Relaxing Regression for a Heuristic GOLOG

Michelle L. Blom and Adrian R. Pearce[1]

Abstract. GOLOG is an agent programming language designed to represent complex actions and procedures in the situation calculus. In this paper we apply relaxation-based heuristics – often used in classical planning – to find (near) optimal executions of a GOLOG program. We present and utilise a theory of relaxed regression for the approximate interpretation of a GOLOG program. This relaxed interpreter is used to heuristically evaluate the available choices in the search for a program execution. We compare the performance of our heuristic interpreter (in terms of the quality of executions found) with a traditional depth-first search interpreter and one guided by a greedy heuristic without a look-ahead on three domains: spacecraft control, mine planning, and task scheduling.

1 Introduction

GOLOG [8] is an agent programming language grounded in the situation calculus [11]. The aim of a GOLOG interpreter is to find an execution of a GOLOG program – a sequence of actions for an agent to perform. To increase the applicability of GOLOG as a tool for programming agents, we do not want just any execution, but one that satisfies desirable properties (eg. minimising resource utilisation). In this paper we consider the application of relaxation-based heuristics often used in classical planning to find an optimal or near optimal execution of a GOLOG program.

Relaxation-based heuristics have been used in classical planning to find optimal or near-optimal plans (w.r.t plan length) with great success [2, 7]. The available choices in the search for a plan are evaluated by solving a relaxed version of the planning problem. This relaxed problem typically ignores the negative effects of actions (such as the depletion of a resource). For each choice, a relaxed plan is formed representing what an actual (legal) plan might look like if that choice is made. The properties of this relaxed plan (eg. length) are used to select the most promising choice for further exploration.

We consider the relaxation-based heuristics explored in [2, 7] and adapt them for use in the interpretation of GOLOG programs. In doing so we develop a heuristic GOLOG interpreter. We assume that we have a numeric evaluation function f_e that assesses the quality of an execution, and that the aim of our heuristic interpreter is to find an execution that minimises this function. This utility function can be derived from factors such as execution cost, resource utilisation, and the achievement of weighted goals or desires.

A GOLOG interpreter is faced with choices each time it comes across a non-deterministic construct (eg. a branch). A typical Prolog implementation of a standard interpreter selects the first available choice (eg. the first program in a branch). In contrast, our heuristic interpreter evaluates each choice by: making the choice, and then finding a relaxed execution of the remaining program. We develop a relaxed GOLOG interpreter for this purpose, relying on a relaxed form of (situation calculus) regression to optimistically determine whether an action is possible in any given situation. This relaxed execution is designed to represent what desirable future may exist by making this particular choice in the interpretation of the program. The evaluation function f_e applied to each relaxed execution assigns a heuristic value to the corresponding choice. Our interpreter opts to pursue the choice with the best (smallest) evaluation.

To assess the performance of our heuristic interpreter, we conduct experiments on three domains. We use an example presented in [13] in which a spacecraft is required to make scientific observations of celestial targets during a limited time window (while maximising scientific return). We additionally design an iron ore mining agent that collects ore to maintain a stockpile of a desired quality, and a task scheduling agent that assigns tasks to workers while maximising worker utilisation. Given a GOLOG program controlling these agents, we show that our interpreter is able to find executions that are higher in quality (eg. make more observations in the example of [13]) than those found by both a traditional interpreter and one guided by a greedy heuristic without a look-ahead.

In the next section we review GOLOG, the situation calculus, and a traditional GOLOG interpreter. We then describe regression, and how it can be altered to operate in a relaxed manner. Following this, we develop our relaxed and heuristic GOLOG interpreters. We present these interpreters within the context of offline planning. We conclude by considering the use of our interpreter in an online setting, and with an analysis of related work. To the best of our knowledge, this is the first work to consider the application of relaxation-based heuristics in the GOLOG interpretation process, without requiring compilation of the problem for use with a classical planner.

2 GOLOG and the Situation Calculus

The situation calculus [11] is a formalism for reasoning about dynamically changing worlds. Key elements in the situation calculus are: *actions*, which change the state of a world; *situations*, which describe a sequence of actions applied to an initial state; and *fluents*, which represent properties of a world. A fluent is *relational* if it represents a true or false value, and *functional* if it evaluates to an alternative value (such as a number or a coordinate).

We describe a world in the situation calculus with a set of: precondition, successor state, initial state and domain independent axioms. These axioms, in conjunction with a set of unique names axioms for actions, form a basic action theory $\mathcal{D}$.

A precondition axiom for an action $A(\vec{x})$ describes the conditions (Π_A) that must be satisfied before $A(\vec{x})$ can be performed in a given situation s, and assumes the form:

$$Poss(A(\vec{x}), s) \equiv \Pi_A(\vec{x}, s)$$

[1] NICTA VRL, Department of Computer Science and Software Engineering, The University of Melbourne, Australia.

