Fragment-Based Planning Using Column Generation

Toby O. Davies, Adrian R. Pearce, Peter J. Stuckey, and Harald Søndergaard
National ICT Australia and
Computing & Information Systems, The University of Melbourne
Melbourne, Australia
firstname.lastname@nicta.com.au

Computing & Information Systems
The University of Melbourne
Melbourne, Australia
harald@unimelb.edu.au

Abstract

We introduce a novel algorithm for temporal planning in *Golog* using shared resources, and describe the Bulk Freight Rail Scheduling Problem, a motivating example of such a temporal domain. We use the framework of column generation to tackle complex resource constrained temporal planning problems that are beyond the scope of current planning technology by combining: the global view of a linear programming relaxation of the problem; the strength of search in finding action sequences; and the domain knowledge that can be encoded in a *Golog* program. We show that our approach significantly outperforms state-of-the-art temporal planning and constraint programming approaches in this domain, in addition to existing temporal *Golog* implementations. We also apply our algorithm to a temporal variant of blocks-world where our decomposition speeds proof of optimality significantly compared to other anytime algorithms. We discuss the potential of the underlying algorithm being applicable to *STRIPS* planning, with further work.

Introduction

The motivation for this work came from the first author's experiences of modifying an existing rail service scheduling tool to handle an apparently small change to the structure of some generated services for a bulk-freight railway serving the Australian mining industry. Modifying the 10,000 lines of C++ data structures and procedures used to generate service plans took nine months. Planning formalisms such as *Golog* and *STRIPS* are extremely general modeling techniques that make them attractive to the constantly changing needs of industrial optimisation problems. However pure heuristic search is often insufficient to solve large-scale industrial problems with hundreds of goals and thousands of time-points. We present the Bulk Freight Rail Scheduling Problem as a simplified example of such a domain.

Multi-agent temporal *Golog* (Kelly and Pearce 2006) and heuristic optimising *Golog* (Blom and Pearce 2010) have been investigated separately. The combination of these features have potential industrial applications in scheduling problems, and the ability to use domain knowledge to supplement the search for solutions is attractive, however *Golog*'s lack of powerful search algorithms has limited its applicability. In this paper we deal with this shortcoming for an important class of planning/scheduling problems.

Resource constrained planning problems are known to be challenging to solve using current technology, even in non-temporal settings (Nakhost, Hoffmann, and Müller 2012). The Divide and Evolve metaheuristic has been used to tackle temporal planning problems (Schoenauer, Savéant, and Vidal 2006), it too repeatedly solves guided subproblems but, unlike our approach, cannot prove optimality.

A key technique behind our approach is linear programming, in particular the dual, which allows us to accurately predict the cost of resource consumption. Linear programming has been used by a number of planning heuristics (van den Briel et al. 2007; Coles et al. 2008; Bonet 2013). However these heuristics have exploited only the primal solutions to the LP, whereas we use both the primal and the dual. Additionally we use the information in a way that cannot be described as a heuristic in the usual sense.

The term fragment-based planning has also been defined in the context of conformant planning (Kurien, Nayak, and Smith 2002), while the intuition is similar: plan-fragments are combined into a consistent plan, the approach is distinct from the use of the term here.

Preliminaries

The Situation Calculus and Basic Action Theories. The *situation calculus* is a logical language specifically designed for representing and reasoning about dynamically changing worlds (Reiter 2001). All changes to the world are the result of *actions*, which are terms in the language. We denote action variables by lower case letters a, and action terms by α, possibly with subscripts. A possible world history is represented by a term called a *situation*. The constant S_0 is used to denote the initial situation where no actions have yet been performed. Sequences of actions are built using the function symbol do, such that $do(a, s)$ denotes the successor situation resulting from performing action a in situation s. Predicates and functions whose value varies from situation to situation are called *fluents*, and are denoted by symbols taking a situation term as their last argument (e.g., $Holding(x, s)$).

Within the language, one can formulate *basic action theories* that describe how the world changes as the result of the available actions (Reiter 2001). These theories, combined with *Golog*, are more expressive than STRIPS or ADL

(Röger and Nebel 2007). Two special fluents are used to define a legal execution: *Poss*(a, s) is used to state that action a is executable in situation s; and *Conflict*(as, s) is used to state that the set of actions *as* may not be executed concurrently.

High-Level Programs. High-level non-deterministic programs can be used to specify complex goals: the goal of a *Golog* program is to find a sequence of actions generated by some path through the program. We use temporal semantics from MIndiGolog (Kelly and Pearce 2006) which builds on *ConGolog* (Giacomo, Lesperance, and Levesque 2000), and refer to these extensions simply as *Golog*. A *Golog* program δ is defined in terms of the following structures:

α	atomic action
$\varphi?$	test for a condition
$\delta_1; \delta_2$	sequential composition
while φ **do** δ	while loop
$\delta_1 \vert \delta_2$	nondeterministic branch
$\pi x.\delta$	nondeterministic choice of argument
δ^*	nondeterministic iteration
$\delta_1 \Vert \delta_2$	concurrent composition

In the above, α is an action term, possibly with parameters, δ is a *Golog* program, and φ a situation-suppressed formula, that is, a formula in the language with all situation arguments in fluents suppressed. 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 $\phi?$ succeeds only if ϕ holds in the current situation. Program $\delta_1\|\delta_2$ expresses the concurrent execution of δ_1 and δ_2. For notational convenience we add:

$\pi(x \in X).\delta$	equivalent to $\pi x.(x \in X)?; \delta$
`foreach` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]}; \cdots ; \delta_{[x/v_n]}$
`forconc` x `in` vs `do` δ	equivalent to $\delta_{[x/v_1]} \Vert \cdots \Vert \delta_{[x/v_n]}$

Here $\delta_{[x/y]}$ denotes the program δ where each occurrence of variable x has been replaced with the value y, and v_i is the ith element of the sequence vs.

Linear and Integer Programming. An Integer Program (IP) consists of a vector $\tilde{\boldsymbol{x}}$ of binary decision variables,[1] an objective linear expression and a set of linear inequalities. We will refer to these inequalities as "resources" throughout this paper, and each decision represented by the variable x_i can be thought of as using $u_{r,i}$ units of some resources $r \in R$, each of which has an availability of a_r.

