In: Proc. 8th European Conference on Logics in Artificial Intelligence (JELIA 2002), Cosenza, Italy, © 2002 Springer.

# Answer Set Planning under Action Costs *

Thomas Eiter[1], Wolfgang Faber[1], Nicola Leone[2], Gerald Pfeifer[1], and Axel Polleres[1]

[1] Institut für Informationssysteme, TU Wien, A-1040 Wien, Austria
{eiter,faber,polleres}@kr.tuwien.ac.at,
pfeifer@dbai.tuwien.ac.at
[2] Department of Mathematics, University of Calabria, I-87030 Rende (CS), Italy
leone@unical.it

**Abstract.** We present $\mathcal{K}^c$, which extends the declarative planning language $\mathcal{K}$ by action costs and optimal plans that minimize overall action costs (cheapest plans). As shown, this novel language allows for expressing some nontrivial planning tasks in an elegant way. Furthermore, it flexibly allows for representing planning problems under other optimality criteria as well, such as computing "fastest" plans (with the least number of steps), and refinement combinations of cheap and fast plans. Our experience is encouraging and supports the claim that answer set planning may be a valuable approach to advanced planning systems in which intricate planning tasks can be naturally specified and effectively solved.

## 1 Introduction

Recently, several declarative planning languages and formalisms have been introduced, which allow for an intuitive encoding of complex planning problems including ramifications, incomplete information, non-deterministic action effects, or parallel actions [13, 18, 17, 19, 12, 4–6]. While these formalisms are designed to generate any plans that establish the planning goals, in practice we are usually interested in particular plans that are optimal with respect to an objective function which measures the quality (or cost) of a plan. Often, this is just the number of time steps to achieve the goal, and many systems are tailored to compute shortest plans (e.g. CMBP [4] and GPT [2] compute shortest sequential plans, while Graphplan [1] computes shortest parallel plans).

However, there are other important objective functions to consider. If executing an action (such as traveling from Vienna to Lamezia Terme) causes some cost, then we may desire a plan which minimizes the overall cost of the actions in that plan. In answer set planning [18], where plans are represented by answer sets of a logic program, this kind of problem has not been addressed so far, to the best of our knowledge.

In this paper, we address this issue and present an extension of the planning language $\mathcal{K}$ [5, 6], where one can associate actions with costs. The main contributions are:

- We define syntax and semantics of a modular extension to $\mathcal{K}$. Costs are associated to an action by a designated `where`-clause describing a cost value.
- Action costs may be dynamic, as they potentially depend on the current stage of the plan when an action is considered for execution. Dynamic action costs have natural applications, such as variants of the well-known Traveling Salesperson example.

* This work was supported by FWF (Austrian Science Funds) under the projects P14781 and Z29-INF

– We sketch how planning with costs can be implemented by mapping it to answer set programming, as realized in a system prototype that we have developed. The prototype, ready for experiments, is available at http://www.dlvsystem.com/K/.
– Finally, we show that our language is capable of easily modeling optimal planning under various criteria: computing (1) "cheapest" plans (which minimize overall action costs); (2) "fastest" plans (with the least number of steps); and combinations of these, viz. (3) fastest plans among the cheapest, and (4) cheapest plans among the fastest. To our knowledge, task (3) has not been addressed in other works so far.

The extension of $\mathcal{K}$ by action costs provides a flexible tool for representing different problems. Moreover, by $\mathcal{K}$'s nature, we get the possibility to easily combine dealing with incomplete knowledge and plan quality, which is completely novel.

Our experience is encouraging and gives further evidence that answer set planning, based on powerful logic programming engines, allows for the development of advanced declarative planning systems in which intricate planning tasks can be naturally specified and decently solved.

For space reasons, we provide proofs and more material in an extended paper [8].

## 2 Review of Language $\mathcal{K}$

In this section, we give a brief informal overview of the language $\mathcal{K}$. We assume that the reader is familiar with action languages and the notions of actions, fluents, goals, and plans and refer to [6] for further details. For illustration, we shall use the following running example, for which a $\mathcal{K}$ encoding is shown in Figure 1.

**Bridge crossing.** Four men want to cross a river at night. It is bridged by a plank bridge, which can only hold up to two persons at a time. The men have a lamp, which must be used in crossing, as it is pitch-dark and some planks are missing. The lamp must be brought back; no tricks (like throwing the lamp or halfway crosses) are allowed. □

A state in $\mathcal{K}$ is characterized by the truth values of fluents, describing relevant properties in the universe of discourse. A fluent may be true, false, or unknown in a state; formally, a state is any consistent set $s$ of (possibly negated) legal fluent instances. Note that in world-state planning, each fluent is either true or false in a state (this can be easily emulated in $\mathcal{K}$).

An action is only applicable if some precondition holds in the current state, and its execution may cause a modification of truth values of some fluents.

