Annals of Mathematics and Artificial Intelligence (2025) 93:699–726
https://doi.org/10.1007/s10472-024-09960-8

RESEARCH

On domain generators for the evaluation of action reversibility in STRIPS

Tobias Schwartz[1,2] · **Jan H. Boockmann**[1] · **Leon Martin**[1]

Accepted: 12 November 2024 / Published online: 29 November 2024

Abstract
Robustness is a crucial requirement for the deployment of AI systems in real-world scenarios. In the context of AI planning, the concept of action reversibility, i.e., the ability to undo the effects of an action using a reverse plan, is a promising direction for achieving robust plans. Plans composed exclusively of reversible actions exhibit resilience against goal changes during the execution of the plan. However, the evaluation of action reversibility systems in STRIPS planning presents a challenge, given that standard planning benchmarks are often not suitable. Early experiments using a naive implementation of an action reversibility algorithm show that the available domain generation approach is susceptible to bias. Building on this existing domain generator, we introduce two slight variations that exhibit entirely different search space characteristics. We assess these domain generators using the naive action reversibility implementation and existing ASP implementations, and demonstrate that different generators indeed favor different implementations. As a follow-up to this line of research, we present a generalized domain generator facilitating the creation of domains with diverse search space characteristics. To finally reduce the utilization of contrived generation patterns, we propose another domain generator based on the Barabási-Albert model yielding less rigid domains. Our experiments demonstrate that these new domain generators can produce a variety of domains with diverse search space characteristics, enabling a less biased evaluation of action reversibility systems.

Keywords STRIPS planning · Action reversibility · Benchmark · PDDL domain generation

Mathematics Subject Classification (2010) 68T20

✉ Tobias Schwartz
tobias.schwartz@uni-bamberg.de

Jan H. Boockmann
jan.boockmann@uni-bamberg.de

Leon Martin
leon.martin.mailbox@gmail.com

[1] Faculty Information Systems and Applied Computer Sciences, University of Bamberg, An der Weberei 5, Bamberg 96045, Germany

[2] Institute for Software Engineering and Programming Languages, University of Lübeck, Ratzeburger Allee 160, Lübeck 23562, Germany

1 Introduction

A classical planning problem, as outlined by [1, 2], is typically solved by identifying a sequence of actions, also known as a plan, that leads from a predefined initial state to a desired goal state. Within this context, the concept of *action reversibility* has been the subject of extensive recent study [3–9]. Action reversibility is utilized to evaluate whether the effects of applying an action can be undone through a *reverse plan*.

In static execution environments with constant goal states, action reversibility may be less critical. However, it becomes crucial in dynamic environments where goal states can change during plan execution, such as autonomous spacecraft control [10] and cloud management [11]. In these scenarios, executing a pre-computed plan with non-reversible actions may prevent reaching an updated goal state, potentially leading to an irrecoverable dead end. Conversely, a plan composed of reversible actions, though potentially longer, offers resilience against goal changes. Such a plan can be rolled back to the initial state if partially executed, allowing the system to reach the updated goal (if it was initially reachable) by computing a new resilient plan from the starting point. In closed-world scenarios, online planning strategies face challenges akin to goal changes in dynamic environments. These strategies are susceptible to dead ends, as observed by [12]. Therefore, investigating action reversibility algorithms, their implementations, and their integration into existing AI planning tools represents a promising research direction.

The works in [5, 7] introduced a framework for action reversibility, which encapsulates different notions of action reversibility that have been studied in previous research [3, 4]. In addition to a complexity analysis, which inherits the PSPACE completeness of basic STRIPS planning [13], the work in [5] proposes a non-deterministic algorithm for computing the generalized notion of *uniform φ-reversibility* introduced in [5]. This concept, similar to generalized planning [14], aims to find a single reverse plan applicable across multiple instances within a domain. However, an implementation and evaluation of the conceptual algorithm are left for future work. In the follow-up paper [6], the authors provide an implementation based on Answer Set Programming (ASP). They evaluate this implementation using STRIPS domains derived from a domain generator and propose a comparison with a procedural implementation as a potential area for future work.

