Proceedings of the Twenty-Ninth AAAI Conference on Artificial Intelligence

A Generalization of Sleep Sets Based on Operator Sequence Redundancy

Robert C. Holte
University of Alberta, Canada
robert.holte@ualberta.ca

Yusra Alkhazraji
University of Freiburg, Germany
alkhazry@informatik.uni-freiburg.de

Martin Wehrle
University of Basel, Switzerland
martin.wehrle@unibas.ch

Abstract

Pruning techniques have recently been shown to speed up search algorithms by reducing the branching factor of large search spaces. One such technique is sleep sets, which were originally introduced as a pruning technique for model checking, and which have recently been investigated on a theoretical level for planning. In this paper, we propose a generalization of sleep sets and prove its correctness. While the original sleep sets were based on the commutativity of operators, generalized sleep sets are based on a more general notion of operator sequence redundancy. As a result, our approach dominates the original sleep sets variant in terms of pruning power. On a practical level, our experimental evaluation shows the potential of sleep sets and their generalizations on a large and common set of planning benchmarks.

Introduction

Depth-first search methods such as IDA* (Korf 1985) have difficulty in domains with many alternative paths to the same state. Pruning techniques aim to overcome this by identifying operators that need not be applied at a given search node because doing so is certain to produce a state that can be reached by a different path that is no more costly than the current one. This can greatly reduce the search time without sacrificing the optimality of the solution found.

One such pruning technique is *sleep sets* (Godefroid 1996; Wehrle and Helmert 2012). Sleep sets use the *commutativity* of operators as the basis for pruning decisions. A different technique, *move pruning* (Holte and Burch 2014), is based on a much richer notion of operator sequence redundancy than commutativity, but is limited to pruning relatively short operator sequences whereas sleep sets can prune arbitrarily long sequences. Consequently, move pruning and sleep sets are incomparable in terms of pruning power: there exist operator sequences that one of them can prune, but the other cannot, and vice versa (Holte and Burch 2014).

The first contribution of this paper is an evaluation and comparison of sleep sets and move pruning on a large set of benchmark problems from the international planning competitions. We show that both pruning methods substantially improve IDA*'s search time and that neither method dominates the other in terms of search time. The second contribution is that we provide several generalizations of sleep sets based on the richer notion of operator sequence redundancy used by move pruning. These generalizations strictly dominate the original concept of sleep sets in terms of pruning power, while preserving completeness and optimality of tree search algorithms like IDA*. We show that in domains where the more general kinds of redundancy exist, search time is substantially reduced by using the more general methods. Our experiments also show that, in contrast to generalized sleep sets and move pruning, only the basic sleep set method is faster than IDA* in terms of total time (i. e., search time + preprocessing time) due to its relatively low computational overhead.

Formal Preliminaries

As in SAS$^+$ (Bäckström and Nebel 1995) and PSVN (Holte, Arneson, and Burch 2014), we define a search problem using finite domain variables. States are represented as a mapping of each variable to a value of its domain. Operators are state transformers, consisting of a precondition and an effect. In this paper, we consider SAS$^+$ operators, where preconditions and effects both consist of variable/value pairs (indicating the required variable values in the preconditions, and the new variable values after applying the operator, respectively). The empty operator sequence is denoted ε. If A is a finite operator sequence then $|A|$ denotes the length of A (the number of operators in A, $|\varepsilon| = 0$), $cost(A)$ is the sum of the costs of the operators in A ($cost(\varepsilon) = 0$), $pre(A)$ is the set of states to which A can be applied, and $A(s)$ is the state resulting from applying A to state $s \in pre(A)$. We assume the cost of each operator is non-negative. A path from state s to state t is an operator sequence A such that $s \in pre(A)$ and $A(s) = t$. A prefix of A is a nonempty initial segment of A ($A_1 ... A_k$ for $1 \leq k \leq |A|$) and a suffix is a nonempty final segment of A ($A_k ... A_{|A|}$ for $1 \leq k \leq |A|$). A left-pointing arrow over an operator sequence denotes the prefix consisting of all operators in the sequence except the last. If $|A| \geq 1$, $\overleftarrow{A} = A_1 \dots A_{|A|-1}$; $\overleftarrow{\varepsilon}$ is defined to be ε.

If $\mathcal{O}$ is a total order on operator sequences, $B >_{\mathcal{O}} A$ indicates that B is greater than A according to $\mathcal{O}$. Throughout this paper we identify an operator o with the operator sequence of length 1 that consists of only o.

Definition 1. *A total order, $\mathcal{O}$, on operator sequences is*

nested *if* $\varepsilon <_{\mathcal{O}} Z$ *for all* $Z \neq \varepsilon$*, and* $B >_{\mathcal{O}} A$ *implies* $XBY >_{\mathcal{O}} XAY$ *for all* $A, B, X,$ *and* Y.

Definition 2. *A* length-lexicographic order $\mathcal{O}$ *is a total order on operator sequences based on a total order of the operators* $o_1 <_{\mathcal{O}} o_2 <_{\mathcal{O}} \ldots$*. For operator sequences* A *and* B*,* $B >_{\mathcal{O}} A$ *iff either* $|B| > |A|$*, or* $|B| = |A|$ *and* $o_b >_{\mathcal{O}} o_a$*, where* o_b *and* o_a *are the leftmost operators where* B *and* A *differ (*o_b *in* B*,* o_a *in the corresponding position in* A*).*

Every length-lexicographic order is a nested order. A total order on operators induces a length-lexicographic order on operator sequences. We will use the same symbol ($\mathcal{O}$) to refer to both the order on operators and the length-lexicographic order on operator sequences it induces.