A successor state axiom for a fluent defines how its value changes. A relational fluent F has a successor state axiom of the form:

$$F(\vec{x}, do(a, s)) \equiv \Phi_F(\vec{x}, a, s) \equiv$$
$$\gamma_F^+(\vec{x}, a, s) \vee F(\vec{x}, s) \wedge \neg\gamma_F^-(\vec{x}, a, s)$$

where: $\gamma_F^+(\vec{x}, a, s)$ denotes how F is made true by action a in situation s; $\gamma_F^-(\vec{x}, a, s)$ denotes how F can be made false by a in s; and $do(a, s)$ denotes the situation that results after action a is performed in situation s.

A successor state axiom for a functional fluent G has the form:

$$G(\vec{x}, do(a, s)) = y \equiv \Psi_G(\vec{x}, y, a, s) \equiv \delta_G(\vec{x}, y, a, s) \vee$$
$$G(\vec{x}, s) = y \wedge \neg(\exists y').\delta_G(\vec{x}, y', a, s)$$

where: $\delta_G(\vec{x}, y, a, s)$ denotes the conditions that must hold for G to evaluate to y in $do(a, s)$; $G(\vec{x}, s) = y$ denotes whether G evaluates to y in the situation s; and $\neg(\exists y').\delta_G(\vec{x}, y', a, s)$ indicates that the action a did not change this evaluation to another value y'.

a	primitive action
$\phi?$	test if ϕ holds
$\delta_1 : \delta_2$	execute program δ_1 then δ_2
$\delta_1 \| \delta_2$	choose to execute δ_1 or δ_2
$\pi\ v.\delta$	choice of argument v in δ
δ^*	execute δ 0 or more times
if ϕ **then** δ_1 **else** δ_2	synchronised conditional
while ϕ **do** δ **end**	synchronised loop
proc $P(\vec{v})$ δ **end**	procedure

Table 1. GOLOG constructs.

The constructs present in the GOLOG programming language are described in [4] and are listed in Table 1. These elements are combined to produce programs. In Example 1, a running example used throughout the paper, a GOLOG program is presented that controls the operation of a spacecraft charged with observing celestial targets.

Example 1. *Consider a spacecraft designed to observe celestial targets within a limited time window [13]. Celestial targets are located in clusters within a 3D space. The spacecraft has an instrument capable of making observations (which must be calibrated prior to its first observation). Making an observation requires turning the instrument to a target (consuming an amount of fuel and time dependent on the magnitude of the turn), and taking an image (consuming power). Fuel is limited, while power is renewable at a constant rate*[2].

A plan for the spacecraft is determined by finding an execution of the GOLOG *program* SPACE.

proc SPACE
switchOn : $\pi\ t_1$. *[caltrgt(t_1)?* : *turnto(t_1)* : *calibrate(t_1)]* :
while $\exists t$. *(trgt(t)* $\wedge \neg$ *obsd(t))* $\wedge$ *timeleft* ***do***
(π *tg. [(trgt(tg)* $\wedge \neg$ *obsd(tg))?* :
turnto(tg) : *takeimage(tg)]* | *wait)*
end *:*
switchOff
end

where: trgt(t) denotes that t is a celestial target; obsd(t) denotes that target t has been observed; timeleft denotes that there is $\geq$ 1 unit of time left; caltrgt(t) denotes that t is a calibration target; and switchon, calibrate(t), switchoff, turnto(t), wait, and takeimage(t), are primitive actions. The craft selects a target to observe until the available time has elapsed. If there is time left but it is not possible to observe a target (due to limited resources) the craft waits for one unit of time before checking whether a new target can be observed.

[2] This is a simplification of the example in [13] in which the power renewal rate is dependent on the orientation of the craft with respect to the sun.

2.1 A Traditional GOLOG Interpreter

We now describe a traditional depth-first search GOLOG interpreter (using the transition semantics of [4]) that we shall refer to throughout the remainder of this paper. These semantics define when an agent can transition from one configuration (δ, s) to another (δ', s') by performing one step in the program δ in situation s, resulting in the situation s' and program δ' left to perform.

For each GOLOG construct, δ_f, a *trans* axiom defines which configurations can be transitioned to from (δ_f, s) by performing a single step of δ_f in situation s. A configuration (δ_f, s) is final (*final*(δ_f, s)) if no steps remain to be performed in δ_f. A selection of *trans* axioms are shown below. We find a legal execution of a program δ in situation s_0 by repeatedly applying these *trans* axioms – starting with the configuration (δ, s_0) – until we reach a final configuration. In the following, $\phi[s]$ denotes whether ϕ holds in situation s, and δ_x^v denotes that v has been replaced with x in δ.

$$trans(a, s, \delta', s') \equiv Poss(a, s) \wedge \delta' = nil \wedge s' = do(a, s)$$
$$trans(\delta_1|\delta_2, s, \delta', s') \equiv trans(\delta_1, s, \delta', s') \vee trans(\delta_2, s, \delta', s')$$
$$trans(\mathbf{while}\ \phi\ \mathbf{do}\ \delta, s, \delta', s') \equiv \exists\gamma.\delta' = (\gamma : \mathbf{while}\ \phi\ \mathbf{do}\ \delta) \wedge \phi[s] \wedge trans(\delta, s, \gamma, s')$$
$$trans(\pi v.\delta, s, \delta', s') \equiv \exists x.trans(\delta_x^v, s, \delta', s')$$

We do not describe the final axioms for each δ_f but refer the reader to [4]. To determine $\phi[s]$, we use regression.

