Temporal Planning with Problems Requiring Concurrency through Action Graphs and Local Search[⋆]

Alfonso E. Gerevini, Alessandro Saetti, and Ivan Serina

Dipartimento di Ingegneria dell'Informazione,
Università degli Studi di Brescia,
via Branze 38, 25123 Brescia, Italy
{gerevini,saetti,serina}@ing.unibs.it

Abstract. We present an extension of the planning framework based on action graphs and local search to deal with PDDL2.1 temporal problems requiring concurrency, while previously the approach could only solve problems admitting a sequential solution. The paper introduces a revised plan representation supporting concurrency and some new search techniques using it, which are implemented in a new version of the LPG planner. An experimental analysis indicates that the proposed approach is suitable to temporal planning with requiring concurrency and is competitive with state-of-the-art planners.

1 Introduction

Recent work on temporal planning, e.g., [1–3], has emphasized the practical and theoretical importance of addressing problems where action concurrency is necessary to find a valid plan, and which cannot be solved by most of the current temporal planners. In this paper, we present an extension of the temporal planning framework based on action graphs and local search [8] to deal with PDDL2.1 problems requiring concurrency. Previously the approach could only solve problems admitting a sequential solution (no necessary action overlapping in the plan), because of a strong assumption in the plan representation on the possible ordering of the start/end times of the durative actions in the plan.

In the proposed revised approach, each domain action a is represented by a pair of instantaneous snap actions, distinguishing the start and end of a, which are denoted by $s(a)$ and $e(a)$, respectively, and by appropriate temporal constraints between the corresponding time points. The preconditions of $s(a)$ are the start and overall conditions of a, while its effects are the start effects of a. The preconditions of $e(a)$ are the end conditions of a, while its effects are the end effects of a.[1] Essentially, a plan is represented

Proceedings of the 21st RCRA workshop on *Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion* (RCRA 2014).
Vienna, Austria, July 17–18, 2014.

[⋆] This paper has been already published in the Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 2010).

[1] A similar compilation of the domain actions into snap actions has been used in [1, 2, 9].

by an action graph modeling the causal/logical relations of the snap actions and a set of temporal constraints between the start/end times of the corresponding domain actions.

The paper introduces some new local search techniques for the revised representation, which are implemented in a new version of the LPG planner [8]. An experimental analysis indicates that our approach is suitable to temporal planning with problems requiring concurrency and is competitive with state-of-the-art planners.

2 Temporal Action Graph with Concurrency

In this section, we give the necessary background and we introduce a plan representation supporting action concurrency. A *temporal action graph* [8], abbreviated with *TA-graph*, for a planning problem is a directed acyclic leveled graph alternating a *fact level* and an *action level*. Like in Graphplan, fact levels contain *fact nodes* labeled by ground predicates, and each fact node f at a level l is associated with a *no-op* action node at level l representing a dummy action having f as its only precondition and effect. Each action level contains one action node labeled by the name of the domain action that it represents, and the no-op nodes corresponding to that level. The edges of the graph directly connect each action node a at a level l with the fact nodes at l representing the conditions of a (*precondition nodes*) and with the fact nodes at level $l+1$ representing the positive effects of a (*effect nodes*). The initial and final action levels contain the special nodes a_{start} and a_{end} representing the problem initial state and goals, respectively.

A TA-graph $\mathcal{A}$ also contains a set $C_{\mathcal{A}}$ of *temporal constraints* between the start/end of the actions in the (partial) plan represented by the graph [8, 7]: (i) ordering constraints generated to deal with mutually exclusive actions (if the domain actions labeling two action nodes are mutex, they are ordered according to the levels of their action nodes and the type of condition/effect involved in the interference); (ii) ordering constraints implied by the causal relations between the planned actions; (iii) temporal constraints encoding the planned action durations. $C_{\mathcal{A}}$ can be specified as a *Simple Temporal Problem* (STP) [4], i.e., a set of constraints of form $y - x \leq t$, where y and x are point variables and $t \in \mathbb{R}$.[2]

A solution of an STP is an assignment of values to its point variables satisfying every constraint in the STP. An STP C has a solution iff the *distance graph* of the STP, indicated with $\mathcal{D}(C)$, does not contain a negative cycle [4]. Given an STP encoding $C_{\mathcal{A}}$, a solution where every variable (vertex of $\mathcal{D}(C_{\mathcal{A}})$) has the shortest distance from the point variable for a_{start} (with a "tolerance" value for handling strict inequalities) can be computed in $O(n^3)$ time for n variables in $C_{\mathcal{A}}$ [4, 6]. A possible schedule of the actions in the plan represented by $\mathcal{A}$ can be derived from this solution, and defines the *time values* associated with the action nodes of $\mathcal{A}$.

