An Atom-Centric Perspective on Stubborn Sets

Abstract
Stubborn sets are an optimality-preserving pruning technique for factored state-space search, for example in classical planning. Their applicability is limited by their computational overhead. We describe a new algorithm for computing stubborn sets that is based on the state variables of the state space, while previous algorithms are based on its actions. Typical factored state spaces tend to have far fewer state variables than actions, and therefore our new algorithm is much more efficient than the previous state of the art, making stubborn sets a viable technique in many cases where they previously were not.

Introduction
Heuristic search is a common approach for classical domain-independent planning. Especially in optimal planning, the search suffers from a state explosion problem that occurs if states can be reached by applying the same actions in different orders. Indeed, Helmert and Röger (2008) showed that even with close-to-perfect heuristics, the number of nodes that must be explored by pure heuristic search (only relying on node expansions and an admissible heuristic) can grow exponentially in the size of the planning task. Since we cannot hope for efficiently computable perfect heuristics, search algorithms are often enhanced with pruning techniques that reduce the size of the explored state space.

One family of such pruning techniques is partial order reduction, which allows the search to ignore some paths to the goal by not considering all permutations of the actions. Intuitively, the idea is to avoid interleaving the solution of independent subproblems but instead solving one subproblem after the other. Partial order reduction was originally introduced by Valmari (1989) for petri-nets in the context of computer-aided verification. Alkhazraj et al. (2012) transferred his concept of strong stubborn sets to classical planning. Later on, Wehrle and Helmert (2014) generalized them with more fine-grained criteria that are still sufficient for optimality-preserving pruning. With suitable decisions at certain choice points, strong stubborn sets strictly dominate the expansion core method (Chen and Yao 2009; Wehrle and Helmert 2012), a partial order reduction technique introduced earlier for planning (Wehrle et al. 2013).

A stubborn set for a state is a set of actions so that all other actions can safely be ignored at its expansion. The concept is inherently action-centric and so are the underlying definitions and algorithms. In this paper, we adopt a more atom-centric perspective on their computation, which gives rise to a significantly faster algorithm. As an additional enhancement, we also contribute a new atom selection strategy, which has a tendency to produce smaller stubborn sets which leads also empirically to a better pruning power.

Background
We consider SAS+ planning tasks (Bäckström and Nebel 1995), extended with non-negative action costs. A task is defined over a finite set V of variables, each associated with a finite domain D(v). A pair (v, d) with v ∈ V and d ∈ D(v) is called an atomic proposition, or atom for short, and we use P to denote the set of all atomic propositions (over an implicit set of variables V). We call all atoms (v, d') with d' ∈ D_v \ {d} the siblings of atom (v, d).

A partial state s maps every variable v from a set vars(s) ⊆ V to a value s[v] from D(v). If vars(s) = V, we call s a state. When it is suitable, we also consider a partial state s as the set of atoms {(v, s[v]) | v ∈ vars(s)} and write (v, d) ∈ s for s[v] = d.

A task is given as a tuple II = (V, A, s_i, s_G) where V is the finite set of variables, A a finite set of actions, s_i a state called the initial state and s_G a partial state called the goal. Each action a ∈ A is defined by its cost c(a) ∈ R+, and two partial states pre(a) and eff(a), called its precondition and effect, respectively. If (v, d) ∈ eff(a) for some atom (v, d), we say that action a achieves (v, d). If (v, d) ∈ pre(a), we say a depends on (v, d). W.l.o.g. we require that no action both depends on and achieves the same atom.

An action a is applicable in state s if pre(a) ⊆ s. Then the successor state s' is given as s'[v] = eff(a)[v] for all v ∈ vars(eff(a)) and s'[v] = s[v] for all other variables. Slightly abusing notation, we write a(s) for the successor state resulting from applying action a in state s.

A goal state is a state s with s_G ⊆ s. A plan is a sequence of actions that are subsequently applicable in s_i and where the resulting state is a goal state. The cost of a plan is the sum of the individual action costs. A plan is optimal if it has minimal cost among all plans. Wehrle and Helmert (2014) pointed out that for correct pruning it is sometimes impor-
tant to only consider so-called strongly optimal plans, which are optimal plans with a minimal number of 0-cost actions among all optimal plans. If there is no plan for a task, the task is unsolvable. The aim of optimal planning is to find an optimal plan or to prove that the task is unsolvable.

Strong stubborn sets aim to prune permuted plans from the search. On a lower level, the permutation of actions is related to the following notion of interference.

**Definition 1** (interference, Wehrle and Helmert 2014). Let \( a_1 \) and \( a_2 \) be actions and let \( s \) be a state of a planning task \( \Pi \). We say that \( a_1 \) and \( a_2 \) interfere in \( s \) if they are both applicable in \( s \), and

- \( a_1 \) disables \( a_2 \), i.e., \( a_2 \) is not applicable in \( a_1(s) \), or
- \( a_2 \) disables \( a_1 \), or
- \( a_1 \) and \( a_2 \) conflict in \( s \), i.e., \( a_2(a_1(s)) \) and \( a_1(a_2(s)) \) are both defined but differ.