*Background knowledge.* Static knowledge which is invariant over time is specified as a disjunction-free Datalog program which we require to have a total well-founded model (and therefore a unique answer set).[1]
In our example, the background knowledge is simply

```
person(joe).  person(jack).  person(will).  person(ave).
```

[1] The well-founded model can be calculated in polynomial time and if it is total it corresponds to the unique answer set.

```
(1)  actions :      cross2(X, Y) requires person(X), person(Y), X != Y.
(2)                 cross(X) requires person(X).
(3)                 takeLamp(X) requires person(X).

(4)  fluents :      across(X) requires person(X).
(5)                 diffSides(X, Y) requires person(X), person(Y).
(6)                 hasLamp(X) requires person(X).

(7)  initially :    caused -across(X). hasLamp(joe).

(8)  always :       executable cross2(X, Y) if hasLamp(X).
(9)                 executable cross2(X, Y) if hasLamp(Y).
(10)                nonexecutable cross2(X, Y) if diffSides(X, Y).

(11)                executable cross(X) if hasLamp(X).

(12)                executable takeLamp(X).
(13)                nonexecutable takeLamp(X) if hasLamp(Y), diffSides(X, Y).

(14)                caused across(X) after cross2(X, Y), -across(X).
(15)                caused across(Y) after cross2(X, Y), -across(Y).
(16)                caused -across(X) after cross2(X, Y), across(X).
(17)                caused -across(Y) after cross2(X, Y), across(Y).

(18)                caused across(X) after cross(X), -across(X).
(19)                caused -across(X) after cross(X), across(X).

(20)                caused hasLamp(X) after takeLamp(X).
(21)                caused -hasLamp(X) after takeLamp(Y), X != Y, hasLamp(X).

(22)                caused diffSides(X, Y) if across(X), -across(Y).
(23)                caused diffSides(X, Y) if -across(X), across(Y).

(24)                inertial across(X).
(25)                inertial -across(X).
(26)                inertial hasLamp(X).

(27)  noConcurrency.

(28)  goal :        across(joe), across(jack), across(will), across(ave)? (i)
```

**Fig. 1.** $\mathcal{K}$ encoding of the Bridge Crossing problem

*Type Declarations.* The ranges of the arguments of fluents and actions must be specified. For instance, line (1) in Figure 1 specifies the arguments of action `cross2`, where two persons cross the bridge together, while line (4) specifies a fluent describing the fact that a specific person is on the other side of the river. The literals after "`requires`" in a declaration are from $\Pi$ or built-in predicates. DLV$^{\mathcal{K}}$ [7], our implementation of $\mathcal{K}$, currently supports built-in predicates "`a < b`", "`a <= b`", and "`a != b`" with the obvious meaning for strings and numbers, predicates "`a = b + c`", "`a = b * c`" for integer arithmetics, and "`#int(X)`" which enumerates all integers (up to a limit set by the user).

*Causation Rules.* Causation rules ("rules" for brevity) are syntactically similar to rules of the language $\mathcal{C}$ [13, 18, 17] and are of the basic form "`caused f if B after A.`"

Informally, this reads: if `B` holds in the current state and `A` held in the previous state, then `f` is known to be true in the current state as well. Both the `if` and `after` parts are optional. A rule is called *static* if its after part is empty, and *dynamic* otherwise.

Rules are used to express effects of actions or ramifications. E.g., the rules (18) and (19) describe the effects of a single person crossing the bridge in either direction.

*Initial State Constraints.* Rules can apply to all states or only to the initial state (which may not be unique). This is expressed by the keywords "`always` :" and "`initially` :" preceding sequences of rules. For example, line (7) enforces the fluent `across` to be false in the initial state for all `X` satisfying the fluent declaration, i.e., for all persons.

*Executability of Actions.* This is expressed in $\mathcal{K}$ explicitly, as in lines (8) and (9) which state that two persons can jointly cross the bridge if one of them has a lamp. The same action may have multiple executability statements. Dually, "`nonexecutable A if B`." prohibits the execution of action `A` if condition `B` is satisfied. For example, line (13) says that two persons cannot cross the bridge together if they are on different sides of the bridge. In case of conflicts, `nonexecutable A` overrides `executable A`.

*Parallel Actions.* $\mathcal{K}$ permits simultaneous execution of actions. If, as on line (27), "`noConcurrency`." is specified, then at most one action at a time can be executed.

*Default and Strong Negation.* $\mathcal{K}$ supports strong negation (written as "-") where knowledge about a fluent `f` may be incomplete, i.e., in any given state neither `f` nor $-$`f` needs to hold. In addition, weak negation ("`not`"), interpreted like default negation in answer set semantics [11], is permitted in rule bodies. This allows for natural modeling of inertia, default properties, and dealing with incomplete knowledge in general.

