New Heuristics for Timeline-based Planning

Riccardo De Benedictis and Amedeo Cesta

CNR, Italian National Research Council, ISTC, Rome, Italy
{name.surname}@istc.cnr.it

Abstract. The timeline-based approach to planning represents an effective alternative to classical planning in complex domains where different types of reasoning are required in parallel. The ILOC domain-independent planning system takes inspiration from both Constraint Programming (CP) and Logic Programming (LP). By solving both planning and scheduling problems in a uniform schema, ILOC is particularly suitable for complex domains arising from real world dynamic scenarios. Despite the planner captures elements that are very relevant for applications, its theory is quite challenging from a computational point of view and its performance are rather weak compared with those of state-of-the-art classical planners, particularly on those domains where such planners, typically, excel. In previous works, a resolution algorithm for the ILOC system has been proposed and enhanced with some (static and dynamic) heuristics that help the solving process. In this paper we propose a first improvement of the data structures underlying the proposed heuristics, producing a more informed heuristic and studying its effectiveness as a solving strategy. We perform tests on different benchmark problems from classical planning domains like the Blocks World to more challenging temporally expressive problems like the Temporal Machine Shop and the Cooking Carbonara problems, showing how the ILOC planner compares with respect to other state-of-the-art planners.

1 Introduction

Most of the current timeline-based planners [23], like EUROPA [20], ASPEN [8], IxTeT [18] and APSI-TRF [17, 6], are defined as complex software environments suitable for generating planning applications, but quite heavy to foster research work on specific aspects worth being investigated. Such architectures are, typically, inherently quite inefficient and, therefore, rely on a careful engineering phase of the domain, possibly supported by the definition of domain-dependent heuristics. Exception made for some works (e.g., [3]), their search control part has always remained significantly under explored.

Mostly based on the notion of partial order planning [28], timeline-based planners have usually neglected advantages from classical planning triggered from the use of GRAPHPLAN and/or modern heuristic search [4, 5, 19]. Furthermore, timeline-based architectures mostly rely on a clear distinction between a module for temporal reasoning and other modules that perform other forms

of constraint reasoning, while there is not enough exploration of other forms of reasoning.

In order to cope with such pitfalls, in a recent work [14] we presented a new framework, called iLoC, able to solve both planning and scheduling problems in a uniform schema. In addition, we described its resolution algorithm and endowed it with some (static and dynamic) heuristics. The initial heuristic followed the general principle of simplifying the initial problem, solving such simplified problem, and then use the solution for guiding the search of the initial, more complex, problem. At this initial stage, we left only causal relations and removed all the other types of constraints from the problem, resulting in a heuristic which, despite allowed us to greatly improve the performance of the reasoner, ended up being too uninformed.

This paper reintroduces some of the "removed" constraints (in particular the disjunctions) in the heuristic, thus enriching the informativeness and enabling improved performances of the resolution algorithm. In particularly we targeted those domains in which performances were worse. To explain our technique, we first introduce the basic principles underlying the iLoC system, then describe the new heuristic, and show how the system reasons about timelines, then compare it with other planners on different domains.

2 iLoC: An Integrated Logic and Constraint Reasoner

The aim here is to describe to the reader a minimalistic core that should be both sufficiently expressive as well as easily extensible so as to adapt as much as possible to the most variety of user requirements. Specifically, the basic core of the iLoC architecture provides an object oriented virtual environment for the definition of objects and constraints among them. Similarly to most object oriented environments, every object in the iLoC environment is an instance of a specific *type*. iLoC distinguishes among *primitive types* (e.g., bools, ints, reals, strings, etc.) and user defined *complex types* (e.g., robots, trucks, locations, etc.) endowed with their *member variables* (variables associated to a specific object of either primitive or complex type), *constructors* (a special type of subroutine called to create an instance of the complex type) and *methods* (subroutines associated with an object of a complex type). Defining a navigation problem, for example, might require the definition of a *Location* complex type having two numeric member variables x and y representing the coordinates of each *Location* instance. In the following, we will address objects and their member variables using a Java style *dot* notation (e.g., given a *Location* instance l, its x-coordinate will be expressed as $l.x$).

Once objects are defined, iLoC allows the definition of constraints among them. For example, in case a robot r should always be more East of a location l, the iLoC user could assert a constraint such $[\![l.x < r.x]\!]$. iLoC considers constraints as logic propositions and, as such, it allows the possibility for negating them (e.g., $\neg[\![l.x \leq 5]\!]$), for expressing conjunctions (e.g., $[\![l.x \leq 10]\!] \wedge [\![l.x \geq 5]\!]$), disjunctions (e.g., $[\![l.x \leq 5]\!] \vee [\![l.x \geq 10]\!]$) and logic implications (e.g., $[\![l.x \geq$

$10]\!] \rightarrow [\![l.y \geq 10]\!]$). In order for a solution to be valid, such constraints must always be consistent among themselves therefore, whenever an inconsistency is detected (e.g., $[\![l.x \leq 10]\!] \wedge [\![l.x \geq 15]\!]$), the system will return a failure.

In addition, it is possible to impose constraints on existentially quantified variables (e.g., $\exists l \in \mathit{Locations} : l.x \geq 10$) as well as universally quantified variables (e.g., $\forall l \in \mathit{Locations} : l.x \leq 100$). By combining logical quantifier and object oriented features, iLoC allows to manage, in one shot, all the instances of a given complex type.