In order to support problems with required concurrency, we have revised the TA-graph using snap actions instead of domain actions for labeling the action nodes. We call the revised representation *TA-graph with concurrency*. Figure 1 shows an example of TA-graph with concurrency $\mathcal{A}$ containing four snap actions generated from two planned

[2] Depending on the type of action condition and effect involved in the causal/mutex relation generating an ordering constraint in $\mathcal{C}_{\mathcal{A}}$, the inequality can be strict or non-strict. Strict inequality in an STP can be handled as shown in [6].

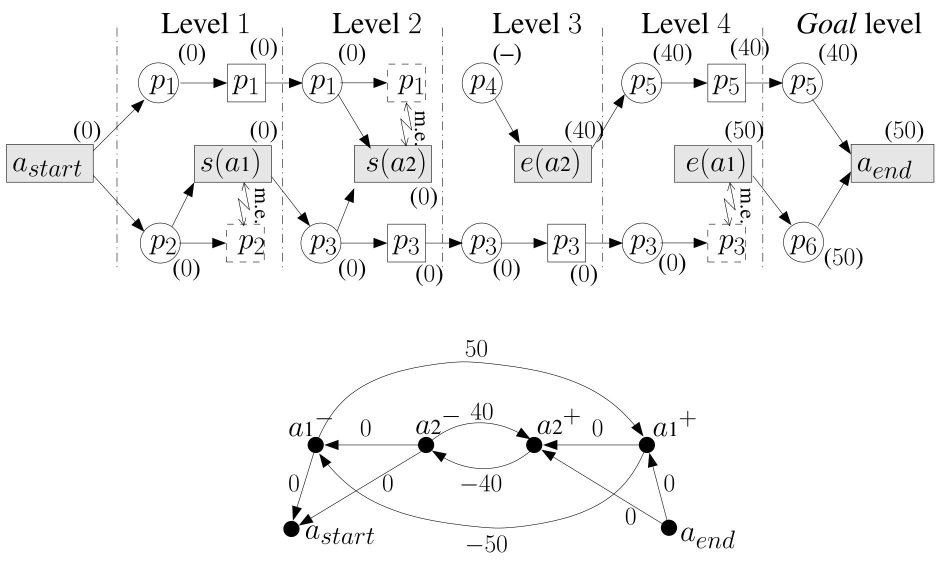


Fig. 1. An example of TA-graph $\mathcal{A}$ with snap actions (action graph on top and $\mathcal{D}(C_{\mathcal{A}})$ at the bottom) supporting the required concurrency of actions $a1$ and $a2$. For the action graph: square nodes are action nodes; circle nodes are fact nodes; dashed nodes are no-ops whose propagation is blocked by mutex actions (abbreviated with "m.e."); the numbers in round brackets are the action start times, with "(–)" indicating an undefined time. For $\mathcal{D}(C_{\mathcal{A}})$: circle nodes are point variables (action start/end times); an edge from x to y labeled t represents constraint $y - x \leq t$; the durations of $a1$ and $a2$ are 50 and 40, respectively.

actions a_1 and a_2 that must be executed concurrently, and the relative distance graph $\mathcal{D}(C_{\mathcal{A}})$. In the example and in the rest of the paper, a^- and a^+ indicate the start and end times of a, respectively. The duration d of an action a is encoded in $C_{\mathcal{A}}$ by constraints $a^+ - a^- \leq d$, $a^- - a^+ \leq -d$. Each causal/mutex relation involving a planned snap action $s(a)$ ($e(a)$) generates an ordering constraint in $C_{\mathcal{A}}$ involving a^- (a^+), with the following exception needed to properly "protect" overall conditions when dealing with interfering actions: if an action node x at a level k interferes with an overall condition of a snap action node $s(a)$ at a level $l < k$ (hence x and $s(a)$ are mutex), then the ordering constraint generated for handling this interference involves a^+ rather than a^-.

