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 the 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 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 by the *convergence heuristic*, that potentially reduces the size

	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. Analysis 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.

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 all 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 problem with trajectories contains 20 times more scalar variables than the optimization without trajectories (each trajectory is represented with 20 waypoints, while a mode-switch is represented with a single waypoint), and is an order of magnitude slower to solve. Second, a good strategy to solve the optimization problem including the trajectories 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 study the scalability of our solver when increasing the 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 blocks. 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 a 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. In this setting, generating a full motion using joint optimization is not the most efficient approach, and could be improved with an heuristic strategy that fixes mode-switches (that are optimized jointly) and

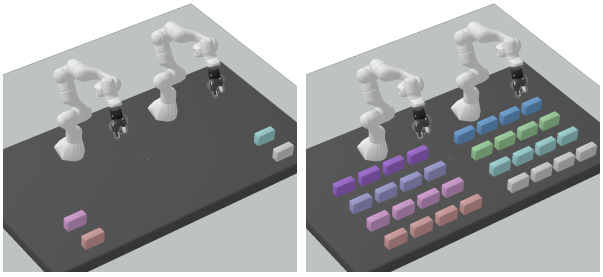

Fig. 6. *Stacking boxes*. The goal is to stack blocks in pairs, e.g. put the orange one on top of the pink and the white on top of the blue block. *Left*: 4 objects. *Right*: 32 objects.

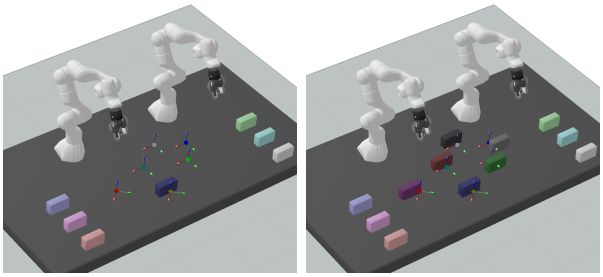

Fig. 7. *Placement in a cluttered table*. Dark colors denote movable obstacles. The frame markers show the goal positions and orientations for the objects. *Left*: 1 obstacle. *Right*: 6 obstacles.

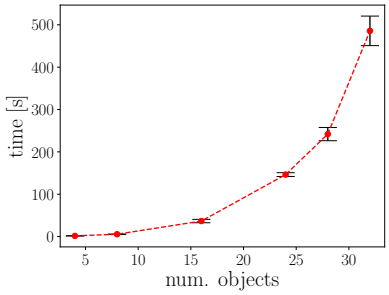

Fig. 8. Computational time in *Stacking boxes*. See Fig. 6. We report mean and standard deviation over 10 runs.

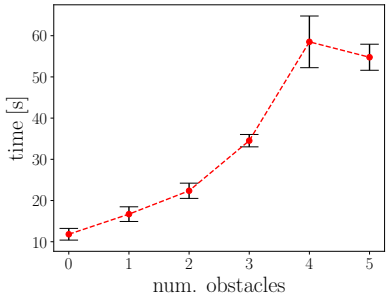

Fig. 9. Computational time in *Placement in a cluttered table*. See Fig. 7. We report mean and standard deviation over 10 runs.

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 and encoding of minimal infeasible subgraphs (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 in the action branching factor).
- *Number of iterations of GNPP*: 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 the 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.