If two actions that are both applicable in a state \( s \) do not interfere in \( s \), we can apply them in any order and will in both cases reach the same state.

The second relevant notion are necessary enabling sets. These are related to disjunctive action landmarks (Helmert and Domshlak 2009), which are sets of actions of which at least one must be applied in every plan. Similarly, necessary enabling sets are sets of actions of which at least one must be applied before a given action is applied in every action sequence from a given set.

**Definition 2** (necessary enabling set, Wehrle and Helmert 2014). Let \( \Pi \) be a planning task, let \( a \) be one of its actions, and let \( Seq \) be a set of action sequences applicable in the initial state of \( \Pi \).

A necessary enabling set for \( a \) and \( Seq \) is a set \( N \) of actions such that every action sequence in \( Seq \) which includes \( a \) as one of its actions also includes some action \( a' \in N \) before the first occurrence of \( a \).

For this paper, we build on the generalized definition of strong stubborn sets by Wehrle and Helmert (2014) but for clarity we omit the concept of envelopes, which permit to safely ignore some actions. Empirically, the known methods for exploiting envelopes did not provide much benefit (Wehrle and Helmert 2014), and they can easily be re-integrated in our work.

**Definition 3** (strong stubborn set). Let \( s \) be a state of planning task \( \Pi = \langle V, A, s, s_G \rangle \) and let \( \Pi_s = \langle V, A, s, s_G \rangle \). A strong stubborn set in \( s \) is a set \( A \subseteq A \) of actions under the following conditions.

If \( \Pi_s \) is unsolvable or \( s \) is a goal state then every \( A \) is a strong stubborn set. Otherwise, let \( Opt \) be the set of strongly optimal plans for \( \Pi_s \) and let \( S_{Opt} \) be the set of states that are visited by at least one plan in \( Opt \). The following conditions must be true for \( A \) to be a strong stubborn set.

- **C1** \( A \) contains at least one action from at least one plan from \( Opt \).
- **C2** For every \( a \in A \) that is not applicable in \( s \), \( A \) contains a necessary enabling set for \( a \) and \( Opt \).
- **C3** For every \( a \in A \) applicable in \( s \), \( A \) contains all actions from \( A \) that interfere with \( a \) in any state \( s' \in S_{Opt} \).

Wehrle and Helmert (2014) showed that the cost of an optimal solution does not change if we preserve for every state in the state space only the outgoing transitions that correspond to an action from a strong stubborn set. Put differently, in each state visited during the search we can prune all actions that are not in the strong stubborn set, while preserving the guarantee to find optimal solutions.

In practice, it is impossible to efficiently determine a minimal strong stubborn set because we do not know \( Opt \) and \( S_{Opt} \). However, if \( C2 \) and \( C3 \) hold for an overapproximation of these sets, they must also hold for the required sets.

Since the set \( S_{Opt} \) cannot be efficiently computed, for \( C3 \) it is common to use a state-independent overapproximation of interference. Alkhazraj et al. (2012) and Wehrle et al. (2013) use a purely syntactic criterion: actions \( a \) and \( a' \) potentially conflict in any state if there is a variable \( v \in vars(\text{eff}(a)) \cap vars(\text{eff}(a')) \) such that \( \text{eff}(a)[v] \neq \text{eff}(a')[v] \). Action \( a \) potentially disables \( a' \) if there is a variable \( v \in vars(\text{eff}(a)) \cap vars(\text{pre}(a')) \) such that \( \text{eff}(a)[v] \neq \text{pre}(a')[v] \). Two actions \( a \) and \( a' \) then potentially interfere if they potentially conflict, \( a \) potentially disables \( a' \), or \( a' \) potentially disables \( a \). With this definition, two actions potentially interfere if there exists some state in which they interfere.

Wehrle and Helmert (2014) strengthen this approach with mutex information: if the preconditions of two actions are mutually exclusive, they cannot both be applicable in a reachable state, so they never interfere in these states.

It is also already intractable to determine whether a given action is an element of \( Opt \). We can still determine a necessary enabling set for \( a \) and \( Opt \) by collecting all achievers of an atom that is not true in \( s \) but which \( a \) depends on. While this set is not minimal, it can be computed efficiently and indeed this is the strategy employed by previous algorithms for constructing strong stubborn sets in planning. Similarly, \( C1 \) get satisfied by picking an atom from the goal that is not true in \( s \) and including all actions that achieve this atom, hence including at least one action from every plan.

**Existing Action-Centric Algorithm**

To satisfy the properties of strong stubborn sets, previous algorithms start from an action set that satisfies \( C1 \) and successively add actions to satisfy \( C2 \) and \( C3 \) until a fixed point is reached. We will adopt the same high-level approach but will differ from this action-centric approach on a lower level.