*Macros.* $\mathcal{K}$ provides a number of macros as syntactic sugar. For example, the `inertial` statement in line (24) informally states that `across(X)` holds in the current state if `across(X)` held at the previous state unless `-across(X)` is explicitly known to hold. This macro expands to "`caused across(X) if not -across(X) after across(X)`."

Moreover, we can "totalize" the knowledge of a fluent by declaring "`total f`." which is a shortcut for a pair of rules "`caused f if not -f`." and "`caused -f if not f`." with the intuitive meaning that unless a truth value for `f` can be derived, the cases where `f` resp. $-$`f` are true will both be considered.

*Planning domains and problems.* In $\mathcal{K}$, a *planning domain* $PD = \langle \Pi, \langle D, R \rangle \rangle$ has a background knowledge $\Pi$, action and fluent declarations $D$, and rules and executability conditions $R$; a *planning problem* $\mathcal{P} = \langle PD, q \rangle$ has a planning domain $PD$ and a *query*

$$q = g_1, \ldots, g_m, \texttt{not}\ g_{m+1}, \ldots, \texttt{not}\ g_n\ ?\ (i)$$

where $g_1, \ldots, g_n$ are ground fluents and $i \geq 0$ is the plan length (see line (28)). An *(optimistic) plan* for $\mathcal{P}$ is a sequence $P = \langle A_1, \ldots, A_l \rangle$ of action instances with a supporting *trajectory* $T = \langle \langle s_0, A_1, s_1 \rangle, \langle s_1, A_2, s_2 \rangle, \ldots, \langle s_{l-1}, A_l, s_l \rangle \rangle$ to the goal, i.e., starting at a legal initial state $s_0$, the actions in $A_1$, $A_2$ etc. are executable and lead to legal successor states $s_1$, $s_2$ etc. such that all literals in $q$ are true in $s_l$.

If, as stated in Figure 1, `joe` initially carries the lamp, our problem has simple five-step plans where `joe` always carries the lamp and brings all others across; e.g.,

$P = \langle$ {`cross2(joe,jack)`}, {`cross(joe)`}, {`cross2(joe,will)`},
{`cross(joe)`}, {`cross2(joe,ave)`} $\rangle$

## 3 Actions with Costs

Using $\mathcal{K}$ and $\texttt{DLV}^{\mathcal{K}}$, it is possible to express and solve involved planning tasks. However, $\mathcal{K}$ and $\texttt{DLV}^{\mathcal{K}}$ offer no means for finding optimal plans with respect to any criteria. In particular, this applies to action costs, which are needed for the following elaboration of the bridge crossing example that is well-known as a brain-teasing riddle.

**Quick bridge crossing.** The four guys need different times to cross the bridge, namely 1, 2, 5, and 10 minutes, respectively. Walking in two implies moving at the slower rate. Is it possible to get all of them across within 17 minutes? □

On first thought, this is not feasible, since the seemingly optimal plan where `joe`, who is the fastest, keeps the lamp and leads all the others across takes 19 minutes. Surprisingly, as we will see, there *is* a better solution.

In order to allow for an elegant and convenient encoding of such optimization problems, we extend $\mathcal{K}$ to the language $\mathcal{K}^c$ where we can assign costs to actions.

### 3.1 Syntax of $\mathcal{K}^c$

Let $\sigma^{act}$ denote the set of action names, and $\mathcal{L}_{typ}$ the set of literals over predicates defined in the background knowledge including built-in predicates. Furthermore, let $\sigma^{var}$ denote the set of variable symbols. $\mathcal{K}^c$ merely extends action declarations in $\mathcal{K}$ to express action costs as follows.

**Definition 1.** *An* action declaration $d$ *in* $\mathcal{K}^c$ *is of the form:*

$$p(X_1, \ldots, X_n) \texttt{ requires } t_1, \ldots, t_m \texttt{ costs } C \texttt{ where } c_1, \ldots, c_k. \qquad (1)$$

*where (1)* $p \in \sigma^{act}$ *has arity* $n \geq 0$*, (2)* $X_1, \ldots, X_n \in \sigma^{var}$*, (3)* $t_1, \ldots, t_m$*,* $c_1, \ldots, c_k$ *are from* $\mathcal{L}_{typ}$ *such that every* $X_i$ *occurs in* $t_1, \ldots, t_m$*,*[2] *(4)* $C$ *is either an integer constant, a variable from the set of all variables occurring in* $t_1, \ldots, t_m$*,* $c_1, \ldots, c_k$ *(denoted by* $\sigma^{var}(d)$*), or the distinguished variable* `time`*, and (5)* $\sigma^{var}(d) \subseteq \sigma^{var} \cup \{\texttt{time}\}$*, and* `time` *does not occur in* $t_1, \ldots t_m$*.*

If $m = 0$, the keyword '`requires`' is omitted; if $k = 0$, the keyword '`where`' is omitted and '`costs` $C$' is optional. Planning domains and problems are defined as in $\mathcal{K}$.