A general form, assuming a set R of inequalities, where c_i, $u_{r,i}$ and a_r are constants, is:

$$\begin{array}{lll} \text{Minimize:} & \sum c_i \cdot x_i & \\ \text{Subject To:} & \sum u_{r,i} \cdot x_i \quad \leq a_r & \forall r \in R \end{array}$$

Finding the optimal solution to an integer program is NP-hard, however the linear program (LP), constructed by replacing the integrality constraints $x_i \in \{0, 1\}$ with a continuous equivalent $x_i \in [0, 1]$, can be optimised in polynomial time. This LP is known as the linear relaxation of the IP.

[1] In general decision variables can be integer but binary decision variables suffice for our purposes.

A model of this form where some variables are continuous and some are integral is called a Mixed Integer Program (MIP). To limit confusion, we will denote binary variables x_i and continuous variables v_i, and assume all $x_i \in \{0, 1\}$ and $v_i \in [0, 1]$ throughout this paper.

The (M)IP is solved using a "branch-and-bound" search strategy where some x_i which is fractional in the LP optimum is selected at each node and two children are enqueued with additional constraints $x_i = 1$ in one branch and $x_i = 0$ in the other. Heuristically constructed integer solutions provide an upper bound, whereas the relaxations provide a lower bound.

Solving the linear relaxation implicitly also optimises the so-called dual problem. Intuitively the dual problem is another linear program that seeks to minimize the unit price of each resource in R, subject to the constraint that it must under-estimate the optimal objective of the primal. We use π_r to denote this so-called "dual-price" of resource r. An estimate of the impact of consuming u additional units of resource r on the current objective can then be computed by multiplying usage by the dual price: $u \cdot \pi_r$.

Dual prices allow us to quickly identify bottlenecks in a system. They give us an upper bound on just how far out of the way we should consider going to avoid them. This leads to the important notion of reduced cost: an estimate of how much a decision can improve an incumbent solution. The reduced cost is the decision's intrinsic cost c_i, less the total dual price of the resources required to make this decision.

Given an incumbent dual solution $\tilde{\boldsymbol{\pi}}$, a decision variable x_i has a reduced-cost $\gamma(i, \tilde{\boldsymbol{\pi}})$, defined as:

$$\gamma(i, \tilde{\boldsymbol{\pi}}) = c_i - \sum_{r \in R} \pi_r \cdot u_{r,i}$$

This is guaranteed to be a locally optimistic estimate, so that, in order to improve an incumbent solution, we only need to consider decisions x_i with $\gamma(i, \tilde{\boldsymbol{\pi}}) < 0$. Due to the convexity of linear programs, repeatedly improving an incumbent solution is sufficient to eventually reach the global optimum, and the non-existence of any x_i with negative reduced cost is sufficient to prove global optimality.

Column Generation and Branch-and-Price. Most real-world integer programs have a very small number of non-zero x_i. This property, combined with the need only to consider negative reduced-cost decision variables, allows us to solve problems with otherwise intractably large decision vectors using a process known as "column generation" (Desaulniers, Desrosiers, and Solomon 2005). The name reflects the fact that the new decision variable is an additional column in the matrix representation of the constraints.

Column generation starts with a restricted set of decision variables obtained by some problem-dependent method to yield a linear feasible initial solution. With such a solution we can compute duals for this restricted master problem (RMP) and use reduced-cost reasoning to prune huge areas of the column space.

Incomplete and suboptimal methods of constructing integer feasible solutions are referred to in the Operations Research literature simply as "Heuristics". To avoid ambiguity we refer to them as "MIP heuristics". These are essential for both finding a feasible initial solution, and providing a worst bound on the solution during the branch-and-bound search process. They are analogues to fast but incomplete search algorithms in planning such as enforced hill climbing.

Column generation then proceeds by repeatedly solving one or more "pricing problems" to generate new decision variables with negative reduced cost and re-solving the RMP to generate new dual prices. Iterating this process until no more negative reduced cost columns exist is guaranteed to reach a fixed point with the same objective value as the much larger original linear program. We can then use a similar "branch-and-bound" approach as in integer programming to reach an integer optimum by performing column generation at each node in the search tree. This process is known as "branch-and-price".

Branching rules used in practical branch-and-price solvers are often more complex than in IP branch-and-bound and are sometimes problem dependent. The branch $x_i = 0$ does not often partition the search-space evenly: there are usually exponentially many ways to use the resources consumed by x_i. Additionally, disallowing the re-generation of the same specific solution x_i by the pricing problem is not possible with an IP-based pricing-problem without fundamentally changing the structure of the pricing-problem.

Consequently, effectiveness of a branching strategy must be evaluated in terms of how effectively the dual-price of the branching constraint can be integrated into the pricing problem. This is another motivation for our hybrid IP/Planning approach, as using a planning-based pricing problem allows us to disallow specific solutions and handle non-linear, time-dependent costs and constraints.

A concrete example of branch-and-price is presented in the Fragment-Based Planning section below.

Big-M Penalty Methods. We noted earlier that to start column generation, an initial (linear) feasible solution is required. There is no guarantee that finding such a feasible solution is trivial. Indeed for classical planning problems, finding a feasible solution is PSPACE-complete in general. No work we are aware of determines the complexity of computing a linear relaxation of a plan.

We avoid this problem by transforming our IP into an MIP where for any constraint having $a_r < 0$ we add a new continuous variable $v_r \in [0, 1]$, representing the degree to which the constraint is violated, and replace the constraint with: $\sum u_{r,i} \cdot x_i \leq |a_r| \cdot v_r - |a_r|$ and $c_v = M$ where M is a large number. This guarantees the feasibility of the trivial solution $x_i = 0$ for all i and all $v_r = 1$. It represents a relaxation of the original IP with the property that, given sufficiently large M, the optimal solution is a feasible solution to the original problem, iff such a solution exists. This is known as a "penalty method" or "soft constraint". The process is similar to the first phase of the simplex algorithm for finding an initial feasible solution to a linear program when all decision variables are known in advance.

The Bulk Freight Rail Scheduling Problem

In the Australian mining industry, it is common for bulk commodities such as coal and iron ore to be transported to ports on railways that are principally or exclusively used for that purpose. This is in contrast to Europe and the US where freight railways often share significant infrastructure with passenger trains. Additionally, bulk freight is nearly always routed as whole trains and usually only stockpiled at the mines and ports, avoiding many complex blocking and routing problems addressed in the literature.

