Exploring Heuristic Action Selection in Agent Programming

Koen V. Hindriks, Catholijn M. Jonker, and Wouter Pasman

EEMCS, Delft University of Technology, Delft, The Netherlands
{k.v.hindriks,c.m.jonker,w.pasman}@tudelft.nl

Abstract. Rational agents programmed in agent programming languages derive their choice of action from their beliefs and goals. One of the main benefits of such programming languages is that they facilitate a high-level and conceptually elegant specification of agent behaviour. Qualitative concepts alone, however, are not sufficient to specify that this behaviour is also nearly optimal, a quality typically also associated with rational agents. Optimality in this context refers to the costs and rewards associated with action execution. It thus would be useful to extend agent programming languages with primitives that allow the specification of near-optimal behaviour. The idea is that quantitative heuristics added to an agent program prune some of the options generated by the qualitative action selection mechanism. In this paper, we explore the expressivity needed to specify such behaviour in the Blocks World domain. The programming constructs that we introduce allow for a high-level specification of such heuristics due to the fact that these can be defined by (re)using the qualitative notions of the basic agent programming language again. We illustrate the use of these constructs by extending a Goal Blocks World agent with various strategies to optimize its behaviour.

1 Introduction

In this paper, we use the well-known Blocks World domain [1] to explore and present evidence for the usefulness of adding expressive programming constructs that allow the specification of utility-based heuristic strategies for action selection to the agent programming language Goal [2]. By means of various examples we illustrate that the new constructs introduced allow for an elegant specification of such strategies. Additionally, we present some experimental results that demonstrate the usefulness of the programming constructs introduced and confirm and slightly extend earlier results available in the literature [1,3,4].

Our objectives are twofold: (i) The first objective is to extend Goal with programming constructs to define a heuristic or utility-based decision capability as an additional action selection mechanism. Such constructs allow the optimization of agent behaviour as well as reduce the amount of nondeterminism present in an agent program. (ii) The second objective is to assess the usefulness of the mechanism by comparing the behaviour of a Goal agent which does not use the mechanism with various instantiations of Goal agents that do use it.

K.V. Hindriks, A. Pokahr, and S. Sardina (Eds.): ProMAS 2008, LNAI 5442, pp. 24–39, 2009.

Although some related work on adding quantitative heuristics based on e.g. resource costs or other decision-theoretic extensions has been done, see e.g. [5,6], as far as we know little research has been done on programming constructs for specifying heuristic action selection in the area of agent programming. [5] allows for defining such decision-theoretic capabilities by means of arbitrary programming languages instead of introducing primitives that reuse the basic concepts of a rational agent programming language as we propose. Moreover, the work extending Golog with decision-theoretic capabilities in e.g. [7] relies on the situation calculus and cannot straightforwardly be incorporated into rational agents that derive their choice of action from their beliefs and goals.

The paper is organized as follows. In Section 2 the Blocks World is briefly introduced and a Goal agent is presented that is able to effectively deal with Blocks World problems. In Section 3 some issues to improve the behaviour of this agent are discussed and a general framework for adding (utility-based) heuristics to an agent programming language is outlined. In Section 4 various heuristics for the Blocks World are presented and it is shown how these can be implemented using the primitives introduced. Section 5 concludes the paper.

2 Designing a Goal Agent for the Blocks World

In this Section, we design a Goal agent that is able to effectively solve Blocks World problems. The Blocks World has been labelled the "Hello World" example for planning [1]. One reason why it is still being used is that it is computationally hard and moreover has some similarities with other, more realistic domains, e.g., it is related to freight operations [1]. Another reason why this domain is still interesting is that it is relatively simple and can be analyzed in detail to gain an understanding of the capabilities needed to deal with it effectively [1,3]. Since, historically, agent programming languages were motivated in part by ideas from reactive planning (see in particular [8,9]), it is interesting to start with this domain for analyzing whether the right features for fine-grained control of action needed to generate near-optimal behaviour are present in agent programming languages.

The Blocks World consist of a finite number of blocks of equal size that are stacked into towers on a table of unlimited size. Each block has a unique name $a, b, c, ...$ representing the fact that different blocks cannot be used interchangeably (which would be the case if only the colour of blocks would be relevant). Some basic axioms of the Blocks World are that no block is on more than one block, no more than one block is on a given block, and every block is either on the table or on another block (see e.g. axiom 4 and 5 in [10], which provides a complete axiomatization of the Blocks World). More realistic versions of this domain have been investigated (e.g., limited table size, varying sizes of blocks; cf. [4]). However, as argued in [1] the elementary Blocks World domain can support systematic experiments and, at least as important for our purposes, allows features relevant to various kinds of reasoning to be abstracted and studied. The Blocks World domain in particular allows for a precise study of various heuristics to ensure that an agent's choice of action generates near-optimal behaviour. Artificial domains such as the Blocks World moreover are hard for general purpose AI systems (e.g. planners), and it is only to be expected that this also holds for programming languages to build rational agents which provide abstract semantic primitives derived from common sense to do so [11]. In this paper some of these difficulties will be explored and discussed. In addition, Blocks World problems allow us to illustrate that programming languages for rational agents provide the expressiveness to construct elegant agent programs that solve such problems, though admittedly the domain is too simple to be convincing by itself.

