# A Conflict-driven Interface between Symbolic Planning and Nonlinear Constraint Solving

Joaquim Ortiz-Haro[1] Erez Karpas[2] Michael Katz[3] Marc Toussaint[1]

***Abstract*— Robotic planning in real-world scenarios typically requires joint optimization of logic and continuous variables. A core challenge to combine the strengths of logic planners and continuous solvers is the design of an efficient interface that informs the logical search about continuous infeasibilities. In this paper we present a novel iterative algorithm that connects logic planning with nonlinear optimization through a bidirectional interface, achieved by the detection of minimal subsets of nonlinear constraints that are infeasible. The algorithm continuously builds a database of graphs that represent (in)feasible subsets of continuous variables and constraints, and encodes this knowledge in the logical description. As a foundation for this algorithm, we introduce *Planning with Nonlinear Transition Constraints (PNTC)*, a novel planning formulation that clarifies the exact assumptions our algorithm requires and can be applied to model Task and Motion Planning (TAMP) efficiently. Our experimental results show that our framework significantly outperforms alternative optimization-based approaches for TAMP.**
**Webpage: https://quimortiz.github.io/graphnlp/**

## I. Introduction

Robot planning involves both discrete and continuous decisions. For example, in Task and Motion Planning (TAMP, [1]), discrete decisions concern which type of interactions with which objects are to be performed, typically formalized using a logic planning language such as STRIPS or PDDL. Continuous decisions concern robot & object poses, motions and potentially forces [2], which need to respect geometric, physical and collision constraints, according to the discrete decisions. The focus of this paper is the case where continuous constraints are formulated as a nonlinear mathematical program (NLP) over continuous variables [3].

Despite recent advances in TAMP solvers, current algorithms struggle in high dimensional configuration spaces (e.g. several robots), large symbolic spaces and constrained environments that require joint optimization. A promising approach to plan in such challenging settings is to efficiently interface state-of-the-art solvers on both sides, in particular, incorporating information about (in)feasibility from continuous solvers back to the logical level. Looking into similar challenges in classical planning formulations, such as Satisfiability Modulo Theory (SMT [4]), a fundamental approach to inform and guide the logical search is to automatically identify and block *minimal conflicts* which guarantee infeasibility of the continuous problem.

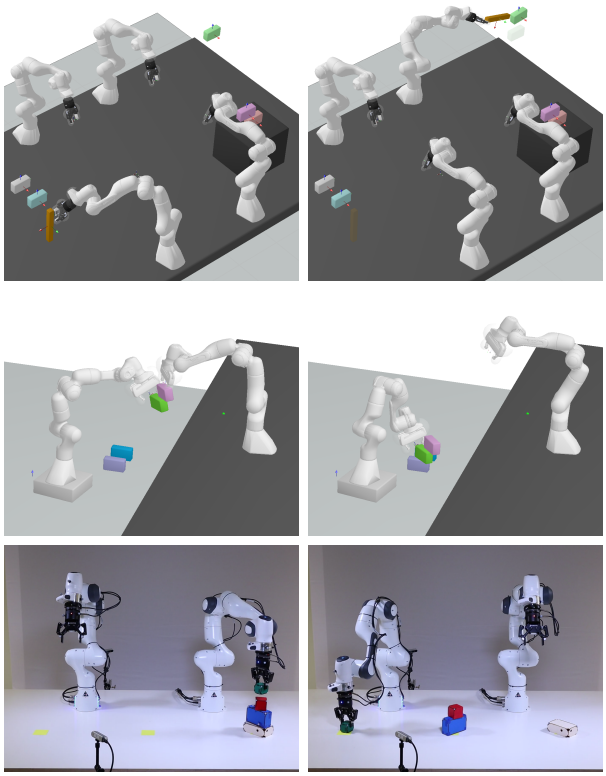

Fig. 1. Task and Motion Planning problems solved by our framework. *Top row*: four robot manipulators use a stick as a tool to reach a distant block. *Mid row*: a heterogeneous team of robots builds a tower. *Bottom row*: two real 7-DOF manipulators solve the Tower of Hanoi puzzle.

In this paper we present the *Graph-NLP Planner (GNPP)*, a novel method to interface a logic solver with an NLP solver, which iteratively detects *minimal infeasible subgraphs*, i.e., minimal subsets of nonlinear constraints that are infeasible, and encodes back this information into the logic problem description. The algorithm continuously builds a database of (in)feasible subgraphs – in a sense learning what is feasible or infeasible – and uses this accumulated information to inform logic search as well as avoid future feasibility checks.

As a foundation for this algorithm we introduce an abstract *Planning with Nonlinear Transition Constraints (PNTC)* formulation, where a symbolic plan implies a factored nonlinear program as a sub-problem and logical predicates of the plan can be related to factors of the NLP. This formulation clarifies the concepts and exact assumptions our algorithm

This research has been supported by the German-Israeli Foundation for Scientific Research (GIF) grant I-1491-407.6/2019. Joaquim Ortiz-Haro thanks the International Max-Planck Research School for Intelligent Systems (IMPRS-IS) for the support.

[1] TU Berlin, Germany.
[2] Technion, Israel.
[3] IBM Research, USA.

builds on, and formally defines the explicit bidirectional relation between the logical and the continuous components of the problem, which is exploited in our solver.

PNTC provides a natural and efficient formulation of TAMP problems in robotics that can be viewed as a variant of a Logic-Geometric Program (LGP) [3]. However, in comparison to LGP, PNTC explicitly defines a factored structure of the implied NLP and a bidirectional mapping between symbols and constraint factors in the resulting NLP. This is exactly the structure we need to inform better the symbolic search and is naturally available in TAMP, making our solver directly applicable to solve TAMP and LGP problems. Our method is evaluated on three robotic TAMP scenarios with complex intrinsic logic-geometric dependencies that require long action sequences, generating a solution in a few seconds – state-of-the-art performance. We deploy the framework in real-world experiments and demonstrate that the solver computes full task and motion plans in real-time.

## II. Related Work

### A. Planning with continuous variables

Classical AI planners have been extended to support planning with numerical constraints on action preconditions (e.g., Metric-FF [5]). Recent versions of the Planning Domain Definition Language (PDDL) include temporal planning with numerical variables [6], [7], [8]. For instance, the COLIN planner [9] includes continuous linear change of numerical variables (e.g., fixed velocities), and encodes the temporal and state evolution constraints implied by a sequence of actions as a linear program. The Scotty planner [10] adds support for control variables which govern the rate of change (e.g., controlling the velocity) by replacing the linear program with a Second Order Cone Program. Perhaps the most closely related work is [11], that extends classical planning with general state constraints. In contrast, the nonlinear constraints of *PNTC* are defined by a sequence of symbolic states and evaluated on consecutive continuous variables. This implies a nonlinear program for the whole sequence of continuous variables, that must be solved jointly, and can model efficiently long-term relationships between the continuous variables without additional discretization.

### B. Task and Motion Planning

Task and Motion Planning in robotics is a prominent example for jointly solving for discrete and continuous variables. Most TAMP solvers rely on a discretization of the configuration space. For instance, PDDLStream [12] uses constrained samplers for generating grasp and kinematic solutions inside PDDL-like planning; and [13] integrates samples of feasible configurations into the planning task through a precompilation.

Some sampling-based TAMP solvers reason explicitly about geometric conflicts. For example, a limited set of predefined predicates such as "is reachable" are used in [14] to combine a black-box task planner with a motion planner. The constraint based approach in [15] incorporates information about geometric infeasibility by blocking the full task plan or, in some special cases, a pair of a (partial) state and an action. Alternatively, our solver detects and encodes any general continuous infeasibility that potentially involves several motion phases. In fact, instead of enumerating possible geometric failure cases, we define nonlinear constraints to model the motion and geometry, and let the solver detect and encode which intrinsic subset is jointly infeasible.

Our method provides an optimization-based formulation of TAMP [16], [17], [18], [19] which leverages nonlinear optimization to jointly compute a motion that satisfies all geometric and physical constraints. In contrast to previous solvers for LGP [18], namely Multi-Bound Tree Search [20], our solver provides a more efficient logic-geometric interface based on detecting infeasible subsets of constraints instead of infeasible action sequences, as our evaluations show.