Before we go into details, we first introduce and analyze the action-centric algorithm. Previously published pseudo-code (Alkhazraj et al. 2012; Al-Khazraj 2017) does not have the level of detail we require for our discussion, but an implementation by Wehrle and Helmert (2014) is available as part of Fast Downward (Helmert 2006). We extracted the pseudo-code as Algorithm 1 from Fast Downward 19.12.\(^3\)

The algorithm collects all actions to be included in the strong stubborn set for a non-goal state \( s \) in a collection stubborn. The actions for which it still needs to ensure \( C2 \) and \( C3 \) are tracked in a collection queue. To avoid clutter in the pseudo-code, we assume that stubborn, queue and the components of the task are globally accessible.

\(^3\)http://www.fast-downward.org/Releases/19.12
Algorithm 1 Action-centric algorithm

1: function COMPUTESTUBBORNSET(s)
2:   stubborn = empty collection
3:   queue = empty collection
4:   procedure MARKASTUBBORN(a)
5:       if a $\notin$ stubborn then
6:           stubborn.add(a)
7:           queue.add(a)
8:   procedure ENQUEUEINTERFERS(a)
9:       for a' $\in$ A potentially interfering with a do
10:          MARKASTUBBORN(a')
11:   procedure ENQUEUESTUBBORNES(atom)
12:       for a $\in$ A with atom $\in$ eff(a) do $\triangleright$ achievers
13:          MARKASTUBBORN(atom)
14:   atom = some unsatisfied goal atom
15:   ENQUEUESTUBBORNES(atom)
16:   while queue is not empty do
17:       a = queue.pop() $\triangleright$ any element
18:       if a is applicable in s then
19:          ENQUEUEINTERFERS(a)
20:       else
21:          atom = some unsatisfied atom from pre(a)
22:          ENQUEUESTUBBORNES(atom)
23:   return stubborn

The overall process for generating a strong stubborn set starts with collecting a set of actions to satisfy C1 (lines 14–15). As long as the other conditions are not yet guaranteed for some action a (lines 16–17), it includes further actions to ensure C2 (lines 20–22) or C3 (lines 18–19), depending on whether a is applicable in state s or not.

Whenever an action should be included in the result (marked as stubborn), the algorithm checks if it has already been included previously and if not includes it and enqueues it for further processing into queue (lines 4–7).

As mentioned above, necessary enabling sets are generated by starting from an atom and collecting all actions that achieve it (lines 11–13).

Complexity Analysis

In the complexity analysis, we use $p_{\text{max}}$ for the maximal size of a partial state occurring as precondition or effect of any action. In typical planning tasks, this is quite a low number. In general, it can be bound by the number $|V|$ of variables.

For an efficient implementation of Algorithm 1, we assume that all state-independent information gets precomputed and stored once for every task (i.e., only once for the entire search, not once for every node that gets expanded). This affects the set of achievers for every atom (used in line 12) and the interference relation (used in line 9).

The achievers can be determined by one pass over all actions that scans the effect and registers the action accordingly. This requires time $O(|A|p_{\text{max}})$ and the result can be stored in space $O(|P||A|)$.

Exploiting pre-sorted action preconditions and effects, the interference relation can be computed in $O(|A|^2p_{\text{max}})$, ranging over all pairs of actions and syntactically testing their potential interference in time $O(p_{\text{max}})$. With no influence on Big-O, we can halve the effort by exploiting that the relation is symmetric. The result can be stored in $O(|A|^2)$. Fast Downward uses a lazy implementation that only performs the computation for an action once it is required.

We now analyze the time complexity of a single call of COMPUTESTUBBORNSET. Using suitable data structures for stubborn (e.g., a bitset) and queue (e.g., an array-based stack), MARKASTUBBORN takes constant time. Then ENQUEUEINTERFERS takes time $O(|A|)$ because there are at most $|A|$ interfering actions and this information has been precomputed. Analogously, ENQUEUESTUBBORNES runs in $O(|A|)$.

In lines 14 and 21 the algorithm selects an unsatisfied atom from a partial state. Wehrle and Helmert (2014) discussed several such atom selection strategies—taken from the literature and new—with different time requirements. To stay general, we account for them with $O(t)$, resulting in time $O(t + |A|)$ for lines 14–15.

Each iteration of the while loop takes time $O(p_{\text{max}})$ for testing applicability plus $O(t + |A|)$ accounting for the more expensive else-case of the if statement. As every action is added to queue at most once, the overall runtime of COMPUTESTUBBORNSET is $O(|A|(|p_{\text{max}} + t + |A|))$. The space complexity for stubborn and queue is $O(|A|)$.

New Atom-Centric Algorithm

The original fixed-point computation from Algorithm 1 tracks (in queue) actions that have already been included in the stubborn set but for which it is not yet sure that C2 and C3 are satisfied.

We now reconsider the overapproximation of the interference relation and necessary enabling sets and what this implies for the computation of strong stubborn sets. We begin with potential interference. Using the notion of sibling atoms, we can paraphrase the set of actions that potentially interfere with action $a$: it consists of all actions $a'$ s.t.