For instance, suppose that in Figure 1 fact p_3 is a start effect of $a1$ and an overall condition of $a2$, and that $\neg p_3$ is an end effect of $a1$. Constraint $a1^- - a2^- \leq 0 \in C_{\mathcal{A}}$ because $s(a1)$ supports precondition node p_3 of $s(a2)$; $a2^+ - a1^+ \leq 0 \in C_{\mathcal{A}}$ because $e(a1)$ interferes with condition overall p_3 of $s(a2)$.

A TA-graph with concurrency $\mathcal{A}$ may contain two types of flaw: unsupported precondition ode (*propositional flaw*), and maximum level snap action node involved in a negative cycle of $\mathcal{D}(C_{\mathcal{A}})$ (*temporal flaw*). This definition of temporal flaw is motivated by the following property (the proof is omitted for lack of space): *the minimum level temporal flaw is an end snap action $e(a)$ identifying the endtime of a planned action a whose duration constraint cannot be satisfied given the duration constraints of the actions that, according to $C_{\mathcal{A}}$, must occur concurrently with a.*

For example, in Figure 1 p_4 at level 3 is a propositional flaw; if the duration of $a2$ were 60, $e(a1)$ would be a temporal flaw, because $\mathcal{D}(C_{\mathcal{A}})$ would contain the negative cycle $a1^+ \rightsquigarrow a2^+ \rightsquigarrow a2^- \rightsquigarrow a1^- \rightsquigarrow a1^+$. A TA-graph with no flaw represents a valid plan.

3 Local Search for TA-graphs with Concurrency

Our approach is based on the local search procedure Walkplan [8]. Starting from an initial TA-graph formed by only action nodes a_{start} and a_{end}, each search step identifies the flaw σ at the *minimum* level, and defines the elements of the search neighborhood of the current TA-graph (search state) as a set of modified TA-graphs where σ is repaired. A relaxed-plan based heuristic is used to select an element from the neighborhood as the best candidate for the next search state. This choice is randomized by a noise parameter helping to escape from possible local minima.

In this section, we propose new definitions of search neighborhood and an extended heuristic function for using Walkplan in the context of TA-graphs with concurrency. The *basic search neighborhood* $N(\mathcal{A})$ *of* $\mathcal{A}$ *for a propositional flaw* σ is obtained by adding/removing a *pair* of "twin" snap-action nodes $s(a)$ and $e(a)$ (with the level of the first preceding the level of the second) such that their addition/removal would remove σ.[3] The addition/removal of $s(a)$ and $e(a)$ requires an update of $C_{\mathcal{A}}$ for adding/removing the duration constraints of a and the appropriate ordering constraints of type (i)-(ii) involving a^- and a^+, which have been described in the previous section.

In the following, x_l denotes an action node x at level l of the current TA-graph $\mathcal{A}$, and l_x the level of x. The *search neighborhood* $N(\mathcal{A})$ *of* $\mathcal{A}$ *for a temporal flaw* $e(a)_l$ is obtained by the following possible graph changes and corresponding updates of $C(\mathcal{A})$:

(T1) removing $e(a)_l$ and its twin action node $s(a)_h$ $(h < l)$,
(T2) removing a snap action node x associated with a point variable on a negative cycle ω in $\mathcal{D}(C_{\mathcal{A}})$ such that the maximum level action node of ω is $e(a)_l$ (also the twin snap action node of x is removed),
(T3) removing $e(a)$ from level l and adding it to a level j, with $h < j < l$, such that the STP formed by the temporal constraints involving the action nodes at levels lower than or equal to j in the resulting TA-graph is consistent.[4]

For instance, if $e(a1)_4$ were a temporal flaw in the graph of Figure 1, an example of T3 would be removing $e(a1)_4$ and adding $e(a1)_2$, since the STP involving only actions $s(a1)_1$ and $e(a1)_2$ would be consistent (in the resulting TA-graph, $a2$ would not be requested to be during $a1$ anymore).

The search neighborhood for a TA-graph with concurrency can be considerably larger that the neighborhood for the original TA-graph, where each graph modification consists of adding or removing only *one* action node (domain action) instead of two snap action nodes. Table 1 gives empirical evidence of this for some problems without required concurrency, which can be solved by both representations. The neighborhood of a TA-graph with snap actions is on average two orders of magnitude larger than the

[3] When a pair of snap-action nodes is added, the graph is extended by two levels, and when a pair is removed, the graph is "shrunk" by two levels. More details in [8].
[4] Level j exists because $e(a)_l$ is the minimum level flaw of $\mathcal{A}$.

neighborhood of a TA-graph with domain actions. As a consequence, the local search procedure with snap actions is significantly slower.

