Optimisation and Relaxation for Multiagent Planning in the Situation Calculus

Toby O. Davies
National ICT Australia and
The University of Melbourne
toby.davies@nicta.com.au

Adrian R. Pearce
National ICT Australia and
The University of Melbourne
adrianrp@unimelb.edu.au

Peter J. Stuckey
National ICT Australia and
The University of Melbourne
pstuckey@unimelb.edu.au

Harald Søndergaard
The University of Melbourne
harald@unimelb.edu.au

ABSTRACT

The situation calculus can express rich agent behaviours and goals and facilitates the reduction of complex planning problems to theorem proving. However, in many planning problems, solution quality is critically important, and the achievable quality is not necessarily known in advance. Existing *Golog* implementations merely search for a *Legal* plan, typically relying on depth-first search to find an execution. We illustrate where existing strategies will not terminate when quality is considered, and to overcome this limitation we formally introduce the notion of *cost* to simplify the search for a solution. The main contribution is a new class of relaxations of the planning problem, termed *precondition relaxations*, based on Lagrangian relaxation. We show how this facilitates optimisation of a restricted class of *Golog* programs for which plan existence (under a cost budget) is decidable. It allows for tractably computing relaxations to the planning problem and leads to a general, *blackbox*, approach to optimally solving multi-agent planning problems without explicit reference to the semantics of interleaved concurrency.

Categories and Subject Descriptors

G.1.6 [**Numerical Analysis**]: Optimization; I.2.8 [**Artificial Intelligence**]: Problem Solving, Control Methods, and Search—*plan execution, formation, and generation*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*multiagent systems*

General Terms

Algorithms

Keywords

Resource-bounded planning, Golog, Lagrangian relaxation, Situation calculus

Appears in: *Proceedings of the 14th International Conference on Autonomous Agents and Multiagent Systems (AAMAS 2015), Bordini, Elkind, Weiss, Yolum (eds.), May 4–8, 2015, Istanbul, Turkey.*

1. INTRODUCTION

Reasoning about quality is essential in many domains, as agents must often make economical use of one or more resources, be it money, fuel, time, or some other domain-specific resource.

We consider multiagent planning problems expressed in *Golog*, an agent language based on the situation calculus. *Golog* is Turing complete—the use of a *Golog* interpreter comes without any guarantee of termination. We can, however, identify a restricted class of problems (or associated *Golog* programs) that have a "bounded benefit" property. We show that budget-limited planning is decidable for this class. We also introduce relaxations that let us reason about how far from optimal a candidate solution is. Relaxations can be viewed as specialized algorithms to prove statements of the form "no solution is more than a multiple of ϵ better than solution x" without the need to enumerate the solution space. Technically our approach is close to the use of Lagrangian relaxation techniques in operations research.

Delete relaxation has previously been applied to *Golog*, to generate heuristically improved execution [5]. In contrast with that work, we focus on *precondition relaxation*, as this is particularly appropriate for multi-agent teams where each agent ideally helps others achieve the preconditions they face. The kind of relaxation we have in mind is much more sophisticated than an "ignore precondition" relaxation. Our precondition relaxations can avoid combinatorial explosions by ignoring the multiple ways concurrent action sequences can be interleaved. To guide search for optimal solutions we apply costs to assuming, and bonuses to causing, the relaxed preconditions. Thus preconditions are treated as a type of shared resource that can be traded between agents at a cost. This is key to the performance improvements we achieve.

Utilisation of shared resources has previously been treated in the situation calculus, but in the context of an explicit interleaved semantics of concurrency [8]. Our approach can frequently minimise—sometimes even ignore—explicit inter-agent action interleaving, heuristically finding high-quality joint executions directly from high-level specifications.

Contribution.

The relax-&-merge algorithm described in this paper generalises the fragment-based planning approach of Davies *et*

al. [7] which shows orders of magnitude improvements over state-of-the-art temporal planners. We define the necessary conditions for modeling domains in this generalized formalism. We propose a new kind of relaxation for planning problems; "precondition relaxations" which adapt Lagrangian relaxation to dynamic logic-based optimisation problems. The technique is of particular interest in a multi-agent setting, because it offers a blackbox approach to planning via collaborative search: An agent is not required to know about other agents' programs or effect axioms for "private" fluents; all that is required is the ability to query those agents for their optimal plans under a given penalty function. We show that well-chosen relaxations can provide large increases in the speed of planning, scaling linearly with the number of interacting agents. Our work allows optimisation using a wider range of search techniques in the situation calculus than previously possible.

Outline.

We recapitulate *Golog* and relaxation broadly, in Sections 2 and 3, respectively. Section 4 introduces bounded-benefit programs and Section 5 introduces precondition relaxation. Section 6 shows how to combine individual agents' relaxed plans and Section 7 shows how to use the result to construct a feasible joint execution.

2. PRELIMINARIES

We assume familiarity with the situation calculus and reasoning based on regression, at the level of Reiter [15], from which we also (mostly) borrow notation and terminology. We use $\mathcal{R}$ for the regression operator; $\preceq$ for the pre-history relation; Φ_f as the regressable successor-state axiom for a fluent f; and ϕ_f^+ and ϕ_f^- for the positive and negative effect axioms of a fluent f respectively.

We use a fragment of *ConGolog* [8], which includes most constructs of the language, except for (recursive) procedures. Hereafter we will simply refer to *Golog*, *ConGolog* and its extensions simply as *Golog*:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequence
if φ **then** δ_1 **else** δ_2	conditional
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$\delta_1 \Vert \delta_2$	concurrency

In the above, α is an action term, possibly with parameters, and φ is a situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. We denote by $\varphi[s]$ the situation calculus formula obtained from φ by restoring the situation argument s into all fluents in φ.