The Bulk Freight Rail Scheduling Problem (BFRSP) is solved by bulk freight train operators and mining companies on a daily basis. The BFRSP can be an operational or tactical problem depending on the degree of vertical integration of the supply chain: where different above-rail operators share a track network they may need to prepare an accurate schedule to be negotiated with the track network operator or other above-rail operators; alternatively schedules may be prepared "just-in-time" to assign crew.

The term "train" is highly overloaded in this domain, so we avoid it. In the following, a *consist* is a connected set of locomotives and wagons, and a *service* is a sequence of movements performed by a consist.

The above-rail operator has a set of partially-specified services collected from its clients. We refer to such partially-specified services as "orders". Each order is a sequence of locations that must be visited in order with time-windows (e.g. a mine, then a port, then the yard). Given this, schedulers must find a path for as many services as possible on the rail network such that no two services occupy the same "block" at the same time. A block is represented by a vertex in the track-network graph.

The objective is to first maximise the number of orders delivered, then to minimise the total duration of services. The number of segments that a single service must traverse varies with the path through the network. Services can dwell for an arbitrary time on some edges to allow others to pass.

More formally, a BFRSP $\langle G, \tau, c, O, T \rangle$ consists of: a track network graph $G = \langle V, E \rangle$ where each block is a vertex in V; a crosstime function $\tau(\langle v, v' \rangle)$ which represents the minimum time a consist may spend at the vertex v' after traversing the edge $\langle v, v' \rangle \in E$; a capacity function $c(v)$ which represents the maximum number of services which may occupy $v \in V$ concurrently; a set of orders O, each of which is a sequence of waypoints of the form $\langle v, t_{min}, t_{max} \rangle$, where $v \in V$ and t_{min} and t_{max} represent the earliest and latest times the waypoint may be visited; and a set of consists T.

The aim of the BFRSP is to satisfy as many orders as possible, with the minimum sum of service durations. An order o is satisfied by a service s if that service visits each waypoint within the specified time window. One service must satisfy one order and be performed by one consist.

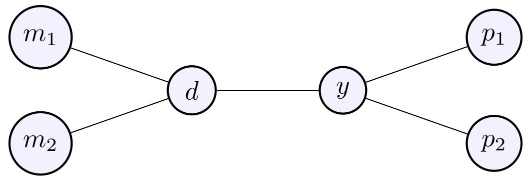

Figure 1: A simple track network

Modeling the BFRSP

We model the problem as a planning problem in *Golog*. The primitive actions available to a consist c are: traverse an edge e with $\texttt{move}(c, e)$; wait for some integral number d of time points with $\texttt{wait}(c, d)$; and perform whatever action is required at a block b (e.g. loading at a mine or unloading at a port) that is a waypoint in some order o with $\texttt{visit}(c, o, b)$. We assume unit duration for these actions.

The following are effect axioms for the domain, which may be transformed into successor state axioms to address the frame problem in the standard way (Reiter 2001). We assume the existence of some predicates to identify the first, last and next waypoints in an order: $\mathit{first}(o, b)$, $\mathit{last}(o, b)$ and $\mathit{next}(o, b, b')$

$$\begin{aligned}
&\mathit{At}(c, b, \mathit{do}(\mathit{move}(c, \langle _, b\rangle), _)).\\
\neg&\mathit{At}(c, b, \mathit{do}(\mathit{move}(c, \langle _, b'\rangle), _)) \subset b \neq b'.\\
&\mathit{Started}(c, o, \mathit{do}(\mathit{visit}(c, o, b), _)) \subset \mathit{first}(o, b).\\
&\mathit{Finished}(c, o, \mathit{do}(\mathit{visit}(c, o, b), _)) \subset \mathit{last}(o, b).\\
&\mathit{Visited}(c, o, b, \mathit{do}(\mathit{visit}(c, o, b), _)).
\end{aligned}$$

Golog effect axioms are essentially first-order predicate logic with data constructors. For example, in the first fluent definition above, '*move*' is a data constructor, whereas '*At*' is a predicate. Hence '$\mathit{At}(c, b, \mathit{do}(\mathit{move}(c, \langle \_, b\rangle), \_))$' reads "consist c is at block b in the situation that results from moving along an edge ending at b."

We need to keep track of allowable next visitable destinations for each consist. A consist c is allowed to visit the first waypoint of an order that no consist has started, so long as c has not started any order; alternatively it may visit the next waypoint in the order it has already started.

$$\begin{aligned}
&\mathit{Dest}(c, o, b, s_0) \subset \mathit{first}(o, b).\\
&\mathit{Dest}(c, o, b, \mathit{do}(\mathit{visit}(c, o', b'), s)) \subset\\
&\qquad \mathit{first}(o, b) \wedge \mathit{last}(o', b') \wedge \neg \mathit{Visited}(_, o, b, s).\\
&\mathit{Dest}(c, o, b, \mathit{do}(\mathit{visit}(c, o, b'), s)) \subset next(o, b, b').\\
\neg&\mathit{Dest}(c, o, b, \mathit{do}(\mathit{visit}(_, o, b), _)).
\end{aligned}$$

Finally, we assume the existence of a $\mathit{time}(s)$ fluent to define our time window checks.

$$\begin{aligned}
&\mathit{InWindow}(o, b, s) \subset\\
&\qquad \exists t, t'. \langle b, t, t'\rangle \in o \wedge t \leq \mathit{time}(s) \leq t'.
\end{aligned}$$

Obviously a consist cannot *move* on an edge that starts at a different node to the end of the last edge it traversed, and a consist can only visit a block that it is on. A waypoint (block) can only be visited once for an order and a consist cannot visit another order's waypoint until it has visited all the waypoints required for any order it has visited any waypoints of (i.e., consists cannot interleave visiting waypoints for different orders).

$$\begin{aligned}
&\mathit{Poss}(\mathit{move}(c, \langle b, b'\rangle), s) \subset \mathit{At}(c, b, s) \wedge \langle b, b'\rangle \in E.\\
&\mathit{Poss}(\mathit{visit}(c, o, b), s) \subset\\
&\qquad \mathit{Dest}(c, o, b, s) \wedge \mathit{InWindow}(o, b, s).\\
&\mathit{Poss}(\mathit{wait}(c, n), s) \subset n \in \mathbb{Z}.
\end{aligned}$$