## III. Problem Formulation

A *Planning with Nonlinear Transition Constraints (PNTC)* problem is a 7-tuple $\langle \mathcal{V}, \mathcal{A}, s_0, g, \Pi, \mathcal{H}, \mathcal{X} \rangle$. The logical component $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ corresponds to a classical planning problem encoded in SAS+ [21]. $\mathcal{V}$ is a finite set of variables and $\mathcal{A}$ is a finite set of action operators. Each variable $v \in \mathcal{V}$ has a finite domain $dom(v)$. A symbolic state $s$ is an assignment to the variables $v \in \mathcal{V}$. A partial state $p$ is an assignment to a subset of variables. $s_0$ is the initial state and $g$ is a partial state that represents the goal. We denote by $\mathcal{P} = \times_{v \in \mathcal{V}}(dom(v) \cup \{\bot\})$ the space of partial states, where $\bot$ means that the partial state does not instantiate a given variable. Each action operator $a \in \mathcal{A}$ is a pair of partial states called preconditions and effects $\langle \text{pre}(a), \text{eff}(a) \rangle$. An action $a$ is applicable in state $s$ if $pre(a) \subseteq s$ and modifies variables in $v \in \text{eff}(a)$. $s' = s[a]$ denotes the resulting state after applying action operator $a$ on $s$.

$\mathcal{X}$ is a finite set of continuous variables $\{x^1 \dots x^K\}$. Each variable takes value in a continuous space $X^k$ (e.g. $\text{dom}(x^k) = X^k = \mathbb{R}^{n_k}$), A continuous state $\underline{x} = \{\underline{x}^1 \dots \underline{x}^K\}$ is a value assignment to $\{x^1 \dots x^K\}$. $\mathcal{H}$ is a finite set of nonlinear piece-wise differentiable constraint functions that are evaluated on a pair of subsets of continuous variables, $\mathcal{H} = \{h_b : X^{b_0} \times X^{b_1} \to \mathbb{R}^{n_b}\}$. The index-sets $b_0, b_1 \subseteq \{1 \dots K\}$ indicate on which subset of variables the function $h_b$ depends on. These functions define nonlinear constraints $h \leq 0$ on two consecutive subsets of continuous variables $(x^{b_0}, x'^{b_1})$.

The logical and continuous components of a *PNTC* are coupled through the mapping $\Pi : \mathcal{P} \times \mathcal{P} \to \mathcal{H} \cup \emptyset$, that maps consecutive partial states $\langle p, p' \rangle$ to a nonlinear constraint function $h_b$ evaluated on $\langle x^{b_0}, x'^{b_1} \rangle$, i.e. $\Pi : (p, p') \mapsto h_b(x^{b_0}, x'^{b_1})$, (the empty set $\emptyset$ highlights that some $\langle p, p' \rangle$ do not generate constraints). This formulation includes dimension-reducing constraints $h = 0$ (rewritten as $h \leq 0$ , $-h \leq 0$) and constraints acting on a single state $\Pi(p) \to h_b(x^{b_0})$.

A solution is a sequence of logical and continuous states $\langle (s_0, \underline{x}_0) \dots (s_n, \underline{x}_n) \rangle$ and action operators $\langle a_1 \dots a_n \rangle$ such that $s_i = s_{i-1}[a_i]$, $g \subseteq s_n$ and $h_b(\underline{x}_i^{b_1}, \underline{x}_{i+1}^{b_0}) \leq 0$, $h_b = \Pi(p, p')$, $\forall p \subseteq s_i, p' \subseteq s_{i+1}, \forall i = 0 \dots n-1$ (using

$\underline{x}_i = \{\underline{x}_i^1 \dots \underline{x}_i^K\}$). Given a fixed logical plan $\langle s_0 \dots s_n \rangle$ the continuous states can be computed by solving the continuous feasibility program, i.e. nonlinear program without costs:

$$\begin{aligned} &\text{find } x_i^k \in X^k, \ \forall i, k \\ &\text{s.t } h_b(x_i^{b_0}, x_{i+1}^{b_1}) \leq 0, \ h_b = \Pi(p, p') \\ &\qquad \forall p \subseteq s_i, p' \subseteq s_{i+1}, \forall i = 0 \dots n-1 \end{aligned} \tag{1}$$

Therefore, a valid logical plan is only a necessary condition for the existence of the full logical and continuous solution and, in practice, valid logical plans often fail at the continuous level.

## IV. Graph-NLP: A Bidirectional Logic-Continuous Interface

Given a fixed sequence of logical states $\langle s_0 \dots s_n \rangle$, we represent the NLP on the continuous variables $x_i^k$ (1) as a graph-NLP, a structured representation that resembles constraint graphs [22], graphical models [23] or factor graphs. We now state the definition and some properties that will later be exploited by our algorithms.

*Definition 4.1:* A graph-NLP $G(\langle s_0 \dots s_n \rangle) = (V, E)$ is a bipartite graph that models continuous variables and constraints (vertices) and their dependencies (edges) for the fixed sequence of logic states $\langle s_0 \dots s_n \rangle$. Formally, $V = X_G \cup H_G$, where $X_G = \{x_i^k, \ i \in 0 \dots n, \ k \in 1 \dots K\}$ and $H_G = \{h_b : h_b = \Pi(p, p') \ \forall p \subseteq s_i, p' \subseteq s_{i+1}, \forall i = 0 \dots n-1\}$, and $E = \{(x_i^k, h_b) : \ h_b \in H_G \text{ depends on } \ x_i^k \in X_G\}$.

In relevant applications, such as TAMP, each constraint $h \in H_G$ depends only on small subsets of variables, which results in sparse and factored graph-NLPs (e.g. Fig. 2).

A subset of variables and constraints is a subgraph-NLP:

*Definition 4.2:* A subgraph-NLP $M \subseteq G$ of a graph-NLP $G = (X_G \cup H_G, E)$ is $M = (X_M \cup H_M, E_M)$ with $X_M \subseteq X_G$, $H_M \subseteq \{h \in H_G \ : \text{Neigh}(h) \subseteq X_M\}$.

A graph-NLP is locally time-connected, and the factors are time-invariant.

*Property 4.1:* (Local time-connectivity) A variable vertex $x_i^k$ is connected to constraints that are evaluated on variables with time-index $i$, $i-1$ or $i+1$.

*Property 4.2:* (Factor time-invariance) A sequence of partial states induces a subgraph $M(\langle p_0 \dots p_L \rangle) = (X_M \cup H_M, E)$, with $H_M = \{h_b : h_b = \Pi(p, p') \ \forall p \subseteq p_l, p' \subseteq p_{l+1}, \ l \in 0 \dots L-1\}$ and $X_M = \{x_l^k, \ l \in 0 \dots L, \ k \in 1 \dots K : \ \exists h \in H_M : \ h \text{ depends on } x_l^k\}$.

*Property 4.3:* If $\langle s_0 \dots s_n \rangle$ contains $\langle p_0 \dots p_L \rangle$ (i.e. $\exists \ i : p_l \subseteq s_{i+l} \ \forall l$), then $M(\langle p_0 \dots p_L \rangle) \subseteq G(\langle s_0 \dots s_n \rangle)$.

*Definition 4.3:* A graph-NLP $G$ is said to be feasible ($\text{Feas}(G) = 1$) if there exists a variable assignment $\underline{x}_i^k, \ \forall x_i^k \in X_G$ such that all constraints $h \in H_G$ are satisfied. Otherwise $G$ is infeasible ($\text{Feas}(G) = 0$). Note that any subgraph $M \subseteq G$ can be evaluated for feasibility.

An assignment can be computed with nonlinear constrained optimization methods such as interior points or augmented Lagrangian (used in our implemenation). The computational complexity (determined by the factorization of a banded diagonal hessian matrix) is $O(nK^3)$.

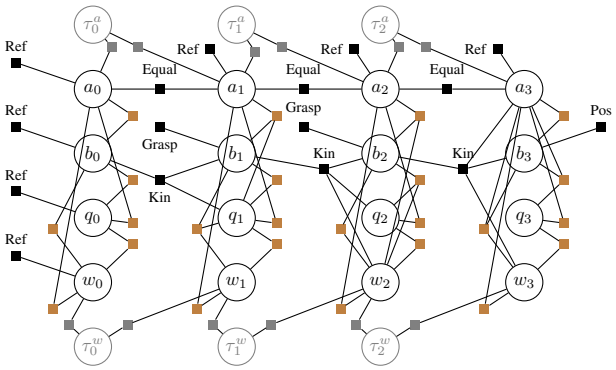


Fig. 2. Graph-NLP of the example domain in Sec. IV-A. Circles are variables and squares are constraints. We display variables for all mode-switch configurations $(a, b, q, w)$, and trajectories variables $(\tau^a, \tau^w)$ (omitting $\tau^b, \tau^q$ and factors that represent collisions between trajectories to keep the illustration clean). Brown squares are collision avoidance constraints. Gray squares are boundary constraints between trajectories and mode-switches. This representation is similar to the constraint graphs in [24].