Program $\delta_1 | \delta_2$ allows for the nondeterministic choice between programs δ_1 and δ_2, while $\pi x.\delta$ executes program δ for *some* nondeterministic choice of a legal binding for variable x. δ^* performs δ zero or more times. Program $\delta_1 \Vert \delta_2$ expresses the concurrent execution (interpreted as interleaving) of programs δ_1 and δ_2. We assume without loss of generality that each occurrence of the construct $\pi x.\delta$ in a program uses a unique fresh variable x.

Formally, the semantics of *Golog* is specified in terms of single-step transitions, using the following two predicates [8]: *(i)* $Trans(\delta, s, \delta', s')$, which holds if one step of program δ in situation s may lead to situation s' with δ' remaining to be executed; and *(ii)* $Final(\delta, s)$, which holds if program δ may legally terminate in situation s.

The definitions of *Trans* and *Final* we use are as in [17]; these are standard [8], except that, following [6], the test construct $\varphi?$ does not yield any transition, but is final when satisfied. Thus, it is a *synchronous* version of the original test construct (it does not allow interleaving). Note that the definition of $Trans(\delta, s, \delta', s')$ has only one successful non-recursive case, where s' is exactly one action longer than s; any successful transition adds exactly one action to the current situation.

Trans and *Final* are used to define the single-step semantics of *Golog*; they are used to look ahead, to solve the projection problem. We define $Trans^*$ to be the transitive reflexive closure[1] of *Trans*. $Trans^*$ is used to define the reachable states: we wish to limit the search to states that are both reachable, and for which any residual program is *Final*. For this purpose we define

$$Do(\delta, s, s') \equiv \exists \delta'.\, Trans^*(\delta, s, \delta', s') \wedge Final(\delta', s')$$

3. REASONING ABOUT OPTIMALITY

A key technique used in operations research to prove solution quality is *relaxation*. Given a problem with a set X of solutions and cost function to be minimized C, a relaxation is a new problem with solutions X' and cost function C' such that $X \subseteq X'$ and for any solution $x \in X$, the relaxed cost is a lower bound on the real cost, that is, $C'(x) \leq C(x)$.

A useful relaxation is one that is easier to solve to optimality, compared to the original problem, and which additionally provides a tractable way to establish how far from optimal a candidate solution is. Let x be a feasible solution to the original problem and let x'^* be an optimal solution to the relaxed problem. The optimality gap is then defined as $\eta = C(x)/C'(x'^*) - 1$. When the gap is 0, the solution x is optimal. Where multiple relaxations are available, the tightest can be used to compute η. This approach is used with Lagrangian relaxation of integer programs.

In many optimisation problems it is easier to optimize a variant of the problem with fewer constraints. For example the resource constrained shortest path problem [13] is NP-complete, whereas there are well-known polynomial-time algorithms, such as Dijkstra's, for the classical, unconstrained shortest path problem.

Lagrangian relaxation softens complicating constraints and incorporates them into the objective function [12, 9]. The idea is to capitalise on algorithms designed for the easier problem while penalizing violations of the complicating constraints. Careful variation of the penalties (the so-called Lagrange multipliers) allows solutions to the relaxed problem to be guided towards feasible areas of the original problem.

Traditionally Lagrangian relaxation is applied to Integer Programming (IP) models. A major contribution of this paper is the extension of Lagrangian relaxation to logic-based optimization in dynamical systems.

[1] *Trans* is a binary relation but we follow convention, writing $Trans(\delta, s, \delta', s')$ rather than $Trans((\delta, s), (\delta', s'))$.

Consider a set R of inequalities to relax in an IP model:

$$\begin{array}{lll} \text{Minimize:} & z(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_r(\tilde{\boldsymbol{x}}) \leq 0 & \forall r \in R \\ & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

We transform it into a relaxed problem:

$$\begin{array}{lll} \text{Minimize:} & z(\tilde{\boldsymbol{x}}) + \sum_{r \in R} \lambda_r \cdot c_r(\tilde{\boldsymbol{x}}) & \\ \text{Subject To:} & c_n(\tilde{\boldsymbol{x}}) \leq 0 & \forall n \in N \\ & \tilde{\boldsymbol{x}} \in \mathcal{Z}^n & \end{array}$$

Note that the effect is to increase the objective when constraints are violated, and decrease it when constraints are strictly satisfied. To solve the original problem, the relaxed problem is optimized and for each violated constraint r, the penalty λ_r is increased. The relaxed problem is then re-optimized, and so the process is iterated. In general, branching is also required to find solutions to discrete problems.

4. COST-AWARE SEARCH

Consider the problem of finding a path for agent a from $\langle 0, 0\rangle$ to $\langle x, y\rangle$ on an infinite Manhattan grid. The agent is allowed to move in each of the cardinal compass directions N, S, E, and W.

A naive *Golog* program to solve this problem is:

```
proc travelto(a, x, y):
  while ¬At(a, x, y) do
    π(d ∈ [N, S, E, W]). move(a, d);
```

A satisfying solution to this problem is uninteresting as there are infinitely many and an optimal path is trivial to compute. However a simple depth first search for solutions to this program might not terminate. To guarantee termination, a significantly more complex and less flexible program may restrict the search to only move towards $\langle x, y\rangle$. Such a program will find an optimal path between two points, but the approach will fail in general, if any obstacles are introduced into the grid.

We introduce a restricted class of *Golog* program and cost functions that allow the simpler and more general *Golog* program to always terminate. Algorithms from classical planning then allow us to compute the optimal solution. We refer to the restrained programs as "bounded-benefit".

Definition 1 (Bounded Benefit). *A bounded-benefit program is a Golog program δ, for which there exists some plan length l_0 and $\epsilon > 0$ such that some lower bound $lb(l)$ on the cost of any reachable situation of length l satisfies $\forall n \in \mathbb{N} : lb(l_0 + n) \geq lb(l_0) + n\epsilon$, and $lb(l_0)$ is finite.*

That is, there are a finite number of beneficial actions that can be performed, and all other actions increase cost by at least ϵ. Note that this is a property of the program and cost function, and may not depend on the precondition axioms of the domain.