This paper is an extended version of our previous work [15] presented at the 12th International Symposium on Foundations of Information and Knowledge Systems (FoIKS) in 2022 for the FoIKS special issue of the Annals of Mathematics and Artificial Intelligence (AMAI) Journal[1]. In this previous work, we followed the call to action of [6] and implemented the conceptual algorithm from [5]. Subsequently, we evaluated the performance of our prototypical Breadth-First Search (BFS) and Depth-First Search (DFS) strategy with the ASP implementation from [6]. BFS and DFS represent archetypal strategies for graph traversal [16] that have been studied in many areas [17]. Hence, they provide an appropriate reference point for comparison and further investigation of domain characteristics in the reversibility context. We showed that the domain generator considered so far does not suffice for evaluating other action reversibility implementations in STRIPS without bias. Hence, we proposed two additional domain generators that result in domains with multiple reverse plans and potential dead ends. Our results indicate that a thorough analysis of the bias contained in generated domains is crucial to accurately evaluate implementations of action reversibility systems. This corresponds to a prevalent problem in research, where the examples necessary to evaluate a novel technique are often not yet available. Typically, researchers then resort to

[1] Parts of Sections 2, 3, 4, and 7 in this paper originate from our FoIKS 2022 paper [15].

creating examples of varying style by hand or crafting a series of structurally similar examples using a generator. In the end, establishing an admissible benchmark based on handcrafted or generated examples is vital to evaluate novel techniques. However, a thorough analysis is required to ensure that a benchmark assembled this way does not contain a systematic bias that favors a particular technique.

In addition to a fundamental revision of the content of our FoiKS 2022 paper [15] in light of recent research on action reversibility in STRIPS [8, 9] and the re-execution of our original experiments, this extended version addresses several additional aspects: When it comes to action reversibility, basic STRIPS offers limited capabilities for expressing more challenging problems. To address this, we discuss and illustrate a domain generator based on a generalized approach for generating STRIPS domains with specific characteristics to alleviate this limitation. To overcome the shortcomings of the previously analyzed elementary domain generators, we introduce a new domain generator based on the Barabási-Albert (BA) model [18], an algorithm for creating scale-free networks. The advantages of this new domain generator are twofold: Firstly, domains generated using this new method can be considered less rigid than those derived from the previous generators that utilize contrived procedures for domain generation. Secondly, the parameters of the BA model facilitate the generation of domains with diverse characteristics. Our experimental results indicate that the BA domain generator provides a promising benchmark basis for evaluating action reversibility in STRIPS, as it enables the creation of scenarios that favor different strategies.

This paper falls under the category of planning in the context of the AMAI Journal due to its focus on STRIPS, action reversibility, and the evaluation thereof. The remainder of this paper is structured as follows: Section 2 covers foundational topics related to action reversibility in STRIPS planning and discusses the challenges arising from the evaluation of action reversibility. Then, Section 3 introduces the first set of elementary domain generators conceptually, before Section 4 discusses exploratory experiments conducted on domains produced by these generators. Fueled by the results, Section 5 proposes the two new advanced domain generators. Following the discussion of our experiments concerning the advanced domain generators in Section 6, Section 7 draws a conclusion with due regard to leads on future work.

2 Background and related work

The first part of this section covers foundational topics including STRIPS planning, the different notions of action reversibility with a particular focus on uniform φ-reversibility. Afterwards, we discuss the challenges of evaluating action reversibility in STRIPS, motivating the research presented in this paper.

2.1 STRIPS planning

Building on the concepts presented in [1] and adhering to the naming conventions of [5], we define a STRIPS planning problem as a tuple $\Pi = (\mathcal{F}, \mathcal{A}, s_0, \mathcal{G})$. This tuple comprises facts $\mathcal{F}$ (which are atomic statements about the world), actions $\mathcal{A}$, an initial state s_0 (a subset of $\mathcal{F}$), and a goal specification $\mathcal{G}$ (also a subset of $\mathcal{F}$). Each action is represented as a tuple $a = \langle pre(a), add(a), del(a)\rangle$, where $pre(a)$, $add(a)$, and $del(a)$ are subsets of $\mathcal{F}$ representing the preconditions, the positive effects, and the negative effects of action a, respectively. Note

that in the remainder of this paper, we will usually display actions in the Planning Domain Definition Language (PDDL) [19], where the effects of an action are displayed in a combined format including both positive and negative effects, the latter indicated by a prepended "not". We make the assumption that actions are well-formed, meaning that the sets of positive and negative effects are mutually exclusive ($add(a) \cap del(a) = \emptyset$), and the preconditions and positive effects do not overlap ($pre(a) \cap add(a) = \emptyset$). An action a is applicable in a state s if and only if its preconditions are met in that state ($pre(a) \subseteq s$). When an applicable action a is applied in state s, it results in a new state $a[s] = (s \setminus del(a)) \cup add(a)$. An action sequence $\pi = \langle a_1, \ldots, a_n \rangle$ is applicable in the initial state s_0 if there exists a sequence of states $\langle s_1, \ldots, s_n \rangle$ such that for each $1 \leq i \leq n$, action a_i is applicable in state s_{i-1} and leads to state s_i ($a_i[s_{i-1}] = s_i$). Applying an applicable action sequence π in the initial state s_0 results in the state $\pi[s_0] = s_n$. The length of an action sequence π is denoted by $|\pi|$.