- $a'$ achieves a sibling of an atom in $\text{pre}(a)$ ($a'$ potentially disables $a$), or
- $a'$ depends on a sibling of an atom in $\text{eff}(a)$ ($a$ potentially disables $a'$), or
- $a'$ achieves a sibling of an atom in $\text{eff}(a)$ ($a$ and $a'$ potentially conflict).

**Observation 1:** We can characterize these actions by only considering the occurrence of individual atoms in their precondition or effect.

**Observation 2:** The same is true for the actions in the necessary enabling set (cf. ENQUEUESTUBBORNES in Algorithm 1).

**Observation 3:** The order in which the actions are processed is not important for the fixed-point computation.\(^2\)

\(^2\)The order can influence dynamic atom selection strategies, but we are not aware of any work that aims for a specific order.
Based on these three observations, we propose the atom-centric Algorithm 2. The core idea is to achieve synergy effects by deferring the inclusion of actions in the stubborn set, instead tracking the atoms that characterize them.

We use two collections for this purpose: achieved contains atoms for which all achievers should get included in the stubborn set, depended serves the same purpose for depending actions. We ensure that every atom gets added to each of these collections at most once by tracking in sets seen_achieved and seen_depended what has already been included earlier. The procedure ENQUEUE_ACHIEVERS demonstrates this for achieved. The procedure ENQUEUE_DEPENDERS, which is defined analogously, is omitted from the pseudo-code.

ENQUEUE_INTERFERS and ENQUEUENES play exactly the same role as in the action-centric algorithm, with the only difference that they do not directly mark actions stubborn but instead mark the corresponding atoms for further processing. This directly translates to the initialization of the algorithm in lines 27 and 28.

The main loop (lines 30–38) processes the atoms from achieved and depended, initiating the previously deferred handling of actions. HANDLE_ACTION adds the action to the stubborn set and triggers the later inclusion of interfering actions and necessary enabling sets to satisfy C2 and C3.

Complexity Analysis

For the analysis, we use $p_{\text{max}}$ as before and $d_{\text{max}}$ for the maximal size of all variable domains.

An efficient implementation of the algorithm precomputes the achievers and dependers of each atom (used in lines 33 and 37) once for the entire search. As discussed in the analysis of the action-centric algorithm, this requires time $O(|A|p_{\text{max}})$ and space $O(|P||A|)$ for storing the result. In contrast to the action-centric algorithm, we do not need to compute and store the interference relation.

With suitable data structures, ENQUEUE_ACHIEVERS, ENQUEUE_DEPENDERS and ENQUEUENES take constant amortized time. As each outer loop of ENQUEUE_INTERFERS iterates over at most $p_{\text{max}}$ atoms and the inner loop over all but one atom for each of these variables, the procedure runs in $O(p_{\text{max}}d_{\text{max}})$. Using again $t$ for the variable selection time, it is then easy to see that HANDLE_ACTION runs in time $O(p_{\text{max}}d_{\text{max}} + t)$.

For the fixed-point iteration, each atom can pass achieved at most once, causing runtime $O(|A|(p_{\text{max}}d_{\text{max}} + t))$ in lines 33–34. Similarly, each atom can cause runtime $O(|A|(p_{\text{max}}d_{\text{max}} + t))$ in lines 37–38 for at most one pass through depended. Hence, the overall runtime of COMPUTE_STUBBORN is in $O(|A|(p_{\text{max}}d_{\text{max}} + t)|P|)$. The space complexity for achieved, depended, seen_achieved and seen_depended is in $O(|P|)$, for stubborn it is in $O(|A|)$.

If we contrast the two algorithms, for the precomputation the action-centric one requires time $O(|A|^2p_{\text{max}})$ and space $O(|A|^2 + |P||A|)$, whereas the atom-centric one only takes time $O(|A|p_{\text{max}})$ and space $O(|P||A|)$. For the computation during search (for each expanded node), the time complexity of the action-centric algorithm is $O(|A|^2 + |P|(p_{\text{max}} + t))$ compared to $O(|A||P|(p_{\text{max}}d_{\text{max}} + t))$ for the atom-centric one. The corresponding space complexities are $O(|ops|)$ and $O(|A| + |P|)$, respectively.