*Property 4.4 (Monotone Infeasibility):* If a subgraph-NLP $M \subseteq G$ is infeasible, $G$ is infeasible. If $G$ is a feasible graph, $M \subseteq G$ is feasible.

*Definition 4.4:* An infeasible subgraph-NLP is minimal if, when removing one or more variables or constraints, the resulting graph is feasible.

*Property 4.5:* The minimal infeasible subgraph is connected. If the graph-NLP $G$ is not connected, the NLP associated to each connected component $G_i$ can be solved independently. The feasibility of $G$ is $\text{Feas}(G) = \bigwedge_i \text{Feas}(G_i)$.

### A. Example Domain

Consider a basic robot manipulation domain, where we have two movable objects, *OA* and *OB*, initially on *OA_init*, *OB_init*, and two robot manipulators, *RQ* and *RW*, on a *Table* with the goal to stack *OA* on top of *OB*.

The logical description contains abstract information of the structure of the configuration (parent-child relations in the kinematic tree) but without defining the continuous relative pose. For example, the logic variable *parent_OA* with domain {*OA_init, Table, RQ, RW, OB*} indicates whether object *OA* is on the initial position, the *Table*, held by one of the robots or on top of object *OB*. The possible logic decision sequences are defined using a classical planning task with two action operators: *pick* and *place*.

Figure 2 shows the graph-NLP that results from applying the logical decision sequence *pick(OB, OB_init, RQ)*, *pick(OB, RQ, RW)*, *place(OB, RW, OA)* from the starting logic state {*parent_OA=OA_init, parent_OB=OB_init, gripper_RQ=free, gripper_RW=free*}.

In each vertical slice, we have continuous variables $\{a, b, q, w, \tau^a, \tau^b, \tau^q, \tau^w\}$, where $a, b$ are the pose of the object with respect to the parent frame in the kinematic chain (for example, using quaternions for the rotation $a, b \in \mathbb{R}^7$) and $q, w$ are the robot joint configurations (using a 7-DOF manipulator $q, w \in \mathbb{R}^7$). These variables represent the configurations at the beginning of each motion-phase and are usually called mode-switches. $\tau^a, \tau^b, \tau^q, \tau^w \in \mathbb{R}^{20 \cdot 7}$ are the

corresponding trajectories during each motion phase (represented with 20 waypoints). These variables are constrained by nonlinear functions that model grasping (*Grasp*), collision avoidance, kinematic switches (*Kin*), position (*Pos*), time-consistency (*equal*) and reference (*Ref*) constraints.

The constraints operate on pairs of consecutive continuous variables, and the constraints that are applied depend on the values of the logical variables. For instance, a transition of logic variables *Parent_OA = OA_init* $\rightarrow$ *Parent_OA = OA_init* generates $a = a'$ (*equal*$(a, a')$). *Parent_OB = RQ* generates *grasp(b)*, which constrains the relative position of the object to be inside the two fingers of the gripper with a correct orientation. Kinematic switch constraints appear when robots pick/place the objects, e.g., a transition *Parent_OB = OB_init* $\rightarrow$ *Parent_OB' = RQ* generates *Kin*$(b, b', q')$ where *Kin* ensures that the absolute pose of *OB* is kept constant when the robot *RQ* picks the object.

## V. Overview: Graph-NLP Planner

Fig. 3 provides an overview of the Graph-NLP Planner (GNPP) for solving a PNTC that we will introduce in the next sections. To simplify the presentation, we briefly outline each step of the algorithm, which are run iteratively:

1) We leverage a state-of-the-art PDDL planner to find a sequence of logical states that is logically-feasible for the current logical planning task.
2) We generate a nonlinear program with an explicit graph structure, called graph-NLP (Sec. IV) that represents the continuous variables and the nonlinear constraints defined by the logical state sequence.
3) An NLP solver evaluates the graph-NLP. If this NLP is feasible, the algorithm terminates and the output is a solution containing all logic and continuous states. Otherwise a minimal conflict in the form of a minimal infeasible subgraph NLP is extracted and all evaluated subgraph NLPs are stored in the database of feasible and infeasible subgraphs.
4) Finally, we reformulate the logical planning task to forbid all plans that would generate a graph-NLP that contains any subgraph that was already found to be infeasible.

## VI. Finding small infeasible subgraphs

In this section, we discuss how to detect a minimal subset of infeasible constraints from a Graph-NLP (Step 3 of the Graph-NLP Planner, Fig. 3) In the worst case, finding an infeasible subgraph of *minimum cardinality* requires solving a NLP for each subset of constraints $O(2^{|H|})$ [25]. Conversely, a *minimal* infeasible subgraph can be found by solving a linear number of problems [26]. This search can be accelerated with a divide and conquer strategy, with complexity $O(\log |H|)$ [27]. Recently [25] presented a technique for finding an approximately minimal subgraph in a convex optimization problem by solving one convex program with slack variables.

Inspired by these works, we propose an algorithm for finding small minimal infeasible subgraphs that exploits the particular structure of the graph in our setting, namely the time structure of graph-NLPs and the semantic information contained in them, as well as the convergence point of the nonlinear optimizer.

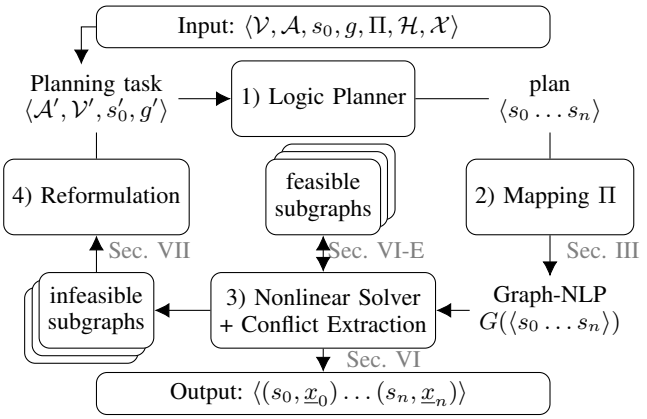


Fig. 3. Overview of the Graph-NLP Planner for solving a PNTC problem $\langle \mathcal{V}, \mathcal{A}, s_0, g, \Pi, \mathcal{H}, \mathcal{X} \rangle$. The solution is a sequence of logic and continuous states $\langle (s_0, \underline{x}_0) \dots (s_n, \underline{x}_n) \rangle$. See Sec. V

### A. Double Binary Search on time index

The first key insight is to exploit the time connectivity of our graph-NLP (Property 4.1). Given an infeasible graph $G(\langle s_0 \dots s_n \rangle)$ we can find a minimal temporal subsequence $\langle s_f \dots s_l \rangle, 0 \leq f \leq l \leq n$ such that $G(\langle s_f \dots s_l \rangle)$ is infeasible with a double binary search that executes $O(\log n)$ calls to a nonlinear optimizer (Properties 4.5 and 4.4). Specifically, we first compute the minimum upper index $l$ such that $G(\langle s_0 \dots s_l \rangle)$ is infeasible. After fixing $l$ we compute the maximum lower index $f$ such that $G(\langle s_f \dots s_l \rangle)$ is infeasible.

### B. Relaxations

Binary search on time exploits the local connectivity in the temporal dimension, but does not detect the infeasible factors inside an infeasible temporal subsequence. To address this issue, we propose to solve a set of relaxations of the graph-NLP that evaluates only a subset of variables and constraints. Each relaxation corresponds to a subgraph-NLP and is, therefore, a necessary condition of feasibility. The algorithm stores the infeasible relaxations as candidates for the minimal subgraph.

The relaxations depend on the semantic information of the variables and constraints and are problem independent but domain specific. Intuitively, we are looking for relaxations that make a graph sparser, smaller and potentially disconnected, while keeping those constraints that define the infeasible subgraph. Section VIII-B presents informative relaxations in the context of Task and Motion Planning.

### C. Leveraging the Convergence Point of the Optimizer

A powerful heuristic to discover a smaller infeasible subset of variables and constraints is to check the convergence point of the optimizer in an infeasible graph.