3 Regression and Relaxed Regression

Regression takes a query involving a situation s and transforms it into an equivalent query that is uniform in the initial situation s_0. A formula W is uniform in the initial situation if the only situation term mentioned in W is s_0 (and several other conditions are satisfied, as described in [10]). This query can then be proven with respect to a set of initial state axioms with a first order theorem prover.

When finding an execution of a GOLOG program, we use regression to determine: if an action a is possible in a situation s ($Poss(a, s)$); and whether a condition within a $test$, $while$, or $if \ldots else$ construct holds.

The regression operator R is defined by a series of rules. These rules look at the structure of a formula and tell us how it can be shifted one step closer to being uniform in the initial situation. For a definition of regressable formulae – situation calculus formulae that can be regressed – refer to [10].

To demonstrate, consider a regressable relational fluent $F(\vec{t}, do(\alpha, s))$, and let $F(\vec{x}, do(a, \sigma)) \equiv \Phi_F(\vec{x}, a, \sigma)$ denote the successor state axiom for the fluent F (defined in Section 2). The regression rule for this fluent is [10]: $R[F(\vec{t}, do(\alpha, s))] = R[\Phi_F(\vec{t}, \alpha, s)]$.

We would like to estimate whether such formulae hold without expending the effort required by an exact regression. This estimation can be used to find an approximate execution of a GOLOG program. To perform this estimation, we define a *relaxed regression* operator RR. To do so we combine the concept of relaxation used in classical planning [2, 7] with the regression rules of standard regression.

In classical planning relaxation is typically conducted by ignoring the negative effects of actions (the effects that cause certain propositions – aspects of the planner's environment – to be false). Propositions that are true in the initial state, or that become true by performing an action, persist – they remain true irrespective of any action performed that may make them false.

We employ this idea in our formalisation of relaxed regression. A relational fluent F holds in a situation $do(\alpha, s)$, for example, if

it holds in situation s or the action α has made it hold. We do not consider whether α causes the fluent to be false. We now define the required properties of RR.

Definition 1. *Given a regressable formula W, and a basic action theory $\mathcal{D}$, RR is defined such that:*

1. *$RR[W]$ is uniform in the initial situation s_0; and,*
2. *If $\mathcal{D} \models W$ then $\mathcal{D}_{s_0} \cup \mathcal{D}_{una} \models RR[W]$,*

where $\mathcal{D}_{s_0}$ is the set of initial state axioms of $\mathcal{D}$, and $\mathcal{D}_{una}$ is the set of unique names axioms of $\mathcal{D}$.

The second property states that if a regressable formula W holds given $\mathcal{D}$, then the transformation of W under relaxed regression holds given $\mathcal{D}_{s_0} \cup \mathcal{D}_{una}$. The converse will not necessarily be the case. If $RR[W]$ holds we infer that W *might* hold, not that it *does* hold. If $RR[W]$ does not hold, we infer that W does not hold. Contrast these properties with those of R [10] – R is both sound and complete as the property $\mathcal{D} \models W$ iff $\mathcal{D}_{s_0} \cup \mathcal{D}_{una} \models R[W]$ holds. RR, on the other hand, is designed to be complete but not sound.

Definition 2 defines the rules of RR, presented alongside their R counterpart (defined as in [10]). Before presenting these rules, we must explain an important observation. It is clear that we need to regress negated formulae and non-negated formulae with different perspectives under relaxed regression. $RR[\neg W]$ is an optimistic determination of whether W does not hold. It is not the case that $RR[\neg W] \equiv \neg RR[W]$ as in traditional regression. To avoid the addition of a new rule for the negation of each type of compound formula, formulae are converted to *negation normal form* (NNF) prior to the application of RR. In this form: implications ($p \rightarrow q$) are eliminated by replacing them with their equivalent representation ($\neg p \vee q$); negation operators are moved inward using De Morgan's laws; and double negations are eliminated. The result is a formula in which the negation operator appears only before atomic sub-formulae.

