Generalizing the Relaxed Planning Heuristic to Non-Linear Tasks

Stefan Edelkamp

Computer Science Department
Baroper Str. 301
University Dortmund
`stefan.edelkamp@cs.uni-dortmund.de`

Abstract. The *relaxed planning heuristic* is a prominent state-to-goal estimator function for domain-independent forward-chaining heuristic search and local search planning. It enriches the state-space traversal of almost all currently available suboptimal state-of-the-art planning systems.
While current domain description languages allow general arithmetic expressions in precondition and effect lists, the heuristic has been devised for propositional, restricted, and linear tasks only. On the other hand, generalizations of the heuristic to non-linear tasks are of apparent need for modelling complex planning problems and a true necessity to validate software. Subsequently, this work proposes a solid extension to the estimate that can deal with non-linear preconditions and effects. It is derived based on an approximated plan construction with respect to intervals for variable assignments. For plan extraction, weakest preconditions are computed according to the assignment rule in Hoare's calculus.

1 Introduction

More and more successful forward-chaining planners apply variants of the *relaxed planning heuristic*, originally proposed by Hoffmann [22] as an extension to the estimated that was applied in the first heuristic search planning system HSP [2]. As one indicator, in the list of participants in the bianual series of the *international planning competition*[1], we observe a drastic increase of planners that incorporate the relaxed planning heuristic. In 2000, the planning systems *FF*, *STAN*, and *MIPS* at least partially applied a relaxed plan analysis. In 2002, the planners *Metric-FF*, *LPG*, *VHPOP*, *SimPlan*, *SAPA*, and *MIPS* implemented refinements and extensions to the heuristic. In 2004, the planners *SGPlan*, *Marvin*, *FAP*, *Fast Diogonally Downward*, *Crikey*, *Roadmapper*, *YAHSP*, *LPG-TD*, *Macro-FF*, and *P-MEP* refer to the notion of *relaxed plans* [12].

With Level(s) 2 (and 3) of PDDL2.1 [15] an agreed standard for metric (and temporal planning) has been introduced to allow richer and more flexible domain models. So-called *fluents* encode numerical quantities to express real-valued attributes. They call for numerical preconditions and effects. As a matter of fact, PDDL2.1 does not per se restrict arithmetic expressions in the precondition and effect lists.

One successful extension of the relaxed planning heuristic to planning problem with numerical state variables has been implemented in the planner Metric-FF [20]. The

[1] ipc.icaps-conference.org

translation of the relaxation principle *ignoring the delete lists* to numeric state variables is involved and has been achieved by introducing negated variables in linear tasks[2]. As a feature, the heuristic can deal with repeated operator application in the relaxed plan. The algorithms presented in this paper shows that the relaxed planning heuristic is in fact more general. The changes we propose require a shift in the relaxation paradigm on how to represent and derive the approximation. Our extension contributes to the fact that, instead of propagating negated variables and introducing upper bounds, it is seemingly better to evaluate the heuristic based on upper *and* lower variable bounds.

The paper is structured as follows. First we introduce relaxed planning, and show where the current proposal for metric domains has its limitations. Next we generalize the relaxation scheme. The implementations base on a procedure that checks, if an operator can be applied with respect to a current vector of minimal and maximal numerical quantities, and on a procedure that adjusts the bounds accordingly. In both cases the combined vector of bounds is recursively evaluated in the expression for each operator. Backward extraction will have to determine weakest preconditions for variable assignments. We close with experiments, related work and concluding remarks.

2 Relaxed Planning

In propositional planning, we have a set of atomic propositions AP of which a subset is *true* in a given state. Planning operators modify the set of atomic propositions. As a prototype, consider STRIPS [14] operators $a = (\mathit{pre}(a), \mathit{add}(a), \mathit{del}(a))$. The application of a to a state S with $\mathit{pre}(a) \subseteq S$ yields the successor state $(S \setminus \mathit{del}(a)) \cup \mathit{add}(a)$.

2.1 Propositional Relaxed Planning

Since there are different ways to *relax* planning problems in order to approximate the state-to-goal distance, we briefly recall, what is meant with *relaxed* in this context. The *relaxed planning problem* for a STRIPS planning instance is constructed as follows [22]. The *relaxation* of an *operator* $a = (\mathit{pre}(a), \mathit{add}(a), del(a))$ is the tripel $(\mathit{pre}(a), \mathit{add}(a), \emptyset)$, so that the delete list is ignored. The *relaxation of a planning problem* is the one in which all operators are relaxed. It is not difficult to see that any solution that solves the original plan also solves the relaxed one, and that any goal can be established in the original task only if it can be achieved in the relaxed one [22].

Estimate h^+ is determined as the length of the shortest plan, that solves the relaxed problem. The heuristic is *consistent* by means that for all planning states u and v, with v being the successor of u, we have $h^+(u) \leq h^+(v) + 1$: if plan with cost $h^+(v)$ has been established, it can be extended to a plan for u by adding the operator that leads from u to v, so that $h^+(u) \leq h^+(v) + 1$. Heuristic h^+ simplifies the state space graph by modifying edges to make it acyclic. The state space size, however, does not necessarily shrink. Unfortunately, the relaxed problem is still computationally hard.