Definition 3. *Given a nested order,* $\mathcal{O}$*, on operator sequences, for any pair of states* s, t *define* $min(s,t)$ *to be the least-cost path from* s *to* t *that is smallest according to* $\mathcal{O}$*.* $min(s,t)$ *is undefined if there is no path from* s *to* t*.*

Throughout this paper, the sequence of operators comprising $min(s,t)$ is $o_1 o_2 \ldots o_{|min(s,t)|}$. The prefix of $min(s,t)$ consisting of the first i operators is denoted P_i ($P_0 = \varepsilon$), and Q_i denotes the suffix of $min(s,t)$ starting at the i^{th} operator ($Q_{|min(s,t)|+1} = \varepsilon$). Hence, $min(s,t) = P_i Q_{i+1} = P_{i-1} o_i Q_{i+1}$ for all i, $1 \leq i \leq |min(s,t)|$. Note that o_i is the last operator in P_i.

Sleep Sets

We use Wehrle and Helmert (2012)'s formulation of sleep sets, adapted to our notation.

Definition 4. *Operators* p_1 *and* p_2 *are* commutative *(or* commute*), denoted* $p_1 \bowtie p_2$*, iff the following conditions hold for every state* s*:*

1. $s \in pre(p_1)$ *and* $s \in pre(p_2) \implies p_1(s) \in pre(p_2)$ *and* $p_2(s) \in pre(p_1)$ *and* $p_1(p_2(s)) = p_2(p_1(s))$
2. $s \in pre(p_1)$ *and* $s \notin pre(p_2) \implies p_1(s) \notin pre(p_2)$
3. $s \in pre(p_2)$ *and* $s \notin pre(p_1) \implies p_2(s) \notin pre(p_1)$.

A sleep set is associated with an operator sequence and is defined recursively based on an operator order $\mathcal{O}$.

Definition 5. *A set* $ss(P)$ *for an operator sequence* P *is a* sleep set *iff the following hold:* $ss(\varepsilon) := \emptyset$ *(the empty set), and, for the non-empty sequence* $P = \sigma p$*,* $ss(P) := \{o \mid (p \bowtie o)$ *and* $(o <_{\mathcal{O}} p$ *or* $o \in ss(\sigma))\}$.

In the context of searching for a (least-cost) path from state s to state t, sleep sets are used as a pruning technique as follows. If P is an operator sequence and p is an operator in $ss(P)$, then p will not be applied after P, i. e., the sequence Pp and all its extensions, will not be considered as solutions. The following theorem shows that this use of sleep sets is safe in the sense that for any state s and any state t reachable from s, it preserves at least one least-cost path from s to t (namely, $min(s,t)$).

Theorem 1. *For any states* s *and* t *(reachable from* s*),* $o_k \notin ss(P_{k-1})$ *for all* k *(*$1 \leq k \leq |min(s,t)|$*).*

This is a special case of the general theorem (Theorem 7) proven below.

Our implementation of sleep sets has a preprocessing (one-time) step, and a computation that occurs for each node generated (the "run-time" step). In the preprocessing step, the commutativity, $o_1 \bowtie o_2$, of every pair of operators o_1, o_2 is tested and the result stored in a Boolean table named $commutes$, i. e. $commutes[o_1, o_2]$ is $true$ iff $o_1 \bowtie o_2$. Commutativity of operators o_1 and o_2 is checked syntactically on the operator definitions and holds if the variables written by o_1's effect are disjoint from the variables of o_2's precondition, and vice versa, and both o_1's and o_2's effects do not set a common variable to a different value. This step is quadratic in the number of operators. In the run-time step, if state s' is reached by operator sequence Po, then $ss(Po)$ is computed by scanning through the operators that are in $ss(P)$ or precede o in the order $\mathcal{O}$, and adding operator o' to $ss(Po)$ iff $commutes[o, o']$. This step is linear in the number of operators. When expanding s', only the operators that do not belong to $ss(Po)$ are applied.

Operator Sequence Redundancy

The notion of operator sequence redundancy we build upon, and its use for move pruning, were first introduced by Taylor and Korf (1993) and later generalized by Holte and Burch (2014).

Definition 6. *Operator sequence* B *is* redundant *with operator sequence* A*, denoted with* $A \leq B$*, iff the following conditions hold:*

1. $cost(B) \geq cost(A)$
2. $pre(B) \subseteq pre(A)$
3. $s \in pre(B) \Rightarrow B(s) = A(s)$

In addition, we write $A \equiv B$ to denote that A is *equivalent* to B, i. e., that $A \leq B$ and $B \leq A$.

Theorem 2 (Holte and Burch, 2014)**.** *Let* $\mathcal{O}$ *be any nested order on operator sequences and* B *any operator sequence. If there exists an operator sequence* A *such that* B *is redundant with* A *and* $B >_{\mathcal{O}} A$*, then* B *does not occur as a consecutive subsequence in* $min(s,t)$ *for any states* s*,* t*.*

Holte and Burch's move pruning works as follows. If P is an operator sequence, p an operator, and there exists an operator sequence A such that $Pp >_{\mathcal{O}} A$ and move pruning analysis establishes that $Pp \geq A$, then, just like with sleep sets, p will not applied after P. Theorem 2 shows that move pruning is safe in the sense that for any state s and any state t reachable from s, it preserves at least one least-cost path from s to t (namely, $min(s,t)$).