Thus, the new algorithm clearly dominates the old one in

```
Algorithm 2 Atom-centric algorithm

1: function COMPUTE_STUBBORN_SET(s)
2:   stubborn = empty collection
3:   achieved, depended = empty collections
4:   seen_achieved, seen_depended = empty collections
5:   procedure ENQUEUE_ACHIEVERS(atom)
6:     if atom \notin seen_achieved then
7:       achieved.add(atom)
8:       seen_achieved.add(atom)
9:   procedure ENQUEUE_DEPENDERS(atom)
10:  ENQUEUE_ACHIEVERS(atom)
11:  procedure ENQUEUE_INTERFERS(atom)
12:   for atom \in pre(a) do
13:     for all siblings atom' of atom do
14:       ENQUEUE_ACHIEVERS(atom')
15:   for atom \in eff(a) do
16:     for all siblings atom' of atom do
17:       ENQUEUE_ACHIEVERS(atom')
18:     ENQUEUE_DEPENDERS(atom')
19:   procedure HANDLE_ACTION(a, s)
20:     if a \notin stubborn then
21:       stubborn.add(a)
22:     if a is applicable in s then
23:       ENQUEUE_INTERFERS(a)
24:     else
25:       atom = an unsatisfied atom from pre(a)
26:       ENQUEUE_DEPENDERS(atom)
27:     atom = some unsatisfied goal atom
28:     ENQUEUE_DEPENDERS(atom)
29:   while achieved is not empty or
30:     depended is not empty do
31:     if achieved is not empty then
32:       atom = achieved.pop()
33:       for a \in A with atom \in eff(a) do
34:         HANDLE_ACTION(a, s)
35:     else
36:       atom = depended.pop()
37:       for a \in A with atom \in pre(a) do
38:         HANDLE_ACTION(a, s)
39:   return stubborn
```
the time and space requirements for the precomputation. For the actual computation of stubborn sets, the new one needs asymptotically more space, but only linear in the number of atoms. In the time requirements the algorithm exhibit a very different profile, which lets us expect that the new atom-centric algorithm works better if variable domains are not too large and the task has many more actions than atoms.

Enhancements

In this section, we discuss two possible enhancements of Algorithm 2. The first one is based on the observation that the algorithm frequently enqueue all siblings of an atom, the second one is a new atom selection strategy.

Shortcut Handling of all Siblings

Since we frequently add all siblings of an atom to one of the queues, we can expect a number of duplicates. Avoiding this overhead should be particularly beneficial if variable domains are large.

From the perspective of a variable, we can track some compact (incomplete) information on what has already been enqueued, for example in achieved. For this purpose, we use a datastructure marked achieved that stores for each variable v one of the following values:

\[ d \in D(v) \] representing that all siblings of \((v,d)\) have been enqueued,

\[ \top \] representing that all atoms for this variable have been enqueued, or

\[ \bot \] representing that we do not have any such information.

The information is incomplete in the sense that we do not track the inclusion of individual atoms, so the value can for example be \(\bot\) or some \(d \in D(v)\) although we have already seen all atoms for the variable. To update and exploit the stored information, we do not simply enqueue all siblings of atom in lines 13–14 and 16–17 of Algorithm 2 but proceed instead as follows:

```
1: \((v, d) = \text{atom}\)
2: if \(\text{marked\_achieved}[v] = \bot\) then
3:   for all siblings \(a\) of \(v\) do
4:     \text{ENQUEUE\_ACHIEVERS}(a)
5:  \text{marked\_achieved}[v] = d
6: else if \(\text{marked\_achieved}[v] \notin \{d, \top\}\) then
7:   \text{ENQUEUE\_ACHIEVERS}(\(v, \text{marked\_achieved}[v]\))
8:  \text{marked\_achieved}[v] = \top
```

If we do not have sufficient information, we add all siblings of atom as before, but remember that all values apart from the one from atom have been added (lines 2–5). If we know that all atoms (value \(\top\)) or all siblings of atom (value \(d\)) have already been added, we do not have to do anything. Otherwise, we add the only missing sibling (whose value is stored in marked achieved[v]) and remember that we now have added all atoms (lines 6–8).

We proceed analogously, when enqueuing all siblings of an atom with ENQUEUE\_DEPENDERS in lines 16 and 18.

Atom Selection Strategy

If an action from the stubborn set is inapplicable, we need to choose an unsatisfied atom from the action precondition as seed for the inclusion of a necessary enabling set. Wehrle and Helmert (2014) already discussed and evaluated several strategies for this choice point.

We want to propose a new strategy, called quick skip. It is easy to see that if the chosen atom has already been seen (included in seen achieved), the algorithm does not enqueue anything within ENQUEUE\_NES. This saves computational effort and—maybe even more importantly—it can potentially lead to more pruning because we do not necessarily grow the stubborn set. Therefore, in line 25 of Algorithm 2 the quick skip strategy chooses some atom from pre(a) ∩ seen achieved whenever this set is not empty.

This selection strategy is related to the static small and dynamic small strategies by Wehrle and Helmert, both of which aim to keep the resulting stubborn set small. The static strategy prefers variables that appear in the effects of fewer actions, the dynamic one prefers atoms with a minimal number of achieving actions that have not yet been included in the stubborn set. Our proposed strategy is closer to dynamic small but less specific. If there is an atom for which all achieving actions have already been scheduled for inclusion, the strategies are equal. Otherwise, our strategy can be combined with any other strategy, leaving another choice point.