A rather straightforward method for managing this kind of problems is to translate them into a Satisfiability Modulo Theories (SMT) problem (see, for example, [26]). There are several available SMT solvers having different performances, capabilities as well as licenses. Since iLoC has been written in Java the only available choices are, to the best of our knowledge, the SMTInterpol [9], the MathSAT 5 [10] and the Z3 [15] solvers[1].

Although this basic core allows the definition of quite complex problems (without providing any demonstration, we can state that NP-Complete problems are covered), some of the problems we are interested in are in PSPACE and thus excluded from the possibility of being modelled with this formalism. In order to overcome these limitations, we need something more powerful. Something that, roughly speaking, is able to "decide" the number of involved variables, together with their value. For this purpose, we have chosen to extend the above formalism by allowing many-sorted first-order Horn clauses[2], i.e., clauses with at most one positive literal, called the *head* of the clause, and any number of negative literals, forming the *body* of the clause. For example, we could use a predicate such as $\mathit{FirstQuadrant}$, with a $\mathit{Location}\ l$ argument, within the clause $\mathit{FirstQuadrant}\,(\mathit{Location}\ l) \Leftarrow [\![l.x \geq 0]\!] \wedge [\![l.y \geq 0]\!]$, for describing locations in the first quadrant of a Cartesian coordinate system. Furthermore, we do not allow constraints in the head of a clause but we slightly relax the "positive" literals in the body by allowing constraints to appear in any logical combination (i.e., we could rewrite the above example as $\mathit{FirstQuadrant}\,(\mathit{Location}\ l) \Leftarrow \neg[\![l.x < 0]\!] \wedge \neg[\![l.y < 0]\!]$).

A consequence of what we have seen is that iLoC planning problems can be described by a collection of clauses. There are two types of clauses: *rules* and *requirements*. A rule is of the form $\mathit{Head} \Leftarrow \mathit{Body}$. While the head of rules is limited to predicates, a rule's body consists of a set of calls to predicates, which are called the rule's *sub-goals*, and a set of constraints, the latter, in any logical combination. We consider rules having the same head as disjunctive. Clauses with an empty head are called requirements and can be calls to predicates (either *facts* or *goals*), or constraints, the latter, in any logical combination. Example of requirements are $\mathit{goal} : \mathit{FirstQuadrant}\,(\mathit{Location}\ l)$, $[\![l.x \geq 5]\!]$ and $[\![l.y \geq 5]\!]$,

[1] While SMTInterpol provides a pure Java implementation, MathSAT and Z3 provide Java wrappers to their native API. We have not found other SMT solvers that provide, directly or indirectly, a Java API.

[2] This means, in general, sacrificing decidability.

through which we are asking the planner to find a location l, among those which are in the first quadrant, having both coordinates greater than or equal to 5.

It is worth highlighting how the object oriented architecture binds with the discussion above. Intuitively, each variable that appears as the argument of a predicate inside rules is considered as universally quantified. Conversely, each variable that appears as the argument of a predicate inside a requirement is considered as existentially quantified. The object oriented architecture, combined with the many-sorted logic, allows to consider only the instances of a specific complex type, rather than all the defined objects, as the allowed values for the object variables.

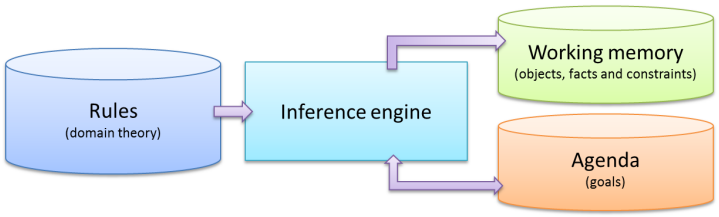


Fig. 1. A high-level view of the ILOC reasoning engine.

From an operational point of view, ILOC uses an adaptation of the *resolution principle* [25] for first-order logic, extended for managing constraints in the more general scheme usually known as *constraint logic programming* (CLP) [1]. Starting from the initial set of objects, facts and constraints, as described by the initial requirements, the reasoner maintains an agenda of the current (sub)goals. Incrementally, the system chooses (sub)goals from the agenda and, by exploiting rules, adds facts and constraints into the working memory. Figure 1 shows a general description of the ILOC reasoning engine.

For each goal $P\left(t_1^g, \ldots, t_i^g\right)$, in general, a branch in the search space is created. Resolution, at first, will try to *unify* goals with existing facts, if any, creating a single branch for all the possible unifications. Specifically, given the existing facts $P\left(t_1^1, \ldots, t_i^1\right), \ldots, P\left(t_1^j, \ldots, t_i^j\right)$, having the same predicate of the goal, the formula $[\![t_1^g = t_1^1 \wedge \ldots \wedge t_i^g = t_i^1]\!] \vee \ldots \vee [\![t_1^g = t_1^j \wedge \ldots \wedge t_i^g = t_i^j]\!]$ is added to the current solution. Intuitively, the purpose of unification is to avoid considering goals whose (any) rule has already been applied. In addition, a branch is also created for each of the rules whose head unifies with the chosen goal and, whenever such a branch is chosen by the resolution algorithm, the body of the corresponding rule is added to the current solution possibly generating further goals to be managed. Summarizing, the basic operations for refining a partial solution π toward a final solution are the following:

1. find the (sub)goals of π (i.e., the agenda).
2. select one such (sub)goals.