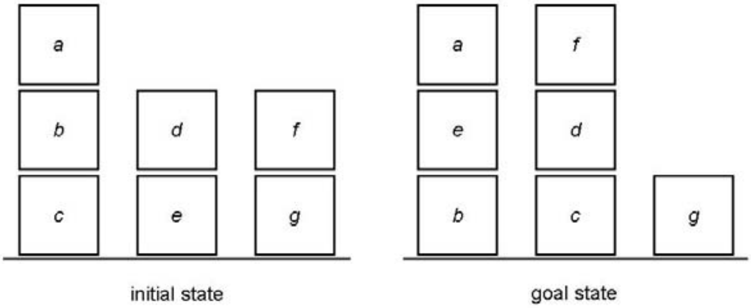

Fig. 1. Example Blocks World problem taken from [1]

The Blocks World planning problem is to transform an initial configuration of towers into a goal configuration, by means of moving one block on the top of a tower onto another tower or to the table; see Figure 1 for an example problem. A block on top of a tower, i.e. without any block on top of it, is said to be *clear*. By definition, there is always room to move a clear block onto the table and therefore the table is also said to be clear. The positioning of towers on the table is irrelevant in a Blocks World problem. The main task of an agent in this domain thus is to restack the blocks on the table according to its given goals. The main choice such an agent faces is which action (moving a block) to select. The performance of a Blocks World agent can be measured by means of the number of moves it needs to turn an initial state or configuration into a goal state. An agent performs optimally if it is not possible to improve on the number of moves it uses to reach a goal state. The problem of finding a minimal number of moves to a goal state is also called the *optimal* Blocks World planning problem. This problem is NP-hard [4], an indication that the problem is not trivial.[1]

Several basic insights help simplify the solving of a Blocks World problem. A block is said to be *in position* if the block in the current state is on top of a block or on the table and should be so according to the goal state, and all blocks (if any) below it are also in position; a block that is not in position is said to be *misplaced*.

[1] It is not within the scope of this paper to discuss the complexity of various proposed Blocks World heuristics for near-optimal planning; see [1,4] on this topic.

In Figure 1 all blocks except block c and g are misplaced. Only misplaced blocks have to be moved in order to solve a problem. A move of block `X` onto another block or onto the table is called *constructive* if in the resulting state block `X` is in position. In the elementary Blocks World with unlimited table size moving a block onto another block should only be done if the move is constructive, i.e., it moves the block in position. A constructive move always decreases the number of misplaced blocks. If in a state no constructive move can be made, we say that the state is in a *deadlock* (see [1] for a detailed explanation). A block is said to be a *self-deadlock* if it is misplaced and above another block which it is also above in the goal state; for example, block a is a self-deadlock in Figure 1. The concept of self-deadlocks, also called singleton deadlocks, is important because on average nearly 40% of the blocks are self-deadlocks [1].

Representing Knowledge and Goals. In the remainder of this paper, we will use Prolog notation to define and specify knowledge and goals. The basic facts and goals to be achieved in the Blocks World can be expressed by means of the predicate `on(X,Y)`. `on(X,Y)` denotes that a block `X` is on `Y`, where `Y` may refer to either another block or the table. We use a predicate `block(X)` to denote that `X` is a block. The predicate `clear(table)` is used to denote that the `table` is clear, i.e. it is always possible to move blocks onto the table. Using the `on` predicate it is possible to formally define a Blocks World planning problem as a pair $\langle I, G\rangle$ where I denotes the initial state and G denotes the goal state. A state is defined as a set of facts of the form `on(X,Y)` that is consistent with the basic axioms of the Blocks World. A state is *complete* if for each block `X` it contains exactly one fact of the form `on(X,Y)`; from now on, we only consider complete states.

In the agent program, all blocks are enumerated to make it clear how many blocks there are. The predicate `above(X,Y)` expresses that block `X` is above *block* `Y` and predicate `tower([X|T])` expresses that the list of blocks `[X|T]` is a tower grounded on the table. We do not require that block `X` is clear, so e.g., `tower([b,c])` holds in the initial state of Figure 1. The Prolog definitions of these concepts are given in the **beliefs** section in Table 1, which is called the *belief base* of the agent. The initial state of Figure 1 is represented in the agent's belief base, which is updated after every action that is performed.

In the **goals** section in Table 1, called the *goal base*, the goal state of Figure 1 is represented. (The clauses for `above(X,Y)` and `tower(T)` are repeated in the goal base. In the current version of GOAL, repetition of such clauses is necessary when they are needed in derivations that use the goal base.) One important difference between the belief and goal base is that individual goals need to be represented as a single conjunction instead of several clauses since it represents a *single* goal. The reason for the distinction is that a goal upon completion, i.e., when it is completely achieved, is removed from the goal base. Achieved goals are removed to satisfy the rationality constraint that an agent does not have any goals it believes to be achieved; the fact that this only happens when the goal has been completely achieved implements a blind commitment strategy.