Weak Stubborn Sets

Besides strong stubborn sets, Valmari (1989) also introduced weak stubborn sets, which provide a stronger pruning power but whose requirements and computation are more involved in general. However, for classical planning, Winterer et al. (2017) proposed a simple variant based on a global notion of interference: Action \(a\) weakly interferes with action \(a'\) if \(a\) and \(a'\) potentially conflict or \(a\) potentially disables \(a'\). The only difference between potential interference and weak interference is whether \(a'\) may potentially disable \(a\) or not.

The definition of weak stubborn sets is then almost identical to Definition 3 with the only difference that condition C3 is replaced with

C3’ For every \(a \in A\) applicable in \(s\), \(A\) contains all actions from \(A\) with which \(a\) weakly interferes.

Weak stubborn sets can equally be used for correct pruning. Since the definition of weak interference is purely syntactical and state- and applicability-independent, it cannot be enhanced with mutex information (Winterer et al. 2017).

Both algorithms for strong stubborn sets can easily be adapted to compute weak stubborn sets: for the action-centric Algorithm 1 we precompute the weak interference relation instead of the potential interference relation; for the atom-centric Algorithm 2 we simply remove lines 12–14.

Empirical Evaluation

We implemented the atom-centric algorithm on top of Fast Downward 19.12, which already contains an implementation of the action-centric algorithm (called “simple stubborn sets” there). For the evaluation, we use the benchmarks of
all optimal tracks of all International Planning Competitions from 1998 to 2018, amounting to 1827 tasks from 65 domains. Each planner run is limited to 1800 seconds and 3.5 GiB. The benchmarks, code and experimental data are published online.\textsuperscript{5}

Before we compare different algorithms and configurations, we evaluate whether the different time complexity of the atom-centric algorithm is promising at all. For this purpose, in Figure 1, we plot the number of atoms against the number of actions in the SAS\textsuperscript{+} planning tasks produced by Fast Downward. We see that the actions frequently outnumber the atoms, often by several orders of magnitude, so the trade-off looks promising indeed.

**Figure 1:** Number of atoms vs. actions for all tasks in the benchmark set. Each mark represents one task. Dashed diagonals show factors 2, 5, 10, and 100.

In the first experiment, we examine how the plain action-centric and atom-centric algorithms compare when they compute the same information. Towards this end, we do not use the mutex-based strengthening of interference and use for both algorithms the same strategy for choosing unsatisfied atoms, namely always picking the first unsatisfied atom according to the fixed variable ordering of Fast Downward.

**Blind Search** With blind search, node expansions are extremely fast, so the relative overhead of computing stubborn sets for each expansion is high. For this reason, we can only expect to benefit from partial order reduction if it leads to significant pruning. On our benchmark set, blind search without pruning solves 710 instances, whereas coverage increases by 25 instances with the atom-centric algorithm (cf. left part of Table 1). Interestingly, computing the same information with the action-centric algorithm leads to a significant coverage decline to 679 instances. As expected, in both cases the total number of expansions is the same and decreases by 24.7% compared to no pruning.

A closer look at the results per domain reveals that even with our more efficient algorithm, using strong stubborn set pruning is not always beneficial, loosing between one and