3. find ways to resolve it.
4. choose a resolver for the (sub)goals.
5. refine π according to that resolver.

The process follows an A* search strategy that aims at minimizing the number of goals in the agenda, proceeding until there are no more goals into the agenda and while all the constraints in the working memory are consistent. Whenever the constraints become inconsistent the system performs a backtracking step.

3 The MinReach Heuristic

Since all the goals must be solved sooner or later, there is almost no difference among *which* goal is solved first. Selecting the "right" goal, however, impacts heavily with the efficiency of the resolution algorithm. In order to overcome this obstacle we can take advantage of some heuristics. In our previous work [14] we have presented a data structure, called *static causal graph*, and we showed how information could be extracted from it to guide the search process.

The static causal graph has a node for each of the predicates that appear in our rules and, for every rule, an edge from the head of the rule to each of the predicates that appear in the body of the same rule. The cost for solving a goal, as suggested by our heuristic, is equal to the number of reachable nodes from the node relative to the predicate associated to the goal. The rough idea behind this strategy is to evaluate goals by considering a kind of worst case scenario where none of the formulas unify. Another way of looking at it is to consider, for each predicate, a new problem having rules without any constraints and a sole goal of the same predicate. We called such a strategy AllReachable (AR) goal selection heuristic.

$A(z) \Rightarrow B(y) \wedge [\![z < y]\!]$
$B(x) \Rightarrow [\![x < 10]\!]$
$C() \Rightarrow D()$
$E() \Rightarrow F() \wedge G()$
$G() \Rightarrow F() \wedge H()$
$G() \Rightarrow I()$

$A(w) \wedge [\![w = 5]\!]$
$B(v) \wedge [\![v = 7]\!]$

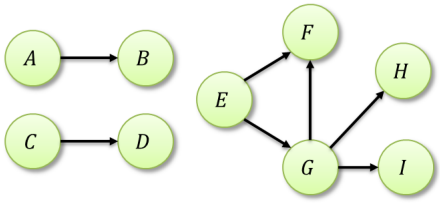


Fig. 2. A set of rules and requirements, along with the associated static causal graph.

Figure 2 shows a set of rules and the static causal graph resulting from them. As an example, the cost for solving an $A(w)$ goal, according to AR, is 1 (since

the sole node B is reachable from node A) while the cost for solving an $E\,(t)$ goal is 4 (since all the nodes F, G, H and I are reachable from node E). Such an heuristic is completely agnostic of disjunctions, putting at the same level all the predicates that appear in the body, whether they were in a disjunction or not, resulting in a too uninformed heuristic and, consequently, in bad performance of the search strategy. Indeed, solving a $G\,()$ goal would be evaluated as having cost 3, regardless of the two (disjunctive) rules having $G\,()$ as head.

A slight improvement to our heuristic is constituted by the addition of disjunctions into the static causal graph by means of two special nodes representing conjunctions (AND nodes) and disjunctions (OR nodes). Figure 3 shows the improved static causal graph generated from the example in Figure 2. The cost for solving a goal is now evaluated as the *minimum* number of reachable nodes starting from the node associated to the goal predicate. The general idea here is the following: whenever the resolution algorithm finds a disjunction, the application of the rule that would lead to the minimum number of formulas should be chosen. We call such a strategy MinReach (MR). As an example, the cost for solving a $G\,()$ goal is now reduced from 3 to 1 since all the nodes F, H and I are reachable from node E, yet introducing a sole formula $I\,()$ (second rule associated to predicate G) is probably preferable than introducing both formulas $F\,()$ and $H\,()$ (first rule associated to predicate G), and far more preferable than introducing all the three formulas $F\,()$, $H\,()$ and $I\,()$ as expected by heuristic AR.

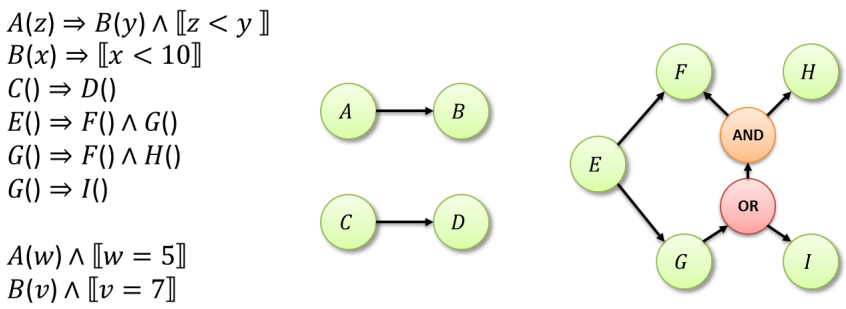


Fig. 3. An AND/OR static causal graph.

