Bounded Intention Planning

Jason Wolfe
Computer Science Division
University of California, Berkeley
jawolfe@cs.berkeley.edu

Stuart Russell
Computer Science Division
University of California, Berkeley
russell@cs.berkeley.edu

Abstract

We propose a novel approach for solving unary SAS^+ planning problems. This approach extends an SAS^+ instance with new state variables representing *intentions* about how each original state variable will be used or changed *next*, and splits the original actions into several stages of intention followed by eventual execution. The result is a new SAS^+ instance with the same basic solutions as the original. While the transformed problem is larger, it has additional structure that can be exploited to reduce the branching factor, leading to *reachable* state spaces that are many orders of magnitude smaller (and hence much faster planning) in several test domains with acyclic causal graphs.

1 Introduction

State-space planners based on forward search have emerged as the overall winners in recent planning competitions. Their success is due in part to cheap and accurate domain-independent heuristics, and in part to easy elimination of repeated states (i.e., transpositions) in the state space. Nevertheless, state-space planners—especially optimal ones—can fail spectacularly on simple problems that allow for many possible interleavings of independent subplans, or contain many actions irrelevant to achieving the goal, which are trivial for alternative approaches such as partial-order planning.

Recent work by Helmert and Röger [2008] has shown that improvements in heuristic technology alone are unlikely to improve this situation, highlighting the need for new techniques in state-space planning to detect and exploit symmetries. Especially vital are techniques to limit the interleaving of independent parts of a solution. Unfortunately, Helmert and Röger conclude that existing such techniques are "difficult to integrate into algorithms that prune duplicates such as A*," and that more generally, "clean, general ideas for exploiting symmetries in planning tasks are still few."

Our key contribution is the observation that a *bounded* set of future *intentions* is sufficient to capture much of the power of hierarchical and partial-order planning. By representing these intentions *within* the problem state, we enable *state-space* planners that avoid interleaving independent subplans and prune irrelevant actions throughout the search space.

Specifically, we extend a unary SAS^+ planning problem (in which every operator affects a single state variable) with *new state variables* representing *intentions* about how each original state variable will be used or changed next: for each variable v, we allow a commitment to a *next action* that will change v, and a *next child variable* that will causally depend on v. Then, the execution of each original action a becomes a multi-step process: first we intend to do a to change v next, then we intend to use v as the next child of each prevail condition of a, and finally we *fire* a, applying its original effects as well as clearing the above intentions. These steps can be interleaved with the steps of actions affecting other variables (e.g., steps achieving prevail conditions of a).

The output of this process is a new SAS^+ instance, which essentially augments the original state with a flat, bounded representation of a partially-ordered hierarchical plan. The set of basic optimal solutions to this augmented instance (after dropping non-*fire* actions) is identical to the set of optimal solutions to the original instance. However, due to the much larger state space, naive search in this formulation (e.g., using uniform-cost search) would be much slower than in the original SAS^+ formulation. To achieve speedups, we exploit the special structure of the augmented instance to heavily prune the set of actions that need to be considered from each state, without sacrificing optimality.

We first show that the set of applicable actions can be partitioned based on their type and the original variable affected, so that search only needs to branch over actions in a *single* (arbitrarily chosen) *partition* at each state. The remaining branching comes in two types: choosing a next intended action, or next intended child, for a single variable (cf. choosing an achiever for an open condition in partial-order planning). Action intentions commit to *what* to do next, but since they are *unordered* they do not rule out necessary interleavings with other variables. Child intentions constrain *when* these intended actions can be fired, only generating branching when necessary to interleave usage of *shared variables* relevant to operators affecting multiple other variables.

Next, we propose a heuristic for choosing an action partition, which favors discharging existing commitments over creating new ones. This leads to our Bounded Intention Planner (BIP), which automatically breaks symmetries between equivalent interleavings, providing provably exponential speedups on independent subproblems, and huge em-

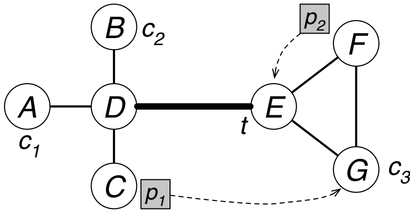

Figure 1: A transportation planning task [Helmert, 2006]. There are two cities, each defined by a graph of locations with roads (thin lines) that can be traversed by cars. There are two cars (c_1 and c_2) in the left city and one (c_3) in the right city, plus a single truck t that can drive between the cities, stopping at locations D and E. The goal is to deliver parcels p_1 and p_2 to the designated locations using the vehicles.

pirical speedups even without them, in several test domains with acyclic causal graphs. BIP also leverages intentions in other ways; for example, intended actions' preconditions provide *subgoals*, which can be used to prune irrelevant actions throughout search.

We conclude with a brief discussion of potential improvements and applications to general planning problems.

2 Background

2.1 Unary SAS$^+$ Representation

Following Bäckström and Nebel [1995], a unary SAS$^+$ planning problem is defined by a tuple $\Pi = (\mathcal{V}, \mathcal{O}, s_0, s_\star)$:

- $\mathcal{V} = \{v_1, ..., v_m\}$ is the set of *multi-valued state variables*. Each variable $v \in \mathcal{V}$ has values chosen from a *domain* $\mathcal{D}_v$. A *state* is a total assignment of variables to domain values, so the *state space* is $\mathcal{S}_\mathcal{V} = \mathcal{D}_{v_1} \times ... \times \mathcal{D}_{v_m}$. The value of variable v in state s is written $s[v]$.