Definition 2. *Let W denote a regressable situation calculus formula in NNF, and $\phi_n(\psi)$ the result of converting a formula ψ into NNF. In the following: σ and s are situations; and α and a are actions.*

1. *W is a situation independent atom or uniform in s_0.*

$$RR[W] = W \qquad R[W] = W$$

2. *W is a possibility predicate $Poss(A(\vec{t}), s)$ where $A(\vec{t})$ is an action. Let $Poss(A(\vec{x}), \sigma) \equiv \Pi_A(\vec{x}, \sigma)$.*

$$RR[W] = RR[\phi_n(\Pi_A(\vec{t}, s))]$$
$$R[W] = R[\Pi_A(\vec{t}, s)]$$

3. *W is a relational fluent $F(\vec{t}, do(\alpha, s))$. Let $F(\vec{x}, do(a, \sigma)) \equiv \Phi_F(\vec{x}, a, \sigma)$ be defined as shown in Section 2.*

$$RR[W] = RR[\phi_n(F(\vec{t}, s) \vee \gamma^+_F(\vec{t}, \alpha, s))]$$
$$R[W] = R[\Phi_F(\vec{t}, \alpha, s)]$$

4. *W is an atomic formula mentioning a functional fluent $G(\vec{t}, do(\alpha, s))$. Let $G(\vec{x}, do(a, \sigma)) = y \equiv \Psi_G(\vec{x}, y, a, \sigma)$ be defined as shown in Section 2. $W|^{\phi}_{\phi'}$ denotes the formula that results from replacing all instances of ϕ in W with ϕ'.*

$$RR[W] = RR[\phi_n((\exists y).(\delta_G(\vec{t}, y, \alpha, s) \vee G(\vec{t}, s) = y) \wedge W|^{G(\vec{t}, do(\alpha, s))}_{y})]$$
$$R[W] = R[(\exists y).(\delta_G(\vec{t}, y, \alpha, s) \vee G(\vec{t}, s) = y \wedge \neg(\exists y').\delta_G(\vec{t}, y', \alpha, s)) \wedge W|^{G(\vec{t}, do(\alpha, s))}_{y}]$$

5. *W is a negated possibility predicate $\neg Poss(A(\vec{t}), s)$ where $Poss(A(\vec{x}), \sigma) \equiv \Pi_A(\vec{x}, \sigma)$.*

$$RR[W] = RR[\phi_n(\neg\Pi_A(\vec{t}, s))]$$
$$R[W] = \neg R[\Pi_A(\vec{t}, s)]$$

6. *W is the negation of a relational fluent $F(\vec{t}, do(\alpha, s))$. Let $F(\vec{x}, do(a, \sigma))$ be defined as in rule 3.*

$$RR[W] = RR[\phi_n(\neg F(\vec{t}, s) \vee \gamma^-_F(\vec{t}, \alpha, s))]$$
$$R[W] = \neg R[\Phi_F(\vec{t}, \alpha, s)]$$

7. *W is the negation of an atomic formula W_1 involving a functional fluent $G(\vec{t}, do(\alpha, s))$. Let $G(\vec{x}, do(a, \sigma))$ be defined as in rule 4.*

$$RR[W] = RR[\phi_n((\exists y).(\delta_G(\vec{t}, y, \alpha, s) \vee G(\vec{t}, s) = y) \wedge \neg W_1|^{G(\vec{t}, do(\alpha, s))}_{y})]$$
$$R[W] = \neg R[W_1]$$

8. *W is a compound formula. The rules for RR are the same as those for R (refer to [10]).*

Theorem 1. COMPLETENESS OF RR *Suppose W is a regressable situation calculus formula and $\mathcal{D}$ is a basic action theory. Then $RR[W]$ is a formula uniform in the initial situation s_0, and:*

$$\mathcal{D} \models (\forall)(W \rightarrow RR[W])^3$$

Consequently, if $\mathcal{D} \models W$ then $\mathcal{D}_{s_0} \cup \mathcal{D}_{una} \models RR[W]$.

Proof: For space reasons we do not include the proof. In this proof we follow the basic steps of the completeness proof for R ([10]).

4 A Heuristic GOLOG Interpreter – Preamble

A GOLOG interpreter makes a sequence of choices in the search for a program execution. These choices are: selecting a path to pursue given a branch construct ($\delta_1|\delta_2$); how to instantiate a variable v in $\pi v.\delta$; and whether to execute a program δ zero times or at least once in δ^*. When faced with a choice, a standard GOLOG interpreter selects the first available option (eg. δ_1 given the branch $\delta_1|\delta_2$). When search reaches a dead-end, the interpreter backtracks to the last choice made and makes a different choice in its place.

A standard interpreter does not consider the impact of its choices on the quality of executions found. We develop a heuristic interpreter that evaluates available choices when faced with a non-deterministic construct. We assume: that there exists an evaluation function f_e that returns the numeric quality of an execution; and that the aim of our interpreter is to find an execution that minimises this function.

Example 2. *Consider the space control domain of Example 1. Let $f_e(s) = -num\_obs(s)$ describe the number of observations made in the execution s of the given* GOLOG *program. The goal in this domain is to maximise the number of targets observed.*

We construct a heuristic interpreter for GOLOG programs by adapting the ideas described in [2, 7]. When faced with a non-deterministic construct, the interpreter has the option of transitioning to one of a number of configurations $[(\delta_{c_1}, s_{c_1}), \ldots, (\delta_{c_n}, s_{c_n})]$. For each of these choices, we solve a relaxed version of the problem – we use a relaxed interpreter (defined in Section 5) to find a relaxed execution (δ_{r_i}, s_{r_i}) of δ_{c_i} in s_{c_i}. We then assign to (δ_{c_i}, s_{c_i}) a numeric evaluation given by $f_e(s_{r_i})$. The available choices are ordered from best (smallest) evaluation to worst (largest). The best configuration is transitioned to, and if that choice leads to a dead-end, the next best configuration is considered upon backtracking. Section 6 presents an implementation of our heuristic interpreter.