Examples will be given in Section 3.3.

### 3.2 Semantics of $\mathcal{K}^c$

Semantically, $\mathcal{K}^c$ extends $\mathcal{K}$ by the cost values of actions at points in time. In any plan $P = \langle A_1, \ldots, A_l \rangle$, at step $1 \leq i \leq l$, i.e. at time $i$, the actions in $A_i$ are executed.

We recall that a ground action $p(x_1, \ldots, x_n)$ is a *legal action instance* of an action declaration $d$ w.r.t. a $\mathcal{K}$ planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$, if a ground substitution $\theta$ for $\sigma^{var}(d)$ exists such that $X_i\theta = x_i$, for $1 \leq i \leq n$ and $\{t_1\theta, \ldots, t_m\theta\} \subseteq M$, where $M$ is the unique answer set of the background knowledge $\Pi$. Any such $\theta$ is called a *witness substitution* for $p(x_1, \ldots, x_n)$. Action costs are now formalized as follows.

[2] Informally, this means that all parameters of an action must be "typed" in the `requires` part.

**Definition 2.** *Let $a = p(x_1, \ldots, x_n)$ be a legal action instance with declaration $d$ of the form (1), let $i \geq 1$ be a time point, and let $\theta$ be a witnessing substitution for $a$ with* `time` $\theta = i$. *Then*

$$cost_{i,\theta}(p(x_1, \ldots, x_n)) = \begin{cases} 0, & \text{if the } \texttt{costs} \text{ part of } d \text{ is empty;} \\ val(C\theta), & \text{if } \{c_1\theta, \ldots, c_k\theta\} \subseteq M; \\ \text{undefined}, & \text{otherwise.} \end{cases}$$

By involving the variable `time` it is possible to define time-dependent action costs. This can be used for complex variants of Traveling Salesperson, where the optimal tour does not only depend on the route taken, but also when a certain connection is used (e.g., imagine the traffic jams on some roads during the weekend, which could increase the cost of a connection). We do not elaborate on this application here due to space restrictions, see [8] or http://www.dlvsystem.com/K/.

Using $cost_{i,\theta}$, we introduce well-defined legal action instances and action costs:

**Definition 3.** *A legal action instance $a = p(x_1, \ldots, x_n)$ is* well-defined *iff for any time point $i \geq 1$, it holds that (i) there is some witness substitution $\theta$ for $a$ such that $cost_{i,\theta}(a)$ is an integer $\geq 0$, and (ii) $cost_{i,\theta}(a) = cost_{i,\theta'}(a)$ holds for any two witness substitutions $\theta, \theta'$ with defined costs.For any well-defined $a$, its unique cost at time point $i \geq 1$ is given by $cost_i(a) = cost_{i,\theta}(a)$ where $\theta$ is as in (i).*

In this definition, condition (i) ensures that some cost value exists, which is an integer number, and condition (ii) ensures that this value is well-defined, i.e., different witness substitutions $\theta$ and $\theta'$ for $a$ can not evaluate the `cost` part to different integer cost values. In our framework, the semantics of a $\mathcal{K}^c$ planning domain $PD = \langle \Pi, \langle D, R \rangle \rangle$ is only well-defined, if all legal instances of action declarations in $PD$ are well-defined. This will be fulfilled if, for instance, the variables $X_1, \ldots, X_n$ in (1) in database terms functionally determine the values of the other variables except `time`, i.e., any tuple of values $(x_1, \ldots x_n)$ for $X_1, \ldots, X_n$ has a unique extension to a tuple of values for all variables in $d$ except `time`. In the rest of this paper, we assume well-definedness of $\mathcal{K}^c$ unless stated otherwise. Violations of well-definedness can be easily detected by introducing designated fluents and rules, see [8] for details. Using $cost_i$, we now define costs of plans.

**Definition 4.** *Let $\mathcal{P}$ be a planning problem. Then, for any plan $P = \langle A_1, \ldots, A_l \rangle$ for $\mathcal{P}$, its cost is defined as $cost_{\mathcal{P}}(P) = \sum_{i=1}^{l} \left( \sum_{a \in A_i} cost_i(a) \right)$. A plan $P$ is* optimal *for $\mathcal{P}$ and fixed plan length $i$, if it has least cost among all plans of length $i$, i.e., $cost_{\mathcal{P}}(P) \leq cost_{\mathcal{P}}(P')$ for each plan $P'$ of length $i$ for $\mathcal{P}$. The cost of $\mathcal{P}$ w.r.t. plan length $i$ is the cost of an optimal plan for $\mathcal{P}$ and $i$.*

Usually one only can estimate some *upper bound* of the plan length, but does not know the exact length of an optimal plan. Although we have only defined optimality for a fixed plan length $i$, we will see in Section 4.1 that by appropriate encodings this can extended to optimality for plans with length *at most* $i$.