<table>
<thead>
<tr>
<th></th>
<th>base</th>
<th>action</th>
<th>atom</th>
</tr>
</thead>
<tbody>
<tr>
<td>airport (50)</td>
<td>22</td>
<td>21</td>
<td>28</td>
</tr>
<tr>
<td>data-network (20)</td>
<td>7</td>
<td>6</td>
<td>12</td>
</tr>
<tr>
<td>freecell (80)</td>
<td>20</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>gdl (20)</td>
<td>15</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>hiking (20)</td>
<td>11</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>miconic (150)</td>
<td>55</td>
<td>50</td>
<td>141</td>
</tr>
<tr>
<td>mprime (35)</td>
<td>19</td>
<td>18</td>
<td>19</td>
</tr>
<tr>
<td>nomystery (20)</td>
<td>8</td>
<td>7</td>
<td>15</td>
</tr>
<tr>
<td>openstacks-08 (30)</td>
<td>22</td>
<td>20</td>
<td>22</td>
</tr>
<tr>
<td>openstacks-11 (20)</td>
<td>17</td>
<td>15</td>
<td>17</td>
</tr>
<tr>
<td>org.-synth-split (20)</td>
<td>10</td>
<td>9</td>
<td>15</td>
</tr>
<tr>
<td>parcprinter-08 (30)</td>
<td>10</td>
<td>30</td>
<td>19</td>
</tr>
<tr>
<td>parcprinter-11 (20)</td>
<td>6</td>
<td>20</td>
<td>13</td>
</tr>
<tr>
<td>parking-11 (20)</td>
<td>0</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>parking-14 (20)</td>
<td>0</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>pegsol-08 (30)</td>
<td>27</td>
<td>27</td>
<td>29</td>
</tr>
<tr>
<td>pegsol-11 (20)</td>
<td>17</td>
<td>17</td>
<td>19</td>
</tr>
<tr>
<td>petri-net-align. (20)</td>
<td>4</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>pipesworld-not. (50)</td>
<td>17</td>
<td>14</td>
<td>16</td>
</tr>
<tr>
<td>pipesworld-l. (50)</td>
<td>12</td>
<td>8</td>
<td>12</td>
</tr>
<tr>
<td>rovers (40)</td>
<td>6</td>
<td>7</td>
<td>10</td>
</tr>
<tr>
<td>satellite (36)</td>
<td>6</td>
<td>6</td>
<td>8</td>
</tr>
<tr>
<td>scanalyzer-08 (30)</td>
<td>12</td>
<td>8</td>
<td>16</td>
</tr>
<tr>
<td>scanalyzer-11 (20)</td>
<td>9</td>
<td>5</td>
<td>13</td>
</tr>
<tr>
<td>snake (20)</td>
<td>11</td>
<td>4</td>
<td>9</td>
</tr>
<tr>
<td>spider (20)</td>
<td>11</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>termes (20)</td>
<td>9</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>tetriss (17)</td>
<td>9</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>tidybot-11 (20)</td>
<td>13</td>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>tidybot-14 (20)</td>
<td>6</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>transport-11 (20)</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>transport-14 (20)</td>
<td>7</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>trucks (30)</td>
<td>6</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>woodworking-08 (30)</td>
<td>8</td>
<td>16</td>
<td>18</td>
</tr>
<tr>
<td>woodworking-11 (20)</td>
<td>3</td>
<td>10</td>
<td>12</td>
</tr>
<tr>
<td>zenotravel (20)</td>
<td>8</td>
<td>7</td>
<td>13</td>
</tr>
<tr>
<td>sum (1138)</td>
<td>436</td>
<td>405</td>
<td>461</td>
</tr>
<tr>
<td>other domains (689)</td>
<td>274</td>
<td>274</td>
<td>351</td>
</tr>
<tr>
<td>total (1827)</td>
<td>710</td>
<td>670</td>
<td>735</td>
</tr>
</tbody>
</table>

Table 1: Coverage of A\textsuperscript{*} with the blind (left), LM-cut (middle), and SCP (right) heuristics, comparing vanilla search (base) with the addition of plain action-centric (action) and atom-centric (atom) pruning. We highlight maximum coverage separately for each heuristic.

three tasks in 13 domains and even five instances in the freecell domain. The positive net benefit stems from the two parcprinter domains with a coverage increase of 20 and 14 and the two woodworking domains with an increase of 8 and 7 tasks. So it seems that these domains are especially suitable for partial order reduction, whereas in other domains the additional overhead does not pay off. Indeed, in woodworking the goal is to process a set of work pieces, each basically corresponding to an independent sub-task. In par-
Atom-centric

Figure 2: Comparison of pruning time (within an A* search with the SCP heuristic) of the action-centric and the atom-centric algorithm for tasks solved by both approaches.

eprinter, the aim is to print a set of pages using several components of an involved printing system. The actions for the different pages can often be arbitrarily interleaved, which can be avoided with partial order reduction.

**LM-Cut** Wehrle and Helmert (2014) used A* search with the LM-Cut heuristic (Helmert and Domshlak 2009) for their evaluation. In this setting, stubborn set pruning is useful overall (cf. middle part of Table 1). 957 tasks are solved without pruning. 982 with the action-centric algorithm and 991 with the atom-centric computation. In a per-domain comparison to the baseline without pruning we never lose more than one task, but coverage increases in seven domains. However, this is again most prominent in the parcprinter (+11 and +7) and the woodworking (+9 and +7) domains. The advantage in comparison to the action-centric algorithm stems from five domains.

**Saturated Cost Partitioning** We also conducted an analogous experiment for A* with a saturated cost partitioning (SCP) heuristic (Seipp, Keller, and Helmert 2020) over pattern databases (Edelkamp 2001) and Cartesian abstractions (Seipp and Helmert 2018). The pattern databases were generated systematically up to pattern size 2 and via hill climbing (Haslum et al. 2007). This SCP heuristic yields state-of-the-art performance for optimal classical planning, because it is both accurate and very fast to evaluate (much faster than LM-cut, for example).

Similar to the results for blind search, using the action-centric algorithm decreases coverage (cf. right part of Table 1). In contrast to the results for LM-cut, using the atom-centric algorithm increases the total coverage by only two tasks, with a decrease by more than one task in four domains. However, if we compare the action-centric against the atom-centric algorithm, we see a clear advantage of the new one in 18 domains, while the opposite is never the case.

**Overall** As both stubborn set algorithms compute the same information, the difference in performance must be attributed to the different computational overhead. Figure 2 compares the total time spent for computing stubborn sets for each task. This data stems from the experiment with the SCP heuristic; the plots for the other configurations (blind and LM-Cut) look similar. We see that with the atom-centric algorithm we can obtain the same pruning power much faster, often by more than an order of magnitude.