Bylander has shown that propositional STRIPS planning is PSPACE complete [5]. By a simple reduction to 3-SAT, he also proved that minimizing the *sequential plan*

[2] Note that the term *non-linear task* has been used for other aspect in planning with a different meaning, e.g. *Pednault* refers to conditional effects as being non-linear.

```
procedure Relax($\mathcal{C}, \mathcal{G}$)
    $P_0 \leftarrow \mathcal{C}$; $t \leftarrow 0$; $A \leftarrow \emptyset$
    while ($\mathcal{G} \not\subseteq P_t$)
        $P_{t+1} \leftarrow P_t \cup \bigcup_{pre(a) \subseteq P_t} add(a)$
        if ($P_{t+1} = P_t$) return $\infty$
        $t \leftarrow t + 1$
    for $i \leftarrow t$ downto 1
        $G_i \leftarrow \{g \in \mathcal{G} \mid level(g) = i\}$
    for $i \leftarrow t$ downto 1
        for $g \in G_i$
            if $\exists a.\ g \in add(a)$ and $level(a) = i - 1$
            $A \leftarrow A \cup \{a\}$
            for $p \in pre(a)$
                $G_{level(p)} = G_{level(p)} \cup \{p\}$
    return $|A|$
```

Fig. 1. Propositional relaxed planning heuristic.

length for propositional relaxed tasks is in fact NP-complete. The corresponding plan existance problem, however, turns out to be computationally tractable, and an optimal *parallel plan* that solves the relaxed problem can be found in polynomial time. This has lead to an *approximation* of h^+, which counts the number of actions of a greedily extracted parallel plan in the unrolled relaxed planning graph.

Figure 1 provides an implementation in pseudo code. The planning goal $\mathcal{G}$ is provided as a set of atoms, while $\mathcal{C}$ denotes the current state. The set A that is constructed reflects the greedily extracted plan. The set of goal atoms in Layer i of the relaxed planning graph is denoted by G_i. Note that in difference to planning graphs in *Graphplan*, in the propositional relaxed planning graphs, atoms and operators are unique. In the sequel of this paper, with the term *relaxed planning heuristic* we will refer to the *approximation of* h^+. This approximation is efficient to compute, so that it is applied in forward-chaining planners to evaluate each expanded planning state.

This advantage, however, comes at a high price. Optimal parallel and optimal sequential plans may have different sets of operators, so that the approximation of h^+ is no longer *consistent* nor *admissible*, and thus fails to serve as a lower bound for the optimal sequential path length. Admissablity, however, is required, when standard heuristic search algorithms like A* [25] are applied to find optimal plans.

Therefore, the relaxed planning heuristic is usually employed in local search or hill climbing planners. Alternative designs of *admissible estimates* are the *max-atom* heuristic [2], defined as the maximal depth of a goal in the planning graph, the *max-pair heuristic* [17], which takes interactions of pairs of atoms into account, and the *pattern database heuristics* [7], that introduces *don't care* symbols in the state vector to relax the planning problem.

The h^+ heuristic and its approximation have been studied yielding an empirical validated [18] and theoretical founded [19] topology of the many benchmark domains.

2.2 Metric Planning

In a metric planning problem *conditions* are constraints of the form $exp = exp' \otimes exp''$, where $\otimes$ is a comparison symbol, and exp' and exp'' are arithmetic expressions over the set of variables and constants. *Assignments* in the effect lists are expressions $v \oplus exp$ with a *variable head* v, an arithmetic term *exp*, and an assignment operator $\oplus$. In grounded representation of a metric planning problem for some $k \in I\!N$ we have the state space

$$\mathcal{S} \subseteq 2^{AP} \times I\!R^k,$$

where 2^{AP} is short for the power set of AP. Consequently a state $S \in \mathcal{S}$ is a pair (S_p, S_n) with propositional part $S_p \subseteq AP$ and numerical part $S_n \in I\!R^k$. For the ease of exposition we assume that all actions are in *normal form*, i.e. all propositional expressions satisfy STRIPS notation. All numerical condition and assignments refer to arithmetic expressions. Comparison operators $\otimes$ are selected from $\{\geq, \leq, >, <, =\}$ with common interpretation. Assignments have operators $\oplus$ in $\{\leftarrow, \uparrow, \downarrow, \nearrow, \searrow\}$, meaning variable *assignment*, *increase*, *decrease*, *scale-up*, or *scale-down*, respectively. This is not a limitation to general PDDL, since ADL constructs with object quantification, negated or disjunctive preconditions and conditional effects can be compiled away[16].

2.3 Numerical Relaxed Plans and Current Limitations

In brief terms, the *numerical extension to the relaxed planning heuristic* [20] generates and analyzes a layered graph, maintaining sets of propositional facts *and* arithmetic conditions in each layer. The planning graph *generation phase* applies relaxed operators until all goal conditions are established. In the backward *extraction phase*, goal propositions and conditions are selected and a greedy procedure selects either a proposition or a condition at a time. It searches for the corresponding operator of the forward phase, and marks its (numerical or propositional) preconditions in the *smallest possible layer* as still to be processed. The matching condition is deleted and the process repeats with the updated sets of propositions and conditions. The returned heuristic estimate is the number of operators in the relaxed plan. Multiple operator application is permitted by a special update option to the condition that has been met.