  We also consider *partial states*, in which variables can take on the *undefined* value u, and let $\mathcal{V}_{s^+}$ be the set of variables defined ($\neq \mathsf{u}$) in partial state s^+. s^+ is *satisfied* by a state s, written $s^+ \sqsubseteq s$, if $\forall v \in \mathcal{V}_{s^+}, s^+[v] = s[v]$.
- $\mathcal{O}$ is the set of *operators* (a.k.a. actions), each of which is defined by a tuple $o = (\mathsf{pre}_o, \mathsf{post}_o, \mathsf{prv}_o)$ of partial states, called the *pre-*, *post-*, and *prevail-conditions* respectively. We assume *unary* operators that affect a single variable v, i.e., $\mathcal{V}_{\mathsf{pre}_o} = \mathcal{V}_{\mathsf{post}_o} = \{v\}$, with prevail conditions allowed on the remaining variables. An operator o is *applicable* in a state s if $\mathsf{pre}_o \sqsubseteq s$ and $\mathsf{prv}_o \sqsubseteq s$. o can then be be *executed* in s, producing a new state $o(s)$ with v set to $\mathsf{post}_o[v]$. This incurs a cost ≥ 0, which we assume is 1 unless otherwise mentioned.

  Denote the partition of operators affecting v by $\mathcal{O}[v]$.
- $s_0 \in \mathcal{S}_\mathcal{V}$ is the *initial state* (we assume it is complete)
- $s_\star$ is the (typically partial) *goal state*

Finally, a *solution* is a sequence of applicable operators $(o_1, ..., o_k)$ that transforms s_0 to a state that satisfies $s_\star$.

As a running example, we consider a transportation planning task (see Figure 1) due to Helmert [2006]. The objective is to deliver a set of packages (p_i) from their current locations to their destinations using a fleet of vehicles. Cars (c_i)

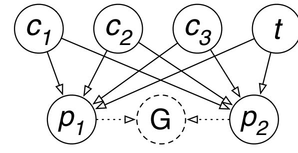

Figure 2: Causal graph of the transportation task. G is a dummy variable representing the goal conditions.

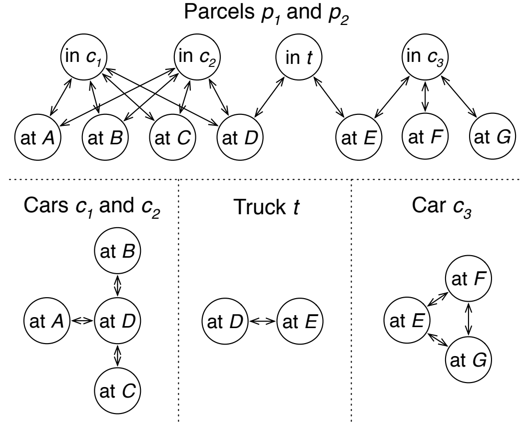

Figure 3: Domain transition graphs for the transportation task.

can drive along roads within a city, and the truck (t) can drive between the cities; all vehicles can hold any number of packages. A SAS$^+$ encoding of this problem has six state variables: one for the location of each vehicle, and one for the status of each parcel (either "at location *l*" or "in vehicle *v*"); and three basic operator types: LOAD, DRIVE, and UNLOAD. For example, operator LOAD(p_2, c_3, F) loads package p_2 into car c_3 at location F; it has prevail condition $c_3 = at\ F$, precondition $p_2 = at\ F$, and postcondition $p_2 = in\ c_3$. The optimal solution to the example problem delivers p_2 using c_3, and p_1 using c_1 or c_2, then t, then c_3. These subplans can be interleaved arbitrarily, except that c_3 must be used to deliver p_2 first in an optimal solution.

2.2 Causal Graphs and Independence

The *causal graph* *CG* of an SAS$^+$ instance captures the causal relationships between its variables [Helmert, 2006]. Formally, *CG* is a directed graph over the state variables $\mathcal{V}$, where there is an edge from s to t iff there exists an operator $o \in \mathcal{O}$ with a prevail condition on s and postcondition on t (see Figure 2). Denote the children of s by *CG*(s). In our example, the causal graph contains an edge from each vehicle variable v to each package variable p, generated by the LOAD$(p, v, \cdot)$ and UNLOAD$(p, v, \cdot)$ operators that have a prevail condition on v and a postcondition on p.

A first observation is that only *ancestors* of v in *CG* (including v itself) are *relevant* for changing v. Define a *subproblem* as the task of achieving a particular value for some variable. If the ancestors of two variables are disjoint, subproblems on these variables are *independent*: they do not interact, and so *any interleaving* of their optimal solutions is an

optimal solution to the combined task. In our example, only subproblems involving different vehicles are independent.