Similar to sleep sets, our implementation of move pruning has two parts, a preprocessing step, and run-time step. As in Holte and Burch (2014)'s implementation, the user specifies a length L and the preprocessing step generates all operator sequences of length L or less and tests each sequence with all the sequences that precede it in the order $\mathcal{O}$. The redundancy test for a given pair of operator sequences is the same complexity as the commutativity test done for sleep sets. This step is quadratic in the number of operator sequences of length L or less, not in the number of operators, and so is considerably more expensive than the preprocessing step for sleep sets even when $L = 2$. The redundancy relation

is computed for all pairs of operator sequences of length L or less and stored in a Boolean table named $redundant$. For example, for $L = 2$, $redundant[o_1, o_2]$ is $true$ iff there exists an operator sequence B of length 2 or less such that $B \leq o_1 o_2$ and $B <_{\mathcal{O}} o_1 o_2$. In the run-time step, if state s' is generated by applying operator o to state s, then operator o' is applied in s' iff $redundant[o, o']$ is $false$. This check takes constant time per operator.

As discussed by Holte and Burch (2014), the pruning power of sleep sets and move pruning is incomparable. On the one hand, sleep sets can prune more than move pruning because sleep sets propagate information along paths: for a path Pp, an operator o is contained in $ss(Pp)$ if o and p are commutative, and $o <_{\mathcal{O}} p$ or o has been in $ss(P)$ already. This recursive nature can propagate o from $ss(P)$ to $ss(Pp)$ and, by the same reasoning, o can be propagated in a sleep set an arbitrary distance. Move pruning does not perform such propagations. The following example, which has previously already been discussed by Wehrle and Helmert (2012) and Holte and Burch (2014), illustrates this more clearly: consider a sequence that consists of operators o_1, o_2 and c which are ordered $o_2 <_{\mathcal{O}} c <_{\mathcal{O}} o_1$ by both techniques. Suppose c is commutative with both o_1 and o_2, then sequences $o_1 o_2 c$, $o_1 c o_2$, and $c o_1 o_2$ are equivalent. Given the ordering, sleep sets will preserve sequence $c o_1 o_2$ and prune the other two sequences. Move pruning will explore both $o_1 o_2 c$ and $c o_1 o_2$ and prune only $o_1 c o_2$. On the other hand, move pruning can be more powerful than sleep sets because it can handle sequences of length $L > 2$. In addition, move pruning with $L = 2$ computes a redundancy relation that is not restricted to the same operators on both side of the inequality, such as $o_1 o_2 \equiv o_3 o_4$, and strict redundancies such as $o_1 o_2 < o_2 o_1$.

Throughout this paper we assume that the operator sequence order used for redundancy analysis is the length-lexicographic order induced by the operator order used to define sleep sets, and hence we use $\mathcal{O}$ for both.

Evaluation of Sleep Sets and Move Pruning

We implemented sleep sets and move pruning within the Fast Downward planning system (Helmert 2006). We evaluate these pruning techniques with IDA*, with cycle detection, using the LM-Cut heuristic (Helmert and Domshlak 2009), which is the state-of-the-art admissible planning heuristic, on all the planning benchmarks from the international planning competitions up to 2011 that contain the features required by LM-Cut (1396 problem instances drawn from 44 domains). In the following, IDA*SS refers to IDA* with pruning based on sleep sets, and IDA*MP refers to IDA* with move pruning. The evaluation is performed on Intel Xeon E5-2660 CPUs that run at 2.2 GHz, with a time limit of 30 minutes (for the total of search time and preprocessing time) and a memory limit of 2 GB per run. For move pruning we set $L = 2$. Times that are less than or equal to 0.1 seconds are shown in the plots as 0.1.

The plots in Figure 1 compare IDA* (y-axis) and IDA*SS (x-axis) in terms of the number of nodes generated (left) and search time (preprocessing time not included) in seconds (right) using logarithmic scales. Each point represents a single problem instance. Points above the diagonal are instances for which IDA*SS outperformed IDA*. To easily gauge how much better/worse one algorithm is than another, reference lines are shown for y=2x, y=10x, and y=50x. Points at the very top of the plot are instances that were solved by IDA*SS but not by IDA* because the latter exceeded the time limit.

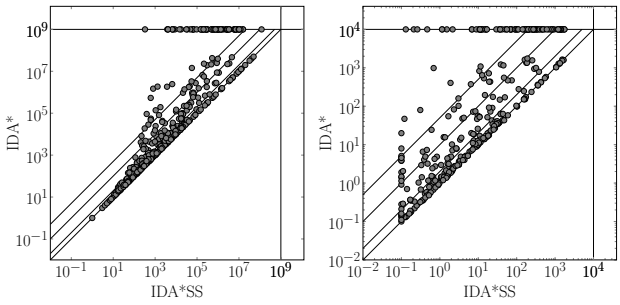

Figure 1: IDA* (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time.

We see that there are many instances in which IDA*SS and IDA* generate exactly the same number of nodes. These are instances in which none of the operators commute or the operators that do commute are not encountered during search. On most of the other problem instances, IDA*SS generates substantially fewer nodes, in some cases as much as 3 orders of magnitude fewer. The search time plot (right) shows that the reduction in the number of nodes generated translates into comparable reductions in search time.