Table 1. Goal Agent Program for Solving the Blocks World Problem of Figure 1

```
:main stackBuilder
{ % This agent solves the Blocks World problem of Figure 1.
   :beliefs{
     block(a), block(b), block(c), block(d), block(e), block(f), block(g).
     on(a,b), on(b,c), on(c,table), on(d,e), on(e,table), on(f,g), on(g,table).
     clear(table).
     clear(X) :- block(X), not(on(Y,X)).
     above(X,Y) :- on(X,Y), block(Y).
     above(X,Y) :- on(X,Z), above(Z,Y).
     tower([X]) :- on(X,table).
     tower([X,Y|T]) :- on(X,Y), tower([Y|T]).
   }
   :goals{
     block(a), block(b), block(c), block(d), block(e), block(f), block(g),
     on(a,e), on(e,b), on(b,table), on(f,d), on(d,c), on(c,table), on(g,table),
     above(X,Y) :- (on(X,Y), block(Y)),
     above(X,Y) :- (on(X,Z), above(Z,Y)),
     tower([X]) :- (on(X,table)),
     tower([X,Y|S]) :- (on(X,Y), tower([Y|S])).
   }
   :program{
     if bel(tower([Y|T])), a-goal(tower([X,Y|T])) then move(X,Y).
     if a-goal(tower([X|T])) then move(X,table).
   }
   :action-spec{
     move(X,Y) {
        :pre{ clear(X), clear(Y), on(X,Z) }
        :post{ not(on(X,Z)), on(X,Y) }
     }
   }
}
```

Actions. Actions of a Goal agent are specified in the **action-spec** section by means of a STRIPS-like specification of a precondition and add/delete lists, see e.g., Table 1. Add/delete lists are specified here as a single list of literals, where a positive literal denotes an add and a negative literal denotes a delete. Note that the precondition in Table 1 allows moving a block `X` on top of another block `Y` even if block `X` initially already is on top of `Y`. Such redundant actions, however, are never generated given the action rules in the **program** section; therefore it is unnecessary to add conditions preventing such moves to the precondition. For the same reason, it is not necessary to add the precondition `not(X=Y)`.

Goal *agent design.* The basic parts of a Goal agent have now been specified. The belief and goal base together are called the mental state of the agent, typically denoted by $m = \langle \Sigma, \Gamma \rangle$. A mental state needs to satisfy some basic rationality constraints: (i) beliefs need to be consistent, (ii) individual goals need to be consistent, and (iii) individual goals in the goal base are not believed to be the case. Actions are executed in Goal by checking whether the preconditions of an action follow from the agent's beliefs and, if so, by updating the beliefs in line with the action's postcondition after executing it. In addition, if as a result of action execution a goal in the goal base has been completely achieved, then this goal is removed from the goal base.

The **program** section specifies the *strategy for action selection* by means of so-called *action rules*. These rules consist of a mental state condition and an action and specify which actions an agent may consider for execution. The mental state condition determines which actions may be executed. In order to express such conditions a belief operator `bel(...)` and a goal operator `goal(...)` are available, which can be combined using conjunction `,` and prefixed with negation `not`. For example, `bel(block(a))` expresses that the agent believes that `a` is a block whereas `bel(on(a,b)), goal(on(a,e))` expresses that the agent believes it has not yet achieved its goal `on(a,e)` (since it believes `a` to be on top of `b`).

The semantics of the belief operator `bel(`φ`)` is that φ follows from the belief base (i.e. $\Sigma \models \varphi$ where $\models$ denotes the classical first order consequence operator; since we use Prolog, additionally the Closed World Assumption is used in practice). The semantics of the goal operator `goal(`φ`)` is slightly different from that of the belief operator; `goal(`φ`)` holds if φ follows from some goal in the goal base (i.e. $\exists\gamma \in \Gamma : \gamma \models \varphi$).[2] It is useful and necessary to have access to the belief base as well as the goal base of an agent. For example, without either of these operators it is not possible to specify that a block is in position, i.e. that its current position is in correspondence with its positions in the goal state. Using both operators, we can express that block `X` is in position by means of `bel(tower([X|T])), goal(tower([X|T]))` for some tower `T`. We call such a (sub)goal a *goal achieved* and introduce the new operator `goal-a(...)` as an abbreviation to denote this fact, i.e.,

$$\texttt{goal-a}(\varphi) \overset{df}{=} \texttt{bel}(\varphi)\texttt{,}\ \texttt{goal}(\varphi)$$

The notion of an *achievement goal*, i.e., a goal not yet believed to be achieved, can also be defined using the belief and goal operator (cf. [12]). We introduce the new operator `a-goal(...)` to denote such goals as an abbreviation for:

$$\texttt{a-goal}(\varphi) \overset{df}{=} \texttt{bel(not(}\varphi\texttt{)),}\ \texttt{goal}(\varphi)$$

Using the achievement goal operator we can represent the fact that block `X` is not in position in the initial state by `a-goal(tower([X|T]))` for `T` a tower. `a-goal(tower([X|T]))` means that in the goal state block `X` must be on top of the tower `T` but in the current state the agent does not believe that this is already the case; `a-goal(tower([X|T]))` thus expresses that `X` is misplaced. This is an important concept in defining any strategy since only misplaced blocks should be moved to solve a Blocks World problem. The definition of a self-deadlocked block also requires the inspection of both the belief as well as the goal base. The concept of a self-deadlock can be quite naturally defined by `a-goal(tower([X|T])), goal-a(above(X,Y))` where the first conjunct expresses that `X` is misplaced and the second conjunct expresses that `X` is above some

[2] This is different from definitions of the goal operator in previous work [2] where the goal operator was used to denote *achievement goals*, which additionally require that the agent does not believe φ. We need the more basic `goal` operator however to express that a block is in position.

block `Y` in both the current state as well as in the goal state. This concept is just as important for solving Blocks World problems since any self-deadlocked block needs to be moved at least twice to reach the goal state. Moving such a block to the table thus will be a necessary move in every plan.