More commonly, subproblems can exhibit *partial* dependence, where solutions only interact via one or more shared ancestor variables. The more frequently two subproblems require different values for a shared variable, the stronger the dependence. Our approach capitalizes on these structures, avoiding all interleaving of independent subproblem solutions, and only generating interleaving to multiplex the use of shared variables for partially dependent subproblems.

The *domain transition graphs* (*DTG*s) complement the causal graph. For each variable $v \in \mathcal{V}$, DTG_v is a directed graph over its domain $\mathcal{D}_v$. Each operator $o \in \mathcal{O}[v]$ generates an edge from $\mathsf{pre}_o[v]$ to $\mathsf{post}_o[v]$; thus, paths in DTG_v correspond to operator sequences affecting v. Figure 3 shows *DTG*s for our example; the parcel transitions are generated by LOAD and UNLOAD operators, and the vehicle transitions by DRIVE operators.

3 Related Work

Various researchers have studied the unary/acyclic problems that are our primary focus, often with an emphasis on complexity and identifying tractable subclasses (e.g., [Bäckström, 1992; Bäckström and Nebel, 1995; Jonsson and Bäckström, 1998; Jonsson, 2009]). A particularly relevant result by Brafman and Domshlak [2003] is that planning remains PSPACE-complete even when restricted to unary effects and acyclic causal graphs (and binary-valued variables).

The intentions in our approach are closely related to commitments in hierarchical planning (e.g., [Knoblock, 1994]) and partial-order planning (e.g., [Mcallester and Rosenblitt, 1991]). A key difference is that we retain the advantages of state-space search, including cheap domain-independent heuristics and easy repeated-state checking.

Numerous works have explored related planning algorithms that work directly on the causal graph. However, these proposals have been suboptimal (e.g., [Haslum, 2007]), incomplete (e.g., [Chen *et al.*, 2008]), or limited to more restricted problem classes (e.g., [Jonsson, 2009]).

Within the state-space setting, Coles and Coles [2010] present several optimizations that prune action choices based on the stated goals of a problem; while powerful, these optimizations can also be fragile (sensitive to problem formulation). Intentions allow us to apply similar optimizations *hierarchically* throughout search, using *subgoals* provided by the preconditions of intended actions. Other pruning methods include exploiting symmetric objects and tunnel macros (e.g., [Fox and Long, 1999; Coles and Coles, 2010]), which should complement our approach.

Most closely related to our approach are techniques for pruning symmetric action interleavings in state-space search. Stratified planning [Chen *et al.*, 2009] is a small modification of state-space search, in which the applicable operators from a state s are pruned based on the last operator o executed to reach s, effectively avoiding some equivalent interleavings. Variables are first assigned numeric *levels* from a topological ordering of the causal graph. Then, from state s, only operators that either (1) affect $\geq$-level variables than o, or (2) were enabled by doing o are considered next; all other actions are pruned.

The Expansion Core (EC) method [Chen and Yao, 2009] is another recent approach for pruning symmetric action interleavings. From each state s, the EC planner builds a *potential dependency graph* on the state variables of the problem, which contains an edge from variable u to v iff u is a potential precondition or potential dependent of v. Roughly speaking, a potential precondition u's current value satisfies a pre(vail) condition of an operator affecting v that lies on a path from the current value of v to its goal value (if applicable); and a potential dependent u can be changed next by an operator that needs a value for v that lies on a path from its current value to its goal value (if applicable). Given this graph, a subset of variables V called a *dependency closure* is selected, which has no outgoing edges, and contains at least one variable $v \in V$ s.t. $s[v] \neq s_\star[v] \neq \mathsf{u}$. Only actions affecting V are considered from s, without sacrificing optimality.

Both the EC method and the "operator partitions" used by BIP can be understood as applications of the "stubborn set" method [Valmari, 1990], which identifies subsets of actions that can be greedily focused on at a given state without sacrificing global optimality.

Despite the connections, there are many important differences between BIP, stratified planning, and the EC approach. Perhaps most importantly, whereas BIP only branches over actions that affect a single variable at each step, both the stratified planner and EC must branch over a (potentially large) set of such partitions. A first consequence is that the stratified planner explores an exponential number of interleavings (in n) given n independent subproblems (it can do any number of actions in subproblem 1 before moving to subproblem 2, and so on). Both BIP and EC are linear in n.

A key difference between EC and BIP arises with *weakly* dependent subproblems. In such cases, BIP makes and maintains commitments to how shared variables will be used next, temporarily decomposing the subproblems so that independence can be exploited (by not switching subproblems until these commitments are discharged). This ability is enabled by the addition of intentions; put simply, the stratified planner and EC do not have enough *memory* to generate such behaviors. Moreover, intentions enable additional advantages, such as goal-directed behavior, which are not available to the other approaches. These claims are borne out by empirical results in several domains, in Section 5.

4 Bounded Intention Planning

4.1 Representation

As summarized above, the first step in our approach is to transform an "original" unary SAS$^+$ instance Π to an "augmented" instance $\bar{\Pi}$ with additional state variables and a modified set of operators. As a precursor to this transformation, we add to Π a Boolean goal variable G, initially *false*, and an operator GOAL that sets G to *true*, with prevail condition $s_\star$.