The heuristic is restricted to *linear tasks*, where expressions are of the form $a_0 v_0 + \ldots + a_k v_k$ for variables v_i and coefficients a_i. In case a coefficient a_i is negated, a surplus variable representing $-v_i$ is included into the state vector to approximate monotonicity in each variable. Many interesting planning tasks, however, include non-linear expressions. As a illustrative example consider the following domain, where a selection of numbers and arithmetic symbols is given, with the task to include the symbols into the sequence to compute a pre-specified target number. The domain has been invented by van der Krogt [28]. He observed that no current PDDL planning system can deal with this simple domain. An operator for introducing multiplication looks as follows.

```
(:action mul
 :parameters (?x ?y ?z - number)
 :precondition (and (active ?x) (active ?y) (non-active ?z))
 :effect (and (active ?z)
              (assign (value ?z) (* (value ?x) (value ?y)))))
```

Given an instance of three numbers $a = 1$, $b = 3$, and $c = 2$, the plan to generate value $e = 8$ is `(add a b d)` and `(mul c d e)`. Breadth-first search exploration produces a plan while expanding 14 planning states. Included in A*, our variant of the numerical relaxed planning heuristic generates the above plan in the optimal number of 2 state expansions. Other examples are referred to in the experiments.

Beside extended planning benchmarks, some of which are possibly better dealt with constraint satisfaction techniques, our main motivation to deal with non-linear expressions in domain descriptions are *software verification domains*, where general expressions in form of variable assignments are by far more frequent.

2.4 Application Area for Non-Linear Planning

Model checking [6] has evolved into one of the most successful software verification techniques. Examples range from mainstream applications such as *protocol validation* and *embedded systems verification* to exotic areas such as *business workflow analysis*, *scheduler synthesis* and *verification*. Automated software checking validates (mostly concurrent) code through an exploration of the space of system's states, consisting of propositional and numerical state variables [1]. Software model checking technology is also effective in automated test case generation.

There are two primary approaches to model checking. First, *symbolic model checking* [23] uses symbolic representations for the state sets based on binary decision diagrams [4]. Property validation in symbolic model checking amounts to symbolic fixpoint computation. *Explicit state model checking* uses a single-state representation for traversing the system's global state space graph. An explicit state model checker evaluates the validity of temporal properties over the model by interpreting its global state transition graph as an extended Kripke structure, and property validation amounts to a partial or complete exploration of the state space.

The success of model checking lies in its potential for *push-button* automation and in its error reporting capabilities. A model checker performs an automated complete exploration of the state space of a software model, commonly using a depth-first search strategy. When a property violating state is encountered the search stack contains an error trail that leads from an initial system state into the encountered state. This error trail greatly helps software engineers in interpreting validation results. The sheer size of the reachable state space of realistic software models imposes tremendous challenges on the algorithmics of model checking technology. Complete exploration of the state space is often impossible, and approximations are needed.

Recent advances have lead to a growth of interest in the use of the technology in AI. Heuristic and local search techniques can be directly integrated into existing model checkers. Different to heuristic search, which improves goal finding in action planning, *directed model checking* accelerates error detection [13]. With the *model checking as action planning* approach, software fragments are translated into a planning problem with planning operators for each source code line, having individual preconditions and effects [8]. However, model checking problems in software verification practice are more expressive, since complex source code instructions have to be dealt with. Therefore, it is apparent that a planning heuristic is applicable to existing model checking technology, only if it can handle non-linear expressions.

```
**Procedure** *Test*$(exp, \min, \max)$
    **if** $(op(exp) = \geq [>])$
        **return**
            $Eval^{+}(left(exp), \min, \max) \geq [>]$
            $Eval^{-}(right(exp), \min, \max)$
    **if** $(op(exp) = \leq [<])$
        **return**
            $Eval^{-}(left(exp), \min, \max) \leq [<]$
            $Eval^{+}(right(exp), \min, \max)$
    **if** $(op(exp) = =)$
        **return**
            $Eval^{+}(left(exp), \min, \max) \geq Eval^{-}(right(exp), \min, \max) \wedge$
            $Eval^{-}(left(exp), \min, \max) \leq Eval^{+}(right(exp), \min, \max)$
```

Fig. 2. Test if an expression is valid within the bounds.

3 Generalized Numerical Relaxed Plans

In the proposed alternative to the numerical planning heuristic, the planning graph is built and analyzed according to two main subroutines: *Test* and *Update*. The former procedure takes a vector of intervals for minimal and maximal variable bounds and tests if a given constraint is satisfied by at least one possible vector assignment. The latter procedure adjusts the bounds according to assignment effects in operators.

Both subroutines refer to function *Eval*(exp) that calculates the maximal and minimal value that an expression *exp* with variables v_i in $[\min^i, \max^i]$ can take. Function *Eval*(exp) is divided into two parts: *Eval*$^{+}(exp)$ computes the maximal value, and *Eval*$^{+}(exp)$ computes the maximal value of expression *exp*. Note that computing good bounds for an expression is not simple and refers to analyzing non-trivial functions. In our case *Eval* is itself an approximation that traverses the arithmetic tree in bottom-up fashion and uses constraint propergation rules to determine the bounds for the individual arithmetic operations. For example, for multiplying of two variables v_i and v_j we compute *Eval*$^{-}(v_i \cdot v_j)$ as $\min\{\min_i \cdot \min_j, \max_i \cdot \min_j, \min_i \cdot \max_j, \max_i \cdot \max_j\}$, and *Eval*$^{+}(v_i, v_j)$ as $\max\{\min_i \cdot \min_j, \max_i \cdot \min_j, \min_i \cdot \max_j, \max_i \cdot \max_j\}$. Unfortunately, this results in *Eval*$^{-}(v_i \cdot v_i) = -100$ for $v_i \in [-10, 10]$, where a refined study reveals that *Eval*$^{-}(v_i \cdot v_i) = 0$. Specialized techniques and refined bounds consistency algorithms can be applied to improve the inference of the bounds [24].