The two action rules in the **program** section of Table 1 implements a simple strategy for a Blocks World agent. As explained, an action rule consists of a mental state condition φ and an action a. If the condition φ holds, the action a is said to be *enabled*. The first rule generates constructive move options the agent can choose from. The second rule allows a move of block `X` to the table if it is misplaced. The condition of this rule is weaker than the first implying that whenever the first rule is applicable the second is applicable as well, meaning that the actions of these rules are enabled. Then the agent *arbitrarily* chooses an enabled action. Note that this agent will never move a block that is in position.

Summarizing, a Goal agent program consists of four sections: a belief base consisting of the agent's beliefs, a goal base with the agent's goals, a program section defining the agent's action selection strategy, and an action specification section with STRIPS-like action specifications. The Goal Blocks World agent contains a specification of the initial state of the Blocks World problem in its belief base, a specification of the goal state in its goal base, a specification of the `move` action in its action specification section, and two action rules that define its strategy for performing either a constructive move in case such a move brings a block in position, or a move to the table if a block is misplaced.

3 Heuristic Action Selection in Agent Programming

Research in planning has shown that in order to plan effectively and be able to generate near-optimal plans for the Blocks World it must be possible to specify various domain-dependent heuristics [11]. The specification of these heuristics in domain-independent planning systems requires the right concepts to express and implement them. If agent programming languages are to match these capabilities, programming constructs with similar expressive power need to be available to program rational agents that use heuristics to improve performance. We argue that in programming languages for rational agents such programming constructs would be most useful if they allow for the specification of such heuristics in terms of the core concepts of beliefs and goals present in these languages.

In this Section we introduce a generic extension of the Goal agent programming language that can be incorporated into other agent languages based on concepts of belief and goal, and add a capability for specifying heuristic selection strategies by means of utility functions. We first briefly introduce the basic concepts needed and discuss the semantics of the extension of Goal with a utility-based action selection mechanism. Then we introduce a programming construct for specifying utility values. In Section 4 we show that the programming constructs we introduce allow for an elegant specification of behaviour that shows improved performance compared with a Goal agent that does not make use of the utility-based selection mechanism.

3.1 Associating Utility Values with Action Execution

The idea is to associate a quantitative number with the execution of an action a in a state m, i.e., to associate a real valued number $U(m, a, m')$ with executing a in state m resulting in a new state m'. A number associated with an action in this way can be perceived of in two different ways. One perspective, the more principled view on what this number represents, is to suggest that the number is a utility value that represents how much value is to be gained from executing the action. It is standard to further decompose such a utility value into two components, a *cost component* that is associated with taking an action in the starting state and a *reward component* that associates a reward with getting to the resulting state (cf. [13]). Alternatively, such a number can be perceived of as a heuristic that only provides an estimation of e.g. the costs of executing an action. Since these different views do not conflict, and in practice it is very intuitive to use concepts such as costs and rewards, in the remainder we will freely use either terminology.

Formally, a utility function can be introduced which is defined in terms of costs and rewards by: $U(m, a, m') = R(m') - C(m, a)$. Here, the *reward function* R should be thought of as representing the *utility* of being in state m'. For example, an agent gains more utility for getting to a state with more blocks in position than to a state with less blocks in position. Likewise, the *cost function* C represents the costs associated with the resources spent. However, a cost function can also be used to indicate that performing an action is a good thing.

3.2 Semantics

Agent programming languages in general, and Goal in particular, quite naturally lead to writing programs that are underspecified (i.e. such programs do not determine a unique action to perform next and thus may underspecify the actual behaviour of an agent).[3] The operational semantics of these languages leaves room for various alternatives as to how to implement the action selection mechanism of an agent. One of the benefits of underspecification is that it facilitates and simplifies the design and programming of an agent, but it may also give rise to suboptimal behaviour (due to ad hoc suboptimal choices). The basic idea now is to introduce another, utility-based mechanism for action selection on top of the qualitative selection mechanism already present in Goal that can be used to further limit the number of choices.

Ideally an agent optimizes the sum of all utility gains over an entire execution run. The set of such runs of an agent with which we would like to associate utility values is given by the qualitative action selection mechanism. A run can

[3] The language Goal does not have constructs to specify complex plans such as sequential composition that could be used to further reduce the choices left open by an agent at any particular moment. The action rules in a Goal program thus naturally lead to more underspecification than is typical of other agent programming languages. The only construct available in Goal to reduce this type of underspecification is the module-construct, see [14].

be formally specified as an infinite sequence of computation steps. Very briefly, a computation step written as $m \xrightarrow{a} m'$ denotes that action a can be performed in state m (i.e. action a is enabled: the precondition of a holds in state m and the condition of the corresponding action rule for a also holds) and results in state m'. A *run* r then can be defined as an infinite sequence $m_0, a_0, m_1, a_1, m_2, \ldots$ such that $m_i \xrightarrow{a_i} m_{i+1}$ (for details, we refer the interested reader to [2]). The set of all such runs is denoted by R_A for agent program A.