Now, $\bar{\Pi} = (\bar{\mathcal{V}}, \bar{\mathcal{O}}, \bar{s_0}, \bar{s_\star})$ is defined as follows:

State Variables

For each original variable $v \in \mathcal{V}$, $\bar{\mathcal{V}}$ contains three variables: v itself, plus two new *intention* variables (see Figure 4):

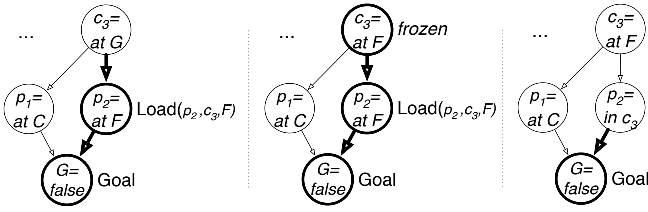

Figure 4: Several (partial) augmented states for the running example. Circles represent original state variables, including their current values. Bold variables have active (non-*free*) action intentions, with the corresponding intention at right. Bold arrows represent current child intentions. Other cars and the truck are omitted for space reasons. *Left:* The initial state, after executing $\text{SETC}(p_2, \mathsf{G})$, $\text{SETO}(\text{LOAD}(p_2, c_3, F))$, $\text{SETC}(c_3, p_2)$. *Center:* The state at left, after $\text{SETO}(\text{DRIVE}(c_3, G, F))$, fire $\text{DRIVE}(c_3, G, F)$, $\text{FREEZE}(c_3, at\ \ F)$. *Right:* The center state, after firing $\text{LOAD}(p_2, c_3, F)$.

First, O_v (next **o**perator on v), with domain $\mathcal{O}[v] \cup \{\textit{free}, \textit{frozen}\}$, representing an intention that the next operator to change v will be O_v. *frozen* represents an intention to *not change* v until its current value is used to satisfy a prevail condition of some operator, and *free* represents no intention.

Second, C_v (next **c**hild of v), with domain $CG(v) \cup \{\textit{free}\}$, representing an intention to use v to satisfy a prevail condition of an operator that affects C_v next (or no such intention). Child intentions can be thought of as abstracted versions of the causal links used in partial-order planning.

Operators

Each original operator is split into stages, while preserving the semantics of the original variables. Specifically, to execute an original operator o that affects v: (1) o must be intended next on v, (2) each prevail variable of o must intend v as its next child, (3) each prevail variable must be *frozen* at the value required by o, and (4) o is executed, and the previous intentions are cleared. For example, Figure 4 shows states where $\text{LOAD}(p_2, c_3, F)$ is ready to fire (center) and has just fired (right). The operators $\bar{\mathcal{O}}$ that accomplish this are:

1. For each $v \in \mathcal{V}$ and $o \in \mathcal{O}[v]$, an operator $\text{SETO}(o)$ with $\mathsf{pre}[\mathsf{O}_v]{=}free$, $\mathsf{post}[\mathsf{O}_v]{=}o$, and $\mathsf{prv}[v]{=}\mathsf{pre}_o[v]$. $\text{SETO}(o)$ adds an intention to change v next by doing o; it requires that no operator is currently intended for v and that v's current value matches the precondition of o, but does not check the prevail conditions of o yet.

2. For each $v \in \mathcal{V}$ and child $c \in CG(v)$, an operator $\text{SETC}(v, c)$ with $\mathsf{pre}[\mathsf{C}_v]{=}\textit{free}$ and $\mathsf{post}[\mathsf{C}_v]{=}c$ that adds an intention to use v to change c next.

3. For each $v \in \mathcal{V}$ and domain value $x \in \mathcal{D}_v$, an operator $\text{FREEZE}(v, x)$ with $\mathsf{pre}[\mathsf{O}_v]{=}\textit{free}$, $\mathsf{post}[\mathsf{O}_v]{=}\textit{frozen}$, and $\mathsf{prv}[v]{=}x$ that freezes the value of v to x until used.

4. For each $v \in \mathcal{V}$ and $o \in \mathcal{O}[v]$, an augmented "fire" operator $\bar{o}$ with the same conditions as o, plus additional pre- and post-conditions on the intentions:

$$
\begin{aligned}
&\mathsf{pre}_{\bar{o}}[\mathsf{O}_v]{=}o, && \mathsf{post}_{\bar{o}}[\mathsf{O}_v]{=}\textit{free}\\
\forall p \in \mathcal{V}_{\mathsf{prv}_o}: &\mathsf{pre}_{\bar{o}}[\mathsf{O}_p]{=}\textit{frozen}, && \mathsf{post}_{\bar{o}}[\mathsf{O}_p]{=}\textit{free}\\
\forall p \in \mathcal{V}_{\mathsf{prv}_o}: &\mathsf{pre}_{\bar{o}}[\mathsf{C}_p]{=}v, && \mathsf{post}_{\bar{o}}[\mathsf{C}_p]{=}\textit{free}
\end{aligned}
$$

Finally, the cost of each $o \in \mathcal{O}$ is assigned to $\text{SETO}(o)$; all other operators in $\bar{\mathcal{O}}$ (including $\bar{o}$) are assigned cost 0.