In order to restrict the search neighborhood for TA-graphs with concurrency, we propose an alternative definition for handling propositional flaws, which constrains the possible ways of adding snap actions and considers an additional type of graph modification. The *restricted search neighborhood $N(\mathcal{A})$ of $\mathcal{A}$ for a propositional flaw σ* at a level l is obtained by the following possible graph changes and corresponding updates of $C(\mathcal{A})$:

(P1) adding two snap action nodes $s(a)_i$ and $e(a)_{l+1}$ s.t. $i \leq l$ and the addition of $s(a)_i$ removes σ,
(P2) adding two snap action nodes $s(a)_{j-1}$ and $e(a)_j$ s.t. $1 < j \leq l$ and the addition of $e(a)_j$ removes σ,
(P3) removing a snap action node x (and its twin snap action node) s.t. the removal of x would remove σ,
(P4) removing a snap action node x from a level $k < l$ s.t. the removal of x would remove σ, and adding it to level $l + 1$.

Remarkably, we have experimentally observed that the size of the restricted neighborhood for snap-actions is similar to the size of the neighborhood for domain actions, and LPG with the restricted neighborhood performs much better than with the basic one defined above.

The elements in $N(\mathcal{A})$ are evaluated using a heuristic evaluation function E consisting of two weighted terms, estimating the *search cost* and *temporal cost* for the elements in $N(\mathcal{A})$, respectively. The temporal cost term is the same as the one defined for TA-graphs with domain actions [8]. In the rest of this section, we focus only on the search cost term of E for TA-graphs with concurrency, considering first the cost for adding a pair of snap actions $s(a)$ and $e(a)$ (changes P1 and P2). For each added snap action, the search cost is estimated by constructing a *temporal relaxed plan*. The algorithm for constructing the relaxed plan is similar to the one given in [8]. The overall search cost of adding a pair of snap actions is the sum of the number of actions forming the relative relaxed plans.

For the addition of $e(a)$ to $\mathcal{A}$, we construct a relaxed plan achieving three sets of facts (the first two of which are also considered in the original heuristic function for TA-graphs without concurrency): (1) the unsupported preconditions of $e(a)$; (2) the supported preconditions of other action nodes in $\mathcal{A}$ that would become *un*supported by adding $e(a)$; and (3) the supported preconditions that would become unsupported by removing the earliest action node becoming a temporal flaw if $e(a)$ were added to $\mathcal{A}$. Intuitively, (3) is a simple way to estimate the search cost for repairing the earliest temporal flaw introduced into $\mathcal{A}$ by adding $e(a)$ through T1.

The relaxed plan $\pi_{s(a)}$ for the search cost of adding $s(a)$ is derived similarly, with the exception that the relaxed plans for $s(a)$ and $e(a)$ have different initial states: for $s(a)$, the initial state $I_{s(a)}$ is derived by applying the actions in $\mathcal{A}$ up to level $l_{s(a)} - 1$; for $e(a)$, it is derived by applying to $I_{s(a)}$ the actions in $\pi_{s(a)}$ (ignoring negative effects), then $s(a)$, and finally the actions in $\mathcal{A}$ from level $l_{s(a)}$ to $l_{e(a)}$.

The search cost for removing a pair of snap action nodes (changes T1, T2, P3), or moving an action node from a level to another one (changes T3, P4) is computed similarly to the cost for adding them, with the difference that the search cost for removing

Problems	N with snap-actions			N with domain actions		
	μ	max	Time	μ	max	Time
`D-01`	3.94	7	0.02	3.45	4	0.01
`D-05`	275.2	2947	0.37	10.90	45	0.02
`D-10`	457.0	70,073	131.5	19.10	108	0.02
`D-15`	48,045	492,964	–	71.47	1518	0.15
`R-01`	39.14	1010	0.07	10.63	33	0.01
`R-05`	405.7	23,708	2.53	23.23	80	0.02
`R-10`	1136	34,181	28.9	76.97	420	0.02
`R-15`	3296	75,026	63.4	86.43	959	0.03

Table 1. Average and max sizes of the basic search neighborhood, and CPU times of LPG using TA-graphs with snap actions and with domain actions for some problems in the "SimpleTime" version of `Driverlog` (`D`) and `Rovers` (`R`). "–" means no solution within 1000 CPU secs.