Bounded benefit programs are related to the problems addressed by classical planning, but are slightly more general. In particular, planning algorithms assume monotonically increasing costs where $l_0 = 0$ and $lb(0) = 0$. Bounded benefit programs can be converted into this form when $lb(l_0)$ is known (rather than merely guaranteed to exist) by defining a new cost function similar to removal of soft-goals in classical planning [11]. Optimal algorithms for this class of problems are well studied in the automated planning literature, and include both "uninformed" search, such as uniform cost search, and "informed" search such as A^*, which is reliant on a heuristic or relaxation.

The simplest class of bounded-benefit programs results when all actions increase cost. In the Manhattan grid example, if all actions have unit cost, every program has bounded-benefit. A less obvious, but interesting class of bounded-benefit programs results when the cost function defines a finite set of "soft goals" that decrease the objective when achieved. For example, in the Manhattan grid example, some grid points may be points of interest for which a reward is available as they are visited the first time (similar to the prize-collecting travelling salesman problem [3]). We use this kind of soft-goal in later sections to guide agents to achieve preconditions for others.

We have introduced bounded-benefit programs in response to the undecidability of planning in *Golog*. In aid of our proof of decidability we introduce the following definitions:

Definition 2 (Agent State). *An agent state is a pair $\langle s, \delta\rangle$ comprising a situation s and a residual program δ.*

Definition 3 (Reachable State). *An agent state $\langle s', \delta'\rangle$ is reachable from $\langle s, \delta\rangle$ if it satisfies $Trans^*(\delta, s, \delta', s')$.*

Definition 4 (Applicable Transitions). *The set of applicable transitions $T(s, \delta)$ is the set of agent states reachable by using exactly one action from $\langle s, \delta\rangle$*

$$T(s, \delta) = \{\langle s', \delta'\rangle \mid Trans(\delta, s, \delta', s')\}$$

Definition 5 (Branching Factor). *A Golog program δ in some domain $\mathcal{D}$ has branching factor $b = max(|T(s', \delta')|)$ over all reachable states $\langle s', \delta'\rangle$ of finite length.*

To have a finite branching factor merely requires that the set of possible actions in any given situation is bounded. This is not an unreasonable restriction, and would be required for a Prolog-based *Golog* implementation.

Theorem 1. *Let $m \in \mathbb{N}$ and let δ be a bounded-benefit program with finite branching. The query $\mathcal{D} \models Do(\delta, S_0, s) \wedge (\forall s' : s' \preceq s \Rightarrow cost(s') \leq m)$ is decidable.*

Proof. For any finite length l, the situations reachable from S_0 in at most l steps can be enumerated (there are at most b^l solutions), for example, by a breadth-first search.

By Definition 1 we can assume the existence of a length l_0 beyond which cost grows linearly, at least. The length of a reachable situation s with $cost(s) \leq m$ is then bounded by $l_0 + (m - lb(l_0))/\epsilon$. □

More practically, we can perform any search algorithm with an additional test for the condition $cost(s) \leq m$ when nodes are expanded, and guarantee that either we exhaust the successors of s, or the cost of the successors eventually exceeds the budget m, by defining a budget-limited *Trans*:

$$Trans_m(\delta, s, \delta', s') \equiv Trans(\delta, s, \delta', s') \wedge Cost(s') \leq m.$$

Importantly this approach does not require a computable lower bound, merely the guarantee that one exists.

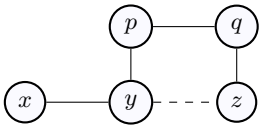

Figure 1: Cooperative navigation with unlockable edges. Initially the edge from y to z is locked. The shortest path from x to z is dependent on the location of other agents capable of unlocking the edge.

5. RELAXING PRECONDITIONS

Relaxing delete effects (by ignoring them) is common in automated planning. The result is "easier" than the original problem because the application of an action without delete effects monotonically increases the set of true facts. Ignoring preconditions is the regression analogue: the regression of a goal formula through an action with no preconditions monotonically decreases the set of open preconditions, guaranteeing the regressed formula will eventually become true in the initial situation, if any such action sequence exists.

Instead of simply ignoring preconditions, we attach costs to assumptions. In a collaborative setting, an agent has a choice between the pursuit of achieving its own precondition, or relying on others to do this. Whichever is easier depends on circumstances but the choice can be informed by using cost to express a benefit to achieving another agent's preconditions.

For a very simple example, consider Figure 1 which depicts an instance of a "cooperative navigation" problem. (We will vary the initial conditions throughout this paper, but Figure 1's graph will remain unchanged in subsequent examples.) The dashed edge denotes an edge that is in a "locked" state; it can be unlocked only by certain agents, and only if they are at appropriate locations. For now assume there are two agents, a and b, with the joint goal for a to reach z. The edge from y to z can only be unlocked by b if it is at y. We assume unit cost to unlock an edge and to traverse any edge. Note that the shortest path for one agent to travel from x to z depends on other agents enabling it.

Primitive actions in this domain are $\texttt{move}(Agt, From, To)$ and $\texttt{unlock}(Agt, From, To)$. Fluent $At(Agt) = Loc$ gives an agent's current location; $Unlocked(From, To)$ represents a usable *From-To* edge while $HasKey(Agt, From, To)$ represents a potential *From-To* edge.

The cost function in this domain is equivalent to plan length. Note that any program has bounded-benefit with this cost function: $l_0 = 0$ and $\epsilon = 1$.

First consider the initial situation where agent a is at x, agent b is at y, and b has a key to unlock $y \rightarrow z$. The minimum cost joint execution is for b to unlock $y \rightarrow z$, and a to travel from x to y to z. If however b is initially at z, the minimum cost path is for x to travel via p and q. Note carefully that a sum of per-agent relaxation will not be admissible here in general: a's optimal path over-estimates the true cost.

Preconditions persist in general, and the best time to allow another agent to achieve our precondition is not necessarily the situation immediately preceding the action requiring that precondition. There are two options as to where we can allow this necessary choice: where we assume a fluent becomes true; and when we allow a fluent we cause to be used by another agent.