Initial State and Goal

The initial state $\bar{s_0}$ of the transformed problem extends s_0, setting all new $\mathsf{O}.$ and $\mathsf{C}.$ variables to *free*, except O_G, which is set to GOAL. The goal $\bar{s_\star}$ of the transformed problem is $\mathsf{G}{=}\textit{true} \wedge (\forall v \in \mathcal{V})\ \mathsf{O}_v \in \{\textit{free}, \textit{frozen}\}$.[1]

4.2 Solutions to $\bar{\Pi}$

The *projection* $P(\mathbf{o})$ of an operator sequence $\mathbf{o} \in \bar{\mathcal{O}}^*$ replaces each operator $\bar{o}$ by o, and drops all other operators.

Theorem 1. *There is a 1-1 correspondence between solutions* $\mathbf{o}$ *to* Π *with cost c, and non-empty* sets *of solutions to* $\bar{\Pi}$ *with projection* $\mathbf{o}$ *and cost c.*

Proof. A solution to Π can be extended to a solution to $\bar{\Pi}$ by expanding each operator o into the sequence SETO(o), then for each prevail variable p in $\mathcal{V}_{\mathsf{prv}_o}$, $\text{SETC}(p, v)$, $\text{FREEZE}(p, \mathsf{prv}_o[p])$, and finally $\bar{o}$. Conversely, the projection of a solution to $\bar{\Pi}$ must be a solution to Π, because only the operators $\bar{o}$ affect the original state variables, and each retains the semantics of o thereupon. Moreover, projection preserves cost, since $\text{SETO}(o)$ and $\bar{o}$ operators are in 1-1 correspondence in solutions to $\bar{\Pi}$. □

4.3 Operator Partitions

Consider the following partitioning $\bar{\mathcal{P}}$ of the operators $\bar{\mathcal{O}}$:

- For each $v \in \mathcal{V}$ and $x \in \mathcal{D}_v$, a partition $SetO_{v=x} = \{\text{SETO}(o) | o \in \mathcal{O}[v] \wedge \mathsf{pre}_o[v]{=}x\} \cup \{\text{FREEZE}(v, x)\}$ that chooses to freeze v, or intend a next operator on v.
- For each $v \in \mathcal{V}$, a partition $SetC_v = \{\text{SETC}(v, c) | c \in CG(v)\}$ that chooses a next intended child for v.
- For each $o \in \mathcal{O}$, a partition $Fire_o = \{\bar{o}\}$ that fires o.

All operators in each partition have identical pre- and prevail-conditions. Define the subset of *applicable* partitions in state s as $\bar{\mathcal{P}}_s$. Only operators from a *single, arbitrary* partition in $\bar{\mathcal{P}}_s$ need to be considered to preserve optimality.

Theorem 2. *For every state s and operator partition p in* $\bar{\mathcal{P}}_s$ *there exists an optimal solution for s that begins with an operator in p.*

Proof. First note that every applicable operator $o \in p \in \bar{\mathcal{P}}_s$ is *independent* of every applicable operator $o' \in p' \in \bar{\mathcal{P}}_s \setminus \{p\}$ outside its partition: no variable appears in any condition of both o and o'. Independence ensures that if we do o on this step, we can always still do o' on the subsequent step; and moreover, $o(o'(s)) = o'(o(s))$.

[1] $\bar{s_\star}$ is not a true partial state, because of the disjunction in the $\mathsf{O}.$ conditions. These conditions are added to simplify the analysis, but are not actually needed to preserve correctness.

To conclude, we argue that no partition is "harmful"—that is, it is always the case that *some* operator o in each partition $p \in \bar{\mathcal{P}}_s$ begins an optimal solution from s. This is straightforward: $Fire_o$ cannot be harmful, since every intended action must eventually be fired; $SetC_v$ branches on all possible values for C_v (except *free*), and no other operators have conditions requiring $\mathsf{C}_v = free$; and likewise for $SetO_{v=x}$ and O_v (the FREEZE$(v, \cdot)$ actions preserve a zero-cost option, in case v's current value is needed next, or v is not needed again). □

To see how this partitioning can avoid generating unnecessary interleavings, consider the example in Figure 4. From the state at left, we can alternate between choosing partition $SetO_{c_3=\cdot}$ and firing the intended operator on c_3 until the state at the center is reached (assume that we do not freeze variables to the wrong values, an issue we address shortly). Then, we can fire LOAD(p_2, c_3, F) to reach the state at right. In this process, we need not consider interleaving actions involving other packages or vehicles. Optimality is preserved because in reaching the state at left, we would have considered SETC(c_3, p_1), which covers the possibility that c_3 might need to pick up p_1 first instead. In short, committing to a first child for c_3 up front temporarily decomposes the state of c_3 from the rest of the problem, allowing us to avoid an exponential number of potential interleavings with operators affecting other parts of the state.

4.4 Action Partition Pruning

When selecting from a partition $SetO_{v=x}$ in state s, if $\mathsf{C}_v = c$, $\mathsf{O}_c = o$, and $\mathsf{prv}_o[v] = x' \neq \mathsf{u}$, we can use this value as a *subgoal*, and prune the options as follows (without sacrificing optimality). If $x = x'$, greedily do FREEZE(v, x'). Otherwise, prune all operators that do not lie on an *acyclic path* through the domain transition graph of v from x to x'. In our running example, this pruning eliminates all branching over paths for the cars in the left city, since their domain transition graphs only admit a single acyclic path from each initial value to each subgoal value. For instance, if we have intended LOAD(p_1, c_1, C) and SETC(c_1, p_1) and c_1 is at D, we can prune the partition $SetO_{c_1=D}$ to the singleton $\{$SETO(DRIVE$(c_1, D, C))\}$ because after driving to A or B, c_1 would have to return to D before reaching C (see Figure 3). This optimization generalizes the irrelevant-action pruning of [Coles and Coles, 2010].