Solving for a single service with a single waypoint is a case of finding a minimum-cost path through G subject to the time-window constraints, where the edge cost is given by τ. The *Golog* program to solve this problem is:

$$\begin{aligned}
&\texttt{proc travelto}(c, o, b, t0, t1):\\
&\quad \texttt{while } \neg\texttt{onblock}(c, b) \texttt{ do}\\
&\qquad \pi(e \in E).\, \texttt{move}(c, e);\\
&\quad \pi(t \in [t0, t1]).\, \texttt{waituntil}(t);\\
&\quad \texttt{visit}(c, o, b)
\end{aligned}$$

To deliver a more complex service with more waypoints we simply `travelto` them in sequence:

$$\begin{aligned}
&\texttt{proc deliverservice}(c, o):\\
&\quad \texttt{foreach } (b, t0, t1) \texttt{ in } o \texttt{ do}\\
&\qquad \pi(dt \in [0..t1 - \texttt{time}()]).\texttt{wait}(dt);\\
&\qquad \texttt{travelto}(c, o, b, t0, t1)
\end{aligned}$$

Up to this point, the problem is simple; existing implementations of *Golog* are able to find high-quality plans on simple networks without even considering action costs. Also most of the complex preconditions of the domain are irrelevant: it is impossible to interleave the satisfaction of different orders and no two trains can occupy the same block.

The *Poss* and *Conflict* clauses required to model these inter-order constraints are easier to model in terms of shared resources. We introduce $\mathit{usage}(\mathit{action}, \mathit{resource}, \mathit{situation})$ and $\mathit{availability}(\mathit{resource}, \mathit{situation})$ functions. These can be considered as additional clauses of the definition of *Conflict* and *Poss* in the following way:

$$\begin{aligned}
&\mathit{Conflict}(as, s) \subset\\
&\qquad \exists r. \sum_{a \in as} \mathit{usage}(a, r, s) > \mathit{availability}(r, s)\\
\neg&\mathit{Poss}(a, s) \subset \exists r. \mathit{usage}(a, r, s) > \mathit{availability}(r, s)
\end{aligned}$$

For the BFRSP domain we define these resources:

$$\mathit{usage}(\mathit{enter}(c, b), \mathit{block}(b, t), s) = 1 \subset \mathit{time}(t, s).$$

for block capacity constraints; and, to ensure that each order is delivered at most once:

$$\mathit{usage}(\mathit{visit}(c, o, b), \mathit{waypoint}(b, o), s) = 1.$$

When there are a set of orders, but only one consist, the consist must still deliver the orders in some sequence, however there exist $n!$ sequences in which n orders may be satisfied. Any of these sequences is valid (as any service can be dropped), so we begin to tackle the combinatorial optimisation aspect of the BFRSP, in which the solver must pick an optimal sequence to satisfy orders.

$$\begin{array}{l}\texttt{proc deliverservices}(c, O):\\ \quad \pi(o \in O).\\ \quad\quad (\texttt{deliverservice}(c, o) \mid noop);\\ \quad\quad \texttt{deliverservices}(c, O \backslash \{o\})\end{array}$$

With more than one consist, each consist must deliver some non-overlapping set of services. Choosing this set is an optimisation problem in its own right and existing implementations of *Golog* will not find any solution.

$$\begin{array}{l}\texttt{proc main}(C, O):\\ \quad \texttt{forconc } o \texttt{ in } O \texttt{ do}\\ \quad\quad \pi(c \in C).\,\texttt{deliverservice}(c, o) \mid noop\end{array}$$

This program has an important structural property that will be exploited by our algorithm: Given an assignment of orders, the joint execution is a set of fragments generated by

$$\pi(o \in O).\pi(c \in T).\,\texttt{deliverservice}(c, o) \mid noop$$

Fragment-Based Planning

Given the set F of all possible executions of the fragment program $\pi(c \in T).\texttt{deliverservice}(c, o)$, all joint executions of `main` are a subset of F where no two fragments visit the same order or occupy the same track segment simultaneously.

Finding the optimal execution is then equivalent to finding the optimal solution to the Integer Program:

$$\begin{array}{lll}\text{Minimize:} & \displaystyle\sum_{f \in F} d_f \cdot x_f + \sum_{o \in O} M \cdot v_o & \\ \text{Subject To:} & \displaystyle\sum_{f \in F_{bt}} x_f \le c(b) & \forall b \in B, \forall t \in T \\ & \displaystyle v_o + \sum_{f \in F_o} x_f = 1 & \forall o \in O\end{array}$$

Here x_f is 1 iff f should be executed in the joint plan. $F_o \subseteq F$ is the set of fragments that satisfy order o, $F_{bt} \subseteq F$ is the set of fragments that are on block b at time t and d_f is the duration of fragment f.

Enumerating F is however prohibitive, and large numbers of potential fragments are uninteresting, or prohibitively costly and will never be chosen in any reasonable joint-execution, nor need to be considered in finding and proving the optimal joint execution.

To avoid enumerating F, we can use delayed column generation as described earlier. To use this approach, we need to re-compute action costs that minimise the reduced cost of the next fragment generated given an optimal solution to the restricted LP. Given dual prices $\pi_{b,t}$ for block/time resources and $\pi_{b,o}$ for waypoints, our fragment planner's action costs become $\pi_{b,t}$ for *move*(c, e), actions executed at time t, given $e \in b$; and $\pi_{b,o}$ for $\texttt{visit}(c, o, b)$ actions.

We provide pseudo-code for the FBP algorithm below, Quine quotes are used around linear expressions such as $[\![expression \le constant]\!]$ to denote constraints given to the LP solver to distinguish them from logical expressions.

To solve the linear relaxation of the joint planning problem, we call $\textsc{LinFBP}(\{v_o \mid o \in O\}, \{o : [\![v_o = 1]\!] \mid o \in O\}, O, \delta)$. Note that the $[\![expr = a]\!]$ form of constraints can be considered a shorthand for two constraints $[\![expr \le a]\!]$ and $[\![-expr \le -a]\!]$.