### 3.3 An optimal solution for crossing the bridge

To model the different times the four guys need to cross the bridge, we extend the background knowledge, where '`max`' determines which of two persons is faster:

```
speed(joe, 1). speed(jack, 2). speed(will, 5). speed(ave, 10).
max(A, B, A) :- speed(_, A), speed(_, B), A >= B.
max(A, B, B) :- speed(_, A), speed(_, B), B > A.
```

Next, we add costs to the action declarations for `cross` and `cross2` (leaving `takeLamp` unchanged, as the time to hand over the lamp is negligible).

```
  cross(X)  requires person(X) costs SX where speed(X, SX).
cross2(X,Y) requires person(X), person(Y), X < Y costs Smax
            where speed(X, SX), speed(Y, SY), max(SX, SY, Smax).
```

As easily seen, the cost of the 5-step plan considered in Section 2 is 19. However, when we also consider longer plans, we can find the following 7-step plan $P$ with cost 17:

$P = \langle$ {cross2(joe, jack)}, {cross(joe)}, {takeLamp(will)}, {cross2(will, ave)}
{takeLamp(jack)}, {cross(jack)}, {cross2(joe, jack)} $\rangle$

$P$ has least cost over all trajectories of any length establishing the goal and thus constitutes an optimal solution of our problem for fixed plan length $i \geq 7$ .

### 3.4 "Crossing the Bridge" under incomplete knowledge

$\mathcal{K}$ is well-suited to model problems which involve qualitative uncertainty such as incomplete initial states or non-deterministic action effects. The extension of $\mathcal{K}$ to $\mathcal{K}^c$ to include costs gracefully applies to so called secure (conformant) plans as well, which must reach the goal under all circumstances [5].

For example, assume we only know that some of the desperate guys has a lamp. If they now ask for a plan to safely cross the bridge, we need a (fast) secure plan that works under all possible initial situations. In $\mathcal{K}^c$, this can be easily modeled by replacing "`initially : hasLamp(joe).`" by the following, where the first statement says that each guy might have a lamp, and the second that at least one guy has one.

```
initially : total hasLamp(X).
            caused false if -hasLamp(joe), -hasLamp(jack),
                            -hasLamp(will), -hasLamp(ave).
```

Clearly, the optimal solution still takes at least 17 minutes, since the original case (where only `joe` has a lamp) is one of the possible initial situations. However, an optimal secure plan now takes at least 8 steps, since we must assure in the first step that either `joe` or `jack` has the lamp. One such a plan with cost 17 is

$P = \langle$ {takeLamp(joe)}, {cross2(joe, jack)}, {cross(joe)}, {takeLamp(will)}
{cross2(will, ave)}, {takeLamp(jack)}, {cross(jack)}, {cross2(joe, jack)} $\rangle$

## 4 Applications

### 4.1 Cost Efficient versus Time Efficient Plans

In this section, we show how our approach can be used to minimize plan length together with the costs of a plan under parallel actions. In [15, 16] various criteria for optimization are proposed for parallel action domains, such as minimizing the total number of actions, or the number of time steps. We will concentrate on the following generalizations of these optimization criteria with arbitrary action costs. Finding

($\alpha$) plans with minimal costs (cheapest plans) for a given number of steps,
($\beta$) plans with minimal time steps (shortest plans),
($\gamma$) shortest among the cheapest plans, and
($\delta$) cheapest among the shortest plans.

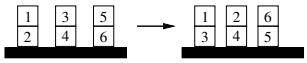


**Fig. 2.** A simple Blocks World instance

*($\alpha$) Cheapest plans for given plan length.* As a guiding example, we refer to the blocks world problem in Figure 2, where we want to find the minimal number of moves (possibly in parallel) to achieve the goal state. As background knowledge $\Pi_{bw}$, we use:

```
block(1). ... block(6).  location(table).  location(B) :- block(B).
```

and the following $\mathcal{K}^c$ program:

```
fluents :   on(B,L) requires block(B), location(L).
            blocked(B) requires block(B).
            moved(B) requires block(B).

actions :   move(B,L) requires block(B), location(L) costs 1.

always :    executable move(B,L) if B!=L.
            nonexecutable move(B,L) if blocked(B).
            nonexecutable move(B,L) if blocked(L).
            nonexecutable move(B,L) if move(B1,L), B!=B1, block(L).
            nonexecutable move(B,L) if move(B,L1), L!=L1.
            nonexecutable move(B,B1) if move(B1,L).

            caused on(B,L) after move(B,L).
            caused blocked(B) if on(_,B).
            caused moved(B) after move(B,_).
            caused on(B,L) if not moved(B) after on(B,L).

initially : on(1,2). on(2,table). on(3,4). on(4,table). on(5,6). on(6,table).

goal :  on(1,3), on(3,table), on(2,4), on(4,table), on(6,5), on(5,table) ? (i)
```