Figure 2 is the same format as Figure 1 but compares IDA*MP (y-axis) to IDA*SS (x-axis). The points at the top of the plot are instances for which the memory required by move pruning's redundancy table exceeded the memory limit but the sleep set commutativity table did not.[1] We see that in the vast majority of the instances, exactly the

[1] Currently, the redundancy table is precomputed in its entirety. In practice, one would not abort search when the memory limit was reached but would instead record as much information as memory allowed and base move pruning on that partial information. In the worst-case this would give a performance equal to IDA*'s.

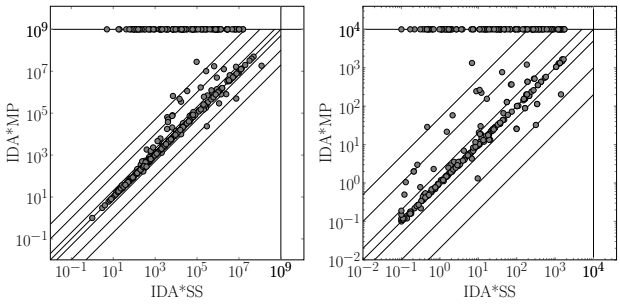

Figure 2: IDA*MP (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time.

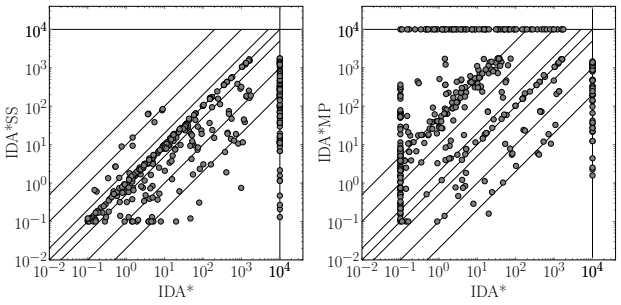

Figure 3: Total time for IDA* (x-axis) vs. IDA*SS (left plot, y-axis) and IDA*MP (right plot, y-axis).

same number of nodes were generated. These are cases in which there was no commutativity or operator sequence redundancy (for L=2), or where the two methods detected exactly the same redundancy. When the methods produce different results, move pruning is superior to sleep set pruning in some cases (points below the diagonal) and inferior in others. This is to be expected if the set of problem instances is truly representative because move pruning detects more kinds of redundancy than just commutativity, but does not pass information forward along a path the way sleep sets do. The search time plot (right) shows that the difference in the number of nodes generated translates into comparable differences in search time.

The conclusions from the data presented so far is that both pruning methods substantially improve IDA*'s performance and that neither pruning method dominates the other. There is, however, one additional consideration, and that is the time required for preprocessing. Figure 3 shows the total time (search time + preprocessing time) required to solve each instance by IDA* (x-axis in both plots), whose total time is its search time, IDA*SS (left plot, y-axis), and IDA*MP (right plot, y-axis). IDA*SS's preprocessing time is small enough that its total time is less than IDA*'s in most cases (points below the diagonal) and its "coverage" (i. e., the number of problem instances that could be solved within the resource limits) is much better than IDA*'s (570 compared to 477, where all instances that could be solved by IDA*SS, but not by IDA* are due to exceeding the time limit). The opposite is true for IDA*MP. Its total time is worse than IDA*'s in the majority of the instances, and is often more than an order of magnitude larger. Consequently, its coverage is worse than IDA*'s (443 compared to 477). Move pruning is most appropriate when its preprocessing step, which is independent of the search goal and start state, can be reused across many instances.

Generalized Sleep Sets

Commutativity of operators, the basis for defining sleep sets, is a very restricted type of operator sequence redundancy. We now show that sleep sets can be defined in terms of much richer notions of operator sequence redundancy. We present this theoretical work in a series of definitions of successively greater generality. We do this because, as the definitions get more general, the computational complexity of the preprocessing and/or run-time step increases and we might get better overall performance using a less general definition. Furthermore, whether a generalized definition is beneficial depends on the properties of the problem domain, so it could happen that any one of the generalized sleep set definitions could be the method of choice in a particular domain.

Because the "or" in Definition 5 ($o <_{\mathcal{O}} p$ or $o \in ss(\sigma)$) reflects very different situations, we separate them and refer to them as "anchor points" and "relay points" respectively.

Definition 7. *A* $\bowtie$-anchor point *for operator* o' *in operator sequence* $p_1 \dots p_n$ *is an index* i' *(*$1 \leq i' \leq n$*) for which* $o' \bowtie p_{i'}$ *and* $o' <_{\mathcal{O}} p_{i'}$.

Informally, anchor points for an operator o' describe positions where o' is added to the sleep set. The operators of a (generalized) sleep set that are not anchor points we call *relay points*, since their role is to relay operator o' to an earlier point in the sequence.

Definition 8. *A* $\bowtie$-relay point *for operator* o' *in operator sequence* $p_1 \dots p_n$ *is an index* i' *(*$2 \leq i' \leq n$*) for which* $o' \bowtie p_{i'}$*, and (*$i' - 1$*) is a* $\bowtie$*-anchor point or* $\bowtie$*-relay point for* o' *in* $p_1 \dots p_{i'-1}$.

Definition 9. *A set* $gss^{\bowtie}(P)$ *for an operator sequence* P *is a* generalized sleep set *with respect to* $\bowtie$ *iff the following hold:* $gss^{\bowtie}(\varepsilon) = \emptyset$*, and, if* $|P| = n > 0$*,* $gss^{\bowtie}(P) = \{o \mid n$ *is a* $\bowtie$*-anchor or* $\bowtie$*-relay point for* o *in* $P\}$.