We assume that the *Golog* search procedure Do returns the fragment f with the least reduced cost $\gamma(f, \tilde{\boldsymbol{\pi}})$, rather than just any valid execution. Our implementation uses uniform-cost search to achieve this.

function $\textsc{LinFBP}(\mathit{Frags}, \mathit{Res}, \mathit{Goals}, \delta)$
  $\mathit{LowBound} \leftarrow 0$
  $\mathit{UpBound} \leftarrow M \cdot \|\mathit{Goals}\|$
  while $(1-\epsilon) \cdot \mathit{UpBound} > \mathit{LowBound}$ **do**
    $\theta, \tilde{\boldsymbol{x}}, \tilde{\boldsymbol{\pi}} \leftarrow \mathit{SolveLP}(\mathit{Frags}, \mathit{Res})$
    $F \leftarrow Do(\pi(g \in \mathit{Goals}).\delta | noop, \tilde{\boldsymbol{\pi}})$
    $\mathit{Frags} \leftarrow \mathit{Frags} \cup \{F\}$
    $d\theta \leftarrow \|\mathit{Goals}\| \cdot \gamma(F, \tilde{\boldsymbol{\pi}})$
    $\mathit{UpBound} \leftarrow \theta$
    $\mathit{LowBound} \leftarrow max(\mathit{LowBound}, \theta + d\theta)$
    for all $r \in F$ **do**
      $[\![e \le a]\!] \leftarrow \mathit{Res}[r]$
      $\mathit{Res}[r] \leftarrow [\![e + u_{F,r} \cdot x_F \le a]\!]$
  return $\theta, \mathit{Frags}, \mathit{Res}, \tilde{\boldsymbol{x}}$

We then use the $\textsc{LinFBP}$ column generation implementation inside a branch-and-price search. We assume that fragments use redundant resources for the purposes of branching. In particular we rely on each fragment to use 1 unit of a resource that uniquely identifies that fragment, so that we eventually find an integral solution if one exists.

function $\text{FBP}(\mathit{Gs}, \delta)$
  $\mathit{Fs} \leftarrow \{v_g \mid g \in \mathit{Gs}\}$
  $\mathit{Rs} \leftarrow \{g : [[1 \cdot v_g = 1]] \mid g \in \mathit{Goals}\}$
  $\mathit{LowBound} \leftarrow 0$
  $\mathit{UpBound} \leftarrow M \cdot \|\mathit{Goals}\|$
  $\mathit{Queue} \leftarrow \{\emptyset\}$
  $\mathit{Fs} \leftarrow \mathit{Fs} \cup \mathit{initfrags}(\mathit{Gs}, \delta)$
  while $(1-\epsilon) \cdot \mathit{UpBound} > \mathit{LowBound}$ **do**
    $\mathit{BranchRs} \leftarrow Pop(\mathit{Queue})$
    $\mathit{LRs} \leftarrow \mathit{Rs} \cup \mathit{BranchRs}$
    $\theta, \mathit{Fs}, \mathit{Rs}, \tilde{\boldsymbol{x}} \leftarrow \textsc{LinFBP}(\mathit{Fs}, \mathit{LRs}, \mathit{Gs}, \delta)$
    if any resources have fractional usage **then**
      if soft constraints satisfied i.e., $\theta < M$ **then**
        $X \leftarrow$ some fractional resource
        $\mathit{Branches} \leftarrow$ branch on $\lceil X \rceil$ and $\lfloor X \rfloor$
        $\mathit{Queue} \leftarrow \mathit{Queue} \cup \mathit{Branches}$
    $\mathit{UpBound} \leftarrow \mathit{SolveIP}(\mathit{Fs}, \mathit{LRs})$
    $\mathit{LowBound} \leftarrow$ minimum θ in Queue

This process can be modified to incrementally return each solution to $\mathit{SolveIP}(\mathit{Fs}, \mathit{Rs})$ as it is computed, and make this an effective anytime algorithm.

We illustrate the iterations of the $\textsc{LinFBP}$ algorithm in Table 1 using a goal non-satisfaction penalty of 100. We solve a BFRSP on the network from Figure 1 with

Iteration		$\tilde{\pi}$	Fragments
1.	$\theta = 200$ $\gamma(f_1) = -91$ $\gamma(f_2) = -91$		f_1, f_2
2.	$\theta = 109$ $\gamma(f_3) = -90$ $\gamma(f_4) = -90$	$\pi_{d,0} = -91$	f_3, f_4
3.	$\theta = 19$	$\pi_{d,3} = -1$ $\pi_{p_1,o_1} = -90$ $\pi_{p_2,o_2} = -90$	

Table 1: LINFBP iterations. $M = 100$

two orders: $o_1 = [\langle m_1, 0, 20\rangle, \langle p_1, 0, 20\rangle, \langle y, 0, 20\rangle]$ and $o_2 = [\langle m_2, 0, 20\rangle, \langle p_2, 0, 20\rangle, \langle y, 0, 20\rangle]$. We omit the first consist argument to all actions and the duals of the goal satisfaction constraints for brevity.

Initially, the dual vector has no non-zero elements other than the goal non-satisfaction constraints. The lowest reduced-cost plans for the two sub-goals are then simply the shortest paths from y to one of the mines m_1 or m_2, then one of the ports p_1 or p_2, then to the yard. That is,

$$f_1 = \begin{bmatrix} move(\langle y, d\rangle); move(\langle d, m_1\rangle); visit(m_1, o_1); \\ move(\langle m_1, d\rangle); move(\langle d, y\rangle); move(\langle y, p_1\rangle); \\ visit(p_1, o_1); move(\langle p_1, y\rangle); visit(y, o_1) \end{bmatrix}$$

$$f_2 = \begin{bmatrix} move(\langle y, d\rangle); move(\langle d, m_2\rangle); visit(m_2, o_2); \\ move(\langle m_2, d\rangle); move(\langle d, y\rangle); move(\langle y, p_2\rangle); \\ visit(p_2, o_2); move(\langle p_2, y\rangle); visit(y, o_2) \end{bmatrix}$$

Given these two new columns, the dual vector detects one of the bottlenecks at block d at time 0, overestimating that over-use of this resource costs 91 units. Using this information the new minimum cost way to achieve each goal now avoids this shared resource by initially waiting, generating fragments $f_3 = [wait(1)] +\!+ f_1$ and $f_4 = [wait(1)] +\!+ f_2$.

With these two additional fragments, the dual vector shows us that the unloading *visit* actions at p_1 and p_2 are a bottleneck (as we now have two ways to satisfy each goal). Obviously these bottlenecks cannot be avoided, however they have a cost of only 90, so if there is a fragment with cost < 10 which also does not use block d at time 3, there could exist a fragment with negative reduced cost. The search shows that no such fragment exists, and this proves (linear) optimality.

Within the FBP algorithm we then solve the IP with the 4 fragments f_1 to f_4, we find one of the primal integral solutions $x_1 = 1$ and $x_3 = 1$ and all other variables 0. Since this solution has the same objective of 19 as the linear relaxation, we have proved this solution optimal.