A *test* of a condition *exp* within the vector of variable bounds relaxes the requirement for accurate assignment information. If *any* assignment vector to the variables in the given ranges satisfies the conditions then the procedure returns *true*. In the pseudocode in Figure 2 we perform a case study according to the comparison operator at the root of the expression *exp* and evaluate both subtrees for the maximal and the minimal possible value. It is not difficult to see, that if *Eval*$^{+[-]}(exp)$ calculates a maximal [minimal] value that an expression *exp* with variables v_i in $[\min^i, \max^i]$ can take, then *Test* returns *true* if there exist an assignment $a \in [\min, \max]$ to v that fulfills *exp*.

```
**Procedure** *Update*$(exp, \min', \max', \min, \max)$
    $v_{\min} \leftarrow Eval^{-}(\min', \max')$
    $v_{\max} \leftarrow Eval^{+}(\min', \max')$
    **if** $(op(exp) = \uparrow)$
        **if** $(v_{\min} < 0)$ $\min_{head(exp)} \uparrow v_{\min}$
        **if** $(v_{\max} > 0)$ $\max_{head(exp)} \uparrow v_{\max}$
    **if** $(op(exp) = \downarrow)$
        **if** $(v_{\min} > 0)$ $\min_{head(exp)} \downarrow v_{\min}$
        **if** $(v_{\max} < 0)$ $\max_{head(exp)} \downarrow v_{\max}$
    **if** $(op(exp) = \leftarrow)$
        **if** $(v_{\min} < \min_{head(exp)})$ $\min_{head(exp)} \leftarrow v_{\min}$
        **if** $(v_{\max} > \max_{head(exp)})$ $\max_{head(exp)} \leftarrow v_{\max}$
```

Fig. 3. Update according to a given expression.

The observation is true for all options that we have devised. In each case we select the weakest condition for the set of variables that is available. E.g. for condition $\geq$ we determine if the maximum variable assignment on the left hand side is larger than the minimum variable assignment on the right hand side.

Figure 3 depicts the implementation for the *Update* procedure according the three main assignment operators $\leftarrow$, $\uparrow$, and $\downarrow$. First, the minimal and maximal evaluation values $v_{\min}$ and $v_{\max}$ are determined with respect to the old bounds. Then the new bounds are updated if the new evaluation exceeds the existing bounds.

If *Eval*$^{+[-]}$(*exp*) calculates a maximal [minimal] value that an expression *exp* with variables v_i in $[\min^i, \max^i]$ can take, then *Update* adjusts the bounds so that each possible outcome of an assignment with variable values in $[\min', \max']$ is in $[\min, \max]$.

The operation modifies $[\min', \max']$ to $[\min, \max]$ given that $[\min, \max]$ is initialized with $[\min', \max']$. Since evaluation determines the lower and upper bound of variables in the expression tree, the three update rules we have devised, re-adjust the bounds conservatively. If, for example, we have an increase in variable h of a value of at least $v_{\min} < 0$ and of at most $v_{\max} > 0$, then the new interval $I = [\min_i + v_{\min}, \max_i + v_{\max}]$ ensures that the variable assignment to v_i will yield a value that is contained in I.

3.1 Relaxed Plan Generation

In Figure 4 we show the plan generation module to construct the relaxed planning graph. The presentation was chosen to be aligned with the one in [20]. For each layer t in the relaxed planning graph, a set of propositions and a vector $(\min_t, \max_t)$ of bounds for each variable is maintained, where $\mathcal{C}$ is the current state, $\mathcal{G}$ is the planning goal description, $p(\cdot)$ denotes the propositions true in a given state, and $v(\cdot)$ denotes the variable assignments with respect to a given state. To select the set of applicable actions A_t, we apply procedure *Test* to the vectors $\min_t$ and $\max_t$ with respect to the precondition

```
**procedure** Relax($\mathcal{C}, \mathcal{G}$)
    $P_0 \leftarrow p(\mathcal{C}); \forall i : \min_0^i \leftarrow \max_0^i \leftarrow v_i(\mathcal{C})$
    $t \leftarrow 0$
    **while** ($p(\mathcal{G}) \not\subseteq P_t$ **or** $\exists exp \in v(\mathcal{G}) : \neg Test(\exp, \min_t, \max_t)$)
        $A_t = \{a \in A \mid pre(a) \subseteq P_t,$
                    $\forall exp \in v(pre(a)) : Test(\exp, \min_t, \max_t))\}$
        $P_{t+1} \leftarrow P_t \cup \bigcup_{pre(a) \subseteq P_t} add(a)$
        $[\min_{t+1}, \max_{t+1}] \leftarrow [\min_t, \max_t]$
        **for** $a \in A_t, exp \in v(eff(a))$
            $Update(exp, \min_t, \max_t, \min_{t+1}, \max_{t+1})$
        **if** (*relaxed problem unsolvable*) **return** $\infty$
        $t \leftarrow t + 1$
```

Fig. 4. Generating the problem graph for the *generalization of the relaxed planning heuristic*.

expressions for each action. The process is continued until all propositional and all numerical goals are satisfied, or the relaxed problem proves to be unsolvable. We have not yet derived the latter criterion. Unfortunately, the simple fixpoint condition $P_t = P_{t+1}$ – as in the propositional case – together with $[\min_t, \max_t] = [\min_{t+1}, \max_{t+1}]$ may not be sufficient, since the growth of some numerical variables can be unbounded. The option applied in [20] pre-computes bounds for *relevant variables*, by tracing the cone of influence for the variables and propositions in the goal description. The bounds are referred to as *max-need*. In our case, we would have have to extend the computation to *min-need*$_i$, so that the computation is terminated if $P_t = P_{t+1}$ and $[\min_{t+1}, \max_{t+1}] \not\subseteq$ [*min-need*, *max-need*].