Given a graph-NLP $G = (X_G \cup H_G, E)$, the nonlinear optimizer aims to compute $x$ s.t $h(x) \leq \mathbf{0}$. Typical optimization methods converge also for infeasible $G$, where we can use the convergence point $x^*$ as a heuristic guess to find a subgraph of $G$ that is infeasible. Specifically, we test the subgraph spanned by the constraints violated at $x^*$, i.e $M' = (X' \cup H', E')$ where $H' = \{h \in H_G : h(x^*) > 0\}$, $X' = \{x \in X_G : \exists h \in \text{Neigh}(x) \text{ s.t } h \in H'\}$. If $M'$ is also infeasible, we consider only $M'$ as a candidate for the minimal infeasible subgraph.

### D. The complete algorithm

We combine these three ideas into an algorithm to find an infeasible subgraph (Alg. 1 of Appendix A[1]). In this algorithm, we alternate between applying relaxations (each relaxation considers only a subset of variables and constraints) that potentially break the full problem into disconnected components, and computing the minimal infeasible time slice inside each connected component (with double binary search). The convergence point of the optimizer is used to reduce the size of the output infeasible subgraph. The algorithm will return the first infeasible subgraph it finds, and therefore it is best to try the relaxations in *loose* to *tight* order, as this will likely result in a smaller infeasible subgraph.

Deciding whether a relaxation should be applied before or after the binary search on the time index is rather arbitrary. To this end, a relevant observation is that solving a small NLP that is feasible is usually an order of magnitude faster than checking that a larger NLP is infeasible. Thus, we try to solve numerous small and feasible problems first.

### E. Database of feasible subgraphs

The graph structure of the graph-NLP is a suitable representation to share information about feasibility between different sequences of logical states. Graphs of different symbolic plans contain common subgraphs, which correspond to sequences of partial states that appear in both plans (potentially at different time indices).

During the execution of the graph-NLP Planner (see Fig. 3), all solved subgraphs are stored either in a feasible or infeasible database. Before solving a nonlinear program in Alg. 1, we check if it is a subgraph of any graph in the *feasible-database*. This check requires a graph isomorphism test [28], based on the adjacency structure and semantic information of variable-vertices, which corresponds to the variable-index $k = 1 \dots K$ (without considering the time index) and the name of the constraint $h \in \mathcal{H}$. Given the available semantic information, the test is fast in practice (complexity close to $(Kn)^2$ instead of the worst case exponential).

### F. Infeasible Subgraphs in TAMP

To conclude the section, Fig. 4 provides two examples of possible infeasible subgraphs of the graph-NLP of the example domain (Fig. 2), together with an intuitive explanation of the underlying reason of continuous infeasibility.

[1]Available in project webpage https://quimortiz.github.io/graphnlp/

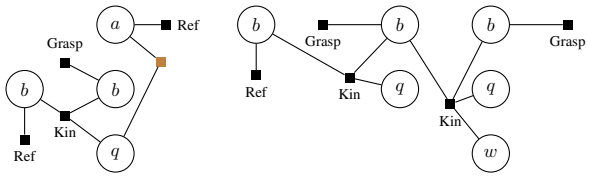


Fig. 4. Two examples of possible infeasible subgraphs of the graph-NLP of the example domain (Fig. 2). *Left*: the robot *RQ* can not pick object *OB* from the initial position if *OA* is on the initial position, e.g. *OA* blocks the grasp of *OB*. *Right*: it is not possible to pick object *OB* with the robot *RQ* and then do a handover to robot *RW*, e.g. due to kinematic constraints, robot *RQ* can only pick the object in a certain way that prevents doing a handover later.

## VII. Logic Reformulation

In this section we discuss how to reformulate the logic task with the information about the continuous infeasibilty (Step 4 of the Graph-NLP planner, Fig. 3). Specifically, given an infeasible subgraph $M$, we modify the logical planning task $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$, to ensure that the logical planner will never generate plans whose graph-NLP contains $M$. The mapping is achieved through a two-step process:

First, we translate the infeasible subgraph $M = (X_M \cup H_M, E_M)$ into a sequence of logical partial states. Recall that each constraint $h \in H_M$ was generated by the forward mapping $\Pi(p, p') \mapsto h$. We now trace this mapping back, to get $(p, p')$ which generated $h$. The relative temporal order of the partial states is kept, resulting in the sequence $\langle p_0 \dots p_L \rangle$

Given an infeasible sequence of partial states $\langle p_0 \dots p_L \rangle$ we introduce a compilation which eliminates plans that satisfy $\langle p_0 \dots p_L \rangle$, similarly to the plan forbidding compilation [29]. Our compilation introduces binary symbolic variables $b_l = \{0, 1\}, l = 0, ..., L$ to indicate whether the path from $s_0$ to $s_k$ contains the infeasible subsequence of partial states. $b_l = 1$ means that the current path contains the first $l + 1$ elements of the infeasible sequence.

Given a planning task $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ and an infeasible sequence $\langle p_0 \dots p_L \rangle$, the new SAS+ task is $\langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$, where:

- $\mathcal{V}' = \mathcal{V} \cup \{b_0, \dots, b_L\}$
- $s_0' = s \cup \{b_l = 0 \mid l = 1, \dots, L\} \cup \{b_0 = 1 \text{ if } p_0 \subseteq s_0;\ b_0 = 0 \text{ otherwise}\}$
- $g' = g \cup \{b_L = 0\}$
- $\mathcal{A}' = \{a' = \text{mod}(a),\ a \in \mathcal{A}\}$

where $a' = \text{mod}(a)$ modifies action $a$ by adding conditional effects to ensure that if action $a$ was executed when $b_{l-1} = 1$, and executing $a$ makes $p_l$ true, then $a$ sets $b_l = 1$ and $b_{l-1} = 0$. Alternatively, if $a$ was executed when $b_{l-1} = 1$, and it does not make $p_l$ true, then $a$ sets $b_{l-1} = 0$. The last binary variable $b_L$ can not transition $1 \to 0$ (i.e. $b_L = 1$ is a dead end). The formal reformulation $a' = \text{mod}(a)$ is shown in the Appendix B. We can now state the proposition which shows that this compilation eliminates exactly all solutions which satisfy $\langle p_0 \dots p_L \rangle$:

*Proposition 7.1:* Let $T = \langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ be a SAS+ planning task, $\langle p_0 \dots p_L \rangle$ be some infeasible sequence, and $T' = \langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$ be the reformulation described above. $\pi$

is a solution of $T'$ iff $\pi$ is a solution of $T$ and the states along $\pi$ do not contain any subsequence of states $\langle s'_i \dots s'_{i+L} \rangle$ such that $p_l \subseteq s'_{i+l}$ for $l = 0, \dots, L$ for some $i$.

Multiple infeasible subsequences are forbidden by iterative reformulation. We are now ready to discuss the properties of our Graph-NLP Planner (Fig. 3):

*Theorem 7.1:* If the underlying classical planner is sound and complete and the nonlinear optimizer always finds a feasible solution if such exists, then the Graph-NLP Planner is sound and complete.

*Proof Sketch:* The proof follows from the fact that any subsequence we forbid can not be part of any feasible solution (because it generates a subgraph found to be infeasible 4.3), together with the fact that our compilation eliminates only plans which contain these subsequences. 7.1). The completeness of the algorithm does not require the infeasible subgraphs to be minimal, nor the mapping $\Pi$. Nonetheless, these properties are desirable for an efficient algorithm.

## VIII. Experimental Results

### A. Benchmark

Our algorithm is evaluated in 3 different simulated scenarios, where the goal is to move obstacles, rearrange and stack up to six blocks to build towers with several robots. The evaluation on real robots is reported in Section VIII-H.

1) *Laboratory (Lab)*: Two 7-DOF manipulator arms execute pick and place actions to build a tower. The solution requires handovers, regrasping and removing obstacles. It is based on the real-world setting, Fig. 5.
2) *Workshop (Work)*: Extension of the *Laboratory* scenario that includes four robots and a stick, that can be grasped and used as a tool to reach blocks, Fig 1.
3) *Field*: Contains a fixed 7-DOF manipulator and a mobile 7-DOF manipulator, with two additional actions operators: *start-move* and *end-move*, for moving the base of the mobile robot on the floor, Fig. 1.

For each scenario we generate 5 different problems by modifying the symbolic goal to increase the complexity in the logical and geometric levels (e.g *Lab_{1,2,3,4,5}*). A subset of these problems, together with the computed solutions, are shown in the project webpage.

### B. Relaxations for finding infeasible subgraphs

The formulation *PNTC* and the solver is general and domain independent. Domain knowledge is introduced through the relaxations used for extracting minimal conflicts (Sec. VI-B). The following relaxations (applicable in any TAMP problem) are used in the benchmark scenarios:

*1) Removal of trajectories:* The remaining graph only contains variables for the mode-switches, considerably reducing the dimensionality of the underlying nonlinear program while still detecting most of the geometric infeasibility.