2.2 Notions of action reversibility

Intuitively, we can think of an action a as being reversible if there exists a reverse plan to undo all the changes introduced by a. Various notions of reversibility have been studied in the academic literature. For instance, the work by [4] introduces the concept of *undoability*. According to this concept, an action a is considered undoable if, for any reachable state s in a STRIPS planning problem Π, there exists a reverse plan that undoes the effects of applying a. The work by [4] also proposes a broader concept of *universal undoability*, which lifts the reachability restriction such that an action a is considered universally undoable if there exists a reverse plan for a in any possible state $s \in 2^{\mathcal{F}}$ that undoes the action's effect. The work by [3] introduces a more restricted definition of reversibility: an action a is deemed reversible if there exists a reverse plan that is independent of the state s where a was applied.

The works by [5, 7] present a comprehensive framework that differentiates various notions of reversibility, all based on the notion of S-reversibility. S-reversibility defines reversibility in the most general sense and is not dependent on a specific STRIPS planning problem, but only on a set of actions and states derived from a set of facts.

Definition 1 (S-reversibility) Given a set of facts $\mathcal{F}$, a set of actions $\mathcal{A}$, and a set of states $S \subseteq 2^{\mathcal{F}}$. An action $a \in \mathcal{A}$ is called S-reversible iff for every state $s \in S$ wherein a is applicable, there exists a reverse plan $\pi = \langle a_1, \ldots, a_n \rangle \in \mathcal{A}^n$ that is applicable in $a[s]$ such that $\pi[a[s]] = s$.

Note that in S-reversibility, the set of states S must be explicitly specified. However, in some cases, it may be more convenient to define S using a propositional formula over the facts in $\mathcal{F}$. In other cases, either all possible states or all reachable states in a given STRIPS planning problem might be of interest. The following definitions capture these different notions of reversibility.

Definition 2 (φ-reversibility) Given a set of facts $\mathcal{F}$, a set of actions $\mathcal{A}$, and a propositional formula φ over $\mathcal{F}$. An action $a \in \mathcal{A}$ is called φ-reversible iff a is S-reversible in the set S of models of φ.

Definition 3 (universal reversibility) Given a set of facts $\mathcal{F}$ and a set of actions $\mathcal{A}$. An action $a \in \mathcal{A}$ is called universally reversible iff a is $2^{\mathcal{F}}$-reversible, i.e., reversible in any possible state.

Definition 4 (reversible in Π) Given a STRIPS planning problem $\Pi = (\mathcal{F}, \mathcal{A}, s_0, \mathcal{G})$, and let $\mathcal{R}_\Pi$ be the set of states reachable from the initial state s_0. An action $a \in \mathcal{A}$ is called reversible in Π iff a is $\mathcal{R}_\Pi$-reversible.

A notion of reversibility is called *uniform* if the reverse plan π used is always the same, regardless of the state. This restriction is particularly interesting for real-world scenarios, because it allows for the precomputation and storage of reverse plans. Clearly, any uniform version of reversibility implies the non-uniform version, as there exists at least one reverse plan π that can reverse the effects of an action a in all relevant states s. For reference, let us formally define the notion of uniform φ-reversibility. Other forms of uniform reversibility are defined similarly.

Definition 5 (uniform φ-reversibility) Given a set of facts $\mathcal{F}$, a set of actions $\mathcal{A}$, and a propositional formula φ over $\mathcal{F}$. An action $a \in \mathcal{A}$ is called uniformly φ-reversible iff a is φ-reversible using always the same reverse plan π.

As also pointed out in [5], Definition 4 aligns with the concept of undoability [4]. Meanwhile, a uniform version of Definition 3 matches the notion of reversibility in [3].

Multiple implementations for solving action reversibility have been proposed based on ASP [6, 8, 9]. While the first programs only solved uniform universal reversibility, recent work also tackled the computationally more difficult problem of uniform φ-reversibility. Note, that even the former problem is already NP-complete [5], though.