The calculations can be performed recursively, initializing [*min-need*$_i$, *max-need*$_i$] with $[-\infty, \infty]$ and further restricting the interval to the numerical conditions in the goal description and action preconditions. We recursively propagate the bounds through the numerical effect lists by determining the weakest preconditions that have to be satisfied if the given post-conditions are met. This will further restrict the intervals [*min-need*$_i$, *max-need*$_i$]. The process is continued until a fix-point is reached. *Irrelevant variables* are those in which [*min-need*$_i$, *max-need*$_i$] $= [-\infty, \infty]$ and can be omitted from the fixpoint requirement.

If algorithm *Relax* terminates with value ∞, then there is no solution to the original planning problem. The two aspects necessary to consider are: i) every plan for the original problem also solves the relaxed one and ii) if $P_t = P_{t+1}$ and $[\min_{t+1}, \max_{t+1}] \not\subseteq$ [*min-need*, *max-need*] once in the relaxed exploration process it will remain the same for all upcoming iterations. Part i) is true, since satisfying any numerical condition or any proposition will be preserved by the relaxation process. If condition *exp* or proposition p will be reachable in the original problem, so it will be in the relaxed one. For part ii) we observe that for all levels in the plan graph we have $P_t \subseteq P_{t+1}$ and $[\min_t, \max_t] \subseteq [\min_{t+1}, \max_{t+1}]$.

The complexity of the algorithm is proportional the size of the plan graph times the maximal length of the condition and effect lists and the maximal expression tree size to evaluate the vector in. Since operators can apply more that once, even in the case of bounded lists this does not necessarily imply polynomial complexity for deciding the relaxed task. In fact, the depth of the graph can be exponential in the binary encoding of the variables values in start and goal description. However, one may introduce so-called "∞ handling" rules as shown in [20] to match the result that deciding propositional relaxed PLANSAT is polynomial [5].

3.2 Relaxed Plan Extraction

For the extraction process as shown in Table 5, we first determine the minimal layers of the goal propositions and target arithmetic conditions in $\mathcal{G}$ to initialize the *pending queues* of requests. More precisely, in each layer in the planning graph we maintain two queues, one for the set of facts that have still to be processed and one for the set of constraints that have to be satisfied and propagated. In the procedure *Extract* the two queues are referred to as $p(G_i)$ and $v(G_i)$, with i being the smallest layer in which the propositions constraints are satisfied.

Next we greedily traverse the constructed graph backwards, layer by layer. To determine the set of matching operators A, for each layer i we process set A_i and reconstruct the vector $\min_{i+1}$ and $\max_{i+1}$ of lower and upper bounds to the variables, using the *Update* procedure. This will ease to determine which of the operators do match. There are two cases. Either the propositional *add* effect of an operator matches our propositional pending queue or a numerical constraint matches one in the arithmetic condition queue. In both cases we delete the matching condition from the queue and propagate the preconditions of the selected action a as yet to be established. The relaxed plan A is extended by action a. The minimal layer for each precondition, as denoted by the variable *level*, can be derived using additional information selected in the forward phase. Alternatively, the layer can be re-computed by applying procedure *Test* to the vectors $\min_j$ and $\max_j$ as computed for the level $j \in \{1, \ldots, i-1\}$.

The remaining pseudo-code fragment – starting with the line "**for** *exp*$'$ $\in v($*eff*$(a)))$" considers how to modify and propagate the numerical expression for which we have found a match, to allow repeated operator application. For instance, say that we have a lower bound on a variable of 10 units and an operator that increases the variable content by 2 units, then it is appropriate for the first application to denote that there are still 8 units to be produced. For the sake of simpler exposition, for the following we assume that we have updates only according to the uniform assignment operator "$\leftarrow$". This is not a restriction, if we allow variables, that appear in the heads of an assignment to re-appear on the right hand side.