Heuristic	10	20	30	40	50
E_{NoT}	0.93	10.8	77.8	–	–
E_T	0.02	0.16	0.91	4.62	17.1

Table 2. CPU time of LPG using two heuristic functions for 5 problems in domain `Match` (10, 20, 30, 40, 50 matches). "–" indicates no solution found with 1000 CPU secs.

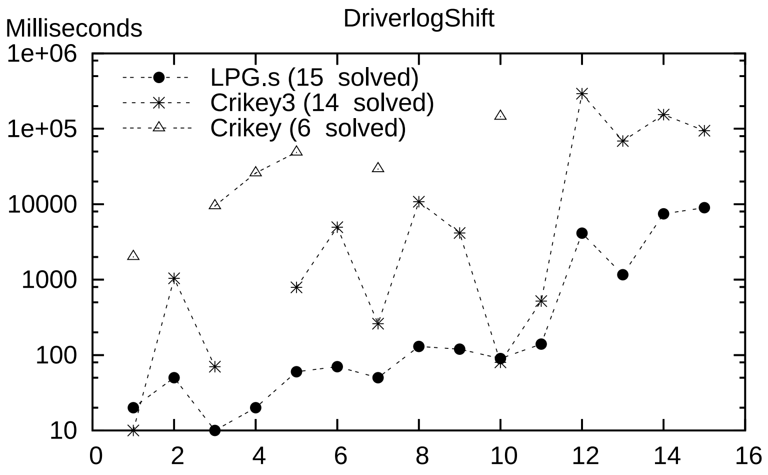
an action node x is estimated by constructing a relaxed plan for the precondition nodes supported by x that would become *un*supported by removing x.

4 Experimental Results

The techniques presented in the previous sections have been implemented in a new version of the LPG planner, which we tested using four domains with problems requiring concurrency: the known domains `Match`, `Matchlift` and `DriverlogShift` [1] and a new variant of `Rovers` Simpletime (`RoversShift`), obtained by adding some ground actions from two new action schemas, which require concurrency in every valid plan. All test domains and problems are available from
`http://lpg.ing.unibs.it/Concurrency/`.

The results for LPG were obtained using the restricted search neighborhood for propositional flaws, and correspond to median values over five runs for each problem considered. All tests were conducted on an Intel Xeon(tm) 2.6 GHz with 3 Gbytes of RAM.

Table 2 gives some results comparing the performance of LPG with the heuristic introduced in the previous section (E_T), which estimates the cost for repairing the temporal flaws generated into the current TA-graph when adding snap action nodes, and with a simpler heuristic that ignores these temporal flaws (E_{NoT}). Although E_T can be computationally much more expensive than E_{NoT}, these results indicate that LPG with E_T performs much better (it is about two orders of magnitude faster). In terms of plan quality, we observed similar performances with E_T and E_{NoT}.



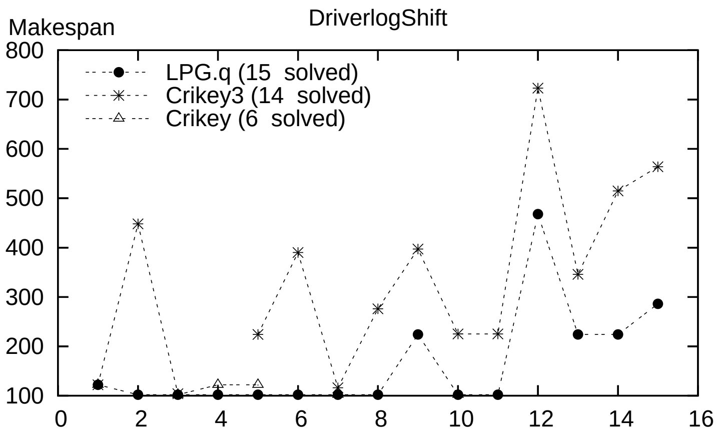
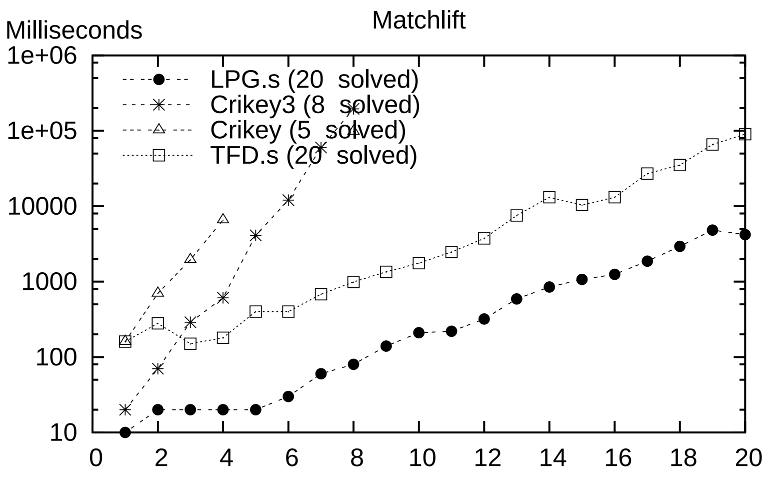