This definition is equivalent to Definition 5, i. e. $gss^{\bowtie}(P) = ss(P)$ for all operator sequences P. We will base our subsequent definitions of generalized sleep sets on Definition 9. They will differ only in replacing $\bowtie$ in the definitions of anchor and relay points by successively more general types of operator sequence redundancy.

Beyond Commutativity

The first generalization of sleep sets is based on the observation that it is not necessary for o' and $p_{i'}$ to be commutative, but only that $o'p_{i'} \leq p_{i'}o'$. The following definitions are based on this observation; in them $\bowtie$ is generalized to $\leq$.

Definition 10. *A* $\leq$-anchor point *for operator* o' *in operator sequence* $p_1 \dots p_n$ *is an index* i' *(*$1 \leq i' \leq n$*) for which* $o'p_{i'} \leq p_{i'}o'$ *and* $o' <_{\mathcal{O}} p_{i'}$.

Definition 11. *A* $\leq$-relay point *for operator* o' *in operator sequence* $p_1 \dots p_n$ *is an index* i' *(*$2 \leq i' \leq n$*) for which* $o'p_{i'} \leq p_i o'$ *and (*$i' - 1$*) is an* $\leq$*-anchor point or* $\leq$*-relay point for* o' *in* $p_1 \dots p_{i'-1}$.

Definitions 7 and 8 are special cases of these definitions because $A \bowtie B \Rightarrow AB \leq BA$ (and also $BA \leq AB$).

Flexible Leapfrogging

The definitions just given use the fact $o'p_{i'} \leq p_{i'}o'$ to leapfrog o' over $p_{i'}$ to get a new operator sequence that is equal or superior to the sequence we started with from a redundancy point of view. There is no reason to require $p_{i'}$ to remain unchanged when o' leapfrogs over it, the properties that are required will still hold if $p_{i'}$ is replaced by some other operator, or even ε, as long as the resulting sequence

is equal or superior to the sequence we started with from a redundancy point of view. Formally, we generalize the requirement that $o'p_{i'} \leq p_{i'}o'$ to $\exists y$ s.t. $o'y \leq p_{i'}o'$, where y can be an operator or ε.

Similarly, we do not require o' to be preserved when it leapfrogs over $p_{i'}$, it could be changed to some other operator z. The definitions of anchor and relay points based on these ideas are as follows.

Definition 12. *A* flexible $\leq$-anchor point *for operator o' in operator sequence $p_1 \dots p_n$ is an index i' ($1 \leq i' \leq n$) for which $\exists y$, an operator or ε, and $\exists z$, an operator or ε s.t. $zy \leq p_{i'}o'$ and $z <_{\mathcal{O}} p_{i'}$.*

Definition 13. *A* flexible $\leq$-relay point *for operator o' in operator sequence $p_1 \dots p_n$ is an index i' ($2 \leq i' \leq n$) for which $\exists y$, an operator or ε, and operator z s.t. $zy \leq p_{i'}o'$ and $(i'-1)$ is a flexible $\leq$-anchor or flexible $\leq$-relay point for z in $p_1 \dots p_{i'-1}$.*

Definitions 10 and 11 are special cases of these definitions in which $y = p_{i'}$ and $z = o'$.

Long Distance Leapfrogging

The aim of the final generalization is to allow sleep sets to take advantage of redundancy information such as $o'p_1 \dots p_n \leq p_1 \dots p_n o'$ when $o'p_i \not\leq p_i o'$ for all i ($1 \leq i \leq n$). In other words, we want o' to be able to leapfrog over entire sequences in a single bound ("long distance leapfrogging"), rather than having it go through a sequence one operator at a time. The definitions of anchor point and relay point that permit long distance leapfrogging are as follows.

Definition 14. *A* long distance $\leq$-anchor point *for operator o' in sequence $p_1 \dots p_n$ is an index i' ($1 \leq i' \leq n$) for which there exist an index k ($1 \leq k \leq i'$), and an operator sequence Y such that*

- ***(A1)*** *$Y \leq p_k \dots p_{i'}o'$, and*
- ***(A2)*** *$\overleftarrow{Y} <_{\mathcal{O}} p_k \dots p_{i'}$.*

Definition 15. *A* long distance $\leq$-relay point *for operator o' in sequence $p_1 \dots p_n$ is an index i' ($2 \leq i' \leq n$) for which there exist an index k ($2 \leq k \leq i'$), operator z, and operator sequence Y such that*

- ***(R1)*** *$zY \leq p_k \dots p_{i'}o'$,*
- ***(R2)*** *$|Y| \leq |p_k \dots p_{i'}|$, and*
- ***(R3)*** *$(k-1)$ is a long distance $\leq$-anchor or long distance $\leq$-relay point for z in $p_1 \dots p_{k-1}$.*

Definitions 12 and 13 are special cases of these definitions in which $k = i'$ and Y here is y in those definitions.

Proof that Generalized Sleep Sets are Safe

The theorems in this section show that pruning with generalized sleep sets based on long distance $\leq$-anchor and relay points is safe. For brevity, we write "anchor" and "relay" points to denote their long distance variants. The theorems' proofs use the following lemmas.

Lemma 3. *The relation "$\leq$" is transitive.*

Proof. Let A, B, C be operator sequences such that $A \leq B$ and $B \leq C$. We show that $A \leq C$.