*2) Removal of collision constraints:* Collision constraints connect all robot configuration and object pose variables in the same time-step, resulting in a densely connected graph (see Fig. 2). Without collisions, the graph becomes sparse, and object and robot variables are only connected by grasping, kinematics and placement constraints.

*3) Removal of Time Consistency:* Time-consistency constraints (*Equal* in Fig. 2) appear when objects are not modified by an action. This relaxation does not consider long term dependencies of the whole manipulation sequence and creates a sparse time-structure.

*4) Removal of robots variables:* The remaining graph considers only the variables for the objects, detecting infeasible placements due to object-object collisions.

### C. Algorithms under Comparison

We compare our approach with two different formulations that combine a logical search with joint nonlinear optimization for solving Task and Motion Planning problems.

*1) One-way interface between Top-K Planning and a nonlinear optimizer (One-way):* This baseline combines Top-K planning [29] to generate a set of different logic plans with a nonlinear optimizer to evaluate the plans. The planner does not receive any information about the geometric reason of infeasibility (only a signal that blocks the evaluated plan).

*2) Multibound Tree Search (MBTS):* The MBTS Solver [20] incrementally builds a tree in a breadth-first order to explore sequences of logic actions that reach the symbolic goal. Instead of solving the full continous optimization problem directly, MBTS computes first relaxed versions (*bounds*) that consider a subset of variables and constraints. The *pose bound* optimizes each mode-switch independently and the *sequence bound* considers the full sequence of mode-switches, that are optimized jointly.

*3) Four Variations of our Graph-NLP Planner (GNPP):* Together with our full planner (*GNPP_trng*), we evaluate three additional versions: *GNPP_t*, *GNPP_tr* and *GNPP_trn* to do an ablation study of the algorithm to extract infeasible subgraphs (Sec VI) The suffixes indicate: $t$ = time search, $r$ = relaxation, $n$ = convergence heuristic and $g$ = feasible graph database.

### D. Metrics

Each algorithm is run 10 times with different random seeds and a timeout of 100 seconds. For each method we report on the number of solved NLPs (*NLP*) and the CPU time (*time*) in table I. Note that "–" means failure to find a solution within 100 seconds with at least 70% success rate.

*Time*[2] provides an objective way to compare algorithms that use different underlying methods. The number of solved NLPs is informative but does not capture the influence of the size and feasibility of NLPs on the running time of the solver.

For each problem, *N* denotes the length of the shortest found plan that is both logical and geometrically feasible and $N_0$ is the length of the logical plan that solves the initial logical task (that is, without considering the continuous information). *N* and $N_0$ are a proxy for the difficulty: the number of candidate plans typically grows exponentially with $N$, and the difference $N - N_0$ shows the impact of the continuous

[2]Experiments are run on Single Core i7-1165G7@2.80GHz

| | *length* | | *One-way* | | *MBTS* | | *GNPP_t* | | *GNPP_tr* | | *GNPP_trn* | | *GNPP_trng* | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | $N_0$ | $N$ | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* |
| Work_1 | 2 | 4 | 77.0 0.0 | 8.2 1.0 | 371 58.5 | 23.5 4.4 | 50.8 12.0 | 5.7 1.2 | 53.0 12.6 | 5.7 1.3 | 60.8 10.5 | 5.6 0.9 | 55.2 1.4 | **5.3** 0.1 |
| Work_2 | 4 | 6 | - | - | - | - | - | - | 113 40.4 | 22.7 9.2 | 94.7 1.7 | 19.7 6.0 | 86.8 1.5 | **16.7** 2.9 |
| Work_3 | 4 | 6 | - | - | - | - | - | - | 105 37.0 | **19.1** 5.3 | 95.6 1.0 | 22.1 5.9 | 86.1 1.5 | 21.4 6.1 |
| Work_4 | 8 | 10 | - | - | - | - | - | - | 282 0.0 | **53.1** 5.6 | 307 1.9 | 56.4 7.6 | 270 1.8 | 55.4 9.0 |
| Work_5 | 8 | 11 | - | - | - | - | - | - | - | - | 355 4.9 | **76.0** 7.0 | 309 5.3 | 76.8 9.4 |
| Lab_1 | 2 | 3 | 25.0 0.0 | 7.1 1.0 | 25.0 0.0 | 4.1 0.5 | 21.0 0.0 | 4.5 0.2 | 25.0 0.0 | **3.2** 0.2 | 30.0 0.0 | 3.3 0.1 | 28.0 0.0 | 3.3 0.2 |
| Lab_2 | 2 | 3 | 12.0 0.0 | 3.1 0.5 | 28.9 0.3 | 3.7 0.5 | 32.0 0.0 | 5.5 0.3 | 46.0 0.0 | 4.3 0.2 | 23.0 0.0 | **2.1** 0.1 | 21.0 0.0 | **2.1** 0.1 |
| Lab_3 | 4 | 5 | 19.0 0.0 | 8.4 1.2 | 34.0 0.0 | 18.5 2.7 | 26.0 0.0 | 5.9 0.4 | 23.0 0.0 | **3.1** 0.2 | 25.0 0.0 | 3.3 0.4 | 24.0 0.0 | 3.2 0.2 |
| Lab_4 | 4 | 9 | - | - | - | - | - | - | - | - | 70.0 0.0 | 6.5 0.6 | 60.1 3.5 | **6.3** 0.4 |
| Lab_5 | 12 | 17 | - | - | - | - | - | - | 87.0 0.0 | **18.7** 1.8 | 93.0 0.0 | 19.1 2.0 | 83.0 0.0 | 19.0 2.0 |
| Field_1 | 2 | 4 | 19.0 0.0 | 7.0 2.0 | 90.0 0.0 | 15.3 3.1 | 19.0 0.0 | 5.7 1.1 | 14.1 0.3 | 2.9 1.4 | 16.0 0.0 | **2.6** 0.3 | 16.0 0.0 | 3.1 1.0 |
| Field_2 | 2 | 6 | - | - | - | - | - | - | 46.0 0.0 | **6.1** 0.4 | 53.0 0.0 | 6.3 0.5 | 52.5 0.5 | 6.3 0.5 |
| Field_3 | 4 | 8 | - | - | - | - | - | - | 75.0 0.0 | **11.7** 1.2 | 84.0 0.0 | 12.3 1.4 | 78.6 0.5 | **11.7** 0.7 |
| Field_4 | 6 | 10 | - | - | - | - | - | - | 67.0 0.0 | **13.2** 1.5 | 77.0 0.0 | 13.6 1.6 | 76.0 0.0 | 13.5 1.3 |
| Field_5 | 6 | 11 | - | - | - | - | - | - | 282 0.0 | 56.5 6.6 | 289 1.0 | **50.7** 5.5 | 262 0.5 | 51.4 6.6 |

TABLE I. Number of NLP evaluations and CPU time, averaged over 10 random seeded runs, with standard deviations in gray.

domain on the logical planner. The approximate branching factor is 12 in *Lab_{1,2,3,4,5}*, 13 in *Field_{2,3,4,5}*, 24 in *Work_{2,3,4,5}*, 4 in *Work_1* and 5 in *Field_1*.

### *E. Comparison to baselines*

Concerning the problems solved, *One-way* and *MBTS* can only solve the easier problems in each scenario while *GNPP_trn/trng* solves all the problems. Our algorithm is significantly faster in the problems solved by *One-way* and *MBTS*, where the more efficient encoding of geometric information reduces the running time.

The success rate of our planners *GNPP_trn/trng* is 100% in all problems except for *Field_5* (80%), *Work_4* (95%) and *Work_5* (90%), where the optimizer fails to solve feasible graph-NLPs in a few runs. The performance of *GNPP_trng* is not affected by the branching factor of the underlying problem and provides good scaling with respect to $N$ and $N - N_0$. The highest computational time correspond to *Field_5* and *Work_5* that require a long plan and detecting collisions between movable objects. In TAMP, the practical size of the graph-NLPs is $O(K^2 n)$ (where $n$ is the length of the action sequence, and $K$ is the number of objects and robots). The domains can be model using a small set ($< 20$) of different types of nonlinear constraints (e.g. Fig. 2).

### *F. Ablation Study*

*1) Analysis of relaxations : GNPP_t* detects conflicts of the form $\langle s_i \dots s_{i+l} \rangle$, while *GNPP_tr* checks relaxations to generate smaller conflicts $\langle p_i \dots p_{i+l} \rangle$. Small conflicts lead to more aggressive pruning of logic plans, and are essential to solve the harder problems (the number of solved problem is 5 vs 13 out of 15). An analysis of the impact of each relaxation is shown in Appendix C, where we conclude that *Removal of trajectories* and *Removal of collision constraints* are the most informative relaxations.