Algorithm 1 Uniform φ-reversibility of an action a (adopted from [5]).

Input : A set of actions $\mathcal{A}$, an action $a \in A$
Output: A formula φ, a reverse plan π

$F^+ = (pre(a) \setminus del(a)) \cup add(a)$
$F^- = del(a)$
$F^0 = \emptyset$
$\pi = \langle\rangle$
while $pre(a) \not\subseteq F^+ \vee F^0 \cap F^- = \emptyset$ **do**
    non-deterministically choose $a \in \mathcal{A}$ such that $pre(a) \cap F^- = \emptyset$
    if a does not exist **then return** $\bot, \langle\rangle$
    $F^0 = F^0 \cup (pre(a) \setminus F^+)$
    $F^+ = (F^+ \setminus del(a)) \cup add(a)$
    $F^- = (F^- \setminus add(a)) \cup del(a)$
    $\pi = \pi \cdot a$
$\varphi = \bigwedge_{l \in F^+ \cup F^0} l \wedge \bigwedge_{l \in F^-} \neg l$
return φ, π

The reversibility notion studied in this paper is uniform φ-reversibility. For reference, Algorithm 1 depicts the non-deterministic algorithm proposed by [5] for computing a reverse plan π and a corresponding formula φ for an action a. The algorithm works by maintaining a placeholder state to which any chosen action is applied. Sets F^+ and F^- denote facts that are true and false within the placeholder state, respectively, while set F^0 indicates the necessary preconditions. The algorithm terminates once all preconditions of a have been restored in the placeholder state. It then outputs the reverse plan π and a formula φ that indicates the states where the reverse plan can be applied. See [5] for more details on the algorithm.

2.3 Benchmarks for evaluating action reversibility

Unlike classical planning problems, the complexity of solving an action reversibility problem is influenced by the planning domain rather than the specific planning instances, i.e., initial and goal state. Therefore, understanding the domain's characteristics is crucial for obtaining an accurate evaluation.

This section explores why current planning benchmarks, such as the International Planning Competition (IPC) [20] planning domains, and the single path domain generator proposed by [6], fall short in providing a representative foundation for accurately evaluating action reversibility systems.

Listing 1 Definitions in PDDL format of the inverse actions `pickup` and `putdown` from the blocks-world domain, adapted from [21].

```
(:action pickup
  :parameters (?ob)
  :precondition (and (clear ?ob) (on-table ?ob) (arm-empty))
  :effect (and (holding ?ob) (not (clear ?ob))
               (not (on-table ?ob)) (not (arm-empty))
  )
)

(:action putdown
  :parameters  (?ob)
  :precondition (holding ?ob)
  :effect (and (clear ?ob) (arm-empty)
               (on-table ?ob) (not (holding ?ob))
  )
)
```

The IPC offers a diverse range of planning domains and corresponding planning instances, encoded in the Planning Domain Definition Language (PDDL) [19]. This allows for an empirical comparison of different planning systems with respect to their performance on a wide range of planning problems with varying characteristics. Recall that Algorithm 1 computes a uniform notion of reversibility, relying solely on the set of actions $\mathcal{A}$. Unlike classical planning, where complexity emerges from the PDDL instances, the assessment of action reversibility systems depends exclusively on the PDDL domains. The work in [4] analyzed the STRIPS domains from IPC'1998 to IPC'2014 through a set of experiments, finding that the majority of actions in these domains are not reversible. Furthermore, it reveals that the reverse plans for the remaining reversible actions are typically quite short, often of length 1. This means that the effects of an action a can be entirely undone in a single step by applying an *inverse action* $\overline{a}$ [22].

For instance, consider the two actions `pickup` and `putdown` from the well-known blocks-world domain [23] depicted in Listing 1. It is apparent that action `pickup` is the inverse action of `putdown`, and vice versa, as they undo each other's effects. Although inverse actions make the computation of the corresponding reverse plans straightforward, solving a classical planning instance whose domain contains such inverse actions does not necessarily become easier.

Note that the existence of small reverse plans in STRIPS planning domains is not necessarily surprising since the complexity of a planning problem primarily arises from the planning instances, including the concrete initial state and goal state. This is also the case for domains studied in the field of real-time planning [12] where domains typically contain only a few actions, and the complexity of the modeled system is encoded in the states. In

contrast, the complexity of the action reversibility problem primarily arises from the action to be reversed in the context of other available actions in the respective domain. Accordingly, the IPC domains in particular are not well-suited for a sophisticated evaluation of action reversibility systems, due to the domains containing very small reverse plans.