One might argue that by introducing disjunctions into the static causal graph we increase the complexity of the evaluation from polynomial to exponential. However, just as the AR heuristic, this graph and, consequently, the costs for each of their nodes, solely depend from the rules, therefore, our heuristic is independent from the requirements and thus can be built once and for ever at the beginning of the solving process, allowing constant-time cost retrieval. Nevertheless the problem can easily be encoded into a MIN-ONE SAT problem (i.e., given a propositional formula, if it is satisfiable, find the variable assignment

that contains the minimal number of positive literals) and let a SAT-solver (e.g., Sat4j [21]) solve it for us. The encoding is trivial:

- a boolean variable is associated to each predicate and to each AND node;
- for each arc $\langle s, t \rangle$, going from source node with boolean variable s to target node with boolean variable t, a clause $(\neg s, t)$ is added;
- for each arc $\langle s, OR \rangle$, going from source node with boolean variable s to an OR target node, we consider the variables $b_1, \ldots, b_n$ associated to all the n nodes directly reachable from the OR node and a clause $(\neg s, b_0, \ldots, b_n)$ is added.

Each predicate can now be evaluated as follows: we assume a unit clause containing the variable associated to the predicate we want to evaluate, solve the resulting MIN-ONE SAT problem, count the number of positive literals associated to predicates and subtract 1, since we don't count the starting node. As an example, the resulting MIN-ONE SAT problem associated to predicate G of Figure 3 is the following (we use lowercase names for the associated boolean variables):

$$(g)\,(\neg a, b)\,(\neg c, d)\,(\neg e, f)\,(\neg e, g)$$
$$(\neg g, and, i)\,(\neg and, f)\,(\neg and, h)$$

resulting in the sole g and i positive literals and, consequently, in an estimated cost of 1.

Similar to what we did in [14] for the AR heuristic, we exploit the MR heuristic both for goal selection and for node selection. Also, we refine the MR heuristic with the less merges dynamic heuristic (see that paper for further details).

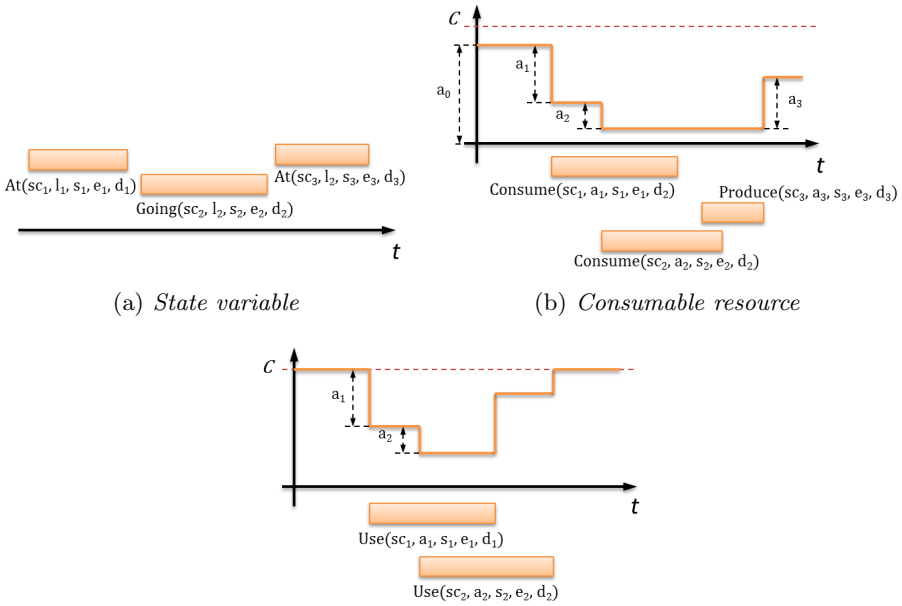
4 Timeline-based Planning and iLoC

The search space of a timeline-based planner has typically *partially specified plans* as nodes and *plan refinement operations* as arcs. Plan refinement operations are intended to further complete a partial solution, i.e., to achieve an open goal or to remove some possible inconsistency. Intuitively, these refinement operations avoid adding to the partial plan any constraint that is not strictly needed for addressing the refinement purpose (this is called the *least commitment principle*). The solving procedure starts from an initial node corresponding to an empty solution and the search aims at a final node containing a solution that correctly achieves the required goals.

A possible approach to the resolution of timeline-based planning problems is to provide the predicates described in the previous sections with numerical arguments in order to represent their starting times, their ending times and their durations. Also, it will be required to define some specific complex types, whose instances will be called *timelines*, in order to add further "implicit" constraints

among the formulas defined "over" their instances. This will also result in a slight adaptation of the resolution procedure in order to check the consistency for every object in the current partial solution so as to make explicit the just mentioned implicit constraints.

What does it mean to define a formula "over" a timeline? We simply add a parameter having the same type as the timeline to the predicates and call such a parameter *scope*. It is worth noting that most timeline-based planners like EUROPA, or APSI-TRF, indeed, consider timelines as a sort of "containers" for formulas. In our approach, since the core reasoning element are the atomic formulas, and consistently with a classical logical approach, we choose to incorporate the timelines "inside" the formulas. In other words, the type of our scope variables will be a "distinguisher" for triggering further reasoning. Furthermore the resulting scope variables are, to all effects, *variables* and, therefore, could be subject to constraints.



(a) *State variable* (b) *Consumable resource*

(c) *Reusable resource*

Fig. 4. Different kinds of timelines with formulas and resource profiles.

In the following we describe the minimal set of the complex types commonly used in timeline-based planning.