*2) Analysis of the convergence heuristic:* The results show that the convergence heuristic is important in problems that require reasoning about the collisions between movable objects, e.g. when the robot must move one object before placing another to avoid a collision. In this case, the relaxations are not informative, while the convergence point of the optimizer in these infeasible problems usually indicates which are the objects that are in collision. *GNPP_tr* solves 13 out of 15 and *GNPP_trn* solves 15 out of 15.

*3) Analysis of database of feasible graphs : GNPP_trng* reduces the number of solved NLPs, from a total average of 1673 to 1508, but there is not improvement in the computational time. We conjecture that the database approach will provide higher benefits in a setting where solving the NLPs requires more time.

### *G. Scalability and limitations*

The problems in each scenario show the performance of the solver when increasing $N$, $N-N_0$ and the branching factor. We conduct two additional experiments in the *Laboratory* scenario to explicitly evaluate the scalability of the method when increasing the number of blocks to be stacked (from 4 to 32) and the number of movable obstacles in a cluttered table (from 1 to 6). Results are shown in Appendix D. The running time of the Graph-NLP Planner scales polynomially with the number of objects and plan length, and the practical bottleneck is the time spent on solving large nonlinear programs (cubic complexity on the number of objects and linear on the plan length). Notably, good scalability is achieved without decomposing the problem, enabling joint optimization of long term geometric dependencies.

The main weakness of our method is that the nonlinear optimizer is not guaranteed to find a solution for a (sub)graph-NLP even if one exists, given that the nonlinear constraints define a non-convex optimization problem (which could break the assumption in Thr. 7.1). However, the extensive experiments demonstrate that the solver is efficient and reliable in relevant use-cases of TAMP.

### *H. Real-time Planning in the real-world*

We demonstrate our solver in a real-world version of the *Laboratory* environment (two 7-DOF Manipulators and up to 6 movable objects), see Fig. 5. The solver is integrated

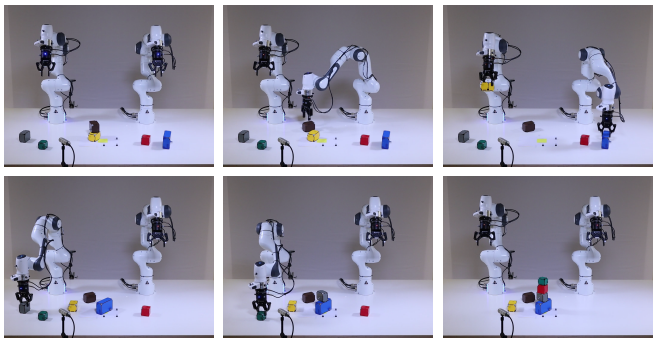

Fig. 5. The task in this real-world experiment is to build the tower *blue-gray-red-green* in the central spot (highlighted in yellow). The solution, computed in only 8.88 seconds with our planner, requires moving the *brown* and *yellow* blocks to avoid collisions and executing a total of 12 actions.

in a *Sense-Plan-Act* pipeline, where we first perceive the scene with an external motion capture system, compute a full (logical and continuous) plan and execute the plan.

The real world evaluation consists of three scenarios: *Tower*, *Hanoi-Tower*, *Obstacles-Tower*, for a total of 11 problems. The symbolic goal is to build a tower of cubes at different locations: *Hanoi-Tower* introduces the classical Hanoi logic constraints and *Obstacles-Tower* requires plans that clear obstacles. The planning time differs across problems: 2.8 s to build a tower of 6 blocks in the center of the table (12 actions), 8.8 s to remove two obstructing blocks and stack 4 blocks (12 actions), 9.4 s to build a Hanoi Tower (12 actions) and 27.2 s to remove obstructing blocks and transfer blocks from the left to right side (16 actions). Recordings of planning and execution are shown in the project webpage.

## IX. Conclusion

We presented a solver that combines nonlinear optimization and PDDL planning for joint optimization of logical and continuous variables in robotic planning. The key contribution is the novel bidirectional interface between logic and continuous constraints, realized through the detection of infeasible subgraphs and a reformulation to inform the logical planner about subgraph infeasibility. The problem formulation is formalized as *PNTC*, which extends classical planning with nonlinear transition constraints.

Our experiments in Task and Motion Planning show that the algorithm is faster and more scalable than Multi-Bound Tree search for LGP, while maintaining the generality and using the same input information. These results are further validated in real-world experiments, where our solver generates plans for two 7-DOF robots with 6 objects in few seconds. As future work, we would like to combine nonlinear optimization with conditional constraint sampling for solving large graph-NLPs, potentially bridging the gap between sample- and optimization-based approaches to TAMP.

## References


[1] C. R. Garrett, R. Chitnis, R. Holladay, B. Kim, T. Silver, L. P. Kaelbling, and T. Lozano-Pérez, "Integrated task and motion planning," *Annual Review of Control, Robotics, and Autonomous Systems*, 2021.

[2] M. Toussaint, J.-S. Ha, and D. Driess, "Describing physics for physical reasoning: Force-based sequential manipulation planning," *IEEE Robotics and Automation Letters*, 2020.

[3] M. Toussaint, "Logic-geometric programming: An optimization-based approach to combined task and motion planning," in *International Joint Conference on Artificial Intelligence, IJCAI*, 2015.

[4] L. De Moura and N. Bjørner, "Satisfiability modulo theories: introduction and applications," *Communications of the ACM*, 2011.

[5] J. Koehler, "Planning under resource constraints." in *ECAI*, 1998.

[6] M. Fox and D. Long, "Modelling mixed discrete-continuous domains for planning," *Journal of Artificial Intelligence Research*, 2006.

[7] W. M. Piotrowski, M. Fox, D. Long, D. Magazzeni, and F. Mercorio, "Heuristic planning for pddl+ domains," in *Workshops at the Thirtieth AAAI Conference on Artificial Intelligence*, 2016.

[8] E. Scala, P. Haslum, S. Thiébaux, and M. Ramirez, "Interval-based relaxation for general numeric planning," in *ECAI 2016*, 2016.

[9] A. J. Coles, A. Coles, M. Fox, and D. Long, "COLIN: planning with continuous linear numeric change," *J. Artif. Intell. Res.*, vol. 44, pp. 1–96, 2012.

[10] E. Fernández-González, B. Williams, and E. Karpas, "Scottyactivity: Mixed discrete-continuous planning with convex optimization," *Journal of Artificial Intelligence Research*, vol. 62, pp. 579–664, 2018.

[11] P. Haslum, F. Ivankovic, M. Ramirez, D. Gordon, S. Thiébaux, V. Shivashankar, and D. S. Nau, "Extending classical planning with state constraints: Heuristics and search for optimal planning," *Journal of Artificial Intelligence Research*, vol. 62, pp. 373–431, 2018.

[12] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Pddlstream: Integrating symbolic planners and blackbox samplers via optimistic adaptive planning," in *Proceedings of ICAPS*, 2020.

[13] J. Ferrer-Mestres, G. Francès, and H. Geffner, "Combined task and motion planning as classical AI planning," *CoRR*, 2017.

[14] S. Srivastava, E. Fang, L. Riano, R. Chitnis, S. J. Russell, and P. Abbeel, "Combined task and motion planning through an extensible planner-independent interface layer," in *ICRA*, 2014.

[15] N. T. Dantam, Z. K. Kingston, S. Chaudhuri, and L. E. Kavraki, "Incremental task and motion planning: A constraint-based approach." in *Robotics: Science and systems*, 2016.

[16] T. Migimatsu and J. Bohg, "Object-centric task and motion planning in dynamic environments," *Robotics and Automation Letters*, 2020.

[17] Z. Zhao, Z. Zhou, M. Park, and Y. Zhao, "Sydebo: Symbolic-decision-embedded bilevel optimization for long-horizon manipulation in dynamic environments," *IEEE Access*, 2021.

[18] M. Toussaint, K. R. Allen, K. A. Smith, and J. B. Tenenbaum, "Differentiable physics and stable modes for tool-use and manipulation planning," in *Robotics: Science and Systems XIV RSS*, 2018.

[19] J. Ortiz-Haro, E. Karpas, K. Michael, and M. Toussaint, "Conflict-directed diverse planning for logic-geometric programming," in *Proceedings of ICAPS*, 2022.