[3] As in [10], $(\forall)\phi$ denotes the universal closure of ϕ w.r.t its free variables.

5 A Relaxed GOLOG Interpreter

Relaxed regression can be used to approximate the interpretation of a GOLOG program. A relaxed interpreter finds a program execution while approximately deciding if an action is possible or not. The result is a relaxed execution of a program in a given situation.

The purpose of the relaxed interpreter is to determine what desirable future may result from making a particular choice in the interpretation of a program. To achieve this, we assume that we have an evaluation function f_p that assesses the quality of a configuration – which may or may not be the same as f_e (the evaluation function for executions). When faced with a range of configurations to transition to, our relaxed interpreter selects the configuration that minimises this function. If this configuration does not lead to an execution, the next best configuration is selected upon backtracking.

Example 3. *Consider the space control domain of Examples 1 and 2. In this domain, we define f_e and f_p differently. Our relaxed interpreter selects targets for observation that require the least degree of movement of the craft's instrument – if the craft only makes small movements, it is likely that more targets can be observed. If f_e were designed to minimise cost, an identical f_e and f_p would be sensible.*

Relaxed interpretation must terminate if we are to extract from it a heuristic value. To ensure termination we assume that: we have a finite domain (ie. v in $\pi\ v.\ \delta$ has a finite range); there is a finite horizon of actions H over which the interpreter searches for an execution[4]; and all paths through a while or iteration construct involve an action being performed. We further discuss the use of a finite action horizon (and its implications) at the conclusion of this section.

Our relaxed interpreter uses the same *trans* and *final* clauses as the traditional interpreter[5] with the exception of the *trans* clause for actions, branches, argument selection, and iteration. The modified versions of these clauses (called *transR*) are defined below. In these clauses, $\delta, \delta_1, \delta_2, \delta'$ denote programs, while S, S' denote situations.

In the *transR* clause for a primitive action: *primAct(A)* holds if A is a primitive action, and *possR(A, S)* holds if A is possible in S where its preconditions have been evaluated by relaxed regression.

transR(H, A, S, δ', S', H') :-
$H > 0$, primAct(A), possR(A, S),
δ' = nil, S' = do(A, S), H' is H-1.

In the *transR* clause for a branch, *transR′(H, δ, S, List)* finds the *List* of configurations that can be transitioned to from (δ, S) under *transR*. The predicate *best(List, δ', S', H')* uses f_p to select the best configuration (δ', S') in *List* (where horizon H' remains).

transR(H, $\delta_1|\delta_2$, S, δ', S', H') :-
transR′(H, δ_1, S, List$_1$), transR′(H, δ_2, S, List$_2$),
append(List$_1$, List$_2$, List), best(List, δ', S', H').

In the *transR* clauses for argument selection and iteration, *transR″(H, δ, S, List)* finds the *List* of configurations that can be transitioned to from (δ, S) under *transRS*, defined below. *sub(V, _, δ, δ_1)* replaces V in δ with an unbound variable to produce δ_1.

transR(H, Prog, S, δ', S', H') :-
(Prog = $\pi V.\delta$; Prog = δ^*),
transR″(H, Prog, S, List),
best(List, δ', S', H').

transRS(H, $\pi V.\delta$, S, δ', S', H') :-
sub(V, _, δ, δ_1), transR(H, δ_1, S, δ', S', H').

transRS(H, δ^*, S, δ:δ^*, S', H') :-
transR(H, δ, S, δ', S', H').

The *transR* clauses defined above are used to find a relaxed execution S' of a program δ in situation S with horizon H as follows (where *transR** is the reflexive transitive closure of *transR*).

doR(δ, S, S', H) :-
transR(H, δ, S, δ', S', H'), (H' = 0; finalR(δ', S')).*

In classical planning, selecting a horizon of actions that we expect a plan to be defined within can be difficult. As we shall see in Examples 4, 5 and 6 in Section 6.1, the structure of a GOLOG program provides a basis from which H can be derived. Given our intended use of the relaxed interpreter as a tool for generating heuristic values, the appropriate selection of H need not be agonised over. A H value that is too low will only yield a (potentially) less informed heuristic. As we shall discuss in Section 8, the use of this look-ahead bound allows us to use our heuristic interpreter in an online setting.

We do not apply relaxed regression to assess the truth of tests, while loop, and if/else conditions. We have found by experiment that this yields the best performance across a range of domains.