First, $cost(A) \leq cost(C)$ because $cost(A) \leq cost(B) \leq cost(C)$. Second, $pre(A) \supseteq pre(C)$ holds because of $pre(A) \supseteq pre(B) \supseteq pre(C)$. Third, $s \in pre(C) \Rightarrow A(s) = C(s)$ holds because $s \in pre(C) \Rightarrow s \in pre(B)$, $C(s) = B(s)$, and $s \in pre(B) \Rightarrow B(s) = A(s)$. Therefore $C(s) = A(s)$ and hence, $A \leq C$. □

Lemma 4. *Let A and B be any operator sequences such that $A \leq B$, and let C be any operator sequence. Then $AC \leq BC$ and $CA \leq CB$.*

Proof. We exemplarily show that $AC \leq BC$ (the proof for $CA \leq CB$ is analogous). For this, we will show that $cost(AC) \leq cost(BC)$, $pre(AC) \supseteq pre(BC)$, and $s \in pre(BC) \Rightarrow AC(s) = BC(s)$.

Firstly, $cost(AC) = cost(A) + cost(C) \leq cost(B) + cost(C) = cost(BC)$.

Secondly, $s \in pre(BC) \Rightarrow s \in pre(B)$ and $B(s) \in pre(C)$. Because $A \leq B$ it follows that $s \in pre(A)$ and $A(s) \in pre(C)$, i.e. that $s \in pre(AC)$.

Thirdly, $s \in pre(BC) \Rightarrow BC(s)$ and $AC(s)$ are both defined. $BC(s) = C(B(s)) = C(A(s)) = AC(s)$. □

Lemma 5. *Let $\mathcal{O}$ be any length-lexicographic order on operator sequences, and A, B, X, Y and Z any operator sequences such that $A <_{\mathcal{O}} B$ and $|X| \leq |Y|$. Then $ZAX <_{\mathcal{O}} ZBY$.*

Proof. If $|A| < |B|$ or $|X| < |Y|$ then $|ZAX| < |ZBY|$ and therefore $ZAX <_{\mathcal{O}} ZBY$. On the other hand, if $|A| = |B|$ and $|X| = |Y|$, then $|ZAX| = |ZBY|$. If o_a and o_b are the leftmost operators where A and B differ (o_a is in A and o_b is in the corresponding position in B), they are also the leftmost operators where ZAX and ZBY differ. Because $A <_{\mathcal{O}} B$ we have $o_a <_{\mathcal{O}} o_b$ and hence $ZAX <_{\mathcal{O}} ZBY$. □

Theorem 6. *For any length-lexicographic order on operator sequences $\mathcal{O}$, any $n \geq 1$, any non-empty operator sequence $P = p_1 \dots p_n$, and any operator $p \in gss(P)$, there exists an operator sequence P' such that*

- ***(P1)*** *$P' \leq Pp$,*
- ***(P2)*** *$\overleftarrow{P'} <_{\mathcal{O}} P$*

Proof. By induction on n.

Base case: $n = 1$. Because $gss(\varepsilon) = \emptyset$, n cannot be a relay point for p in P, it must be an anchor point. Therefore there exist k and Y such that $A1$ and $A2$ hold with $i' = n = 1$ and $o' = p$. Because $n = 1$, we must have $k = 1$. Setting $P' := Y$ satisfies the requirements of the theorem because P1 and P2 are exactly A1 and A2, respectively.

Inductive case: Assuming the theorem is true for all m, $1 \leq m \leq n$, we must prove it is true for $n+1$. Let $P = p_1 \dots p_{n+1}$ be any operator sequence of length $n+1$ and p any operator such that $p \in gss(P)$.

If $n+1$ is an anchor point for p in P there exist k and Y such that $A1$ and $A2$ hold with $i' = n+1$ and $o' = p$. Setting $P' := p_1 \dots p_{k-1}Y$ satisfies the requirements of the theorem for the following reasons.

P1: P1 follows directly from $A1$ and Lemma 4 (by prepending $p_1 \dots p_{k-1}$ to both sides of the $A1$ inequality).

P2: If Y is not the empty sequence, then $\overleftarrow{P'}$ is equal to $p_1 \dots p_{k-1}\overleftarrow{Y}$ and P2 follows directly from $A2$ and Lemma 5 (by prepending $p_1 \dots p_{k-1}$ to both sides of the $A2$ inequality). If Y is the empty sequence, then P2 holds trivially because $\overleftarrow{P'} = p_1 \dots p_{k-2}$ is a prefix of P.

If $n+1$ is not an anchor point for p in P it must be a relay point, in which case there exist k, z, and Y such that $R1$, $R2$, and $R3$ hold with $i' = n+1$ and $o' = p$. From $R3$ and the inductive hypothesis, there exists an operator sequence $\overline{P'} = \overline{p'}_1 \dots \overline{p'}_{|\overline{P'}|}$ such that

($\overline{P1}$) $\overline{P'} \leq p_1 \dots p_{k-1}z$,

($\overline{P2}$) $\overleftarrow{\overline{P'}} <_{\mathcal{O}} p_1 \dots p_{k-1}$.

Setting $P' := \overline{P'}Y$ satisfies the requirements of the theorem for the following reasons.