4.5 Child Partition Pruning

When selecting from a partition $SetC_v$ in state s, if any child c has an intended operator o s.t. forall $p \in \mathcal{V}_{\mathsf{prv}_o}$, $s[p] = \mathsf{prv}_o[p]$, $\mathsf{O}_p \in \{free, frozen\}$, and $\mathsf{C}_p \in \{free, v\}$, operator SETC$(v, c)$ can be greedily executed (discarding alternatives in $SetC_v$). Allowing o to fire leads to a strictly less constrained state, and so optimality is not compromised. For instance, suppose there are ten parcels at F to be LOADed into c_3, which is also at F. This optimization eliminates branching over which parcel(s) to load (first); all ten are greedily picked up in sequence before c_3 is moved. As a further improvement, we also prune children from $SetC_v$ that are not ancestors of any unachieved precondition of GOAL. For instance, once p_1 has been delivered, we can greedily assign SETC(v, p_2) for any vehicle v without considering p_1 as a child.

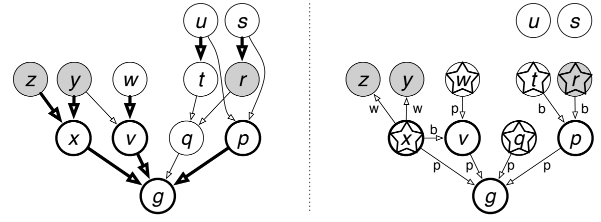

Figure 5: *Left:* A causal graph. Bold lines/circles indicate active intentions. Grayed variables are frozen. Links to bolded circles (with intended actions) have prevail requirements. The variables are lettered according to an inverse depth-first topological ordering. *Right:* The corresponding VPG, with edges labeled by type. Starred variables are sources.

4.6 Partition Selection

This section describes a heuristic for selecting an operator partition that focuses effort on resolving existing commitments necessary for reaching the goal, while maximizing the applicability of the above pruning rules.

To compute this heuristic, given a state s of $\bar{\Pi}$, we first define a *variable precedence graph* VPG_s over the variables $\mathcal{V}$. An edge from u to v in VPG_s indicates that an intention on u must change before the intended action on v can fire. There are three types of edges (see Figure 5):

- **Wait:** For each $v \in \mathcal{V}$ s.t. O_v=*frozen* $\wedge$ $\mathsf{C}_v \neq$*free*, a wait edge $\mathsf{C}_v \rightarrow v$. This captures the fact that v cannot become unfrozen until an operator fires on its intended child C_v.
- **Block:** For each $v \in \mathcal{V}$ s.t. $\mathsf{O}_v \notin \{free, frozen\}$, and each prevail variable $p \in \mathcal{V}_{\mathsf{prv}_{\mathsf{O}_v}}$ s.t. $\mathsf{C}_p \notin \{free, v\}$, a block edge $\mathsf{C}_p \rightarrow v$. This captures that the prevail variable p of operator O_v is currently reserved for another variable C_p, and O_v cannot fire until an operator fires on C_p.
- **Prevail:** For each $v \in \mathcal{V}$ s.t. $\mathsf{O}_v \notin \{free, frozen\}$, and each prevail condition variable $p \in \mathcal{V}_{\mathsf{prv}_{\mathsf{O}_v}}$ s.t. $\mathsf{C}_p \in \{free, v\} \wedge (\mathsf{O}_p \neq frozen \vee \mathsf{C}_p \neq v \vee s[p] \neq \mathsf{prv}_{\mathsf{O}_v}[p])$, a prevail edge $p \rightarrow v$. This captures the conditions on p for O_v to fire. We skip the prevail edge when $\mathsf{C}_p \notin \{free, v\}$, since the block edge captures a stronger condition in this case.

If the component of VPG_s with G contains a cycle, state s is *deadlocked* and can be pruned immediately. Otherwise, we restrict our attention to the set of *source* variables S in VPG_s, which are places where immediate progress can be made towards the goal. If an intended operator can be fired. that is always our first choice. Otherwise, we must add new intentions that work towards achieving the prevail and "intentional" conditions of existing intended operators. Specifically, we select an operator partition as follows.