The main idea is to associate a utility value with each possible run of an agent and to actually execute that run which maximizes utility. In this setup, an agent first (pre)selects possible actions which it may execute in each state using its action selection mechanism based on qualitative action rules. In other words, action rules define the search space in which the agent needs to find an optimal run. The benefit is that this search space typically is significantly reduced compared to the search space induced by the set of all enabled actions in a state, i.e. actions whose preconditions hold.

Given a utility function U it is easy to extend this function to a run. We use m_i^r to denote the ith mental state in run r and similarly a_i^r denotes the ith action in run r. A utility value can be associated with a run r then as follows:

$$U_\delta(r) = \sum_{i=0}^{\infty} \delta^i \cdot U(m_i^r, a_i^r, m_{i+1}^r)$$

where δ is a discount factor in the range $\langle 0, 1]$, intuitively accounting for the fact that utility realized now is more valuable than utility in the future. The meaning of a Goal agent A that uses the utility-based action selection mechanism on top of the qualitative one then can be defined as the set of runs r that maximize the associated utility $U(r)$, i.e., the meaning of a utility-based Goal agent is defined by:

$$U_A = \max_{U(r)}\{r \mid r \in R_A\}$$

The semantics of a utility-based Goal agent as defined above requires *infinite look-ahead.* That is, to select an action in any state requires the agent to compute the utility of all possible runs before performing that action to ensure utility is maximized over the complete computation. Computationally, such a requirement is not feasible and, therefore, we associate a *finite horizon* h with the computation of a utility. In case $h = 0$, the agent does not perform any look ahead at all. For $h > 0$, an agent would require a lookahead facility before taking action, of depth h. Formally, this finite horizon constraint can be defined on an arbitrary computation r at time point i by:

$$U(r, i, h) = \sum_{j=i}^{i+h-1} \delta^{j-i} \cdot U(m_j^r, a_j^r, m_{j+1}^r)$$

Here, $U(r, i, h)$ denotes the (discounted) utility associated with the actions performed from time point i to $i + h$. Using this definition, the meaning U_A^h of a utility-based Goal agent with a finite horizon h is defined by $U_A^h = \sigma_A^h(\infty)$, where σ_A^h is defined by the following inductive definition:

$$
\begin{aligned}
\sigma_A^h(-1) &= R_A, \\
\sigma_A^h(i) &= \max_{U(r,i,h)} \{r \,|\, r \in \sigma_A^h(i-1)\} \text{ if } i \geq 0. \\
\sigma_A^h(\infty) &= \bigcap_{i=0}^{\infty} \sigma_A^h(i).
\end{aligned}
$$

The operator $\sigma_A^h(i)$ for $i \in \mathbb{N}$ selects those runs that have highest utility over the next h steps from time point i on, where the runs inspected are those runs that have survived the same selection using a horizon h at earlier time points $j < i$. At time point $i = 0$ the latter set coincides with the set of all possible runs R_A of agent A. An inductive definition is needed here to reflect the fact that an agent performs a look-ahead again after each step it performs. The limit $\sigma_A^h(\infty)$ defines the new meaning of a Goal agent that uses a look-ahead h to filter runs.

The following proposition partly justifies the definition of σ.

Proposition 1

$$R_A = U_A^0 \tag{1}$$

$$U_A = \sigma_A^\infty(0) \tag{2}$$

$$U_A \supseteq U_A^\infty, \; \textit{i.e.,} \; \max_{U(r)}\{r \mid r \in R_A\} \supseteq \sigma_A^\infty(\infty) \tag{3}$$

The first item of Proposition 1, $R_A = U_A^0$, expresses that the utility-based semantics with a horizon $h = 0$, i.e. no look-ahead, coincides with that of the original semantics that does not take utility into account. The second item $U_A = \sigma_A^\infty(0)$ shows that infinite look-ahead can be defined by means of the σ-operator; simply do infinite look-ahead at time point 0. It is easy to see that $U_A^\infty \subseteq U_A$, since $U_A^\infty \subseteq \sigma_A^\infty(0)$. The fact that U_A is not the same as $U_A^\infty = \sigma_A^\infty(\infty)$ is due to the fact that σ_A^h defines a step by step process and evaluates maximum continuations in each state and does not just once evaluate a global property of a run. Proposition 1 thus shows that in the limit case the semantics defined by the σ-operator approximates that of U_A. The advantage of the σ_A^h-operator over the infinite-lookahead U_A is that for finite horizons h it can implemented. Finally, we do not necessarily have that $U_A^h \subseteq U_A$ for a finite h, since σ_A^h may select runs that have high utility on initial prefixes of length h of the run but over the complete run do worse than other runs with low utility on initial prefixes of length h.

3.3 Specifying Quantitative Utility Values

In order to incorporate the assignment of quantitative values to transitions of a Goal program, such programs are extended with a new **utility** section and the following notation is introduced for representating utility:

```
value(<initial-state-cond>, <action-descr>, <successor-state-cond>) = <utility-expr>
```

The `initial-state-cond` as well as the `successor-state-cond` refer to arbitrary mental state conditions, i.e., conditions that are combinations of `goal(...)` and bel(...) operators. In addition, the constant true - which holds in any mental state - may be used here as well. The action-descr part refers to any action description that is allowed in GOAL, e.g., in the Blocks World move(X,Y). Variables are allowed in both mental state conditions used to characterize the initial or resulting state, as well as in the action description. The same holds for the utility-expr part, which denotes a numerical expression which may involve basic arithmetic operators such as addition and multiplication. The action description part may also be filled with a special don't care label any.