P1: We need to show that $P' \leq Pp$, i. e., that $\overline{P'}Y \leq Pp$. From $\overline{P1}$, we have $\overline{P'} \leq p_1 \dots p_{k-1}z$. By Lemma 4 (appending Y to both sides of the inequality) this implies $\overline{P'}Y \leq p_1 \dots p_{k-1}zY$. By $R1$ and Lemma 4 (prepending $p_1 \dots p_{k-1}$ to both sides of the inequality), we have $p_1 \dots p_{k-1}zY \leq p_1 \dots p_{k-1}p_k \dots p_{n+1}p = Pp$. By the transitivity of "$\leq$" (Lemma 3) we have $P' = \overline{P'}Y \leq Pp$, i.e. P1 is true.

P2: We need to show that $\overleftarrow{P'} <_{\mathcal{O}} P$, i.e. that $\overleftarrow{\overline{P'}Y} <_{\mathcal{O}} P$.

Let $M = \overleftarrow{(qY)}$, where q is the last operator in $\overline{P'}$ if $\overline{P'} \neq \varepsilon$, and $q = \varepsilon$ if $\overline{P'} = \varepsilon$. Then $|M| = |\overleftarrow{(qY)}| \leq |Y|$. Combining this with $R2$, we get $|M| \leq |p_k \dots p_{n+1}|$. Using this fact in Lemma 5 together with the inequality $\overleftarrow{P'} <_{\mathcal{O}} p_1 \dots p_{k-1}$ from $\overline{P2}$ we have $\overleftarrow{P'}M <_{\mathcal{O}} p_1 \dots p_{k-1}p_k \dots p_{n+1} = P$. But $\overleftarrow{P'}M = \overleftarrow{(\overline{P'}Y)}$, hence P2 is true. □

The following proves the safety of generalized sleep sets.

Theorem 7. *For any length-lexicographic order on operator sequences $\mathcal{O}$ and any states s and t (reachable from s), $o_k \notin gss(P_{k-1})$ for all k, $1 \leq k \leq |min(s,t)|$.*

Proof. It is true if $k = 1$, since $P_0 = \varepsilon$ and $gss(\varepsilon) = \emptyset$. For $k \geq 2$, we will derive a contradiction from $o_k \in gss(P_{k-1})$. If $o_k \in gss(P_{k-1})$ then, by Theorem 6, there exists an operator sequence P' such that

(P1) $P' \leq P_{k-1}o_k = P_k$,

(P2) $\overleftarrow{P'} <_{\mathcal{O}} P_{k-1}$.

Let $P = P'Q_{k+1}$. From P1 and Lemma 4 (appending Q_{k+1} to both sides of the inequality), we have $P = P'Q_{k+1} \leq P_kQ_{k+1} = min(s,t)$. In particular, P is a least-cost path from s to t. Let q = the last operator in P' if $P' \neq \varepsilon$, and $q = \varepsilon$ if $P' = \varepsilon$. From P2, and Lemma 5 (appending qQ_{k+1} to the lefthand side of the inequality and o_kQ_{k+1} to the righthand side) we get $P'Q_{k+1} <_{\mathcal{O}} P_kQ_{k+1}$, i.e. $P <_{\mathcal{O}} min(s,t)$. These two facts about P contradict $min(s,t)$ being the least-cost path that is smallest according to $\mathcal{O}$. □

Relationship to Move Pruning

To see the very close connection between move pruning and long distance $\leq$-anchor points (Def. 14), we define move pruning in terms of "move pruning points".

Definition 16. *A* move pruning point *for operator o' in operator sequence $p_1 \dots p_n$ is an index i' ($1 \leq i' \leq n$) for which there exist an index k ($1 \leq k \leq i'$) and an operator sequence W such that*

(MP1) *$W \leq p_k \dots p_{i'}o'$, and*

(MP2) *$W <_{\mathcal{O}} p_k \dots p_{i'}o'$.*

If n is a move pruning point for o' in $P = p_1 \dots p_n$, then move pruning would refuse to apply o' after P, just as sleep sets would refuse to apply o' after P if n was an anchor point for o' in P. There are two differences between move pruning points and long distance $\leq$-anchor points.

First, while move pruning allows $\mathcal{O}$ to be any nested order, sleep sets require it to be a length-lexicographic order. Although there do exist nested orders that are not length-lexicographic, in practice move pruning always applies a length-lexicographic order, so overall, this will not give move pruning a practical advantage over sleeps sets.

The second difference is more interesting. Let $B = p_k \dots p_{i'}o'$. Definition 16 requires $W <_{\mathcal{O}} B$ (MP2), but Definition 14 requires $\overleftarrow{W} <_{\mathcal{O}} \overleftarrow{B}$ (A2). The latter implies the former (Lemma 5) but not vice versa (e.g. $ab <_{\mathcal{O}} ac$ could be true but $a <_{\mathcal{O}} a$ is always false). Changing A2 in Definition 14 to $zY <_{\mathcal{O}} p_k \dots p_{i'}o'$ would render Theorems 6 and 7 false. The reason can be traced back to Definition 13, which allows an arbitrary operator z to be substituted for o' at a relay point. As a result, generalized sleep sets dominate the original version of sleep sets in terms of pruning power, but not move pruning (at least not on a theoretical level). We believe it is possible to restrict the definition of relay point (Definition 15) so that it is safe to allow any move pruning point to be an anchor point. That would make generalized sleep sets dominate *both* sleep sets and move pruning. To summarize, generalized sleep sets and move pruning are incomparable to each other in terms of pruning power. Move pruning can be more powerful since the definition of move pruning points is weaker than long distance $\leq$-anchor points, and hence can lead to more pruning than sleep sets in some cases. On the other hand, like sleep sets, generalized sleep sets can propagate operators along paths of arbitrary length, thereby pruning paths that cannot be pruned by move pruning. Deciding which technique is more suitable than the other depends mainly on the domain of choice.