State variables. They are used to describe the "state" of a dynamical system as, for example, the position of a specific object at a given time or a simple

manufacturing tool that might be operating or not. The semantics of a state variable (and thus the implicit constraints we need to make explicit) is simply that, for each time instant $t \in \mathbb{T}$, the timeline can assume only one value. Figure 4(a) represents an example of state variable with three atomic formulas (parameter types are omitted for sake of space). The example shows a robot r_0, a state variable of type *Robot*, which might be *At* a given location or might be *Going* to another location. We thus have the two predicates $At\,(sc, l, s, e, d)$ and $Going\,(sc, l, s, e, d)$ each having a parameter sc of type *Robot* describing the scope of the formulas and parameters l, s, e and d respectively for the location, the start, the end and the duration. The planner will take care of adding the proper constraints for avoiding the temporal overlapping of the incompatible states (i.e., all the formulas which have the same scope and do not unify) or for "moving" the states on other instances of type *Robot* (i.e., choosing another value, for example r_1, for the scope of the formula).

Resources. They are entities characterized by a *resource level* $\mathcal{L} : \mathbb{T} \to \mathbb{R}$, representing the amount of available resource at any given time, and by a *resource capacity* $\mathcal{C} \in \mathbb{R}$, representing the physical limit of the available resource. We can identify several types of resources depending on how the resource level can be increased or decreased in time. A *consumable resource* is a resource whose level is increased or decreased by some activities in the system. An example of consumable resource is a reservoir which is produced when a plan activity "fills" it (i.e., a tank refueling task) as well as consumed if a plan activity "empties" it (i.e., driving a car uses gas). Consumable resources have two predefined rules, each having an empty body, and a predicate $Produce\,(sc, id, a, s, e, d)$ ($Consume\,(sc, id, a, s, e, d)$) as head, so as to represent a resource production (consumption) on the consumable resource sc of amount a from time s to time e with duration d (we use an id parameter to prevent unification among these formulas). In addition, the consumable resource complex type has four member variables representing the initial and the final amount of the resource, the min and the max value for the resource level. Quite popular in the scheduling literature, *reusable resources* are similar to consumable resources where productions and consumptions go in tandem at the start and at the end of the activities. Reusable resources can be used for modelling, for example, the number of programmers employed on a given project for a given time interval. Reusable resources have one predefined rule having an empty body and a predicate $Use\,(sc, id, a, s, e, d)$ as head so as to represent an instantaneous production of resource sc of amount a at time s and an instantaneous consumption of the same resource sc of the same amount a at time e. In addition, the reusable resource type has a member variable for representing the capacity of the resource. Figures 4(b) and 4(c) represent, respectively, an example of consumable resource and an example of reusable resource with some associated formulas.

By introducing these complex types, we require the reasoner to add further constraints so as to avoid object inconsistencies (e.g., different states overlapping for some state variable; resource levels $\mathcal{L}$ exceeding resource capacity $\mathcal{C}$ or going lower than min, etc.). We chose to refine our resolution process by introducing a

step for detecting such inconsistencies and for adding required constraints which would remove them. The resulting basic operations for refining a partial solution π toward a final solution are thus the following:

1. find the (sub)goals of π.
2. select one such (sub)goals.
3. find ways to resolve it.
4. choose a resolver for the (sub)goals.
5. refine π according to that resolver.
6. check for any object inconsistency and remove it.

Similar to [7], we use a lazy approach for detecting inconsistencies. Namely, we let the underlying SMT solver to extract a solution given the current constraints and, in case some inconsistency is detected we add further constraints so as to remove the inconsistency. A simple example should clarify the idea. Let us suppose in a given partial solution there are two formulas describing a state variable sv_k having two overlapping states s_i and s_j, we solve the inconsistency by adding the constraint $[\![s_i.start \geq s_j.end]\!] \vee [\![s_j.start \geq s_i.end]\!] \vee [\![s_i.scope \neq s_j.scope]\!]$ preventing further overlapping of these states on the same state variable. The core idea for solving resource inconsistencies follows a very similar schema.

5 Preliminary Results

To assess the value of our heuristic, we have endowed ILOC with the proposed MINREACH (MR) heuristic and tried to compare the resulting system with different planners on different benchmarking problems. Specifically, we have selected four planners that are interesting for their features and compared them with ILOC: ILOC(AR) is the previous version of ILOC exploiting the simpler ALL-REACHABLE (AR) heuristic, VHPOP [27] shares with our planner the partial ordering approach, OPTIC [2] and COLIN (see [11]) are both based on a classic FF-style forward chaining search [19]. All the test have been executed with default configurations for every planner.