In the **utility** section of a GOAL program, multiple lines of value statements are allowed that apply to the same transition. In case multiple value statements apply to the same transition the multiple values assigned to that transition are added together by taking the sum of the values. As a simple example, the statements value(true,move(X,table),true)=1 and value(bel(on(X,Y)),any,true)=2 are both applicable to a transition that starts in a state where bel(on(a,b)) holds and in which action move(a,table) is taken, and therefore the values 1 and 2 need to be added to give a total value of 3. Using the value construct we can define various useful abbreviations for reward and cost components as follows:

```
cost(<initial-state-cond>, <action>) ≝ -1·value(<initial-state-cond>, <action>, true)
reward(<successor-state-cond>) ≝ value(true, any, <successor-state-cond>)
```

Note that according to these definitions both costs and rewards are conditional on the beliefs as well as the goals of an agent.

For practical reasons, it is useful to introduce a case statement to define a complex value function based on case distinctions. Inside a case statement conditional expressions of the form <state-cond>:cost(<action-descr>)=<utility-expr> and <state-cond>:reward=<utility-expr> are allowed. By using a case-statement, costs and/or rewards are assigned to a transition using the *first* case that applies, i.e., that value is returned associated with the first condition <state-cond> that holds (assuming, of course that an action description, if present, matches as well). Various examples of the use of this statement are provided below.

In the extension of GOAL quantative values are assigned only to actions that an agent has preselected given its current goals. This reflects the fact that qualitative goals have priority over any quantitative preferences. That is, the first priority of a GOAL agent is to achieve its qualitative goals, whereas its second priority then becomes to do this such that utility is maximized.

4 Heuristic Action Selection in the Blocks World

As explained above, the GOAL Blocks World agent never moves a block that is in position. The agent will only move a misplaced block to the table or move a block onto another block. Note that the agent will only move a block X onto another block Y if this move puts X in position, and such a move thus is constructive. Also note that if a block can be moved onto another block the second action rule of the agent also allows to move this block to the table. In almost all Blocks World states multiple actions are feasible and in line with the semantics of GOAL an action then is selected randomly. The semantics thus allows for various strategies of action selection and does not enforce any of these strategies.

A number of alternative heuristics or strategies have been proposed in the literature [1,3,4]. We explore several of these to illustrate the use of utility values to guide the action selection mechanism of an agent. One of the most straightforward strategies for solving a Blocks World problem is to first unstack all (misplaced) blocks and then to move all blocks in position. This strategy has been called the Unstack-Stack (US) strategy [1]. It is clear that this strategy is compatible with the Goal agent program presented in Table 1. Note that this strategy will only worsen the behaviour of the agent by never making a constructive move during the unstack phase even if such moves are available. We have implemented and experimented with it mainly for reasons of comparison. The following code needs to be added to the utility section:

```
case{
  bel(Y=table): cost(move(X,Y)) = 1.                    % unstack has priority
  true:         cost(move(X,Y)) = 2.                    % otherwise
}
```

USG Heuristic

A first idea to improve the behaviour of the agent is to give priority to *constructive* moves over other moves. The reason that this may improve behaviour is simple: the move has to be made anyway, brings the current state closer to the goal state, and may make it possible to perform another constructive move next. Using the `cost` construct to assign costs to actions we have to make sure that a constructive move always has an associated cost less than that for other types of moves. Since as we noted above, any block that satisfies `bel(tower([X|T])), a-goal(tower([X,Y|T]))` can be constructively moved, the cost function can be defined as follows:

```
case{
  bel(tower([Y|T]), a-goal(tower([X,Y|T])): cost(move(X,Y)) = 1.  % a constructive move
  true:                                     cost(move(X,Y)) = 2.  % otherwise
}
```

GN1G Heuristic

A second heuristic to get closer to near-optimal behaviour is to prefer moving a block that is *self-deadlocked* over moving other blocks when no constructive move is available. As explained above, a self-deadlocked block is a misplaced block above a block it has to be above in the goal state as well. As a result, such a block has to be moved twice (once to the table, and once in position) and it makes sense to do this first when no constructive move is available.[4] The addition of this heuristic to the program requires the more complex conceptual condition that defines a self-deadlock identified above. Here we can slightly simplify, however, because costs of an action are only computed if the action is enabled, i.e. the corresponding action rule condition is satisfied. This means that a block `X` in an enabled action `move(X,Y)` is misplaced and we do not need to repeat it; the part

[4] It does not make any difference whether a constructive or self-deadlocked move is made first; we follow [3,1,4] in preferring to make a constructive move here.

of the definition still required then is `goal-a(above(X,Z))`. For the same reason we also do not need to check whether the block to be moved is clear.

```
case{
  bel(tower([Y|T]), a-goal(tower([X,Y|T]))): cost(move(X,Y)) = 1. % a constructive move
  goal-a(above(X,Z)):                         cost(move(X,Y)) = 2. % X is a self-deadlock
  true:                                       cost(move(X,Y)) = 3. % otherwise
}
```