**Enhancements to the Action-Centric Algorithm**

In the second experiment, we evaluate the two enhancements to the atom-centric algorithm. The shortcut handling of siblings (sib) does not change the behavior of the algorithm and should hence only have an impact on runtime. The new atom selection strategy quick skip (qs) should have a tendency to produce smaller stubborn sets, so we would also hope for more pruning. We compare the qs strategy to the default strategy (FD) and the two strategies dynamic small (ds) and static small (ss) by Wehrle and Helmert (2014), all of them with and without the sib enhancement.

Table 2 shows that with the SCP heuristic, the dynamic small and static small strategies solve many fewer tasks than the Fast Downward and quick skip strategies, and that quick skip has a slight edge over Fast Downward. In contrast, the shortcut handling of siblings only has a very mild impact on
Figure 3: Comparison of pruning ratio of the atom-centric algorithm with strategies FD and qs, using $A^*$ with SCP.

Figure 4: Comparison of pruning ratio (left) and pruning time (right) of EC vs. our best configuration with SCP.

coverage, sometimes negative, sometimes positive. It works best in combination with quick skip, where the small impact is exclusively positive with the blind and SCP heuristics, and coverage decreases by 1 in terms with LM-cut (not shown in Table 2). The combination of quick skip with shortcut handling of siblings achieves the highest total coverage with all three heuristics, and dominates the other strategies also in a per-domain comparison, except for four domains when using blind search and two domains when using $A^*$ with LM-cut and SCP.

To analyze the pruning ratio of the different methods, we run the search with pruning and accumulate the number of successors of all expanded states as $n_{\text{all}}$ and sum up the size of the corresponding stubborn sets as $n_{\text{gen}}$. The pruning ratio is then defined as $1 - n_{\text{gen}}/n_{\text{all}}$, giving values between 0 and 1, where 0 represents no pruning and 1 would mean that all successors were pruned. Figure 3 plots the pruning ratio of the action-centric algorithm with the $FD$ strategy (the best previous selection strategy according to Table 2) to the new quick skip strategy (both with SCP), highlighting domains with larger differences. We observe consistent positive impact on the pruning power, which is particularly pronounced in the logistics and woodworking domains.

Comparison to the State of the Art

In the third experiment, we compare our best configuration (atom-centric algorithm with qs and sib) to the configuration reported as state of the art for computing strong stubborn sets by Wehrle and Helmert (2014), namely “SSS-EC full/mutex” (EC), which computes stubborn sets in a way that dominates the expansion core method (Chen and Yao 2009) and enhances action interference with mutexes. With our best configuration, total coverage increases significantly for all three heuristics, in particular for the two faster-to-compute ones (+53 with blind search, +9 with LM-Cut, +40 with SCP). A deeper analysis reveals that our configuration is on-par with respect to pruning power, but requires a much lower computation time. To illustrate this, we compare the pruning ratio and pruning time for $A^*$ with SCP in Figure 4. For the other heuristics the results look qualitatively similar.

Since not all domains are equally suited for partial-order reduction, many recent IPC planners (e.g., Alkhazraji et al. 2014) disable pruning if after 1000 expanded states the pruning ratio is at most 20%. We evaluated the impact of this approach on our best configuration (atom-centric algorithm with both enhancements) and on the previous state of the art (EC). The method has only a mild impact on our algorithm: overall, coverage increases, but in some domains fewer tasks are solved. This is very different for the slower EC method, which greatly benefits from this approach, bringing it almost on par with our configuration.

Weak Stubborn Sets

We repeated the previous experiments with the action-centric algorithm and its enhancements using weak instead of strong stubborn sets. The results are somewhat inconclusive. With the blind heuristic, there is a visible positive impact on total coverage (from +2 to +7), but with LM-cut and SCP, the impact is minimal and coverage increases or decreases by up to two tasks.

Discussion and Future Work

We proposed an atom-centric algorithm that computes the same stubborn sets as an earlier action-centric algorithm with a different profile wrt. time and space complexity. The new algorithm requires less space and we saw that it is much faster on common planning benchmarks. One limitation of our algorithm is that it is no longer possible to enhance the interference relation with mutex information. However, already without any enhancements, our algorithm outperforms the best previous algorithm (EC), which makes use of such mutex information (1136 vs. 1101 solved tasks with $A^*$ + SCP). The new atom selection strategy quick skip does not only further speed up the computation but also leads to smaller stubborn sets and thus to more pruning.

In classical planning, stubborn sets have not only been used for state-space search but have also been adapted for fork-decoupled search (Gnad, Hoffmann, and Wehrle 2019). Beyond the classical planning fragment, they have been applied to fully observable non-deterministic planning (Winterer et al. 2017), planning with resources (Wilhelm, Steinmetz, and Hoffmann 2018) as well as for goal recognition design (Keren, Gal, and Karpas 2018). In future work, it would be interesting to examine whether the general idea of an atom-centric perspective can also be beneficially applied in these settings.
References