To generate more interesting problems, the planning community leverages generators producing PDDL instances. Prominent examples are the generators provided in the FF Domain Collection [24] or the set of generators in [21], some of which were used to generate benchmarks for the IPC. However, these problem generators are domain-specific and only generate start and goal definitions for a particular domain, often using hand-crafted heuristics. As this is a time-consuming task, previous work in [25] has studied how a preceding domain analysis can be used to support the generation of PDDL instances. While relying on a similar conceptual approach, namely generators that produce task descriptions, this paper is concerned with the generation of PDDL domains (not PDDL instances) to evaluate action reversibility systems (not planners) in a systematic way.

3 Elementary domain generators

This section discusses the single path domain generator proposed in [6] and introduces two slight syntactic variations: a multiple paths and a dead ends domain generator. Both variations have been proposed and explored by us in [15] and provide a first attempt at generating PDDL domains with structurally interesting search spaces tailored to evaluating action reversibility in STRIPS.

The three domain generators operate on a common principle: They accept an integer i as input and produce PDDL domains with varying predicates and actions. Each domain requires computing a reverse plan for the `del-all` action. The search space characteristics vary significantly among the generators. However, they share a common trait: as the input i grows, the complexity of computing a reverse plan also increases. In essence, the single path generator yields a linear reverse plan, while the multiple paths and dead ends generators create domains with multiple reverse plans and potential dead ends, respectively.

Listing 2 Single path domain generator (adopted from [6]).

```
(define (domain singlePath-<i>)
(:requirements :strips)
(:predicates (f0) ... (f<i>))

(:action del-all
:precondition (and  (f0) ... (f<i>))
:effect (and  (not (f0)) ... (not (f<i>))))

(:action add-f0
:effect (f0))

...

(:action add-f<i>
:precondition (f<i-1>)
:effect (f<i>)))
```

3.1 Single path domains

To address the shortcomings of the IPC domains with respect to evaluating action reversibility, the authors of [6] created a custom-built PDDL domain generator for evaluating their action reversibility implementation. The pattern of PDDL domains generated by this single path domain generator is shown in Listing 2. From a high-level point of view, the generator takes an integer i as input and produces a domain named `singlePath-<i>`, with `del-all` designated as the action to be reversed. Higher i values result in linearly longer reverse plans, achieved by introducing additional predicates and actions required for reversal. The action `del-all` removes facts `f0` through `f<i>` from the current state. To reverse this, each fact must be reinstated. Each fact has a corresponding `add-f<i>` action, where `add-f1` requires `f0` to be present, `add-f2` requires `f1`, and so on. `add-f0` has no preconditions. Therefore, to reverse `del-all`, each `add-f<i>` action must be executed once in ascending order.

The reverse plan length for action `del-all` increases linearly with respect to the input i, specifically $|\pi| = i$. In contrast, the number of possible states in the state space increases exponentially with respect to i, specifically $|S| = 2^{i+1}$, because the number of predicates in the generated PDDL domain is $i + 1$ (refer to line 3 in Listing 2). This usually varies from PDDL generators employed in classical planning, which produce different planning instances based on a fixed set of predicates and actions.

The search space generated by this domain consist of a single reverse plan that includes all actions except for `del-all` in ascending order. Note that the existence of a single reverse plan is not a limitation when used for evaluating the performance of ASP implementations [6, 8, 9] as they operate on a guess-and-check basis. The guess-and-check approach initially enumerates all possible action sequences up to a given length, and only retains applicable paths. Therefore, the performance of the approach is expected to depend primarily on the length of the reverse plan, rather than the characteristics of the search space.

Consequently, the domains obtained from this single path generator are only appropriate for evaluating approaches that adhere to a similar guess-and-check pattern and do not take into account the structure of the search space. This, however, does not apply to search-based algorithms, which consider the applicability of actions during the traversal of the search space when searching for a reverse plan. For these algorithms, both the length of the reverse plan and the structure of the search space are anticipated to influence their performance. Accordingly, we deem the single path domain generator insufficient for evaluating arbitrary action reversibility systems that may consider the structure of the search space.