SDG Heuristic

Although the heuristic costs associated with move actions above is quite natural, not quite the same behaviour but similar performance could have been achieved quite elegantly also by using the reward function instead of the cost function by making use of the counting operator `#`.

```
reward(true) = #T^goal-a(tower([X|T])-#T^Y^[a-goal(tower([X|T])),goal-a(above(X,Y))]
```

The first term in the utility expression `#T^goal-a(tower([X|T]))` counts the number of blocks in position in a state, whereas the second term

`#T^Y^[a-goal(tower([X|T])),goal-a(above(X,Y))]`

counts the number of self-deadlocks in a state. Also note the use of the abstraction operators `T^` and `Y^` which, as in Prolog, existentially quantify variables `T` (∃T) and `Y` (∃Y) to ensure that we do not count variation over these variables. In the Blocks World domain the abstraction over `T` is not strictly necessary since in any state a block can be present at most in one tower, but the abstraction over `Y` is required since a block may be above multiple other blocks in both the belief as well as goal state. Rewards increase by either increasing the number of blocks in position or by decreasing the number of self-deadlocks in a state. The heuristic values associated by the `reward` function with performing a constructive move or breaking a self-deadlock are identical. This is different from the `cost` function introduced earlier which always prefers to perform a constructive move first if possible. As noted above, however, since a self-deadlock has to be moved twice in any optimal plan anyway this preference does not result in behaviour that is closer to optimal behaviour.

A third heuristic is adapted from a proposal in [3], and focuses on those cases where neither a constructive nor any self-deadlock move can be made. In that case some block has to be moved to the table, and we pick the block on the tower that has the lowest number of blocks that are neither in position nor self-deadlocked. This number is called the *deficiency* of the tower and is added as an additional third case to the previous cost function defined above.

```
case{
  bel(tower([Y|T]), a-goal(tower([X,Y|T]))): cost(move(X,Y))=1. % a constructive move
  goal-a(above(X,Z)):                         cost(move(X,Y))=2. % X is a self-deadlock
  bel(tower([X|T]),length([X|T],H),last(T,B)), goal-a(on(B,table)): % compute deficiency
         cost(move(X,Y)) = H-#[bel(member(Y,T)), goal-a(tower[Y|U]))]
           -#Z^[bel(member(Y,T)), a-goal(tower([Y|U])), goal-a(above(Y,Z))].
  true:                                 cost(move(X,Y)) = #bel(block(X))+1. % otherwise.
}
```

DSG Heuristic

Results. Although our main aim has been to introduce expressive programming primitives for defining (utility-based) heuristics, it is interesting to briefly discuss the results of running the heuristics discussed. The various heuristics defined above were implemented in our prototype GOAL implementation. This prototype is implemented in Java and SWI-prolog. The default goal behaviour (RSG), which selects one of the applicable actions at random instead of picking the one that has maximum utility, was also measured.

To generate random start and end states in the blocks world, the BWSTATES algorithm of [1,15] was used, whereas the BWOPT algorithm of [1,15] was used to determine the optimal plan length. To run the experiments, 100 problems were generated, each consisting of a random start and end state, for worlds of size 10 up to 120 blocks with step size 10. Each problem was solved using the various utility heuristics. The agents used a horizon of 1. The performance is then computed as the number of steps it took to solve that problem divided by the optimal plan length.

Figure 4 shows the average performance as a function of the number of blocks. The standard deviations on the performance are all in the order of 0.04 and have been left out to improve readability of the Figure. The dashed lines show the results that were found by Slaney [1], the labels ending with G refer to heuristics defined in GOAL.

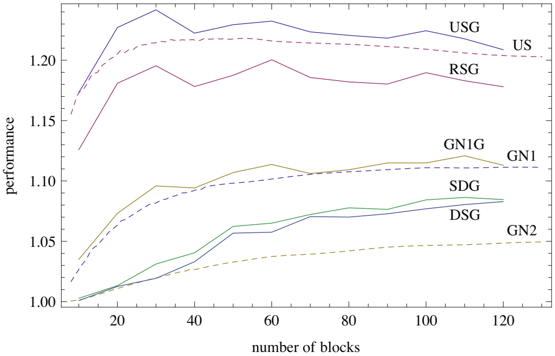

Fig. 2. Performance results

Given the relatively large standard deviations on our measurements, the USG and GN1G heuristics match Slaney's results for the US and GN1 heuristics. The various utility functions USG, GN1G, SDG and DSG were claimed to be a set of incremental improvements on the basic heuristic USG, which is confirmed by the performance results. At 120 blocks and with respect to the optimal performance of 1.0, the GN1G performs twice as good as USG, and the SDG and DSG adds another 37% to the performance of GN1G. The standard goal RSG also performs as expected: better than the USG algorithm but worse than GN1G as it still can do non-constructive moves when a constructive move is possible. The DSG heuristic is only a marginal improvement over the SDG heuristic. Even though the improvement is small, our results confirm the claim in [3] that the deficiency heuristic optimizes performance and adds some new evidence that this improvement is consistent at least for worlds of up to size 120 blocks.