We choose to use the first of these, as there will always be a bounded choice of situations in which we can assume a fluent. We then force agents to take the benefit associated with causing a fluent at the point it was actually caused.

That is, in $do([\texttt{move}(a, x, y), \texttt{move}(a, y, z)], S_0)$ agent a can assume that $y \rightarrow z$ was unlocked in S_0 or after a's first action, and thus may choose either of these two penalties to apply. However in $do([\texttt{move}(b, p, y), \texttt{unlock}(b, y, z)], S_0)$ agent b has no choice and must take the bonus available after the first action.

In the cooperative navigation domain nothing negates the preconditions we are relaxing. However, in general an agent could negate an assumed precondition before the point at which it was required. To avoid this, we introduce some analysis functions that allow us to determine the situations from which an assumed fluent would have persisted. We assume that fluents have successor state axioms transformed from effect axioms ϕ^+ and ϕ^- in the standard way [15]. $lasteff(f, s) = s'$ computes the last situation which would have had either a positive or negative effect on f:

$$\begin{array}{lcl} lasteff(f, S_0) & = & S_0 \\ lasteff(f, do(\alpha, s)) & = & \begin{cases} do(\alpha, s) & \textbf{if } \phi_f^{\pm}(\alpha, s) \\ lasteff(f, s) & \textbf{otherwise} \end{cases} \end{array}$$

Here $\phi_f^{\pm}(\alpha, s) \equiv \phi_f^+(\alpha, s) \vee \phi_f^-(\alpha, s)$. For example, in the situation $s = do(\texttt{move}(a, x, y), S_0)$:

$$\begin{array}{lcl} lasteff(At(a), s) & = & s \\ lasteff(Unlocked(y, z), s) & = & S_0 \end{array}$$

We then restrict situations in which f can be assumed before situation s, to situations s' with $lasteff(f, s) \preceq s' \preceq s$.

To transform a regressable query in a basic action theory in such a way that we can apply penalties, we need to detect which assumptions Δ are made. For this purpose we define Precondition Relaxed Regression $\mathcal{PRR}_R^{\Delta}$ in terms of a set of relaxable fluents R. This can be considered a kind of abductive reasoning, but differs in that any relaxable fluent $F(s)$ must be explicitly assumed if it is ever true, even if it is caused by some action already present in s.

In the cooperative navigation example, $Unlocked(y, z)$ must be assumed (recorded in Δ) to ensure that agent a can find the optimal plan. That is, a must assume that the (y, z) edge becomes unlocked before attempting $move(a, y, z)$.

Precondition relaxed regression is defined like usual regression, except for the cases for atomic fluents:

$$\begin{array}{l} \mathcal{PRR}_R^{\Delta}(f(S_0)) = \\ \quad ([\![f(S_0)]\!] \notin R \wedge f(S_0)) \vee ([\![f(S_0)]\!] \in R \wedge [\![f(S_0)]\!] \in \Delta) \end{array}$$

$$\begin{array}{l} \mathcal{PRR}_R^{\Delta}(f(s)) = \\ \quad (s = do(\alpha, s')) \wedge [\![f(s)]\!] \notin R \wedge \mathcal{PRR}_R^{\Delta}(\Phi_f(\alpha, s')) \vee \\ \quad [\![f(s)]\!] \in R \wedge \exists s''.(lasteff(f, s) \preceq s'' \preceq s \wedge [\![f(s'')]\!] \in \Delta) \end{array}$$

Here Φ_f is the successor state axiom for f. The brackets $[\![\ldots]\!]$ have no semantic significance; we use them simply as a reminder that Δ and R store *representations* $[\![f(s)]\!]$ of fluents f to evaluate at situations s. In the definition, any relaxable fluent $f \in R$ required to be true to satisfy the formula is in Δ, evaluated at some point between when it is required and the last effect on f.

We assume that any regressed formula is in negation normal form, that is, only atomic fluents are negated, not whole

expressions. Additionally we assume that the negation of a fluent is represented as a separate fluent with opposite effect axioms, i.e., $\neg f(s)$ is replaced with $f'(s)$ where f' has effect axioms $\phi^+_{f'} = \phi^-_f$ and $\phi^-_{f'} = \phi^+_f$. This is simply syntactic sugar that simplifies our formulae and proofs.

We can now use the assumptions, Δ, computed in the precondition relaxed problem to compute penalties (Lagrange multipliers). We define an *AssumptionCost* function that computes the total cost of those assumptions:

$$AssumptionCost(\lambda, \Delta) = \sum_{f(s) \in \Delta} \lambda(f, s)$$

We also define a *Bonuses* function that calculates the benefit of causing a fact, by summing those relaxed fluents that actually hold in s:[2]

$$Bonuses(R, \lambda, s) = -\sum_{f(s) \in R} \lambda(f, s) \cdot \langle \Phi_f[s] \rangle \cdot \langle lasteff(f, s) = s \rangle$$

We combine these components with the original cost to define a penalized cost function:

$$Cost_\lambda(R, \Delta, s) = Cost(s) + AssumptionCost(\lambda, \Delta) - Bonuses(R, \lambda, s)$$

We illustrate this concept on a handful of situations in the cooperative navigation domain using the following penalties:

$$\begin{aligned} \lambda_1(Unlocked(x, y), s) = 9 \quad & \subset time(s) = 0 \\ \lambda_1(Unlocked(x, y), s) = 1 \quad & \subset time(s) = 1 \\ \lambda_1(\_, s) = 0 \quad & \textbf{otherwise} \end{aligned}$$

The assumption necessary to perform

$$do([\texttt{move}(a, x, y), \texttt{move}(a, y, z)], S_0))$$

is just $\Delta = \{Unlocked(x, y, do(\texttt{move}(a, x, y), S_0))\}$. Using it, we compute the penalties: $AssumptionCost(\lambda_1, \Delta) = 1$ and no bonuses. Importantly,

$$do([\texttt{move}(b, p, y), \texttt{unlock}(b, y, z), \texttt{move}(b, y, z)], S_0)$$

still requires the assumption

$$Unlocked(x, y, do([\texttt{move}(b, p, y), \texttt{unlock}(b, y, z)], S_0))$$

in spite of the fact that the action sequence causes this assumed fluent. This ensures that all relaxable fluents are treated uniformly across all agents.