[20] M. Toussaint and M. Lopes, "Multi-bound tree search for logic-geometric programming in cooperative manipulation domains," in *Int. Conf. on Robotics and Automation, ICRA*, 2017.

[21] C. Bäckström and B. Nebel, "Complexity results for $SAS^+$ planning," *Computational Intelligence*, vol. 11, no. 4, pp. 625–655, 1995.

[22] R. Dechter, "Constraint networks," 1992.

[23] D. Koller and N. Friedman, "Probabilistic graphical models: principles and techniques," 2009.

[24] C. R. Garrett, T. Lozano-Pérez, and L. P. Kaelbling, "Sampling-based methods for factored task and motion planning," *The International Journal of Robotics Research*, 2018.

[25] Y. Shoukry, P. Nuzzo, A. L. Sangiovanni-Vincentelli, S. A. Seshia, G. J. Pappas, and P. Tabuada, "Smc: Satisfiability modulo convex programming," *Proceedings of the IEEE*, 2018.

[26] E. Amaldi, M. E. Pfetsch, and L. E. Trotter, "Some structural and algorithmic properties of the maximum feasible subsystem problem," in *Int. Conf. on Integer Progr. and Combinatorial Optimization*, 1999.

[27] U. Junker, "Preferred explanations and relaxations for over-constrained problems," in *AAAI*, 2004.

[28] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento, "A (sub) graph isomorphism algorithm for matching large graphs," *IEEE transactions on pattern analysis and machine intelligence*, 2004.

[29] M. Katz, S. Sohrabi, O. Udrea, and D. Winterer, "A novel iterative approach to top-k planning," in *ICAPS*, 2018.

## APPENDIX

### A. Algorithm for finding infeasible subgraphs

Algorithm 1 shows an heuristic way to combine the three algorithmic ideas ("time binary search", "relaxations" and "convergence heuristic") to detect small infeasible subgraphs (see Sec. VI). It achieves a good experimental performance with an easy implementation. More complex and efficient conflict extraction strategies will be explored in future work.

**Algorithm 1** Finding a small infeasible subgraph

**Input:** Graph-NLP $G(\langle s_0 \dots s_n \rangle)$,
Relaxation Rules $R = \{r\}$
**Output:** Small Infeasible Subgraph $M \subseteq G$
**for** $r$ in $R$ **do**
  $G_r \leftarrow r(G)$  ▷ $r$ deletes variables and constraints
  $CG_r \leftarrow$ **ConnectedComponents**$(G_r)$
  **for** $g$ in $CG_r$ **do**
    feasible, $[t_f, t_l] \leftarrow$ **BinarySearchTime**$(g)$
    **if** not feasible **then**
      $g_s \leftarrow$ **Convergence Heuristic**$(g[t_f, t_l])$
      **return** $g_s$

### B. Logic reformulation

Given a planning task $\langle \mathcal{V}, \mathcal{A}, s_0, g \rangle$ and an infeasible sequence $\langle p_0 \dots p_L \rangle$, the new SAS+ task is $\langle \mathcal{V}', \mathcal{A}', s_0', g' \rangle$, where:

- $\mathcal{V}' = \mathcal{V} \cup \{b_0, \dots, b_L\}$
- $s_0' = s \cup \{b_l = 0 \mid l = 1, \dots, L\} \cup \{b_0 = 1 \text{ if } p_0 \subseteq s_0;\ b_0 = 0 \text{ otherwise}\}$
- $g' = g \cup \{b_L = 0\}$
- $\mathcal{A}' = \{a' = \text{mod}(a),\ a \in \mathcal{A}\}$

To formally describe $a' = \text{mod}(a)$, we treat the partial assignment $p_l$ as a set of facts, and add the following conditional effects to $a$: $(\bigwedge_{f \in p_0 \setminus \text{eff}(a)} f) \triangleright (b_0 \rightarrow 1)$; and for every $l = 1, \dots, L$: $(b_{l-1} = 1) \wedge (\bigwedge_{f \in p_l \setminus \text{eff}(a)} f) \triangleright (b_{l-1}, b_l) \rightarrow (0, 1)$ and $(b_{l-1} = 1) \wedge \neg(\bigwedge_{f \in p_l \setminus \text{eff}(a)} f) \triangleright (b_{l-1}) \rightarrow 0$, where the notation $A \triangleright B \rightarrow 1$ means "if $A$ , then $B \rightarrow 1$". If the effects of $a$ are inconsistent with $p_l$ (that is, one of the effects of $a$ assigns a different value to one of the variables in $p_l$), the expression $(\bigwedge_{f \in p_l \setminus \text{eff}(a)} f)$ always evaluates to false.

Finally, we remark that avoiding a sequence of states which satisfies $\langle p_0 \dots p_L \rangle$ can be encoded as a PDDL 3 trajectory constraint. The above-mentioned compilation is a special case.

### C. Experimental study of relaxations

We analyze in detail the 4 relaxations to detect minimal conflicts in TAMP problems (see Sec. VI-B):

- Removal of trajectories (*No Traj*)
- Removal of collision constraints (*No Col*)
- Removal of Time Consistency (*No Time*)
- Removal of robots variables (*No Robot*)

In the experimental evaluation, our algorithms *GNPP_tr*, *GNPP_trn*, and *GNPP_trng* check the following relaxation rules before solving the complete Graph-NLP.

1) *No Robot + No Col + No Traj*
2) *No Col + No Traj*
3) *No Robot + No Time + No Traj*
4) *No Time + No Traj*
5) *No Traj*

Namely, we first apply (1) to the original graph-NLP and check if it is feasible (breaking the full graph into disconnected components, and doing binary search on the time index, see Alg. 1). If feasible, apply (2) to original graph and check if it feasible. If feasible, apply (3) and so forth. The algorithm stops the first time it finds an infeasible subgraph. If all tested relaxations are feasible, we solve for the Full Graph-NLP.

To analyze the influence of each individual relaxation: "*Remove Collisions*", "*Remove Robots*" and "*Remove Time Consistency*", we evaluate the following relaxation rules:

- Without *No Col* (*GNPP_tnr1*)
  1) *No Robot + No Time + No Traj*
  2) *No Robot + No Traj*
  3) *No Time + No Traj*
  4) *No Traj*
- Without *No Time* (*GNPP_tnr2*)
  1) *No Robot + No Col + No Traj*
  2) *No Robot + No Traj*
  3) *No Col + No Traj*
  4) *No Traj*
- Without *No Robot* (*GNPP_tnr3*)
  1) *No Col + No Time + No Traj*
  2) *No Col + No Traj*
  3) *No Time + No Traj*
  4) *No traj*

Results are shown in Table II. We use the version of our algorithm called *GNPP_trn* (with relaxation rules *"r"*, search on the time index *"t"* and convergence heuristic *"n"*). *GNPP_tnr0* is the default implementation using all the relaxations. *GNPP_tnr1*, *GNPP_tnr2*, and *GNPP_tnr3* apply three different relaxations rules (see above).

The worst performing variation is *GNPP_trn1* (without *No Col*). This highlights that *No col* relaxation is fundamental to detect small conflicts in TAMP, as it makes the Graph-NLP sparse and, when combined with the search on the time index, highly disconnected. The best performing variation is *GNPP_trn3*, suggesting that *No Robot* relaxation is in fact not useful to improve the running time. Finally, we remark that the overall performance of each variation is influenced