After selecting the numerical assignment that fires, we need to determine the associated *weakest preconditions* to be included in the pending queue in its individual smallest available layer. This is another precondition that has to be satisfied for the relaxed plan. In this case the (minimal) layer for the new goal *has to be* be computed using procedure *Test* and the vectors $(\min_j, \max_j)$ for increasing level index $j \in \{1, \ldots, i\}$. Note that the currently active Layer i is included in the range for j.

```
**Procedure** *Extract*$(\mathcal{G})$
$A \leftarrow \emptyset$
**for** $i \in \{1, \ldots, t\}$
    $p(G_i) \leftarrow \{g \in p(\mathcal{G}) \mid \mathit{level}(g) = i\}$
    **for** $\mathit{exp} \in v(\mathcal{G})$
        **if** $\mathit{Test}(exp, \min_i \max_i)$
            $v(G_i) \leftarrow v(G_i) \cup \{exp\}; v(\mathcal{G}) \leftarrow v(\mathcal{G}) \setminus \{\mathit{exp}\}$
**for** $i \in \{t, \ldots, 1\}$
    $[\min_{i+1}, \max_{i+1}] \leftarrow [\min_i, \max_i]$
    **for** $a \in A_i$
        **for** $\mathit{exp} \in v(\mathit{eff}(a))$
            $\mathit{Update}\ (\mathit{exp}, \min_i, \max_i, \min_{i+1}, \max_{i+1})$
        **for** $e \in \mathit{add}(a)$
            **if** $e \in p(G_i)$
                $A \leftarrow A \cup \{a\}; p(G_i) = p(G_i) \setminus \mathit{add}(a)$
                **for** $p \in p(\mathit{pre}(a))$:
                    $p(G_{\mathit{level}(p)}) \leftarrow p(G_{\mathit{level}(p)}) \cup \{p\}$
                **for** $\mathit{exp} \in v(\mathit{pre}(a))$
                    $v(G_{\mathit{level}(\mathit{exp})}) \leftarrow v(G_{\mathit{level}(\mathit{exp})}) \cup \{exp\}$
        **for** $\mathit{exp} \in v(G_i)$
            **if** $\mathit{Test}(\mathit{exp}, \min_{i+1}, \max_{i+1})$
                $A \leftarrow A \cup \{a\}; v(G_i) = v(G_i) \setminus \{\mathit{exp}\}$
                $p(G_i) = p(G_i) \setminus \mathit{add}(a)$
                **for** $p \in p(\mathit{pre}(a))$
                        $p(G_{\mathit{level}(p)}) \leftarrow p(G_{\mathit{level}(p)}) \cup \{p\}$
                **for** $\mathit{exp} \in v(\mathit{pre}(a))$
                        $v(G_{\mathit{level}(\mathit{exp})}) \leftarrow p(G_{\mathit{level}(\mathit{exp})}) \cup \{exp\}$
                **for** $\mathit{exp}' \in v(\mathit{eff}(a))$
                    $h \leftarrow \mathit{head}(\mathit{exp}')$
                    $\mathit{exp} \leftarrow \mathit{exp}[h \setminus \mathit{exp}']$
                **for** $j \in \{1, \ldots, i\}$
                    **if** $(\mathit{Test}(\mathit{exp}, \min_j, \max_j))\ l \leftarrow j$
                $v(G_l) \leftarrow v(G_l) \cup \{\mathit{exp}\}$
**return** $|A|$
```

Fig. 5. Extracting the relaxed plan for the proposed *numerical relaxed planning heuristic*.

As a consequence, we need an option to derive the *weakest precondition* for a given *expression* with respect to an *assignment operator*. To solve this problem we go back to the early stages of verifying *partial correctness* of computer programs, namely to the *Hoare calculus*. In the calculus we have rules that are of the form

$$\frac{\textit{Premises}}{\textit{Conclusion}}.$$

Available rules are *skip* (the trivial operation) *assignment*, *composition*, *selection*, *iteration*, and *consequence*. For example, if S_1 and S_2 were programs and p, q, and r were conditions, then the *composition rule* says

$$\frac{\{p\}\ S_1\ \{q\}; \{q\}\ S_2\ \{r\}}{\{p\}S_1; S_2\{r\}}.$$

The *selection rule* is written as

$$\frac{\{p \wedge B\}\ S_1\ \{q\}; \{q \wedge \neg B\}\ S_2\ \{r\}}{\{p\}\ \textbf{if}\ (B)\ \textbf{then}\ S_1\ \textbf{else}\ S_2\ \{q\}},$$

while the *iteration rule* is denoted as

$$\frac{\{p \wedge B\}\ S\ \{p\}}{\{p\}\textbf{while}\ (B)\ \textbf{do}\ S\ \{p \wedge \neg B\}}.$$

A proof of partial correctness starts with simple instructions and certified conditions on variable values. An analysis covers incrementally growing programm fragments, while maintaining respective pre- and postcondition lists. A program is *totally correct* if it is partially correct and terminating. Here we are only interested in the *assignment rule* to derive the weakest precondition of an assignment. It is denoted as

$$\{p[x \setminus t]\}\ x \leftarrow t;\ \{p\},$$

where x is a variable, p the postcondition of the assignment, and $[x \setminus t]$ is the substitution of t in x. As an example we take a program that merely consists of the assignment $u \leftarrow 3x + 17$, and a postcondition p of the form $u < 5x$. To find the weakest precondition for S with respect to p we have $t = 3x + 17$, so that $p[u \setminus t]$ is equal to $3x + 17 < 5x$, or equivalently $x > 8.5$.

The application in the program *Extract* is as follows. Suppose that we have an assignment of the form $h \leftarrow exp$ in one of the effects of the selected operator a and a postcondition of the form exp'. Both expressions exp and exp' are given in form of arithmetic trees. The weakest precondition is now found substituting each occurrence of h in exp' by exp. It will be simplified and inserted in the appropriate pending queue.

4 Experiments

To illustrate practical feasibility of our approach, we have implemented the heuristic estimate in our heuristic search forward chaining planner MIPS [9]. The planner has been

id	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
A*	2	7	5	7	3	2	3	11	13	2	3	2	5	5	3	2	3	9	2	2
BFS	5	33	10	26	7	2	59	22	65	6	4	5	28	7	38	2	5	40	2	2

Table 1. Number of expanded nodes in *Arithmetic* Domain.

extended to PDDL2.2 [11] to deal with timed initial literals and derived predicates [10]. For IPC-4 it has been used to check solvability especially temporal benchmark designs [21]. In our implementation we terminate relaxed planning graph construction, if the depth of the relaxed planning graph exceeds a pre-specified threshold. Since the relaxed planning graph reflects a simplified parallel execution of actions, goals for benchmark problems often appear in shallow depth.