6 An Implementation of a Heuristic GOLOG

Our heuristic interpreter finds an execution of a GOLOG program in the same way as a traditional interpreter – with a set of transition axioms. Our interpreter utilises the same final axioms as the standard interpreter but defines new transition axioms for most of the GOLOG constructs. The *transH* clauses for the primitive action, test, if ... else, and procedure constructs are the same as those in a traditional interpreter[6]. Let *Prog* $= \delta_1 : \delta_2$, $\delta_1|\delta_2$, $\pi v.\delta$, δ^*, or $\mathbf{while}\ \phi\ \mathbf{do}\ \delta\ \mathbf{end}$. The *transH* clauses in our heuristic interpreter for each of the *Prog* constructs have the following form:

transH(Prog, S, δ', S', H) :-
transH′(Prog, S, List), best′(List, H, δ', S').

where *transH′(Prog, S, List)* finds the *List* of configurations that can be transitioned to from *(Prog, S)* (under *trans*), and *best′(List, H, δ', S')* selects the best configuration in *List* to transition to. It does so by finding a relaxed execution of each configuration with *doR* and applying the evaluation function f_e to this execution resulting in a heuristic value for the corresponding choice.

The *transH* clauses defined above are used to find an execution S' of a program δ in situation S with horizon H as follows (where *transH** is the reflexive transitive closure of *transH*).

doH(δ, S, S', H) :- transH(δ, S, δ', S', H), finalH(δ', S').*

Remark 1. *Our heuristic interpreter is not optimal (admissible).*

Executions found by our relaxed interpreter are not necessarily the best or even possible. The heuristic values assigned to configurations that the heuristic interpreter can choose from do not necessarily represent which choices will definitely lead to the best legal executions. In addition, once a choice is made by the search algorithm, we do not backtrack until we arrive at a deadend.

6.1 Experimental Results

We compare the results of our heuristic interpreter with a traditional interpreter, and an interpreter in which choices are made according to a greedy heuristic without look-ahead by relaxation, in three domains. Each of our examples demonstrate situations in which the choices made during interpretation greatly affect the quality of the

[4] The relaxed interpreter will find relaxed executions of up to H actions long.
[5] With the addition of bookkeeping related to the finite action horizon H.
[6] With the addition of a finite action horizon H to pass to our relaxed interpreter when evaluating configurations.

execution found, and where some choices (that may not appear at first to be favourable) need to be made to find the best executions.

Example 4. *Our first collection of experiments is conducted in the spacecraft control domain of Examples 1, 2 and 3.*

In this domain, executions are evaluated with respect to the number of targets observed. Our relaxed and greedy interpreters select targets for observation that require the least degree of movement of the craft's instrument. The effect of relaxation on the preconditions of actions in this example removes the need to check for adequate resources and time to turn to and take images of targets.

A set of test cases have been generated with a specific number of celestial and calibration targets, fuel, power, and time limits (Table 2 shows the results of a subset of these tests). The locations of the target clusters were randomly defined – each coordinate (x, y, and z) assigned a random integer between -100 and 100. The position of each target within a cluster was randomly defined and lies within a small deviation of this location. The action horizon was set to the number of time steps available in each test case.

N_T	FUEL	POWER	TIME	N_{CL}	O_S	O_G	O_H
10	500	200	150	6	4	6	6
10	500	200	150	6	1	1	4
20	600	300	250	9	4	4	10
20	600	300	250	9	1	3	7
30	700	200	350	11	4	5	16
30	700	200	350	11	4	10	17
40	800	300	450	14	9	21	21
40	800	300	450	14	8	17	24
50	900	400	550	18	8	20	30
50	900	400	550	18	7	22	28

Table 2. Observations made in executions found by the standard (O_S), greedy (O_G), and heuristic (O_H) interpreters. N_T and N_{CL} denote the number of celestial targets and clusters in each test.

In 33/50 (46/50) of the tests, our heuristic interpreter found executions that made more observations than those found by the greedy (traditional) interpreter. In the remainder, the compared interpreter pairs yielded executions with equal observation counts.

Example 5. *Consider an iron ore mine consisting of a series of iron ore blocks located across the mine landscape. A mining robot is designed to travel to a block, blast the block, collect the ore, and transfer the ore to a stockpile. Each block has a percentage amount of iron, silica, alumina, and phosphorus. Some blocks are dependent on others – and cannot be mined until they are mined. A mining robot can only travel between a block and the stockpile if there is a safe (non-hazardous) path between the two. The robot is designed to build a stockpile of a desired tonnage and target defining the desired bounds on the percentages of iron, silica, alumina, and phosphorus.*

A plan for the miner is determined by finding an execution of the GOLOG *program* MINEOP.

proc MINEOP
  powerup :
  while *stockrequired* ***do***
    π *blk. [available(blk)? :*
      moveto(blk) : *blast(blk)* : *mine(blk)* :
      moveto(stockpile) : *combine(blk)]*
  end :
  powerdown
end

where: stockrequired denotes that the stockpile requires more ore; available(blk) denotes that blk is a block that is available for mining (ie. it has not been previously mined and all blocks it depends on have been mined); and powerup, moveto(object), blast(blk), mine(blk), combine(blk), and powerdown are primitive actions.