In many domains, the only rational choice is to assume the fluent in the least penalized situation where that fluent would persist to s:

$$\min_{\lambda(f,s)} (s' \mid lasteff(f, s) \preceq s' \preceq s)$$

This observation is important in reducing the computational burden of the projection problem. The obvious exception to this is when a fluent has already been assumed in order to enable an earlier action, in which case there may be no need to assume the same fluent twice. E.g., if an agent takes the route $y \to z \to q \to p \to y \to z$, the agent need only assume that $y \to z$ is unlocked once, before starting the circuit.

[2]We use $\langle$ and $\rangle$ as Iverson brackets. $\langle P \rangle$ denotes a 0-1 variable which takes the value 1 iff condition P holds.

Definition 6 (Precondition-relaxed projection). *The precondition relaxed projection problem with a set R of relaxed fluents, $\Delta \subseteq R$ of fluents assumed within the execution s, and a penalty function λ, is defined*

$$PRDo_\lambda(R, \Delta, \delta, S_0, s) \equiv \mathcal{PRR}^\Delta_R(Do(\delta, S_0, s))$$

Lemma 1. *For any choice of relaxed fluents R, any legal execution is legal in the precondition-relaxed problem, i.e.,*

$$(\mathcal{D} \models Do(\delta, S_0, s)) \supset \forall R. \exists \Delta. (\mathcal{D} \models PRDo(R, \Delta, \delta, S_0, s)),$$

Proof. $\Delta \subseteq \{f(s) \in R \mid \mathcal{R}[f(s)]\}$, the set of assumptions required is at most the set of relaxed fluents that actually hold in the original problem. □

Lemma 2. *For any choice of relaxed fluents R and penalty function λ satisfying $\forall f, s. \lambda(f, s) \geq 0$, the penalized cost of any solution s^* to the original problem is no more than the non-penalized cost.*

Proof sketch. By contradiction.

Since s^* is a solution to the non-relaxed problem, for each f which is a precondition of any action in s^*, it must hold that $\forall s \preceq s^* : f(lasteff(f, s))$, as any subsequent effect would either negate f, causing s^* not to be a solution, or it would cause f in which case that subsequent situation would be the cause of f.

Assume the penalized cost is greater than the non-penalized cost. Then there exists some precondition f assumed in some situation $s \preceq s^*$ where the cost of assuming f exceeds the bonus for causing f (because the restriction on λ means there cannot be a penalty for causing f):

$$min(\lambda(f, s') \mid lasteff(f, s) \preceq s' \prec s) > \lambda(f, lasteff(f, s))$$

But it is impossible that the minimum of a set that includes the value $\lambda(f, lasteff(f, s))$ exceeds that value, hence we have a contradiction. □

Theorem 2. *For any choice of relaxed fluents R and penalty function λ satisfying $\forall f, s. \lambda(f, s) \geq 0$,*

$$\min_{Cost_\lambda(\Delta, S)} \{S \mid \mathcal{D} \models PRDo(R, \Delta, \delta, S_0, S)\}$$

is a relaxation of $\min_{Cost(S)} \{S \mid \mathcal{D} \models Do(\delta, S_0, S)\}$.

Proof. Directly from Lemmas 1 and 2. □

For our relaxation to remain decidable, we place a slightly different restriction on the penalty function: it should have finitely-many ways that actions can have negative cost. An action can have negative cost if any precondition or postcondition has a non-zero penalty.

Theorem 3. *Let δ be a bounded-benefit program and R a set of fluents to relax such that the branching factor of δ remains bounded. For any choice of relaxed fluents R (where the branching factor remains bounded) and penalty function λ and bounded-benefit program δ. If the penalty function λ has a finite number of solutions to $\lambda(f, s) \neq 0$, and a finite total magnitude $M = \sum |\lambda(f, s)|$, then the precondition relaxed problem is itself a bounded-benefit program.*

Proof sketch. By assumption there is a bounded total benefit that can be gained, given an initial lower bound. Defining $lb'(l) = lb(l) - M$ gives a lower bound for the relaxed problem with the necessary properties. □

When lb and M are computable and known, we can use lb' to apply informed search algorithms in subproblems.

Note that when R satisfies the requirements of Theorem 3, an infinite number of penalty functions can be systematically generated. Importantly, to estimate the joint cost, one need not know any agent's program, merely be able to query agents for their optimal relaxed plan given a set of relaxed fluents and penalty function.

6. MULTI-AGENT RELAXATIONS

We now have a mechanism to generate per-agent relaxed plans from penalties. The next logical step is to merge these into a relaxed joint execution. There are several approaches to merging agent plans, the most general approach is to treat agent i's plan as a literal program: $do([\alpha_1, \ldots, \alpha_n], S_0)$ becomes $\delta_i \equiv \alpha_1; \cdots; \alpha_n$; then to merge a pair of such literal programs for agents i and j by taking the minimum cost plan generated by $Do(\delta_i \| \delta_j, S_0, s)$. Other merge operators cannot have solutions that are not also solutions to this general merge operator.