5 Conclusion

We have introduced new programming constructs that add expressiveness to the Goal programming language and allows to specify utility-based heuristics using high-level concepts such as beliefs and goals. The construct can be added to any programming language that is based on these agent concepts. Thus, high-level agent programming concepts are combined naturally with a utility-based action selection capability.

Similar ideas have been proposed in [5,7]. [7] discusses an extension of Golog with a decision-theoretic component called DTGolog. Since Golog is an extension of the situation calculus there are many differences between our work and that of [7]; one of the more important ones is that heuristics in the programming language Goal can be defined using the concepts of belief and goal, which gives additional expressive power not present in [7]. [5] extends the AgentSpeak(L) language with a decision-theoretic capability but allows the use of arbitrary programming languages to do so instead of adding a new programming construct to the language itself. Finally, it would be interesting to compare our work with the specification of heuristics in planners such as TLPlan [11]. TLPlan allows for specifying heuristics using temporal logic to guide search for planning from scratch. The extension of Goal in contrast assumes this search space has been predefined by means of action rules, which may be further pruned by means of the utility-based action selection capability introduced in this paper. It remains for future work to compare the expressiveness of both approaches.

Several example heuristics and related results were presented which show that the addition of a construct to specify quantitative heuristics for action selection may significantly improve performance which cannot be achieved as elegantly without it or not at all.

The action selection mechanism based on quantitative heuristics we introduced in this paper assumes a look-ahead mechanism that is not always applicable in dynamic environments. As argued in [16], however, it is useful to combine look-ahead or planning with action execution. By combining the concept of a module introduced in [14] and the look-ahead action selection mechanism discussed here we can achieve a similar “local” planning mechanism in Goal that is performed within the context of such a module. We leave discussion of this combination for future work.

The extension of Goal proposed here does not allow the use of probabilistic concepts which are available in decision-theoretic approaches. Future work could be to include these as well, but a proper integration of probabilistic concepts into Goal would require an extension of the basic language as well to be able to execute actions with probabilistic effects. Another interesting idea is to allow agents to *learn* the priorities they should associate with actions. For example, reinforcement learning techniques could be deployed within GOAL to learn optimal policies.

References

1. Slaney, J., Thiébaux, S.: Blocks World revisited. Artificial Intelligence 125, 119–153 (2001)
2. de Boer, F., Hindriks, K., van der Hoek, W., Meyer, J.J.: A Verification Framework for Agent Programming with Declarative Goals. Journal of Applied Logic 5(2), 277–302 (2007)
3. Romero, A.G., Alquézar, R.: To block or not to block? In: Lemaître, C., Reyes, C.A., González, J.A. (eds.) IBERAMIA 2004. LNCS, vol. 3315, pp. 134–143. Springer, Heidelberg (2004)
4. Gupta, N., Nau, D.S.: On the Complexity of Blocks-World Planning. Artificial Intelligence 56(2-3), 223–254 (1992)
5. Bordini, R., Bazzan, A., Jannone, R., Basso, D., Vicari, R., Lesser, V.: AgentSpeak(XL): Efficient Intention Selection in BDI agents via Decision-Theoretic Task Scheduling. In: Proc. of the 1st Int. Joint Conference on Autonomous Agents and Multi-Agent Systems (AAMAS 2002), pp. 1294–1302 (2002)
6. Thangarajah, J., Padgham, L., Winikoff, M.: Detecting and avoiding interference between goals in intelligent agents. In: Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003) (2003)
7. Boutilier, C., Reiter, R., Soutchanski, M., Thrun, S.: Decision-Theoretic, High-level Agent Programming in the Situation Calculus. In: Proc. of the 17th National Conference on Artificial Intelligence (AAAI 2000), pp. 355–362 (2000)
8. Ingrand, F., Georgeff, M., Rao, A.: An architecture for real-time reasoning and system control. IEEE Expert 7(6) (1992)
9. Rao, A.S.: AgentSpeak(L): BDI Agents Speak Out in a Logical Computable Language. In: Perram, J., Van de Velde, W. (eds.) MAAMAW 1996. LNCS, vol. 1038, pp. 42–55. Springer, Heidelberg (1996)
10. Cook, S., Liu, Y.: A Complete Axiomatization for Blocks World. Journal of Logic and Computation 13(4), 581–594 (2002)
11. Bacchus, F., Kabanza, F.: Using Temporal Logics to Express Search Control Knowledge for Planning. Artificial Intelligence 116(1-2), 123–191 (2000)
12. Cohen, P.R., Levesque, H.J.: Intention Is Choice with Commitment. Artificial Intelligence 42, 213–261 (1990)
13. Boutilier, C., Dean, T., Hanks, S.: Decision-theoretic planning: Structural assumptions and computational leverage. Journal of AI Research 11, 1–94 (1999)
14. Hindriks, K.: Modules as policy-based intentions: Modular agent programming in goal. In: Dastani, M., El Fallah Seghrouchni, A., Ricci, A., Winikoff, M. (eds.) ProMAS 2007. LNCS, vol. 4908, pp. 156–171. Springer, Heidelberg (2008)
15. `http://users.rsise.anu.edu.au/~jks/bwstates.html` (January 2008)
16. de Giacomo, G., Levesque, H.J.: An incremental interpreter for high-level programs with sensing. Technical report, Department of Computer Science, University of Toronto (1998)