| | length | | *GNPP_tnr* | | *GNPP_tnr1* | | *GNPP_tnr2* | | GNPP_tnr3 | |
|---|---|---|---|---|---|---|---|---|---|---|
| | *N_0* | *N* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* | *NLP* | *time* |
| Work_1 | 2 | 4 | 57.3 1.9 | 5.3 0.1 | 93.6 2.3 | 4.8 0.1 | 88.8 1.4 | 5.3 0.1 | 46.8 0.7 | 4.7 0.0 |
| Work_2 | 4 | 6 | 95.0 0.0 | 15.6 3.5 | 195 0.5 | 33.7 6.0 | 164 1.4 | 15.4 2.6 | 78.5 1.9 | 14.7 4.1 |
| Work_3 | 4 | 6 | 96.2 2.4 | 15.7 3.5 | 196 0.5 | 30.2 2.2 | 164 0.4 | 16.3 3.7 | 78.1 1.6 | 13.0 3.1 |
| Work_4 | 8 | 10 | 308 2.4 | 52.5 4.3 | 1042 2.8 | 111 0.8 | 744 5.1 | 57.1 3.3 | 535 1.2 | 38.3 2.8 |
| Work_5 | 8 | 11 | 353 0.4 | 71.7 5.6 | - | - | 874 24.7 | 79.6 5.7 | 633 2.4 | 51.8 3.6 |
| Lab_1 | 2 | 3 | 30.0 0.0 | 3.6 0.1 | 57.0 0.0 | 4.9 0.1 | 46.0 0.0 | 3.6 0.1 | 30.0 0.0 | 3.4 0.2 |
| Lab_2 | 2 | 3 | 23.0 0.0 | 2.1 0.1 | 31.0 0.0 | 2.2 0.1 | 27.0 0.0 | 2.1 0.1 | 25.0 0.0 | 2.1 0.1 |
| Lab_3 | 4 | 5 | 25.0 0.0 | 3.4 0.3 | 50.6 1.1 | 4.2 0.3 | 42.0 0.0 | 3.4 0.2 | 30.0 0.0 | 3.4 0.2 |
| Lab_4 | 4 | 9 | 71.8 4.9 | 6.9 0.6 | 135 0.0 | 7.9 0.5 | 164 3.8 | 7.6 0.5 | 110 6.8 | 6.5 0.3 |
| Lab_5 | 12 | 17 | 93.0 0.0 | 20.4 0.9 | 285 0.0 | 27.0 1.2 | 278 0.0 | 20.5 1.4 | 207 0.0 | 14.3 0.7 |
| Field_1 | 2 | 4 | 16.0 0.0 | 3.2 1.1 | 27.0 5.6 | 3.6 0.4 | 23.0 0.0 | 3.1 0.9 | 16.0 0.0 | 2.5 0.2 |
| Field_2 | 2 | 6 | 53.0 0.0 | 6.2 0.3 | 99.0 0.0 | 12.9 0.5 | 75.0 0.0 | 6.5 0.3 | 56.0 0.0 | 6.1 0.3 |
| Field_3 | 4 | 8 | 84.0 0.0 | 12.6 0.8 | 226 26.9 | 27.6 2.1 | 141 4.9 | 12.8 1.1 | 138 0.0 | 13.0 0.8 |
| Field_4 | 6 | 10 | 77.0 0.0 | 13.4 0.7 | 247 1.9 | 30.5 1.2 | 195 0.0 | 14.5 1.0 | 188 0.0 | 14.5 0.6 |
| Field_5 | 6 | 11 | 289 0.0 | 50.3 1.1 | 842 | 103 | 739 56.6 | 57.5 8.5 | 573 1.1 | 46.4 0.9 |
| total | | | 1672 | 283 | 3526 | 404 | 3766 | 305 | 2743 | 235 |

TABLE II. Evaluation of different relaxations in TAMP. Number of NLP evaluations and CPU time, averaged over 10 random seeded runs, with standard deviations in gray. "–" indicates consistent failure to solve a problem within 100 seconds.

by the *convergence heuristic*, that potentially reduces the size of the infeasible subgraph and outputs small conflicts even when no informative relaxation is used.

Note that the relaxation *No Traj* is applied in all the relaxation rules, and we only compute trajectories if the previous relaxations are found to be feasible (that is, when solving for the full Graph-NLP). The justification is twofold: first, the nonlinear optimization with trajectories contains 20 times more scalar variables than the optimization without trajectories (as each trajectory is represented with 20 waypoints, while a mode-switch corresponds to a single waypoints), and is an order of magnitude slower to solve. Second, to solve the optimization problem including the trajectories, a good strategy is to 1) compute the mode-switches, 2) warmstart the trajectories with a linear interpolation between the mode-switches, and 3) reoptimize the trajectories and mode-switches jointly.

### *D. Extended study of Scalability: Objects and Obstacles*

In this section, we conduct a study of the scalability of our solver to when increasing number of objects and obstacles. These results extend the Benchmark in the experimental Results (Sec. VIII and Table I).

The new problems are based on the *Laboratory* scenario.

- *Stacking Boxes*. Two 7-DOF manipulator arms execute pick and place actions to stack blocks in small towers of two. We increase the number of blocks in each instance, from 4 to 32. See Fig. 6.
- *Placement in a cluttered Table*. Two 7-DOF manipulator arms execute pick and place actions to place 6 blocks into their goal position and orientation, which requires to detect possible collisions and move obstacles around the table. We increase the number of movable obstacles from 1 to 6. See Fig. 7.

Results are shown in Fig. 9 and Fig. 8.

*Stacking Boxes* - Our solver scales polynomially to the number of objects in the scene, while relying on joint optimization and not using hand-crafted problem decompositions or sampling of partial solutions. The number of evaluated action plans scales linearly with the number of objects, but the computational time spent on solving the (sub)graph-NLPs increases polynomially with the size of the nonlinear optimization problem. In fact, the largest problem requires a motion plan of 32 actions. The optimization of the full motion using joint optimization is not the most efficient approach in this setting, and could be improved with an heuristic strategy that fixes mode-switches (that are

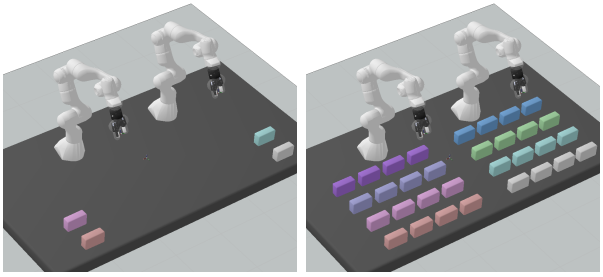

Fig. 6. *Stacking boxes*. The goal is to stack blocks in pairs: e.g. put orange on top of pink and white on top blue. *Left*: 4 objects. *Right*: 32 objects.

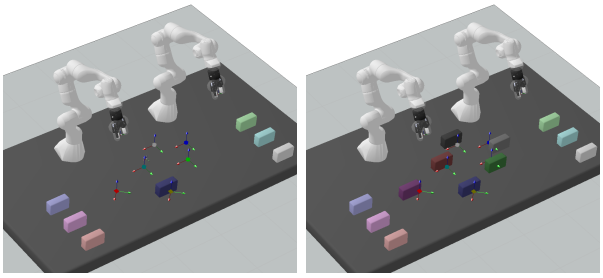

Fig. 7. *Placement in a cluttered table*. Dark colors denote movable obstacles. The frame markers show the goal position of the objects. *Left*: 1 obstacle. *Right*: 6 obstacles.

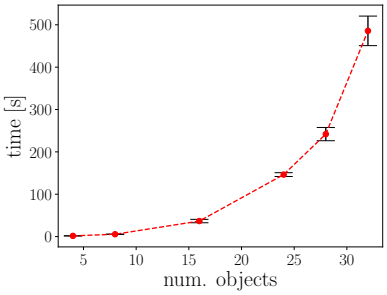


Fig. 8. Computational time in *Stacking boxes*. See Fig. 6. We report mean and standard deviation over 10 runs.

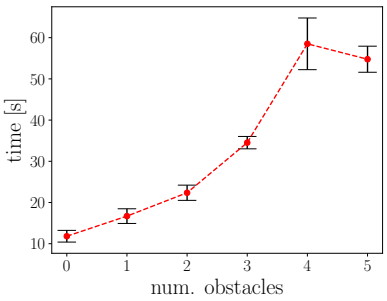


Fig. 9. Computational time in *Placement in a cluttered Table*. See Fig. 7. We report mean and standard deviation over 10 runs.

optimized jointly) and solves the trajectory individually for each step.

*Placement in a cluttered table* - In this setting, our solver scales linearly with the number of obstacles. This is achieved by the efficient detection of minimal infeasible subgraphs, and encoding of the conflict into the logic problem description (in this case, which obstacles are blocking the placement of a block).

Based on the results of the main benchmark (Tab. I) and the scalability study (Fig. 8 and 9), we can conclude that the running time of our solver depends on:

- *Solving the Logic Problem*: fast in practice using the logic encoding of geometric conflicts and a state-of-the-art PDDL planner (the worst case time complexity is exponential on the action branching factor).
- *Number of iterations*: the efficient detection and encoding of geometric information achieves practical linear complexity (the worst case is exponential in the action branching factor).
- *The number of evaluated subgraphs*: for each new symbolic sequence, the algorithm to detect infeasible subgraphs evaluates a number of subgraphs that is linear in the number of objects (in practice) and logarithmic in the length of the action sequence. Worst case is exponential in number of objects.
- *Solving nonlinear programs*: the theoretical time complexity is cubic in the number of objects and robots and linear in the length of the action sequence. The practical complexity is worse, because the nonlinear optimizer usually requires more iterations to solve larger NLPs.