Each move is penalized with cost 1, minimizing the total number of moves. The instance has a two-step parallel solution which involves six moves:

$P = \langle\ \{\mathtt{move}(1,\mathtt{table}), \mathtt{move}(3,\mathtt{table}), \mathtt{move}(5,\mathtt{table})\},$
$\{\mathtt{move}(1,3), \mathtt{move}(2,4), \mathtt{move}(6,5)\}\ \rangle$

However, there is a sequential plan with only five moves:

$P = \langle\ \{\mathtt{move}(3,\mathtt{table})\},\ \{\mathtt{move}(1,3)\},\ \{\mathtt{move}(2,4)\},\ \{\mathtt{move}(5,\mathtt{table})\},\ \{\mathtt{move}(6,5)\}\ \rangle$

This plan can be parallelized to have 3 steps, but not to have 2 steps. For any length $\geq 3$, we obtain optimal plans involving 5 actions. Consequently, the minimal number of steps for a maximally parallelized plan with a minimal number of actions is 3.

*($\beta$) Shortest Plan.* For optimization ($\beta$) we assume that no action costs are specified in the original problem, and minimizing time steps is our sole target. We will show a general preprocessing method for $\mathcal{K}^c$ planning problems which, given an upper bound $i$ of time steps, guarantees plans with a minimal number of steps. Many typical applications have an inherent upper bound for the plan length. In blocks world for a configuration with $n$ Blocks, any goal configuration can be reached within at most $2n - s_i - s_g$ steps, where $s_i$ and $s_g$ represent the numbers of stacks in the initial and goal states.[3] Therefore, 6 is an upper bound for the plan length of our simple instance.

First, we add a new distinct fluent `gr` and a new distinct action `finish` to our domain and extend the `always` section of the program replacing the original goal:

```
fluents : gr.
actions : finish costs time.
always :  executable finish if g1, ..., gm, not gm+1, ..., not gn, not gr.
          caused gr after finish.
          caused gr after gr.
goal :    gr ? (i)
```

where $\mathtt{g}_1, \ldots, \mathtt{g}_m, \mathtt{not\ g}_{m+1}, \ldots, \mathtt{not\ g}_n$ are the original goal literals. Intuitively, `finish` represents a final action, which always has to be executed to finish the plan. The later it occurs, the more expensive is the plan. The fluent `gr` has the meaning "goal reached".

Furthermore, we want the action `finish` to occur exclusively and any occurrence of any other actions should be blocked as soon as the goal has been reached. Therefore, for any action `A` we add `not gr` to the `if` part of any executability condition for `A` and add a rule: `nonexecutable A if finish.`

Finally, to avoid any inconsistencies from static or dynamic effects, as soon as the goal has been reached, we add `not gr` to the `if` part of any causation rule of the original program except `nonexecutable` rules.[4]

If now $P' = \langle A_1, \ldots, A_j, A_{j+1}, \ldots, A_l \rangle$ is an optimal cost plan for the modified $\mathcal{K}^c$ planning problem $\mathcal{P}min$ for plan length $l$ where $A_j = \{\mathtt{finish}\}$, then $P'' = \langle A_1, \ldots, A_{j-1} \rangle$ is a minimal length plan for the original planning problem and all $A_{j+1} = \ldots = A_l = \emptyset$. Using this method, we obtain all 2-step parallel plans for our blocks world example.

Note that this approach for minimizing plan length is only efficient, if we know an upper bound close to the optimum. Searching for a minimum length plan by simply iteratively increasing the plan length could be more efficient when no such bound is known, as the search space might explode with a weak upper bound.

*($\gamma$) and ($\delta$)* In the last section, no costs were specified in the original program. If we want to find the *shortest among the cheapest plans* with arbitrary action costs, we have to set the costs of all actions higher than the highest possible cost value of action `finish`. Obviously, the highest cost for `finish` is the plan length $i$. Thus, we simply modify all action declarations by multiplying the original costs `C` with factor $i$:

[3] We can trivially solve any blocks world problem sequentially by first unstacking all blocks ($n - s_i$ steps) and then building up the goal configuration ($n - s_g$ steps).

[4] There is no need to rewrite `nonexecutable` rules because the respective actions are already "switched off" by rewriting of the executability conditions.

```
A requires B costs C1 where C1 = i * C, D.
```

This lets all other action costs take priority over the cost of `finish` and we can compute plans satisfying criterion ($\gamma$). Applying this rewriting to our blocks world example, a possible plan for length $i = 7$ is

$P = \langle$ {move(3, table)}, {move(1, 3), move(5, table)},
{move(2, 4), move(6, 5)}, {finish}, $\emptyset$, $\emptyset$, $\emptyset$ $\rangle$

As mentioned above, 6 is an upper bound for the plan length, but plan length $i = 7$ is needed for the final `finish` action. Analogously, in order to compute the cheapest among the shortest plans, the cost function of `finish` has to be adapted, such that the costs of `finish` take priority over all other actions costs. To this end, we set these costs high enough, by multiplying them with a factor higher than the sum of all costs of all legal action instances, i.e. the costs of all actions executed in parallel in $i$ steps. We thus can compute solutions for optimization criterion ($\delta$).