Fragment-Based Planning in other domains

The resources we use in the BFRSP are only consumed by actions, and had a finite, non-renewable availability. Additionally, fragments can only have negative interactions between one another. The choice of one fragment can only prevent the choice of another. However we would like to be able to choose fragments such that one may satisfy an open precondition of another.

We handle this case in a similar way to goal-satisfaction constraints by generalising our resources so that they can be generated in addition to being consumed. While we did not call our goal-satisfaction constraints resources, it can be considered that the fragments in the BFRSP generated 1 unit (i.e., consumed -1 units) of a "goal-satisfied" resource, which had a requirement of 1 unit (i.e., an availability of -1 unit).

If we consider the case where a set of fragments F_{pre} satisfies a precondition of F_{post}, then a constraint of the form

$$\sum F_{post} - M \cdot \sum F_{pre} \leq 0$$

is sufficient to guarantee that fragments from F_{post} can only be chosen if the precondition has been satisfied. If F_{post} fragments immediately invalidate this precondition then a constraint of the form

$$\sum F_{post} - \sum F_{pre} \leq 0$$

is more appropriate, and forces at most a single f_{post} to be chosen per time the precondition is satisfied. In either case, as most resources (including preconditions) will be time-indexed (like the block constraints in BFRSP) in many domains there will likely be constraints forcing $\sum F_{post} \leq 1$.

Which form to choose is an exercise for the modeler, and an algorithm for turning a basic action theory into resource-based constraints is beyond the scope of this paper.

Both of the forms fall into our existing resource framework, and can be handled without modification to the FBP procedure. However the *Poss* fluent now needs to be relaxed to allow fragments to assume that a precondition has been satisfied by another fragment, at a cost derived from the π_{pre} dual variables.

If we know each action's resource consumption from the *usage* function described above, and the usage is independent of the actions performed in other fragments, then the usage of a resource r by a fragment f is the sum over all actions a performed in that fragment of $usage(a, r_{time(f)}, f)$.

Given this, a *Golog* program of the form `forallconc` x `in` G `do` δ, can generate an equivalent FBP model:

$$\text{Minimize:} \quad \sum_{f \in F} c_f \cdot x_f + \sum_{g \in G} M \cdot v_g$$

$$\text{Subject To:} \quad \sum_{f \in F} u_{fr} \cdot x_f \leq a_r \qquad \forall r \in R$$

$$v_g + \sum_{f \in F_i} x_f = 1 \qquad \forall g \in G$$

F_g is the set of all legal fragments generated by the *Golog* program $\delta_{[x/g]}$. F is the union for all $g \in G$ of F_g . R is the set of all resources used by any fragment in F. c_f is the cost of executing the actions contained in fragment f. $x_f \in \{0, 1\}$ is 1 iff f should be executed in the joint plan. u_{fr} is the net usage of resource r by fragment f. a_r is the total availability of resource r. π_r will denote the optimal dual prices of each resource r in the solution of this problem.

We refer to the set G as the "fragmentation dimension" and δ as the "fragment program".

If these resources totally describe the potential interactions between the different sub-goals $\delta_{[x/g]}$, then the optimal solution to this FBP model is the optimal joint execution.

We present results in the next section for a multi-agent variant of blocks-world. We use the set of blocks as the fragmentation dimension, and pseudocode for the fragment program in this example is:

proc handleblock(hand, block):
$\quad$ $\texttt{wait}(1)*$;
$\quad$ $(\texttt{pickup}(hand, block); \texttt{drop}(hand, table)|\texttt{noop})$;
$\quad$ $\texttt{wait}(1)*$;
$\quad$ $(\texttt{pickup}(hand, block); \texttt{drop}(hand, dest(block))|\texttt{noop})$;
$\quad$ $On(block, dest(block))?$

Given this fragment program, we can see that other fragments will be required to satisfy the *Clear* fluent required by pickup and drop. Consequently $\mathit{Clear}_{b,t}$ becomes a generatable resource for each block at each time point, with initial availability of 1 if both $t = 0$ and b is initially clear; and 0 otherwise.

Performing the action pickup(h, b) at time t consumes one unit of $\mathit{Clear}_{b,t}$ and generates one unit of $\mathit{Clear}_{b',t+1}$, assuming b was on b'.

Similarly, $\mathit{Occupied}_{h,t}$ is a (non-generatable) resource with availability 1 which is consumed by picking up a block with hand h or waiting when h is holding a block.

The actions we have described thus far produce *Clear* resources at specific time points, however the clearness of a block persists if it is not picked up and nothing is put down on it. To handle this we introduce an explicit zero-duration persist(r) action, which consumes one unit of r_t and generates one unit of $r_{time()}$ for some non-deterministically chosen $t < time()$. These persist(Clear_b) actions are inserted immediately before the pickup and drop actions in handleblock above.

Results

We compare our FBP implementation in these two domains with: MIndiGolog (Kelly and Pearce 2006), a temporal golog interpreter; POPF2 (Coles et al. 2010) and YAHSP2 (Vidal 2011) two heuristic satisficing planners; CPT4 (Vidal and Geffner 2006), a temporal planner based on constraint programming; and CPX, a constraint programming solver using Lazy Clause Generation (Ohrimenko, Stuckey, and Codish 2009). 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. All of the temporal planners were the versions used in the 2011 IPC.

We present results from a simple variant of the BFRSP with a fixed number of waypoints per order: a mine, a port and then the train yard. Results in Tables 2 and 3 represent results on the track network depicted in Figure 1 and a high level network of an Australian mining company. We assume the capacity at all nodes except y is 1, and the y has unbounded capacity, and there are as many consists as orders. The "m/p" column is the number of mines each port makes orders for. All models of the problem were given symmetry-breaking constraints regarding assignment of consists to orders. We report "time to first solution" in these experiments, meaning the time taken to generate a schedule that delivered all orders.

Table 2 shows that the FBP approach scales to problems more than an order of magnitude larger than either a MiniZinc model based on scheduling constraints solved with the lazy clause generation solver cpx, or a temporal PDDL model solved with popf2. The constraint-programming technique of Lazy Clause Generation is state-of-the-art for many scheduling problems, and cpx is competitive with our approach until memory limits were reached.

There is a significant overhead of the FBP algorithm in the smaller models, this is likely explained by our pure python implementation when compared with the highly engineered, compiled implementations of both popf2 and cpx. Unscientifically, interpreted python can expect a $30\times$ to $100\times$ slowdown compared to an equivalent algorithm in C or C++.