We start the comparison by solving the Blocks World domain, a workhorse for the planning community. As known, in this domain a set of cubes (blocks) are initially placed on a table. The goal is to build one or more vertical stacks of blocks. The catch is that only one block may be moved at a time: it may either be placed on the table or placed atop another block. Because of this, any blocks that are, at a given time, under another block cannot be moved. We used the 4-operator version of the classic Blocks World domain, as found on the IPC-2011 website, as a starting point. Specifically, for each block, we defined a state variable for representing what is on top of the block (i.e., either another block or the value "Clear") and a state variable for representing if the block is on the table or not. An additional state variable has been defined for modeling the robotic arm modeling values that represent either the arm holding a block or the value "Empty". Finally, we defined an "Agent" complex type for modeling

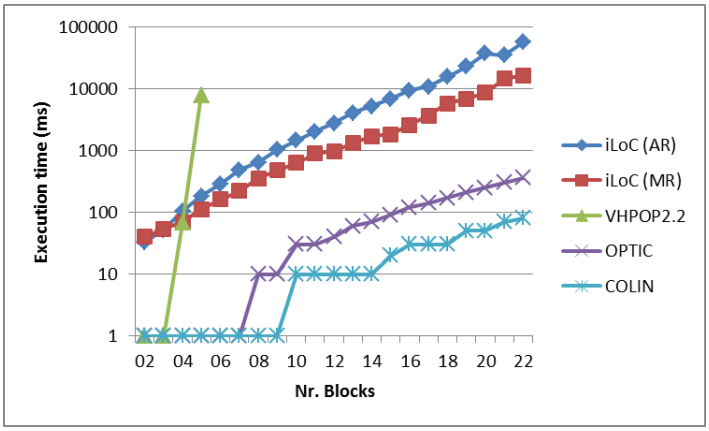


Fig. 5. Blocks world.

the agents' actions. Rules have been defined so as to have an atomic formula for each effect of the PDDL actions as head and an atomic formula for the actions as body, aside from rules having an atomic formula for each PDDL action as head and an atomic formula for their preconditions and effects as body. Temporal constraints have been conveniently added for guaranteeing that preconditions precede actions and effects follow actions.

As shown in Figure 5, despite the introduction of our heuristic planners endowed with "classical heuristics" still perform significantly better than our approach, nevertheless we were able to boost the system performance appreciably, allowing us to find solutions up to, approximately, one third of the time it was required before.

We have also checked our system with two other problems, namely the Temporal Machine Shop [13] and the Cooking Carbonara domain [22]. Both these problems are *temporally expressive* (see [12]) since they require concurrency for being solved.

The first problem is the only temporally expressive problem of the International Planning Competition (IPC) and, within the same competition, it is solved by the sole ITSAT planner (see [24]). The problem models a baking ceramic domain in which ceramics can be baked while a kiln is firing. Different ceramic types require a different baking time. While a kiln can fire for at most 20 minutes at a time (and then it must be made ready again), baking a ceramic takes, in general, less time, therefore we can save costs by baking them altogether. Additionally, similar to [24], we have slightly complicated the domain by considering the possibility for ceramics to be assembled, so as to produce different structures which should be baked again to obtain the final product. Specifically, for each kiln we defined a state variable for distinguishing either the kiln is "Ready" or "on Fire". In addition, each kiln has associated a reusable resource for representing its capacity. For each ceramic piece we defined a state

variable for representing either the piece is "Baking" (with an additional parameter for representing the kiln in which is baking), or the piece is "Baked", or the piece is "Treating", or the piece is "Treated". Similarly, for each ceramic structure we defined a state variable for representing either the structure is "Assembling", or the structure is "Assembled", or the structure is "Baking" (with an additional parameter for representing the kiln in which it is baking), or the structure is "Baked". Rules force these values to appear in time, in each state variable, in the intuitive manner (i.e., in the order in which these values have just been introduced). The interesting aspect, however, is that ceramic structures can bake concurrently with ceramic pieces both *while* (hence the temporal expressiveness) the kiln is firing.

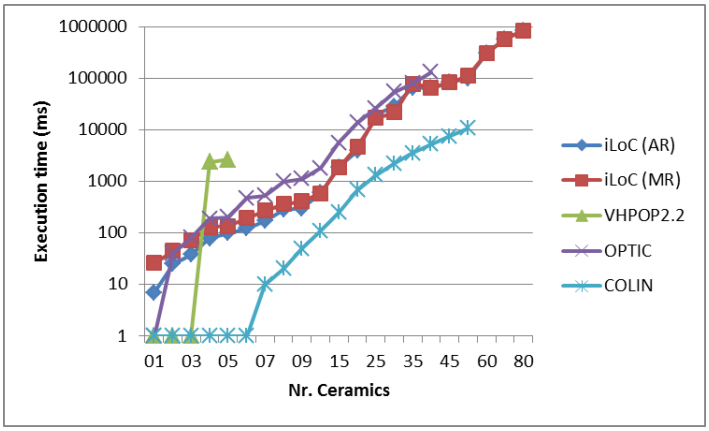


Fig. 6. Temporal machine shop.

The Cooking Carbonara domain represents another temporally expressive problem in which the aim is the preparation of a meal, as well as its consumption by respecting constraints of warmth. Problems cooking-carbonara-n allow to plan the preparation of n dishes of pasta. The concurrency of actions is required to obtain the goal because it is necessary that the electrical plates work in a way that water and oil are hot enough to cook pasta and bacon cubes. It is also necessary to perform this baking in parallel to serve a dish that is still hot during its consumption. Specifically, for each plate we defined a reusable resource for representing its (unary) capacity. For each pot we defined a state variable for distinguishing either the pot is "Boiling" (with an additional parameter for representing the plate on which is boiling) or the pot is "Hot". For each pan we defined a state variable for distinguishing either the pan is "Boiling" (with an additional parameter for representing the plate on which is boiling) or the pan is "Hot". Each portion of spaghetti has associated a state variable for distinguishing either the portion is "Cooking" (with an additional parameter for representing the pot in which is cooking) or the portion has been "Cooked".