Unfortunately, grounded and simplified benchmark problems in all planning competitions are at most linear. This lack of expressiveness is due to a compromise with respect to existing planning technology. Subsequently, we selected three different options for experimental evaluation: 1) problems that are non-linear by definition, 2) existing but modified benchmark domains, and 3) challenging linear domains. Although all problems agree with PDDL syntax, no other planning system can solve either of the domains. In the first two cases this is due to the lack in expressivity, while in the second case this is a due to a lack in performance[3].

First, we generated random instances to the *Arithmetic* domain with target value 24. As depicted in Table 1 the exploration efforts for A* and *breadth-first search* (BFS) are very small. A*'s exploration shows a considerable improvement to the uninformed one. Both algorithms found optimal solutions. In contrast, *enforced hill climbing* often fails to establish a plan.

It is not difficult, to introduce non-linearities to existing benchmarks. In *ZenoTravel* we squared some preconditions, artificially making the domain non-linear. Since determining that variables are strictly positive can be as hard as the planning problem itself, actual planners will not be able to simplify the above formula. As shown to the left in Table 2, the planner solved all modified instances. The table is headed by problem *id*, the number of expanded *nodes*, the CPU search *time* on a 1.8 GHz Windows PC with 256 MB memory, the obtained solution *value*, and the *length* of the established plan.

We also conducted experiments in two domains provided by Amol Mali: one version of *Jugs*, where some pouring actions and goal constraints are non-linear, and *Karel, the Robot* domain, where the distance to move is non-linearly dependent to the current positition, and some goal constraints are also non-linear. We could solve selected *Karel* domains, e.g. *Karel-2* required 98 expansions, 0.15s CPU time to produce a 5-step plan, while in *Jugs* the heuristic turns out to be too weak: complete jug fillings span the entire range for the *content* variables in the first layer of the relaxed planning graph.

Table 2 highlights that the approach is efficient in linear domains. On a slightly larger machine (2.66 GHz Linux PC with 512 MB memory) we could generate the

[3] In the competition 2004, *SGPlan* turned out to be the only other system that solved entire *Settlers*, while *P-MEP* was announced to be a system that has non-linear expressivity.

id	nodes	time	value	length
1	1	0.06	13,564	1
2	10	0.12	7,567	8
3	26	0.13	10,455	8
4	15	0.15	23,422	11
5	37	0.17	12,009	14
6	53	0.20	28,642	14
7	40	0.21	11,977	16
8	53	0.29	41,364	20
9	32	0.36	19,921	23
10	74	0.43	559,60	28
11	31	0.36	34,362	17
12	73	0.47	22,650	26
13	129	0.55	45,358	33
14	59	3.63	1,815	39
15	67	7.77	76,602	40
16	287	15.60	117,050	55
17	709	35.48	153,021	78
18	1,474	67.35	85,896	72
19	2,522	128.75	131,528	103
20	2,429	158.17	283,688	113

id	nodes	time	value	length
1	551	4.48	142	58
2	39	1.40	30	35
3	2,984	17.17	880	101
4	569	5.36	51	42
5	99	5.25	44	68
6	999	14.57	44	67
7	17	4.29	62	16
8	0	2.13	-	-
9	8,515	295.59	2,741	270
10	404	59.15	3,979	159
11	695	42.18	2,694	168
12	251	58.92	1,878	133
13	35,377	1,358	3,897	218
14	761	84.39	1,815	304
15	399	164.60	3,688	261
16	893	194.62	4,827	275
17	13,232	2,401	9,738	285
18	7,803	846.75	8,784	338
19	171,465	725.44	6,428	415
20	23,643	3,728.05	0	286

Table 2. Results in *Modified Zeno Travel* (left) and in *Settlers* (right).

first report in solving the entire problem suite of *Settlers*. Our planner also detects that Problem 8 is unsolvable; the goal requires a railway from *location6* to *location3*, in contrast to connectivity status of the two locations.

5 Conclusion

In this work we extended the mixed propositional and numerical relaxed planning heuristic [20], which itself is an extension to the propositional relaxed planning heuristic and the relaxed planning heuristic for restricted tasks. As the existing approach to translate *ignoring the delete lists* to numeric state variables restricts to at most linear tasks, the question was how to tackle non-linear expressions that are available in PDDL2.1 and do appear frequently in practice. In difference to the introduction of upper bounds and negated variables together with a transformation process beforehand, the proposed alternative applies upper and lower bounds. The algorithmic considerations of the generalization are tricky but base on simple subroutines. It enables to deal with complex problems and shows a way of understanding the heuristic in more detail.

The relaxed planning heuristic has been recently applied to *conformant planning problems* [3] and has also been integrated in a two stage scheduling approach for *temporal planning* [9]. In case of conformant planning, the planing graph construction is associated with an *implication graph* to derive information on atoms that are known to be true or false in a given level, while the temporal planning apporach parallelizes sequential plans with respect to action duration and action dependency. By *scheduling partial, relaxed and final plans*, non-linear expressions are universal for Level 1-3 PDDL2.1 planning problems. Moreover, with relaxed plans for non-linear tasks we are not limited to PDDL2.1 planning. In concurrent work, we have extended our planning approach to PDDL2.2 [11] including *derived predicates* and *timed initial literals*. Our implementation solved relaxed non-linear problems in various domains that planners like *Metric-FF* cannot deal with.