We assume that the actions in s originating from s_1 are distinguishable from those in s_2 using a function $agent(\alpha)$ that returns the agent responsible for this action. We can then define a predicate to un-merge a joint execution into a per-agent equivalent:

$$m^{-1}(agt, S_0, s') = S_0$$
$$m^{-1}(agt, do(\alpha, s), s') = (s' = do(\alpha, s) \wedge agent(\alpha) = agt) \vee (s' = s \wedge agent(\alpha) \neq agt)$$

This is an inverse of any merge operator, regardless of its definition. This also allows us to determine the agent causing a fluent in a joint execution:

$$CausedBy(f, s, agt) \equiv lasteff(f, do(\alpha, s)) = do(\alpha', s') \wedge agent(\alpha') = agt$$

A more practical approach to merging, which we use in our implementation, is temporal merging based on the temporal semantics of *MIndiGolog* [10]. Each agent's plan maintains a per-agent notion of time, and the merged plan must ensure that the actions in the resulting plan are a topological sort of the actions in the agent plans that is consistent with the timestamp for that action. that is, if two agents, i and j perform actions α_i and α_j then, for each s and s' representing a prefix of the merged joint execution,

$$do(\alpha_i, s) \preceq do(\alpha_j, s') \supset time_i(do(\alpha_i, s)) \leq time_j(do(\alpha_j, s')).$$

must hold for the merge to be temporally consistent.

In the cooperative navigation domain, under the assumption that each action takes one unit of time, a temporal consistent merge of $do([\texttt{move}(a, x, y), \texttt{move}(a, y, z)], S_0)$ and $do([\texttt{unlock}(b, x, y), \texttt{move}(b, x, p), \texttt{move}(b, p, q)], S_0)$ would be

$$do([\texttt{move}(a, x, y), \texttt{unlock}(b, x, y), \texttt{move}(a, y, z), \texttt{move}(b, x, p), \texttt{move}(b, p, q)], S_0)$$

but

$$do([\texttt{move}(a, x, y), \texttt{unlock}(b, x, y), \texttt{move}(b, x, p), \texttt{move}(b, p, q), \texttt{move}(a, y, z)], S_0)$$

would not be temporally consistent. Namely, $\texttt{move}(a, y, z)$, performed at time 2 in the per-agent plans, is performed after $\texttt{move}(b, x, p)$ in the merged plan, but this second action was performed at time 1 in the per-agent plan.

This approach massively reduces the computational overhead of merging plans, and also allows us to apply penalty functions more consistently between agents by taking time into account. For example, in the multi-agent navigation domain from Figure 1, the penalty for assuming $x \rightarrow y$ is unlocked at time 0 is high, as this is un-achievable.

In computing a multi-agent relaxation, we would like to totally avoid merging plans to compute a lower bound. To this end we define a class of (penalized) cost functions that allow us to simply sum the per-agent costs. We refer to such cost functions as "merge-consistent".

Definition 7 (Merge-consistent cost). *The penalty function $\lambda(f, s)$ and the cost function $Cost_\lambda$ are consistent with a merge operation $m(\delta_1, \delta_2)$ under a relaxed set R of fluents if for all relaxed plans s_1 and s_2, with assumption set $\Delta = \Delta_1 \cup \Delta_2 \subset R$,*

$$Cost_\lambda(R, \Delta, m(s_1, s_2)) = Cost_\lambda(R, \Delta_1, s_1) + Cost_\lambda(R, \Delta_2, s_2)$$

Merge-consistent penalties are quite easy to develop in practice within temporal domains, when penalties and costs are consistently applied at the same time points. Unfortunately this approach alone is not always sufficient to guarantee that the per-agent relaxed optima (which are the only solutions we want to consider ideally) can be merged to produce the optimal joint execution.

Referring again to Figure 1, now assume that a needs to plan a path from x to z and b one from p to z. As usual b may unlock the locked edge, and $Unlocked(y, z)$ is relaxed in all situations except S_0. The optimal solution is for both agents to travel via $y \rightarrow z$. We see the two non-penalized per-agent optima are $do([\texttt{move}(a, x, y), \texttt{move}(a, y, z)], S_0)$ and $do([\texttt{move}(b, p, q), \texttt{move}(b, q, z)], S_0)$.

No penalty function we can give to b can give any incentive to travel via $y \rightarrow z$, as b will necessarily pay the same penalty for assumption as it gets in bonuses, unless b performs some intermediate action, which would be suboptimal. The reason for this is that the precondition can benefit multiple agents simultaneously. Hence we duplicate $Unlocked(y, z)$ as $Unlocked_a(y, z)$ and $Unlocked_b(y, z)$ with identical successor state axioms. Each agent then requires only its own copy of this fluent to perform the action, and this allows the bonuses to exceed the penalties. Importantly, this transformation also guarantees that the Δs for each agent do not overlap, and therefore no assumption costs will be double counted when a single assumption could be used in the relaxed joint program $\delta_1 \| \delta_2$.

DEFINITION 8 (SHARED RELAXATION). *A shared relaxation is a domain $\mathcal{D}$, a set of fluents R and merge operator m such that*

$$\begin{aligned}\mathcal{D} \models Do(\delta_1 \| \delta_2, S_0, s) \supset \\ \exists \Delta_1, \Delta_2 : PRDo(R, \Delta_1, \delta_1, S_0, s_1) \\ \wedge\, PRDo(R, \Delta_2, \delta_2, S_0, s_2) \wedge m(s_1, s_2) = s \\ \wedge\, \forall f \neg \exists s' \preceq s : \\ f(m^{-1}(1, s')) \in \Delta_1 \wedge f(m^{-1}(2, s')) \in \Delta_2\end{aligned}$$

That is, it is a relaxed domain where sufficiently many fluents are relaxed to ensure that each agent can operate independently in the relaxed domain and thus all legal executions in the original can be generated by merging relaxed single-agent plans, with no assumptions shared between agents. This non-overlap restriction guarantees that penalties cannot be "double counted".