Let $f_e(s) = distance(s, target)$ denote how far away from the desired $target$ the stockpile is in execution s. Distance is defined as the difference between a stockpile component value and its target divided by its acceptable range. The sum of distances for each component (eg. iron etc.) forms f_e. Our relaxed and greedy interpreters select blocks to mine that are closest in composition to this target. The relaxation of action preconditions removes the need to check for hazards along the paths between blocks and the stockpile.

A set of test cases have been generated with a specific number of blocks (of equal tonnage) and hazards (Table 3) – each block and hazard located at a coordinate whose components are assigned a random integer between 0 and 500. The location of the stockpile is also randomly defined. The blocks are divided into groups, forming a chain of blocks that can only be mined in sequence. The stockpile target is randomly defined, and the composition of each block is defined by adding or subtracting a random deviation from the stockpile target. The action horizon chosen in each test is 5$\times$ the number of its blocks.

N_{Blocks}	R_{To}	N_G	N_H	D_S	D_G	D_H
5	1500	3	5	6.12	6.97	2.55
10	2500	5	5	3.27	5.90	0.75
15	3500	6	5	4.29	1.78	1.70
20	4500	6	10	1.85	1.43	2.11
25	5500	8	10	2.62	4.09	0.86
30	6500	11	15	1.54	4.10	0.36
35	7500	10	15	1.23	2.58	0.34
40	8500	12	20	2.37	1.58	0.81
45	8500	17	20	4.68	3.91	0.33

Table 3. Distance from stockpile target in executions found by the standard (D_S), greedy (D_G), and heuristic (D_H) interpreters. N_{Blocks}, R_{To}, N_G, and N_H denote the number of blocks, required stockpile tonnage, number of block groups, and hazards.

In 51/60 (56/60) of the tests, executions found by our heuristic interpreter led to a closer-to-target stockpile than those found by the greedy (traditional) interpreter. In 3/60 (1/60) of the tests, executions found by our heuristic interpreter resulted in a lower quality stockpile than those found by the greedy (traditional) interpreter.

Example 6. *Consider a set of workers W, each $w_i \in W$ with a set of skills S_i. A scheduling agent is designed to select a set of jobs for the workers to perform within a limited time window, where each job consumes one unit of time. Each job consists of a set of tasks, each task requiring a specific subset of skills to complete. Only one job can be scheduled at a time, and only if there is a worker able to perform each of its tasks (where each worker can only perform one task per job). Jobs have dependencies such that some jobs can only be scheduled if other jobs have been performed before them.*

A plan for the scheduler is determined by finding an execution of the GOLOG *program* SCHEDULE.

proc SCHEDULE
  while *timeleft* ***do***
    π *job. [available(job)? :*
      $\{$ π *tk. [(task(tk, job)* $\wedge \neg$ *assigned(tk, job))? :*
        π *w. [worker(w)? : assign(w, tk, job)]]* $\}^*$:
      scheduled(job)?]
  end
end

where: timeleft denotes that there is $\geq$ 1 unit of time left; $available(job)$ holds if job has not been scheduled and each of its dependencies have been scheduled; $task(tk, job)$ denotes that tk is a task of job; $assigned(tk, job)$ denotes that tk in job has

been assigned to a worker; $worker(w)$ *denotes that* w *is a worker;* $scheduled(job)$ *holds if all the tasks in* job *have been assigned a worker; and* $assign(w, tk, job)$ *is a primitive action assigning* tk *(of* job*) to worker* w*.*

Let $f_e(s) = -tasks(s)$ describe the number of tasks that have been assigned to workers in execution s. Our relaxed and greedy interpreters select jobs to schedule that consist of the most tasks for assignment. In this example, relaxation of action preconditions removes the need to check whether a worker has already been assigned to a task on a job before assigning them to another.

A set of test cases have been generated with a specific number of workers, jobs, and maximum number of tasks per job. The number of tasks in each job is a randomly generated number between 1 and this maximum. The jobs are divided into groups forming a chain of jobs that can only be scheduled in sequence (with jobs later in the sequence more likely, but not necessarily, to have more tasks). Each worker is assigned a randomly selected subset of skills (from a set of 10), and each task is assigned a set of skills such that there are at least two workers able to complete it. The action horizon in each test is the number of jobs $\times$ the maximum number of tasks per job.

Due to space limitations, we only summarise the results of these tests. In 31/40 (38/40) of the tests, executions found by our heuristic interpreter resulted in more task assignments than those found by the greedy (traditional) interpreter. In 2/40 of the tests, the heuristic interpreter could not find a solution within 15 minutes. On the remainder, the compared interpreter pairs resulted in equally good executions.

Speed Comparison Traditional interpretation is much faster than our heuristic interpreter. In each of the test cases in Table 2, traditional interpretation takes at most 2 seconds. In the most complex test case of this example (with 50 targets) our heuristic interpreter takes 98 seconds to find a solution. The average time required to find a solution in the set of 50 target test cases was 71 seconds.

In each domain, we also compared our heuristic interpreter to one that uses actual interpretation (not relaxed) to evaluate choices. On test cases in Table 2, this interpreter was on average 8$\times$ (at most 19 and at least 5$\times$) slower than our heuristic interpreter.