For each bacon portion we defined a state variable for distinguishing either the bacon is "Cooking" (with an additional parameter for representing the pan in which is cooking) or the bacon has been "Cooked". Each egg has associated a state variable for distinguishing either the egg is "Being beaten" or the egg has been "Beaten". Finally, for each carbonara portion we defined a state variable for distinguishing either the portion is "Cooking" (with an additional parameter for representing the plate on which should be cooked), or the portion has been "Cooked", or someone is "Eating" the portion or the portion has been "Eaten". Again, rules force values to appear in time, in each state variable, in the intuitive manner (i.e., in the order in which these values have just been introduced). Furthermore, carbonara portions should be cooking after spaghetti, bacon and eggs have been correctly prepared, hence requiring spaghetti to be "Cooking" *while* the water in pots is "Hot" as well as bacon to be "Cooking" *while* the oil in pans is "Hot". Finally, cooking carbonara portions, boiling water in pots and oil in pans should be performed *while* plates are available.

Experimental results on these domains (figures 6 and 7) show that the heuristic does neither guarantee a substantial improvement nor the overhead produces a significant worsening (performance remains almost unchanged). Specifically, in the first problem iLoC performs almost inline with those of state-of-the-art planners. Even though COLIN performs better than iLoC, it is not able to solve problems with more than 50 ceramics since it runs out of memory (we used the default configuration for the planner). In the Cooking Carbonara domain, however, by removing the maximum duration for plate firing, the problem is reduced to a basic scheduling problem hence allowing iLoC to outperform state-of-the-art solvers. This behavior can be explained by observing that these problems are biased toward a temporal kind of reasoning rather than a causal kind of, therefore they find minimum benefit from the improvements introduced in the new heuristic which is mostly oriented toward causal aspects.

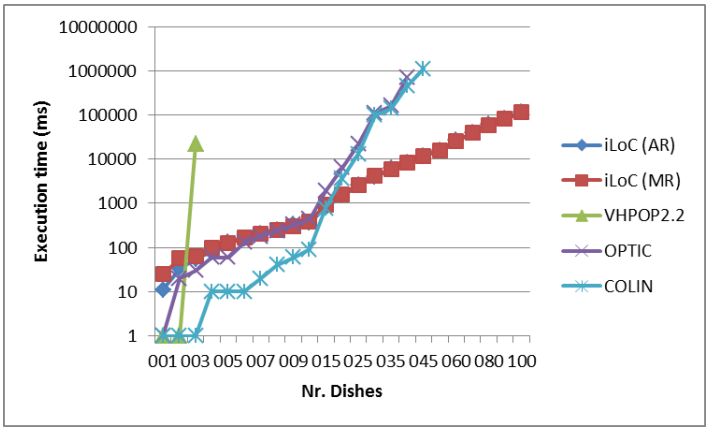


Fig. 7. Cooking carbonara.

A separate discussion it is worth doing concerns the expressiveness of iLoC. All the competing planners use the PDDL2.1 language (see [16]) for modeling their planning problems and, in general, it is quite cumbersome to impose temporal constraints among plain PDDL actions. In the Cooking Carbonara domain, for example, it is important that the cooking happens before the eating but eating should not start too late to avoid that food becomes cold. In [22] a PDDL extension is proposed to overcome this issue and to model properly the domain, however, none of the available planners supports this extension and thus they have been evaluated in a simplified domain in which the warmth constraint decays and dishes can be served anytime after they have been cooked. It is worth noting how this constraint is naturally captured in the iLoC modelling language by creating a rule having as head an action and as body a second action in conjunction with a constraint among the temporal parameters of the two actions.

6 Conclusions

This paper has introduced a new general heuristic for the iLoC planner that improves the planner performance with respect to those of a previous work. The initial heuristic was the result of a too strong simplification and therefore was probably too uninformed. With the present work we started in the direction of reintroducing parts previously neglected. In particular by introducing disjunctions we produced a heuristic that allowed us to improve the performance of the resolution algorithm, especially on those domains in which the performance were weaker.

The iLoC planner already had comparable (or even better) performance of other planners in those domains in which temporal reasoning constitutes the main reasoning requirements (i.e., temporally expressive domains). For this reason we focused on those domains in which the temporal aspects were negligible compared to the causal ones. The current results are still not competitive with respect to those of other planners, nevertheless we succeeded in improving performance on the class of problems not very suited for the timeline-based approach.

We are pursuing a domain-independent planner able to solve efficiently a wider spectrum of planning problems, therefore, work is still needed at heuristic level to reduce the differences with respect to classical approaches.

Acknowledgments. Authors work is partially funded by the Ambient Assisted Living Joint Program under the SpONSOR project (AAL-2013-6-118).