In the context of the 2004 international planning competition, the forward state-space planner *(P-)MEP* [26], for *(parallel) more expressive planner*, is announced to handle PDDL2.1 expressivity and to apply a related technique of bounding intervals to generate a numerical relaxed planning graph and to extract a relaxed plan. Similar to MIPS it also supports ADL functionality and for Level 3 planning as it applies temporal reasoning to find a schedule after a sequential plan has been found. When comparing the two approaches we first realize that planning graph construction is similar in both cases. S-MEP [27] constructs the planning graph allowing each operator to apply at most once. In P-MEP – the participating planner in IPC-4 – this problem has been fixed. The refinement strategy merely reduces to relevant variables and is very different to the plan extraction phase we propose. The approach does not support weakest preconditioning. In some sense (P-)MEP truncates the planning graph of the forward phase, instead of greedily extracting plans as done in Metric-FF and MIPS. As IPC-4 has shown, P-MEP has efficiency deficies. For example, it cannot solve any of the instances in *Settlers*.

Although we proved that a generalization of the relaxed planning estimate is available for full PDDL, we strongly believe that there is much more efforts needed to make the tool appropriate e.g. for software model checking problems. Therefore, next we will address CSP techniques for evaluation and a fine-grained or mixed representation of the set of possible values that is available for a numerical variable. One option is to keep precise numeric information available until a certain threshold on the number values is exceeded. In this case we might swap back to finite constraint variable domains. We are certain that extending the approach is compatible with the *Test*-and-*Update* scheme for the construction of the relaxed planning graph and the extraction of the relaxed plan.

As the core motivation of the work is apply the relaxed planning heuristic to model checking, in future we may try implementing the estimate in an existing model checker.

Acknowledgments The work is supported by *Deutsche Forschungsgemeinschaft* (DFG) in the projects *Heuristic Search* (Ed 74/3) and *Directed Model Checking* (Ed 74/2).

References

1. B. Bérard, A. F. M. Bidoit, F. Laroussine, A. Petit, L. Petrucci, P. Schoenebelen, and P. McKenzie. *Systems and Software Verification*. Springer, 2001.
2. B. Bonet and H. Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001.
3. R. Brafman and J. Hoffmann. Conformant planning via heuristic forward search: A new approach. In *International Conference on Automated Planning and Scheduling (ICAPS)*, 2004. 335-364.

4. R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):142–170, 1992.
5. T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, pages 165–204, 1994.
6. E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.
7. S. Edelkamp. Planning with pattern databases. In *European Conference on Planning (ECP)*, 2001. 13-24.
8. S. Edelkamp. Promela planning. In *Workshop on Model Checking Software (SPIN)*, pages 197–212, 2003.
9. S. Edelkamp. Taming numbers and durations in the model checking integrated planning system. *Journal of Artificial Research (JAIR)*, 20:195–238, 2003.
10. S. Edelkamp. Extended critical paths in temporal planning. In *Proceedings ICAPS-Workshop on Integrating Planning Into Scheduling*, pages 38–45, 2004.
11. S. Edelkamp and J. Hoffmann. PDDL2.2: The language for the classical part of the 4th international planning competition. Technical report, University of Freiburg, 2003.
12. S. Edelkamp, J. Hoffmann, M. Littman, and H. Younes. *Proceedings Fourth International Planning Competition, International Conference on Automated Planning and Scheduling*. Jet Propulsion Laboratory, 2004.
13. S. Edelkamp, S. Leue, and A. Lluch-Lafuente. Directed explicit-state model checking in the validation of communication protocols. *International Journal on Software Tools for Technology (STTT)*, 2004.
14. R. Fikes and N. Nilsson. Strips: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2:189–208, 1971.
15. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Research (JAIR)*, 20:61–124, 2003.
16. B. C. Gazen and C. Knoblock. Combining the expessiveness of UCPOP with the efficiency of graphplan. In *European Conference on Planning (ECP)*, pages 221–233, 1997.
17. P. Haslum and H. Geffner. Admissible heuristics for optimal planning. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 140–149, 2000.
18. J. Hoffmann. Local search topology in planning benchmarks: An empirical analysis. In *International Joint Conferences on Artificial Intelligence (IJCAI)*, pages 453–458, 2001.
19. J. Hoffmann. Local search topology in planning benchmarks: A theoretical analysis. In *Artificial Intelligence Planning and Scheduling (AIPS)*, pages 379–387, 2002.
20. J. Hoffmann. The Metric FF planning system: Translating “Ignoring the delete list” to numerical state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.
21. J. Hoffmann, S. Edelkamp, R. Englert, F. Liporace, and S. Thiebaux. Towards realistic benchmarks for planning: the domains used in the classical part of IPC-4. In *Proceedings Fourth International Planning Competition*, pages 8–15, 2004.
22. J. Hoffmann and B. Nebel. Fast plan generation through heuristic search. *Journal of Artificial Intelligence Research*, 14:253–302, 2001.
23. K. L. McMillan. Symbolic model checking. In M. K. Inan and R. P. Kurshan, editors, *Verification of Digital and Hybrid Systems*, pages 117–137. Springer, 1998.
24. K. Meriott and P. Stuckey. *Programming with Constraints*. MIT Press, 1998.
25. J. Pearl. *Heuristics*. Addison-Wesley, 1985.
26. J. Sanches, M. Tang, and A. D. Mali. P-MEP: Parallel more expressive planner. In *Proceedings Fourth International Planning Competition*, pages 53–55, 2004.
27. J. Sanchez and A. D. Mali. S-MEP: A planner for numeric goals. In *Proceedings IEEE International Conference Tools with Artificial Intelligence (ICTAI)*, pages 274–283, 2003.
28. R. van der Krogt, M. de Weerdt, and C. Witteveen. Exploiting opportunities using planning graphs. In *UK Planning and Scheduling SIG (PlanSig)*, pages 125–136, 2003.