7 Related Work

There are several existing techniques that combine GOLOG and classical planning. In [1], GOLOG programs are compiled into PDDL – a domain description language used by many classical planners. A classical planner can then be used to find an action sequence that is an execution of the GOLOG program. The underlying action theory in [1] must be described in PDDL, rather than a (more expressive) situation calculus basic action theory (BAT)[7]. Our approach allows the domain to be characterised by a BAT. In addition, the compilation of a GOLOG program into PDDL has potential limitations (highlighted in [9]) – the addition of new fluents and actions (increasing state-space size), and the loss of control knowledge expressed in loop constructs. As we remain within the situation calculus and GOLOG, our approach does not suffer from these limitations.

The embedding of classical planning within a GOLOG program is considered in [5]. This work looks at problems that in part are suited for representation in GOLOG, but have stages in which general planning is required (such as the navigation of an entity across a grid). While this approach and ours combine GOLOG and classical planning, our technique looks at applying classical planning techniques to the whole interpretation process (rather than individual steps) to find an optimal or near optimal program execution (while the other does not). Our interpreter is designed to work with problems that are suited for representation in GOLOG, while still involving combinatorial optimisation, and not general planning.

For brevity, we restrict our discussion to the above works and only briefly mention another GOLOG interpreter that finds optimal program executions. DTGOLOG [3] is one such work, but is designed for stochastic environments and searches the entire execution space to find an optimal *policy* for program interpretation.

[7] Relative expressiveness of BATs and PDDL is analysed in [6, 12].

8 Conclusion & Future Work

In this paper we have described a heuristic GOLOG interpreter that is able to simulate the interpretation of a GOLOG program as a means of evaluating the available choices in the search for an execution. This simulation relies on a relaxed variant of regression (relaxed regression) that we have presented in this paper. We have considered the successful relaxation-based heuristics used in classical planning [2, 7] and described how they can be used to find a (near) optimal execution of a GOLOG program given an evaluation function.

We have presented our interpreter within an offline setting. The interpreter can be extended for use in an online setting by adding techniques to handle sensing actions during relaxed interpretation (which is conducted offline). Within our relaxed interpreter, we can view sensing actions as being a choice construct, where each possible sensing result is a choice. The interpreter can then select the sensing result most beneficial for it (ie. it assumes the best case scenario), just as it would a choice between actions. As a finite action horizon bound is used during relaxed interpretation, our interpreter can be used even with non-terminating programs (in an online setting).

As future work, it would be interesting to investigate the relative performance of our interpreter with the compilation approach of [1] on the subset of GOLOG programs and BATs that are suitable for compilation into PDDL.

REFERENCES

[1] J. Baier, C. Fritz, M. Bienvenu, and S. McIlraith, 'Beyond classical planning', in *AAAI*, pp. 1509–1512, (2008).

[2] B. Bonet and H. Geffner, 'Planning as heuristic search', *Artificial Intelligence*, **129**, 5–33, (2001).

[3] C. Boutilier, R. Reiter, M. Soutchanski, and S. Thrun, 'Decision-theoretic, high-level agent programming in the situation calculus', in *AAAI*, pp. 355–362, (2000).

[4] G. de Giacomo, Y. Lespérance, and H. J. Levesque, 'ConGolog, a concurrent programming language based on the situation calculus', *Artificial Intelligence*, **121**(1-2), 109–169, (2000).

[5] J. Claßen, P. Eyerich, G. Lakemeyer, and B. Nebel, 'Towards an integration of Golog and planning', in *IJCAI*, pp. 1846–1851, (2007).

[6] P. Eyerich, B. Nebel, G. Lakemeyer, and J. Claßen, 'GOLOG and PDDL: What is the relative expressiveness?', in *PCAR*, (2006).

[7] J. Hoffmann and B. Nebel, 'The FF Planning System: Fast Plan Generation Through Heuristic Search', *JAIR*, **14**, 253–302, (2001).

[8] H. J. Levesque, R. Reiter, Y. Lespérance, F. Lin, and R. B. Scherl, 'Golog: A logic programming language for dynamic domains', *Journal of Logic Programming*, **31**(1-3), 59–83, (1997).

[9] Ronald P. A. Petrick, 'P^2: A baseline approach to planning with control structures and programs', in *ICAPS 2009 Generalized Planning*, pp. 59–64, (2009).

[10] Fiora Pirri and Ray Reiter, 'Some contributions to the metatheory of the situation calculus', *Journal of the ACM*, **46**(3), 325–361, (1999).

[11] Raymond Reiter, 'The frame problem in situation the calculus', *Artif. Intell. and Mathematical Theory of Computation*, 359–380, (1991).

[12] G. Röger, M. Helmert, and B. Nebel, 'On the Relative Expressiveness of ADL and Golog: The Last Piece in the Puzzle', in *KR*, (2008).

[13] D. E. Smith, J. Frank, and A. K. Jónsson, 'Bridging the Gap Between Planning and Scheduling', *Knowledge Engineering Review*, **15**, (2000).