In our example, there are 36 possible moves; theoretically, we would have to set the costs of action `finish` to time $* 36 * i$. Though, at most 6 blocks can be moved in parallel, and it is sufficient to set the costs of `finish` to time $* 6 * i =$ time $* 42$. Accordingly, the action declarations are modified as follows:

```
actions :   move(B,L) requires block(B), location(L) costs 1.
            finish costs C where C = time * 42.
```

An optimal plan for the modified program for plan length (at most) 7 which in fact amounts to a two-step plan, is:

$P = \langle$ {move(1, table), move(3, table), move(5, table)},
{move(1, 3), move(2, 4), move(6, 5)}, {finish}, $\emptyset$, $\emptyset$, $\emptyset$, $\emptyset$ $\rangle$

## 5 Implementation

We briefly describe how planning under action costs can be implemented using a translation to answer set programming. We will define an extension $lp^w(\mathcal{P})$ of the logic program $lp(\mathcal{P})$ as defined in [7], such that its optimal answer sets (i.e., those minimizing weak constraint violation, see [10, 3]) correspond to the optimal cost plans for a planning problem $\mathcal{P}$.

We recall that in $lp(\mathcal{P})$ fluent and action literals are extended by an additional time parameter, and executability conditions as well as causation rules are modularly translated into corresponding program rules and constraints; disjunction is used for guessing at each point in time the actions which should be executed in the plan.

The translation $lp^w(\mathcal{P})$ for $\mathcal{K}^c$ problem $\mathcal{P}$ includes all rules of $lp(\mathcal{P}')$ from [7] for the $\mathcal{K}$ problem $\mathcal{P}'$ which results from $\mathcal{P}$ by omitting all cost parts of action declarations. In addition, for any action declaration $d$ of the form (1) with nonempty `costs` part, the following two statements are included (let $\overline{X} = X_1, \ldots, X_n$):

$$cost_p(\overline{X}, T, C) \text{ :- } p(\overline{X}, T),\ t_1, \ldots, t_m, c_1, \ldots, c_k, U = T + 1. \quad (2)$$

$$:\sim\ cost_p(\overline{X}, T, C).\ [C :] \quad (3)$$

In statement (2), $T$ and $U$ are new variables and each occurrence of `time` is replaced by $U$. Statement (3) is a *weak constraint*. Intuitively, a weak constraint denotes

a property which should preferably hold, and statement (3) associates a cost $C$ to the weak constraint, which can be interpreted as a penalty to be considered if the weak constraint is not satisfied. An optimal answer set is an answer set for which the sum of the penalties of violated weak constraints is minimal (We refer to [10, 3] for details). For example, the `cross` action defined in Section 3.3 is translated to:

$\texttt{cost}_{\texttt{cross}}(\texttt{X}, \texttt{T}, \texttt{SX})$ :- $\texttt{cross}(\texttt{X}, \texttt{T})$, $\texttt{person}(\texttt{X})$, $\texttt{speed}(\texttt{X}, \texttt{SX})$, $\texttt{U} = \texttt{T} + 1$.

$:\sim$ $\texttt{cost}_{\texttt{cross}}(\texttt{X}, \texttt{T}, \texttt{SX})$. [SX :]

As shown in [7], the answer sets of $lp(\mathcal{P})$ correspond to trajectories of optimistic plans for $\mathcal{P}$. We have similar results for plans with action costs.

**Theorem 1 (answer set correspondence).** *Let $\mathcal{P} = \langle PD, q\rangle$ be a (well-defined) $\mathcal{K}^c$ planning problem. Then, for each optimistic plan $P = \langle A_1, \ldots, A_l\rangle$ of $\mathcal{P}$ and supporting trajectory $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{l-1}, A_l, s_l\rangle\rangle$ of $P$, there exists an answer set $S$ of $lp^w(\mathcal{P})$ representing this trajectory such that the sum of weights of violated weak constraints equals $cost_{\mathcal{P}}(P)$, and vice-versa.*

**Corollary 1 (optimal answer set correspondence).** *For any well-defined $\mathcal{K}^c$ planning problem $\mathcal{P}$, the (trajectories $T = \langle\langle s_0, A_1, s_1\rangle, \ldots, \langle s_{l-1}, A_l, s_l\rangle\rangle$ of) optimal plans $P$ for $\mathcal{P}$ correspond to the optimal answer sets $S$ of $lp^w(\mathcal{P})$.*

Using these results, we have implemented an experimental prototype for planning in $\mathcal{K}^c$, which can be downloaded from http://www.dlvsystem.com/K/. Further documentation on techniques and usage of the prototype is available there and in [7]. For a more detailed discussion of the translation and the prototype we refer to [8].