THEOREM 4. *Let λ be a penalty function, merge-consistent with a merge operator m under a shared relaxation R. Let A be a set of agents numbered 1 to n, let Δ_i be the set of assumptions made by agent i, and let s_i be agent i's relaxed plan. For all joint executions s which are legal in the original domain:*

$$\begin{aligned}\exists s_1 \ldots s_n : s = m(s_1, \ldots, s_n) \wedge \\ \sum_{i \in A} Cost_\lambda(R, \Delta_i, s_i) \leq Cost(s)\end{aligned}$$

PROOF SKETCH. By Definition 8, $s_1 \ldots s_n$ exist. By Definition 7, $\sum_{i \in A} Cost_\lambda(R, \Delta_i, s_i)$ is equivalent to $Cost_\lambda(R, \Delta, s)$. Namely, as no two Δ_i overlap, no penalties can be double counted. So by Lemma 2, $Cost_\lambda(R, \Delta, s) \leq Cost(s)$. □

Obviously, to be practically useful such a relaxation must be computationally easier than the original problem. Choosing such a set is an exercise for the modeler. Table 1 compares the precondition-relaxed runtime with a non-relaxed implementation in *MIndiGolog* and observe speedups in excess of 9000× in some cases. Importantly, we can see both from Definition 7 and our experimental results that the complexity of computing the relaxation for n agents is in $O(n)$ assuming the penalty function and remaining domain are unchanged.

7. CONSTRUCTING JOINT EXECUTIONS

We now have methods to guide agents towards behaviours that benefit the overall objective in the precondition relaxed problem. We would like to use a similar approach to achieve the same aim in the original problem.

We can also guide other agents away from hard to satisfy assumptions, and towards causing helpful preconditions. By violating the non-negativity assumption in Lemma 2, we can also do the converse: guide agents towards assumptions, and away from causing interfering effects.

THEOREM 5. *Given a feasible joint execution s_j such that $Do(S_0, \delta_1 \| \delta_2, s_j) \wedge s_j = m(s_1, s_2)$, there exist relaxed precondition sets R_1 and R_2 such that $PRDo(R_1, \Delta_1, S_0, \delta_1, s_1) \wedge PRDo(R_2, \Delta_2, S_0, \delta_2, s_2)$.*

PROOF SKETCH. By Lemma 1, $PRDo(R, \Delta, S_0, \delta_1 \| \delta_2, s)$ must hold for any R. If we then constrain R_i to be a superset of each precondition f of any action whose last effect was caused by a different action, that is, for all agents agt the following should hold for each fluent f true in any situation $s \preceq s_j$:

$$\begin{aligned}CausedBy(f, s, agt') \wedge f(s) \wedge agt' \neq agt \Rightarrow \\ m^{-1}(agt, s, s') \wedge [\![f(s')]\!] \in R_i \wedge [\![f(s)]\!] \in R\end{aligned}$$

Each R_i is sufficient to allow each agent to perform the action sequence required using $\mathcal{PRR}$. Any superset of R_i allows at least the same solution set. □

To construct a feasible joint execution we use a form of Lagrangian relaxation. We solve the relaxed problem for each agent and then check whether there is a feasible interleaving. If not, we will discover some relaxed fluents that are assumed but not caused by any agent. The penalty for these fluents will then be increased and the process iterated.

Consider the situation in the cooperative navigation domain, where a is at x and b is at y initially and can unlock $y \rightarrow z$. If we relax the $Unlocked(y, z)$ fluent with penalty 0, a's optimal relaxed plan is $do([\texttt{move}(a, x, y), \texttt{move}(a, y, z)], S_0)$, which is exactly the plan a should execute. However, with the same penalty, b's optimal plan is to do nothing. We observe that these action sequences cannot be interleaved to form a legal joint execution. The reason for this is that there is a relaxed fluent assumed by a that is not generated by any agent. If instead we relax the same fluent with a penalty of 2, a will perform the same action sequence (but incur 2 additional cost units of penalty), and b will perform $do(\texttt{unlock}(b, y, z), S_0)$, gaining 2 units in bonuses. Importantly, agent a has no knowledge whatsoever of agent b, and vice-versa. Even the central planner setting the penalties has no specific a priori knowledge of b's capability to unlock $y \rightarrow z$, only by giving b sufficient incentive to achieve this precondition for a is this capability discovered. All that each agent knows is that there may exist an agent capable of causing $Unlocked(y, z)$, and an agent that may rely on the same fluent.

Table 1 shows results from the relaxations used in the agent-based rail scheduling application described in [7]. In this application we schedule a set of train services on a shared rail network subject to mutual exclusion constraints (two trains cannot simultaneously occupy the same track section). In the results we present, these mutual exclusion constraints are relaxed, and we apply a temporal penalty function and merge operator. As this represents a scheduling problem, we compare our approach to a state-of-the-art solver for scheduling problems, `cpx` [18].

The penalties were determined by the optimal dual solution to a linear program modeling the mutual exclusion constraints. The per-agent relaxed plans were then incrementally added to a pool, the linear program was re-solved, and the process was iterated. This process is an application of Branch-and-Price [4]—for details, see [7]. The results illustrate the speedup that can be gained by relaxing the right preconditions, and that relax-&-merge can be used to find feasible joint executions significantly faster than existing approaches.

Table 1 shows the effectiveness of using the relax-&-merge approach to constructing feasible joint executions. It also shows the average time to solve a full joint relaxation extrapolated from the average time to solve a single agent's relaxation. The time is extrapolated because the relaxation

Agents	*Golog*	cpx	Relaxed*	Relax-&-Merge
2	0.4	0.7	0.1	1.6
4	—	1.7	0.2	2.0
8	—	7.5	1.1	7.6
16	—	37.9	4.6	29.7
32	—	—	12.5	50.3
64	—	—	25.3	150.3

Table 1: Time to first solution in seconds of the Bulk Freight Rail Scheduling Problem described in [6]. (— denotes runtime exceeds 1800s, or memory usage exceeds 4GB). (* time to *optimal* solution of the precondition relaxed problem)

was stopped early whenever an agent plan that changed the violated constraints was generated. In the table we compare a special case of relax-&-merge (fragment-based planning [7]) with (column 2) *MIndiGolog* [10], a Prolog-based multi-agent Golog interpreter; and (column 3) `cpx`, a constraint programming solver using Lazy Clause Generation [14]. The implementation of [7] was instrumented to find the proportion of time spent on relaxation (column 4). All experiments were performed on a 2.4 GHz Intel Core i3 with 4GB RAM running Ubuntu 12.04. Our implementation used Gurobi 5.1, CPython 2.7.3. We used the binary version of `cpx` included with MiniZinc 1.6.