(A) If any partition $Fire_{\mathsf{O}_v}$ for $v \in S$ is applicable, choose one arbitrarily. (B) Otherwise, if any partition $SetO_{v=s[v]}$ is

		baseline		∞-strat	EC	BIP		BIP - action pruning		BIP - child pruning	
inst	cost	states	ms	states	states	states	ms	states	ms	states	ms
5-2	8	27104	593	20423	2014	42	38	58	42	292	217
6-1	14	385484	7065	226446	161014	104	151	236	148	1523	1170
4-2	15	600382	11840	341105	352896	141	102	609	342	371	263
5-1	17	1302034	28884	610240	625680	101	98	263	158	5006	4614
4-0	20	2628508	56441	1566447	1710618	132	290	2175	1721	19013	15704
6-3	24	5451635	129786	2747824	3476092	292	226	4856	4130	56598	50117
5-0	27	7066822	177433	3968054	4706820	493	885	25099	18091	456532	537575
9-1	30					1738	1912				
10-0	45					169168	300603				

Table 1: States generated and total runtimes in milliseconds to optimally solve representative LOGISTICS instances from IPC2 using uniform-cost search. Baseline is the ordinary SAS^+ formulation, ∞-strat is the stratified planner with levels chosen from a topological ordering, EC is the expansion core method, BIP is the bounded intention planner with all pruning, and the remaining two algorithms are BIP with a single named pruning type ablated. A sample of instances solved by the baseline within 10 minutes and 512 MB are shown (plus two harder instances); they are sorted by optimal solution cost. Runtimes for ∞-strat and EC are omitted, since we made no attempt to optimize these algorithms (note that time per state generated should be no less than for baseline).

applicable where the action pruning conditions are met, select one arbitrarily. (C) Otherwise, if any partition $SetC_v$ is applicable where at least one child has an intended operator with prevail condition on v, select the *last* one according to a depth-first topological sort of the causal graph *CG*. (D) Finally, if none of (A-C) are applicable, select an arbitrary applicable partition on a source variable. See Section 6 for discussion and analysis of this heuristic.

5 Results

We report empirical results examining the runtime and number of states generated to optimally solve LOGISTICS instances from the second International Planning Competition (IPC),[2] and several variants of a taxi domain.

Our implementation uses the LAMA preprocessor [Richter and Westphal, 2010] to convert STRIPS instances into SAS^+. Search is carried out by a Clojure implementation of uniform-cost search. Our BIP has not been optimized (e.g., VPGs are computed naively from scratch for each state), and is currently a constant factor of about 50 times slower than baseline (per state examined); we estimate that this constant could be brought down to 5 or less with some simple optimizations.

Table 1 shows runtimes and states generated to optimally solve LOGISTICS instances, for baseline, stratified, EC, and BIP planners. On the hardest problems solved by all algorithms, BIP examines *four orders of magnitude* fewer states and finds a solution more than *200 times faster* than baseline. Moreover, BIP was able to solve nine more instances than baseline (two shown). In contrast, on all but the smallest problem instance, both the stratified planner and EC beat the baseline by less than a factor of three.

Figure 6 shows results in several *taxi* domains, wherein a fleet of taxis must deliver passengers from randomly chosen sources to randomly chosen destinations on a 3x3 grid. We experiment with three formulations: *individual*, where each passenger has their own taxi, and the causal graph is an inverted tree; *pairwise*, where each i of n taxis can pick up passenger i or $i+1 \mod n$, and the causal graph is a DAG; and *single*, where a single taxi must deliver all passengers, so the passenger subproblems are most tightly coupled. In all but the pairwise case, the action pruning of BIP infers that passengers should not be put down at intermediate locations; we help level the playing field by giving other algorithms this constraint explicitly (designated by "+c"); in the pairwise case, this constraint actually changes the set of optimal solutions, which we discuss later. In all settings, we see at least an order of magnitude improvement using BIP over baseline, with the advantage increasing as the passenger subproblems become less dependent. As expected, the stratified planner and EC are a small constant factor better than baseline, except on the individual setting where EC is able to exploit the complete independence of passengers.

6 Conclusion

Our bounded intention planner (BIP) brings some of the benefits of hierarchical and partial-order planning to a forward state-space setting. We have proven its optimality for general unary domains, and empirically demonstrated orders of magnitude reduction in reachable state spaces and optimal planning times compared to a standard state-space search, in several domains with acyclic causal graphs. BIP is guaranteed to avoid all interleaving of actions from completely independent subproblems, and minimizes interleaving in the presence of weakly dependent subproblems.

The performance of BIP depends on two main factors. First, performance improves with the amount of (weak) independence available to exploit. In domains like MICONIC and the pairwise taxi domain with no constraint, subproblems are so tightly coupled that no advantage is seen (after picking up a passenger, we have to consider handing them off to every other vehicle at every possible intermediate location). Second, good performance depends on our partition heuristic (see Section 4.6) using cases (A-C) as much as possible.

[2] LOGISTICS is one of only two previous IPC domains we have identified with unary operators (in their most natural SAS^+ formulation). The other is MICONIC, for which our algorithm evaluates roughly twice the states of baseline (results not shown).