The temporal planner popf2 was chosen because it was the best performing of the temporal satisficing planners in IPC 2011 with support for numeric fluents or initial timed literals one of which is required to model time-windows.

popf2 performs very poorly when there is more than one order to any mine or port. We presume this is because standard planning heuristics cannot effectively detect the bottleneck in the track network at block d in Figure 1.

However, popf2's handling of time windows has limited it's performance in similar domains (Tierney et al. 2012), to rule this out, we relax the complicating time-window constraints, which also allows us to test other temporal planners on this domain where we see similar results. Table 3 shows that, as contention for block d increases, solution times in the heuristic search planners increase very sharply. This leads to the somewhat surprising result that a decomposition approach outperforms heuristic search as interaction increases. We also outperform the constraint programming approach used in cpt4, note that this is an optimal planner rather than the anytime algorithms implemented in yahsp2 and popf2.

This result is not seen in the blocks-world domain where both enforced hill climbing and Weighted-A* find solutions very quickly. However we still see significant speedups in proof of optimality versus both popf2 and yahsp2 even in this quite sequential domain. It should be noted that neither are optimal planners, but as both are complete anytime algorithms, like FBP, we believe this is still a meaningful comparison. FBP also proves optimality faster than the optimal temporal planner cpt on the largest instance.

This is obviously not a totally fair comparison, as the *Golog* model can encode more domain knowledge, however we believe this is a desirable property in a modeling language, and closer to a real-world optimisation setting, where modellers can and will provide as much useful information to any solver they use as is feasible. Additionally, existing *Golog* implementations do not perform well on either of these problems, and PDDL planners are among the most well studied alternatives and provide the fairest comparison that can be considered state-of-the-art.

$\|V\|$	$\|O\|$	m/p	*Golog*	popf2	cpx	FBP
6	2	1	0.4	**0.3**	0.7	1.6
6	4	1	—	—	**2.3**	3.3
6	4	2	—	—	**1.7**	2.0
6	8	2	—	—	**7.5**	7.6
6	16	2	—	—	37.9	**29.7**
6	32	2	—	—	—	**50.3**
6	64	2	—	—	—	**150.3**
6	128	2	—	—	—	**418.8**
6	256	2	—	—	—	**589.7**
16	44	11	—	—	—	**315.0**
16	88	11	—	—	—	**363.0**

Table 2: BFRSP: Time to first solution for increasing track network vertices, $|V|$, and orders, $|O|$, in seconds (1800s time limit, 4GB memory)

$\|V\|$	$\|O\|$	m/p	popf2	yahsp2	cpt4	FBP*
6	2	1	0.3	**0.0**	**0.0**	1.6
6	4	1	8.4	**0.0**	0.8	3.3
6	4	2	—	**0.0**	0.7	2.0
6	8	2	—	**0.3**	—	7.6
6	16	2	—	42.5	—	**29.7**
6	32	2	—	—	—	**50.3**

Table 3: BFRSP without time windows: Time to first solution in seconds (time limit 1800s, 4GB memory, FBP results are from the non-relaxed problem)

Blocks	popf		yahsp2		cpt		FBP	
	1st	opt	1st	opt	1st	opt	1st	opt
3	0.0	3.5	0.0	0.0	0.0	0.0	0.3	0.5
4	0.0	53.0	0.0	0.0	0.0	0.0	0.3	0.7
5	0.0	—	0.0	0.1	0.0	0.1	0.2	0.6
6	0.1	—	0.0	6.4	0.1	0.1	0.5	0.8
7	0.9	—	0.1	—	0.2	0.2	0.6	0.8
8	0.1	—	0.0	—	0.4	0.4	1.0	1.9
9	17.9	—	0.0	—	1.7	1.7	5.0	6.2
10	0.3	—	0.0	—	1.7	1.7	1.3	2.0
11	1.4	—	0.0	—	5.2	5.2	2.0	3.8
12	—	—	0.0	—	12.6	12.6	4.9	6.8

Table 4: Blocks-world: Time to first / optimal solutions in seconds (time limit 120s, 4GB memory)

Conclusions and Further Work

We see from our experimental results that our Fragment-Based Planning approach scales to an important class of industrial problems while sacrificing little of the flexibility of the underlying planning formalism, and *Golog* language.

Our approach fares significantly better on the BFRSP than on blocks world. We believe this is because this domain combines the strength of the two core technologies of state-based search and MIP: the MIP detects global, largely sequence-independent bottlenecks and guides the overall search; and state-based search handles the sequential aspects of shortest path finding that might otherwise require a large number of variables to encode as an LP. Whereas, in blocks world, the LP detects the desired sequence of block handling, and the search solves the relatively trivial problem of how much waiting is required to ensure the blocks are picked up and put down at the optimal time. We suspect that this is a bad model for FBP, as it splits the work unevenly between master and subproblems, and relies on the LP to detect impossible sequences, something that even uninformed search should be better at. More work is required to explore what fluents are best to relax into linear constraints; what forms those constraints should take; the choice of fragmentation dimension; and the fragment program. Nonetheless, this approach has proven effective in this sequential domain.

Our implementation lacks a number of significant engineering improvements that could be applied, most significantly: better MIP heuristics; and better branching strategies. We use a simple variant of the general diving MIP heuristics described in (Joncour et al. 2010), though problem specific MIP heuristics could yield better performance, such as (Jampani and Mason 2008).

Additionally we use very simplistic branching rules: we branch only on individual fragments being included, rather than e.g. pairs of resources as in Ryan-Foster branching (Ryan and Foster 1981). This leads to a very unbalanced branch-and-price tree and can lead to exponentially increased runtimes if the problem is not proven optimal at the root. Note that while the FBP algorithm we describe allows Ryan-Foster branching, our implementation does not branch on such "good" redundant resources.

Cost-optimal planning in *Golog* is required to solve the pricing problem in FBP. This is not a problem considered in the existing literature, and FBP might benefit from analogues to classical planning heuristics. Alternatively, as the pricing problem has a cost bound, bounded cost search algorithms for *Golog* might be beneficial. This is still an open area of research even for *STRIPS* planning (Haslum 2013).

Existing fast but incomplete algorithms such as enforced hill climbing or causal chains (Lipovetzky and Geffner 2009) might compute a better starting set of fragments and/or inspire FBP-specific MIP heuristics.