8. RELATED WORK

Baier et al. [2] have considered the compilation of a restricted class of Golog programs using an intermediate language for capturing temporal logic preferences. This could allow PDDL3-compliant classical planners to tackle problems of a similar nature, utilising a range of well studied tractable relaxations. However, PDDL is less expressive than *Golog* in general [16]. Moreover, the application of classical approaches to multi-agent planning problems requires knowledge of each agent's goals and transition function. In contrast, the approach proposed in this paper requires only that agents can generate optimal plans given a penalty function.

Delete relaxation is the most common of these relaxations and has been applied to *Golog* to generate a heuristically good execution [5]. However [5] does not use the delete relaxation to generate lower bounds as no attempt is made to approximate the optimal solution to the relaxed problem, this could limit the accuracy of information derived from this heuristic. The alternative search strategies that are applicable to the bounded-benefit programs we introduce in this paper could allow this issue to be addressed.

In contrast to delete relaxation, our approach introduces *precondition* relaxation and demonstrates its applicability to both computing relaxations in multi-agent problems, and constructing feasible joint executions.

Abduction has been used extensively in planning, however only one published approach the authors are aware of uses abduction to synthesise plans. In this approach planning is achieved using predicates distinguished as *abducible*, and is formalised in the event calculus [19], however this work does not consider optimisation or multi-agent applications. The use of costs in conjunction with abduction has also been used in multi-agent reasoning for plan recognition [1].

9. CONCLUSIONS

We have introduced a notion of cost to the situation calculus, and described a logical framework to prove the relative quality of solutions to projection problems in *Golog* without the need to enumerate the solution space. These enable more control of the search algorithm, and with further work could allow the integration of informed search techniques from automated planning.

Our main contribution is precondition relaxations, a class of relaxations that are particularly applicable to multi-agent domains. Our experimental results show that relaxations can be chosen which yield dramatic speedups and scale linearly with increasing numbers of interacting agents. These relaxations can be usefully and systematically varied, so as to not only improve the lower bounds they generate, but also to generate feasible joint executions from relaxed per-agent plans.

We have explained the restrictions on domains where these relaxations can be applied and described some simple transformations that can be applied, to ensure these properties hold. Additionally we describe temporal merging which represents additional constraints that can be added to a domain such that the overhead of merging per-agent plans can be effectively reduced.

Importantly, these relaxations and approaches to search can be applied without knowledge of any agent's program, so long as those agents can be queried for their optimal plan under a given penalty function.

ACKNOWLEDGEMENTS

NICTA is funded by the Australian Government through the Dept. of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

REFERENCES

[1] D. Appelt and M. Pollack. Weighted abduction for plan ascription. *User Modeling and User-Adapted Interaction*, 2(1-2):1–25, 1992.

[2] J. A. Baier, C. Fritz, M. Bienvenu, and S. A. McIlraith. Beyond classical planning: Procedural control knowledge and preferences in state-of-the-art planners. In *Proc. AAAI'08*, volume 3, pages 1509–1512. AAAI Press, 2008.

[3] E. Balas. The prize collecting traveling salesman problem. *Networks*, 19(6):621–636, 1989.

[4] C. Barnhart, E. L. Johnson, G. L. Nemhauser, M. W. P. Savelsbergh, and P. H. Vance. Branch-and-price: Column generation for solving huge integer programs. *Op. Res.*, 46(3):316–329, 1998.

[5] M. L. Blom and A. R. Pearce. Relaxing regression for a heuristic GOLOG. In *STAIRS 2010: Proc. Fifth Starting AI Researchers' Symposium*, pages 37–49. IOS Press, 2010.

[6] J. Classen and G. Lakemeyer. A logic for non-terminating Golog programs. In *Principles of Knowledge Representation and Reasoning*, pages 589–599. AAAI, 2008.

[7] T. Davies, A. R. Pearce, P. J. Stuckey, and H. Søndergaard. Fragment-based planning using column generation. In *Proc. ICAPS'14*, pages 83–91, 2014.

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

[9] M. L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 50(12 supplement):1861–1871, 2004.

[10] R. F. Kelly and A. R. Pearce. Towards high level programming for distributed problem solving. In S. Ceballos, editor, *IEEE/WIC/ACM Int. Conf. Intelligent Agent Technology (IAT-06)*, pages 490–497. IEEE Comp. Soc., 2006.

[11] E. Keyder and H. Geffner. Soft goals can be compiled away. *Journal of Artificial Intelligence Research*, 36(1):547–556, 2009.

[12] C. Lemaréchal. Lagrangian relaxation. In M. Jünger and D. Naddef, editors, *Computational Combinatorial Optimization*, volume 2241 of *LNCS*, pages 112–156. Springer, 2001.

[13] K. Mehlhorn and M. Ziegelmann. Resource constrained shortest paths. In *Algorithms—ESA 2000*, volume 1879 of *LNCS*, pages 326–337. Springer, 2000.

[14] O. Ohrimenko, P. Stuckey, and M. Codish. Propagation via lazy clause generation. *Constraints*, 14(3):357–391, 2009.

[15] R. Reiter. *Knowledge in Action: Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT Press, 2001.

[16] G. Röger and B. Nebel. Expressiveness of ADL and Golog: Functions make a difference. In *Proc. AAAI'07*, volume 2, pages 1051–1056. AAAI Press, 2007.

[17] S. Sardina and G. De Giacomo. Composition of ConGolog programs. In *Proc. IJCAI'09*, pages 904–910, 2009.

[18] A. Schutt, F. Thibaut, P. J. Stuckey, and M. G. Wallace. Solving RCPSP/max by lazy clause generation. *Journal of Scheduling*, 16:273–289, 2013.

[19] M. Shanahan. An abductive event calculus planner. *Journal of Logic Programming*, 44:207–239, 2000.