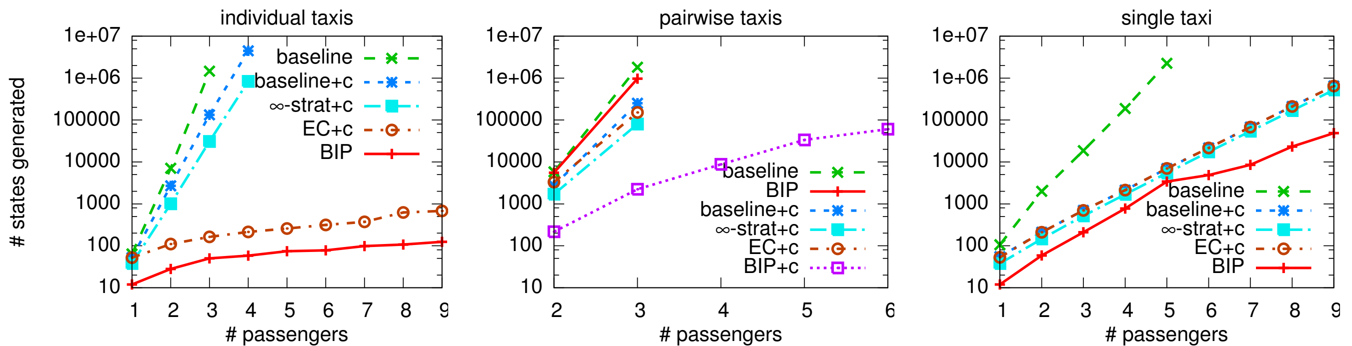

Figure 6: Number of states generated to optimally solve random instances of several "taxi" problem types. "+c" indicates an added constraint that passengers can only be dropped at their destinations. Only instances solved within 512 MB are shown.

In this case, BIP works *hierarchically*, reserving variables as prevail conditions, adding intended operators to help achieve those conditions, and so on, like the example given in Section 4.3. This happens often; in fact, the final case (D) was not actually executed in any problem tested above.

Many interesting extensions of the approach are possible, including: exploiting symmetric objects; further pruning of operator partitions (e.g., using action costs); solving augmented instances hierarchically using state abstraction [Wolfe *et al.*, 2010]; and adding techniques for efficient state-space planning such as heuristics and landmarks, which will be *at least* as informative as in the original state space (in the worst case, one can just drop the intention variables).

Most important, of course, will be generalizations to non-unary problems. We are currently investigating several approaches, ranging from clustering variables co-occurring in postconditions, to more direct generalizations of BIP to handle multiple postconditions. Preliminary results on several domains are encouraging, but further study is needed to see how and to what degree the observed speedups can carry over to general planning problems.

7 Acknowledgements

We thank the anonymous reviewers for many helpful suggestions. This research was supported by the National Science Foundation, award #0904672.

References

[Bäckström and Nebel, 1995] Christer Bäckström and Bernhard Nebel. Complexity Results for SAS+ Planning. *Computational Intelligence*, 11:625–655, 1995.

[Bäckström, 1992] Christer Bäckström. Equivalence and Tractability Results for SAS+ Planning. In *KR*, pages 126–137. Morgan Kaufmann, 1992.

[Brafman and Domshlak, 2003] Ronen I. Brafman and Carmel Domshlak. Structure and Complexity in Planning with Unary Operators. *JAIR*, 18:315–349, 2003.

[Chen and Yao, 2009] Yixin Chen and Guohui Yao. Completeness and optimality preserving reduction for planning. In *IJCAI*, pages 1659–1664, 2009.

[Chen *et al.*, 2008] Yixin Chen, Ruoyun Huang, and Weixiong Zhang. Fast Planning by Search in Domain Transition Graphs. In *AAAI*, pages 886–891, 2008.

[Chen *et al.*, 2009] Yixin Chen, You Xu, and Guohui Yao. Stratified Planning. In *IJCAI*, pages 1665–1670, 2009.

[Coles and Coles, 2010] Amanda Coles and Andrew Coles. Completeness-Preserving Pruning for Optimal Planning. In *ECAI*, pages 965–966, 2010.

[Fox and Long, 1999] M. Fox and D. Long. The Detection and Exploitation of Symmetry in Planning Problems. In *IJCAI*, pages 956–961, 1999.

[Haslum, 2007] Patrik Haslum. Reducing Accidental Complexity in Planning Problems. In *IJCAI*, pages 1898–1903, 2007.

[Helmert and Röger, 2008] Malte Helmert and Gabriele Röger. How good is almost perfect? In *AAAI*, pages 944–949, 2008.

[Helmert, 2006] Malte Helmert. The Fast Downward Planning System. *JAIR*, 26:191–246, 2006.

[Jonsson and Bäckström, 1998] Peter Jonsson and Christer Bäckström. State-variable planning under structural restrictions: Algorithms and complexity. *Artificial Intelligence*, 100:125–176, 1998.

[Jonsson, 2009] Anders Jonsson. The Role of Macros in Tractable Planning. *JAIR*, 36:471–511, 2009.

[Knoblock, 1994] Craig Knoblock. Automatically Generating Abstractions for Planning. *Artificial Intelligence*, 68:243–302, 1994.

[Mcallester and Rosenblitt, 1991] David Mcallester and David Rosenblitt. Systematic Nonlinear Planning. In *AAAI*, pages 634–639, 1991.

[Richter and Westphal, 2010] Silvia Richter and Matthias Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *JAIR*, 39:127–177, 2010.

[Valmari, 1990] Antti Valmari. Stubborn sets for reduced state space generation. In *ICATPN*, pages 491–515, 1990.

[Wolfe *et al.*, 2010] Jason Wolfe, Bhaskara Marthi, and Stuart J. Russell. Combined Task and Motion Planning for Mobile Manipulation. In *ICAPS*, 2010.