References

1. Apt, K.R., Wallace, M.G.: Constraint Logic Programming Using ECL^{i}PS^e. Cambridge University Press, New York, NY, USA (2007)

2. Benton, J., Coles, A., Coles, A.: Temporal Planning with Preferences and Time-Dependent Continuous Costs. In: Twenty-Second International Conference on Automated Planning and Scheduling (2012)
3. Bernardini, S., Smith, D.: Developing Domain-Independent Search Control for EUROPA2. In: Proceedings of the Workshop on Heuristics for Domain-independent Planning at ICAPS-07 (2007)
4. Blum, A., Furst, M.L.: Fast Planning Through Planning Graph Analysis. In: IJCAI. pp. 1636–1642. Morgan Kaufmann (1995)
5. Bonet, B., Geffner, H.: Planning as Heuristic Search. Artificial Intelligence 129(12), 5–33 (2001)
6. Cesta, A., Cortellessa, G., Fratini, S., Oddi, A.: Developing an End-to-End Planning Application from a Timeline Representation Framework. In: IAAI-09. Proceedings of the 21^{st} Innovative Applications of Artificial Intelligence Conference, Pasadena, CA, USA (2009)
7. Cesta, A., Oddi, A., Smith, S.F.: A Constraint-based Method for Project Scheduling with Time Windows. Journal of Heuristics 8(1), 109–136 (2002)
8. Chien, S., Tran, D., Rabideau, G., Schaffer, S., Mandl, D., Frye, S.: Timeline-Based Space Operations Scheduling with External Constraints. In: ICAPS-10. Proc. of the 20^{th} Int. Conf. on Automated Planning and Scheduling (2010)
9. Christ, J., Hoenicke, J., Nutz, A.: SMTInterpol: An Interpolating SMT Solver. In: Model Checking Software - 19th International Workshop, SPIN 2012, Oxford, UK, July 23-24, 2012. Proceedings. pp. 248–254 (2012)
10. Cimatti, A., Griggio, A., Schaafsma, B., Sebastiani, R.: The MathSAT5 SMT Solver. In: Piterman, N., Smolka, S. (eds.) Proceedings of TACAS. LNCS, vol. 7795. Springer (2013)
11. Coles, A.J., Coles, A.I., Fox, M., Long, D.: COLIN: Planning with Continuous Linear Numeric Change. Journal of Artificial Intelligence Research 44, 1–96 (May 2012)
12. Cushing, W., Kambhampati, S., Mausam, Weld, D.S.: When is Temporal Planning *Really* Temporal? In: Proceedings of the 20th International Joint Conference on Artifical Intelligence. pp. 1852–1859. IJCAI'07, Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2007)
13. Cushing, W., Weld, D.S., Kambhampati, S., Mausam, Talamadupula, K.: Evaluating temporal planning domains. In: Boddy, M.S., Fox, M., Thibaux, S. (eds.) Proceedings of the Seventeenth International Conference on Automated Planning and Scheduling, ICAPS 2007, Providence, Rhode Island, USA, September 22-26, 2007. pp. 105–112. AAAI (2007)
14. De Benedictis, R., Cesta, A.: Integrating Logic and Constraint Reasoning in a Timeline-based Planner. In: AI*IA 2015 - XIVth International Conference of the Italian Association for Artificial Intelligence (2015)
15. De Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)
16. Fox, M., Long, D.: PDDL2.1: An Extension to PDDL for Expressing Temporal Planning Domains. Journal of Artificial Intelligence Research 20, 61–124 (2003)
17. Fratini, S., Pecora, F., Cesta, A.: Unifying Planning and Scheduling as Timelines in a Component-Based Perspective. Archives of Control Sciences 18(2), 231–271 (2008)

18. Ghallab, M., Laruelle, H.: Representation and Control in IxTeT, a Temporal Planner. In: AIPS-94. Proceedings of the 2nd Int. Conf. on AI Planning and Scheduling. pp. 61–67 (1994)
19. Hoffmann, J.: FF: The Fast-Forward Planning System. AI Magazine 22(3), 57–62 (2001)
20. Jonsson, A., Morris, P., Muscettola, N., Rajan, K., Smith, B.: Planning in Interplanetary Space: Theory and Practice. In: AIPS-00. Proceedings of the Fifth Int. Conf. on AI Planning and Scheduling (2000)
21. Le Berre, D., Parrain, A.: The Sat4j library, release 2.2. JSAT 7(2-3), 59–6 (2010)
22. Maris, F., Régnier, P.: TLP-GP: Un planificateur pour la rsolution de problmes temporellement expressifs. Revue d'Intelligence Artificielle 24(4), 445–464 (2010)
23. Muscettola, N.: HSTS: Integrating Planning and Scheduling. In: Zweben, M. and Fox, M.S. (ed.) Intelligent Scheduling. Morgan Kauffmann (1994)
24. Rankooh, M.F., Mahjoob, A., Ghassem-Sani, G.: Using Satisfiability for Non-optimal Temporal Planning. In: Logics in Artificial Intelligence - 13th European Conference, JELIA 2012, Toulouse, France, September 26-28, 2012. Proceedings. pp. 176–188 (2012)
25. Robinson, J.A.: A Machine-Oriented Logic Based on the Resolution Principle. Journal of the Association for Computing Machinery 12(1), 23–41 (1965)
26. Sebastiani, R.: Lazy Satisability Modulo Theories. JSAT 3, 141–224 (2007)
27. Simmons, R.G., Younes, H.L.S.: VHPOP: Versatile Heuristic Partial Order Planner. CoRR (2011)
28. Weld, D.S.: An Introduction to Least Commitment Planning. AI Magazine 15(4), 27–61 (1994)