## 6 Related Work and Conclusion

We have presented an extension of the language $\mathcal{K}$ which allows for the formulation of various optimality criteria of desired plans by means of variable action costs, and we sketched a translation to answer set programming with weak constraints. In fact, our implementation also supports computing admissible plans, i.e., plans the costs of which stay within a given limit (see [8]).

In the last years, it has been widely recognized that plan length alone is only one criterion to be optimized in planning. Several attempts have been made to extend heuristic search planners to allow for special heuristics respecting action costs, e.g. [9, 14].

A powerful approach is given in [20], where planning with resources is described as a structural constraint satisfaction problem (SCSP). The problem is solved by local search combined with global control. However, [20] promotes the inclusion of domain-dependent knowledge; the general problem has an unlimited search space, and no declarative high-level language is provided. Among other related approaches, [15] generalizes the "Planning as Satisfiability" approach to use integer optimization techniques for encoding optimal planning under resource production/consumption. In [16] an extension of the action language $\mathcal{C}$ is mentioned which allows for an intuitive encoding of resources and costs, but optimization is not considered in that framework.

A crucial difference between resource-based approaches and ours is that the former build on fluent values, while our approach hinges on action costs. This is a somewhat

different view of the quality of a plan. We plan to generalize our framework such that dynamic fluent values may contribute to action costs. Further possible extensions include negative action costs, which are useful for modeling producer/consumer relations among actions and resources, and different priorities (cost levels) to increase the flexibility and allow for optimizing different criteria at once. Another aspect to be explored is the computational complexity of $\mathcal{K}^c$, complementing the results in [6].

## References


1. A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.
2. B. Bonet and H. Geffner. Planning with incomplete information as heuristic search in belief space. In *Proc. AIPS'00*, pp. 52–61, 2000.
3. F. Buccafurri, N. Leone, and P. Rullo. Enhancing disjunctive datalog by constraints. *IEEE TKDE*, 12(5):845–860, 2000.
4. A. Cimatti and M. Roveri. Conformant planning via symbolic model checking. *Journal of Artificial Intelligence Research*, 13:305–338, 2000.
5. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Planning under incomplete knowledge. In: *Proc. CL-2000*, pp. 807–821, LNCS 1861, Springer, 2000.
6. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning: Semantics and complexity. Technical Report INFSYS RR-1843-01-11, Inst. f. Informationssysteme, TU Wien, December 2001.
7. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. A logic programming approach to knowledge-state planning, II: The DLV$^{\mathcal{K}}$ system. Technical Report INFSYS RR-1843-01-12, Inst. f. Informationssysteme, TU Wien, December 2001.
8. T. Eiter, W. Faber, N. Leone, G. Pfeifer, and A. Polleres. Answer set planning under action costs. Manuscript, 2002.
9. E. Ephrati, M. E. Pollack, and M. Mihlstein. A cost-directed planner: Preliminary report. In *Proc. AAAI-96*, pp. 1223–1228. AAAI Press, 1996.
10. W. Faber. Disjunctive Datalog with Strong and Weak Constraints: Representational and Computational Issues. Master's thesis, Inst. f. Informationssysteme, TU Wien, 1998.
11. M. Gelfond and V. Lifschitz. Classical negation in logic programs and disjunctive databases. *New Generation Computing*, 9:365–385, 1991.
12. E. Giunchiglia. Planning as satisfiability with expressive action languages: Concurrency, constraints and nondeterminism. In *Proc. KR 2000*, pp. 657–666. Morgan Kaufmann, 2000.
13. E. Giunchiglia and V. Lifschitz. An action language based on causal explanation: Preliminary report. In *Proc. AAAI '98*, pp. 623–630, AAAI Press, 1998.
14. P. Haslum and H. Geffner. Admissible heuristics for optimal planning. *AIPS'00*, pp. 140–149. AAAI Press, 2000.
15. H. Kautz and J. P. Walser. State-space planning by integer optimization. In *AAAI'99*, pp. 526–533. AAAI Press, 1999.
16. J. Lee and V. Lifschitz. Additive fluents. In *Proc. AAAI 2001 Spring Symposium on Answer Set Programming*, pp. 116–123, AAAI Press, 2001.
17. V. Lifschitz and H. Turner. Representing transition systems by logic programs. *LPNMR'99*, pp. 92–106.
18. V. Lifschitz. Answer set planning. *ICLP'99*, pp. 23–37. MIT Press, 1999.
19. N. McCain and H. Turner. Satisfiability planning with causal theories. *KR'98*, pp. 212–223. Morgan Kaufmann, 1998.
20. A. Nareyek. Beyond the plan-length criterion. In *Local Search for Planning and Scheduling, ECAI 2000 Workshop*, LNCS 2148, pp. 55–78. Springer, 2001.