Fig. 2. CPU times and makespan of LPG.s, LPG.q, Crikey, Crikey3, TFD.s and TFD.q for the `DriverlogShift` domain. On the x-axis, we have the problem names simplifies by numbers. On the y-axis, we have CPU time (upper plots) or makespan for the computed plans (bottom plots).

Figures 2 –4 compare the performance of LPG and three recent prominent temporal planners supporting required concurrency: Crikey [1], Crikey3 [2] and Temporal Fast Downward (TFD) [5]. LPG and TFD were tested in terms of CPU time used to compute a solution, denoted by LPG.s and TFD.s, and in terms of quality of the best plan computed using at most 1000 CPU seconds, denoted by LPG.q and TFD.q.

From Figures 2 –4 we can observe that: LPG.s solves many more problems and is generally faster than Crikey and Crikey3; for `Matchlift`, LPG.s is about one order of magnitude faster than TFD.s (TFD does not run with `DriverlogShift` and `RoversShift`); the plans computed by LPG.q are generally better than those computed by all the other compared planners, and often they are much better. For `Match`,



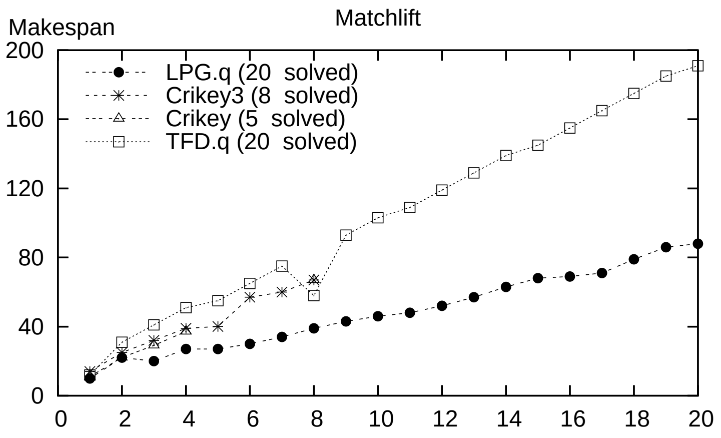
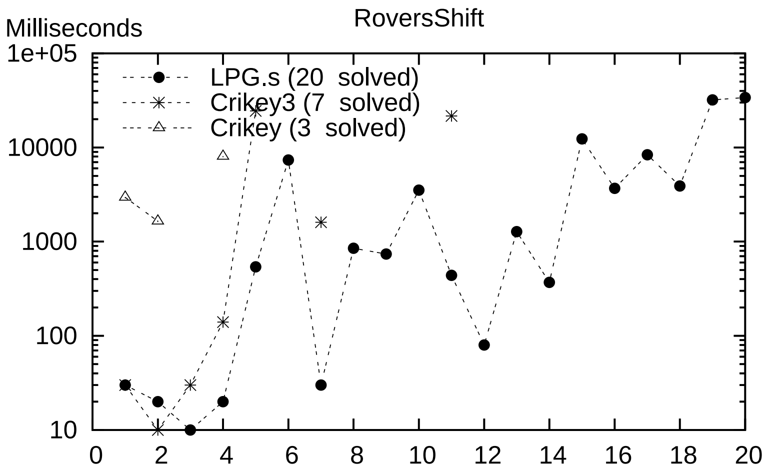


Fig. 3. CPU times and makespan of LPG.s, LPG.q, Crikey, Crikey3, TFD.s and TFD.q for the `Matchlift` domain. On the x-axis, we have the problem names simplifies by numbers. On the y-axis, we have CPU time (upper plots) or makespan for the computed plans (bottom plots).