Many of these enhancements are well studied and implemented in *STRIPS* planners, and only the *Do* call in the FBP algorithm is *Golog*-specific. If similar explicit relaxations and fragment goals can be provided, this could be replaced by a call to an optimal planner, with additional `assume-`[*fact*]`(?time, ...)` actions for each fact with the action cost derived from the dual prices.

The effect of these time-dependant costs on the performance of *STRIPS* planners would also need to be investigated. Few resources will normally have non-zero dual prices, which could help limit the branching factor.

How to select the fragment goals is less obvious. Planning with preferences could be used to choose some subset of goals to achieve, however it is not obvious how to ensure this planning problem is easier than the original.

Additionally, in order for FBP to work directly with PDDL, work will be required to automatically identify: a "fragmentation dimension"; "fragment generator"; and the fluents to relax. Approaches used in SGPlan (Chen, Wah, and Hsu 2006) and factored planning (Amir and Engelhardt 2003; Brafman and Domshlak 2006) appear promising. Neither of these are anytime algorithms, so combining their decompositions with FBP's search algorithm could be an interesting avenue of further work.

Acknowledgements

Thanks to Nir Lipovetzky for his invaluable help modeling the BFRSP in PDDL, and to Biarri Optimisation Pty. Ltd. for sharing their experience with the BFRSP. NICTA is funded by the Australian Government through the Department of Communications and the Australian Research Council through the ICT Centre of Excellence Program.

References

Amir, E., and Engelhardt, B. 2003. Factored planning. In *18th International Joint Conference on Artificial Intelligence (IJCAI)*, 929–935.

Blom, M. L., and Pearce, A. R. 2010. Relaxing regression for a heuristic GOLOG. In *STAIRS 2010: Proceedings of the Fifth Starting AI Researchers' Symposium*, 37–49. Amsterdam, The Netherlands: IOS Press.

Bonet, B. 2013. An admissible heuristic for SAS+ planning obtained from the state equation. In Rossi, F., ed., *International Joint Conference on Artificial Intelligence (IJCAI)*, 2268–2274. IJCAI/AAAI.

Brafman, R. I., and Domshlak, C. 2006. Factored planning: How, when, and when not. In *ICAPS Workshop on Constraint Satisfaction Techniques for Planning and Scheduling Problems*, 7–14.

Chen, Y.; Wah, B. W.; and Hsu, C.-W. 2006. Temporal planning using subgoal partitioning and resolution in SGPlan. *Journal of Artificial Intelligence Research* 26:323–369.

Coles, A. I.; Fox, M.; Long, D.; and Smith, A. J. 2008. A hybrid relaxed planning graph-LP heuristic for numeric planning domains. In *Proceedings of the Eighteenth International Conference on Automated Planning and Scheduling (ICAPS 08)*, 52–59.

Coles, A. J.; Coles, A. I.; Fox, M.; and Long, D. 2010. Forward-chaining partial-order planning. In *Proceedings of the Twentieth International Conference on Automated Planning and Scheduling (ICAPS 10)*, 42–49. AAAI Press.

Desaulniers, G.; Desrosiers, J.; and Solomon, M. M. 2005. *Column Generation*. Springer.

Giacomo, G. D.; Lesperance, Y.; and Levesque, H. J. 2000. ConGolog, a concurrent programming language based on the situation calculus. *Artificial Intelligence* 121(1-2):109–169.

Haslum, P. 2013. Heuristics for bounded-cost search. In *Proceedings of the Twenty-third International Conference on Automated Planning and Scheduling (ICAPS 13)*, 312–316.

Jampani, J., and Mason, S. J. 2008. Column generation heuristics for multiple machine, multiple orders per job scheduling problems. *Annals of Operations Research* 159(1):261–273.

Joncour, C.; Michel, S.; Sadykov, R.; Sverdlov, D.; and Vanderbeck, F. 2010. Column generation based primal heuristics. *Electronic Notes in Discrete Mathematics* 36:695–702.

Kelly, R. F., and Pearce, A. R. 2006. Towards high level programming for distributed problem solving. In Ceballos, S., ed., *IEEE/WIC/ACM International Conference on Intelligent Agent Technology (IAT-06)*, 490–497. IEEE Computer Society.

Kurien, J.; Nayak, P. P.; and Smith, D. E. 2002. Fragment-based conformant planning. In *AIPS*, 153–162.

Lipovetzky, N., and Geffner, H. 2009. Inference and decomposition in planning using causal consistent chains. In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 09)*, 217–224.

Nakhost, H.; Hoffmann, J.; and Müller, M. 2012. Resource-constrained planning: A Monte Carlo random walk approach. In *Proceedings of the Twenty-second International Conference on Automated Planning and Scheduling (ICAPS 12)*, 181–189.

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

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

Röger, G., and Nebel, B. 2007. Expressiveness of ADL and Golog: Functions make a difference. In *Proceedings of the 22nd National Conference on Artificial Intelligence - Volume 2*, AAAI'07, 1051–1056. AAAI Press.

Ryan, D. M., and Foster, B. A. 1981. An integer programming approach to scheduling. In Wren, A., ed., *Computer Scheduling of Public Transport Urban Passenger Vehicle and Crew Scheduling*. Amsterdam, The Netherlands: North Holland. 269–280.

Schoenauer, M.; Savéant, P.; and Vidal, V. 2006. Divide-and-evolve: A new memetic scheme for domain-independent temporal planning. In Gottlieb, J., and Raidl, G. R., eds., *Evolutionary Computing in Combinatorial Optimization*, volume 3906 of *Lecture Notes in Computer Science*, 247–260. Springer.

Tierney, K.; Coles, A. J.; Coles, A.; Kroer, C.; Britt, A. M.; and Jensen, R. M. 2012. Automated planning for liner shipping fleet repositioning. In *ICAPS*, 279–287.

van den Briel, M.; Benton, J.; Kambhampati, S.; and Vossen, T. 2007. An LP-based heuristic for optimal planning. In Bessière, C., ed., *Principles and Practice of Constraint Programming - CP 2007*, volume 4741 of *Lecture Notes in Computer Science*. Springer. 651–665.

Vidal, V., and Geffner, H. 2006. Branching and pruning: An optimal temporal POCL planner based on constraint programming. *Artificial Intelligence* 170(3):298–335.

Vidal, V. 2011. YAHSP2: Keep it simple, stupid. In García-Olaya, A.; Jiménez, S.; and López, C. L., eds., *The 2011 International Planning Competition: Description of Participating Planners, Deterministic Track*. 83–90.