Experiments with Generalized Sleep Sets

We repeated the experiment of the earlier section using the most general of the sleep set definitions. We restricted the redundancy analysis to operator sequences of length $L = 2$ or less, which makes long distance leapfrogging conceptually the same as flexible leapfrogging (called IDA*GSS in the following). Figure 4 shows the results in the same format as Figure 1. In the vast majority of the cases IDA*GSS (y-axis) generates exactly the same number of nodes as IDA*SS (x-axis), meaning that in those problem instances,

the generalized method does not find any redundancies except the commutativity that basic sleep sets find. In the other problem instances, we see the number of nodes generated by IDA*GSS can be substantially smaller, by as much as an order of magnitude. For example, in the largest task solved by both techniques in domains from the IPC-11, this is the case in Elevators (10,136,319 nodes with IDA*SS vs. 1,045,250 nodes with IDA*GSS), Openstacks (7,253,474 vs. 1,066,192), Transport (2,474,615 vs. 783,932) and Woodworking (3,644,753 vs. 1,181,201). The overhead for using the more general method is higher than the overhead for basic sleep sets, which is reflected in the search time plot in the points with x $\leq 10^1$ being slightly above the diagonal. The preprocessing time for IDA*GSS is comparable to the one for IDA*MP which, as we saw in the first experiment, is so large that these methods have worse coverage than plain IDA* (421 and 443, respectively, compared to 477). Hence, they are most appropriate when their preprocessing steps, which are independent of the search goal and start state, can be reused across many instances. To illustrate this, we have calculated for each domain a value (which we call the breakeven point) that represents the number of instances that would need to be solved such that the preprocessing starts paying off. Suppose solving a single problem without the preprocessing takes time t_0, on average, solving it after doing the preprocessing takes time t_1 seconds, and the preprocessing itself takes time t_2. To solve N problems takes time $N \times t_0$ without preprocessing and takes time $t_2 + N \times t_1$ with preprocessing. The breakeven points are the values of N when $N \times t_0 = t_2 + N \times t_1$. In several domains, the values of N are relatively small (< 50), showing that the precomputation can amortize rather quickly.

Finally, we also ran experiments with $L = 3$, which resulted in some additional pruning and, of course, higher preprocessing time again.

Note that the preprocessing times for all the methods increases with the number of operators and these planning domains have a large number of operators because they are fully grounded. If the domains were encoded directly in a language that did not require full grounding of the operators, such as PSVN (Holte, Arneson, and Burch 2014), the preprocessing times would be much smaller.

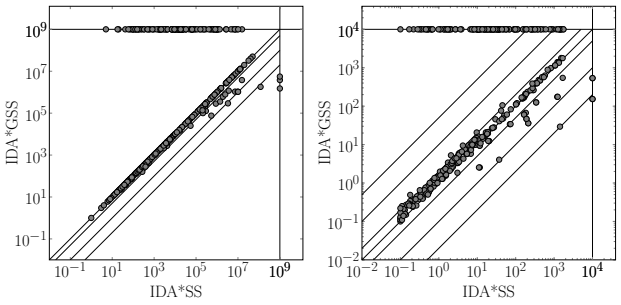

Figure 4: IDA*GSS (y-axis) vs. IDA*SS (x-axis). Left: Number of nodes generated. Right: Search time.

Conclusions

We have given the first evaluation of sleep sets and move pruning on the benchmark planning domains. Our experiments show that both methods substantially reduce the number of nodes generated and search time of IDA* and that the preprocessing time for sleep sets is small enough that IDA* with sleep set pruning usually is faster than IDA* even when preprocessing time is taken into account. For move pruning, the preprocessing time is sufficiently large that it is best used when the preprocessing can be amortized over many problem instances. Furthermore, we have provided a generalized notion of sleep sets based on operator sequence redundancy and showed that the generalized definitions can further improve the performance of sleep sets. The increased computational overhead suggests that generalized sleep sets are particularly suited for formalisms like PSVN where operators are not fully grounded. It will be interesting to further investigate this direction in the future.

Acknowledgments

This work was supported by the Natural Science and Enginerring Research Council of Canada (NSERC), by the Swiss National Science Foundation (SNSF) as part of Robert Holte's visit to the University of Basel and as part of the project "Safe Pruning in Optimal State-Space Search (SPOSSS)", and by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Center "Automatic Verification and Analysis of Complex Systems".

References

Bäckström, C., and Nebel, B. 1995. Complexity results for SAS$^+$ planning. *Computational Intelligence* 11(4):625–655.

Godefroid, P. 1996. *Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer-Verlag.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *Proc. ICAPS 2009*, 162–169.

Helmert, M. 2006. The Fast Downward planning system. *Journal of Artificial Intelligence Research* 26:191–246.

Holte, R. C., and Burch, N. 2014. Automatic move pruning for single-agent search. *AI Communications* 27(4):363–383.

Holte, R. C.; Arneson, B.; and Burch, N. 2014. PSVN Manual (June 20, 2014). Technical Report TR14-03, Computing Science Department, University of Alberta.

Korf, R. E. 1985. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence* 27(1):97–109.

Taylor, L. A., and Korf, R. E. 1993. Pruning duplicate nodes in depth-first search. In *Proc. AAAI 1993*, 756–761.

Wehrle, M., and Helmert, M. 2012. About partial order reduction in planning and computer aided verification. In *Proc. ICAPS 2012*.