Score	LPG.s	LPG.q	Crikey	Crikey3	TFD.s	TFD.q
Speed	70.8	42.3	9.4	0.5	3.2	3.0
Quality	53.9	72.2	30.0	32.7	22.7	24.2

Table 3. IPC-6 performance scores of the compared planners for `DriverlogShift`, `Match`, `Matchlift` and `Rovershift`. Higher scores correspond to better performance.

we observed similar performance gaps (the plots for this domain are omitted for lack of space).



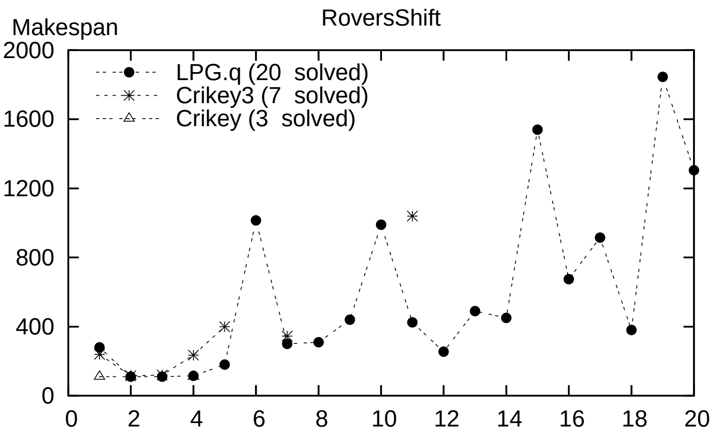


Fig. 4. CPU times and makespan of LPG.s, LPG.q, Crikey, Crikey3, TFD.s and TFD.q for the `Rovershift` domain. On the x-axis, we have the problem names simplifies by numbers. On the y-axis, we have CPU time (upper plots) or makespan for the computed plans (bottom plots).

Table 3 gives results about the performance of the compared planners using the IPC-6 performance score functions. This analysis confirms that LPG.s/q performs better than the other compared planners.

Finally, we have compared the new version of LPG with Crikey and TFD also using the Simpletime versions of the known domains `Driverlog` and `Rovers` which do *not* require concurrency. These results indicate that LPG is generally much faster and generates better quality plans w.r.t. the compared planners.

5 Conclusions and Future Work

Handling required concurrency in temporal planning is practically and theoretically important. We have proposed an extension of a prominent approach to satisficing planning for handling required concurrently in PDDL2.1 problems, and experimentally demonstrated its effectiveness. Current and future work includes the design of more powerful heuristic techniques for dealing with temporal flaws, new techniques for dynamically splitting the domain actions "when needed" during search, and additional experiments.

References

1. A. Coles, M. Fox, K. Halsey, D. Long, and A. Smith. Managing concurrency in temporal planning using planner-scheduler interaction. *Artificial Intelligence*, 173(1):1 – 44, 2009.
2. A. Coles, M. Fox, D. Long, and A. Smith. Planning with problems requiring temporal coordination. In D. Fox and C. P. Gomes, editors, *AAAI*, pages 892–897. AAAI Press, 2008.
3. W. Cushing, S. Kambhampati, Mausam, and D. S. Weld. When is temporal planning really temporal? In M. M. Veloso, editor, *IJCAI*, pages 1852–1859, 2007.
4. R. Dechter, I. Meiri, and J. Pearl. Temporal constraint networks. *Artificial Intelligence*, 49:61–95, 1991.
5. P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In A. Gerevini, A. E. Howe, A. Cesta, and I. Refanidis, editors, *ICAPS*. AAAI, 2009.
6. A. Gerevini and M. Cristani. On finding solutions in temporal constraint networks. In *IJCAI*, pages 1460–1465, 1997.
7. A. Gerevini, A. Saetti, and I. Serina. On managing temporal information for handling durative actions in LPG. In *AI*IA 2003: Advances in Artificial Intelligence*, pages 91–104. Springer-Verlag, Berlin-Heidelberg, New York, 2003.
8. A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)*, 20:239–290, 2003.
9. D. Long and M. Fox. Exploiting a graphplan framework in temporal planning. In *Proceedings of the Thirteenth International Conference on Automated Planning and Scheduling (ICAPS-03)*, pages 52–61, Menlo Park, CA, USA, 2003. AAAI Press.