Listing 3 Our multiple paths domain generator.

```
 1  (define (domain multiplePaths-<i>)
 2  (:requirements :strips)
 3  (:predicates (f0) ... (f<i>))
 4
 5  (:action del-all
 6  :precondition (and  (f0) ... (f<i>))
 7  :effect (and  (not (f0)) ... (not (f<i>))))
 8
 9  (:action add-f0
10  :effect (f0))
11
12  ...
13
14  (:action add-f<i>
```

```
15  :precondition (f<i-1>)
16  :effect (and (f<i>) (not (f0)) ... (not (f<i-1>)))))
```

3.2 Multiple paths domains

Building upon the single path domain generator, we propose an enhancement allowing the generation of domains that yield multiple paths, each representing a valid reverse plan. We have named this new domain generator the *multiple paths domain generator*. Thereby produced domains yield multiple paths by ensuring that permutations of the reverse plan are also viable. This is achieved by modifying the effects of the `add-f<*>` actions, as illustrated in Listing 3. Instead of simply adding the predicate `f<j>`, these actions now remove all preceding predicates from `f0` to `f<j-1>`. This necessitates the reinstatement of the just removed predicates to obtain the next predicate `f<j+1>`. This minor syntactical modification effectively transforms the characteristics of the search space, causing the length of the reverse plan to increase from linear to quadratic in relation to the input i. More specifically, the length of the reverse plan is now represented by the equation $|\pi| = \sum_{k=1}^{i} k$. However, it is crucial to point out that despite this change, the number of actions remains linear in relation to the input i.

Clearly, the computation of a reverse plan that has a quadratic length in relation to i requires considerably more time compared to a plan of linear length. However, considering that *all* feasible paths constitute valid reverse plans, the performance of search-based methods employing a DFS strategy is expected to remain relatively consistent with the single path domain, provided that both reverse plans have the same length. Conversely, search-based methods that utilize a BFS strategy are likely to experience performance degradation due to the increased size of the search space. The guess-and-check strategy, as employed by the ASP implementations referenced in [6, 7, 9], is expected to be affected more significantly by the increased length of the reverse plan, rather than the changes in the search space structure.

Listing 4 Our dead ends domain generator.

```
 1  (define (domain deadEnds-<i>)
 2  (:requirements :strips)
 3  (:predicates (f0) ... (f<i>) (token))
 4
 5  (:action del-all
 6  :precondition (and  (f0) ... (f<i>) (token))
 7  :effect (and  (not (f0)) ... (not (f<i>))))
 8
 9  (:action consume
10  :precondition (token)
11  :effect (not (token)))
12
13  (:action add-f0
14  :effect (f0))
15
16  ...
17
18  (:action add-f<i>
19  :precondition (f<i-1>)
20  :effect (and (f<i>) (not (f0)) ... (not (f<i-1>)))))
```

3.3 Dead ends domains

The search space of the domains generated by the two so far discussed generators, single paths and multiple paths, share a characteristic: all paths in the search space are viable and eventually lead to a valid reverse plan. Conceptually, this property heavily favors search-based approaches that follow a DFS strategy. In real-world planning scenarios, performing an arbitrary action does not necessarily guarantee progress towards the goal. In retrospect, applying an action in classical planning might have been superfluous for reaching the desired goal state, or worse, it could have led to a dead end, a state in which the goal can no longer be reached. From a search perspective, this necessitates backtracking to a previous configuration and the pursuit of a different path. As a result, a DFS strategy can be significantly slower in this context, as it may follow lengthy paths that ultimately lead to dead ends. On the other hand, dead-end paths that exceed the length of the reverse plan should have minimal impact on a BFS, despite the increased size of the search space. However, additional information, such as the number of viable paths, is required to determine whether a BFS strategy is indeed more efficient than a DFS strategy.

PDDL domains with dead ends can be constructed in a manner akin to the multiple paths domain generator. When computing reverse plans for an action a, the starting state is the state $a[s]$, that is, the state after a has been applied. This state is defined by $(pre(a) \setminus del(a)) \cup add(a)$, which mirrors the initial value of F^+ in Algorithm 1. It is important to note that for the multiple path domain generator, this starting state is empty because $pre(del\text{-}all) = del(del\text{-}all) \wedge add(del\text{-}all) = \emptyset$. Although the search space derived from this domain generator does not contain dead ends, this characteristic can be incorporated: If a reverse plan exists for action a and $pre(a) = del(a) \wedge add(a) = \emptyset$, then the search space for computing action reversibility will not contain a dead end. This is proven by the necessary existence of actions that lack preconditions and can be used to generate predicates, such as the actions `add-f<*>`. Without such actions, a reverse plan would not be feasible.

Our proposed *dead ends domain generator* constructs domains exhibiting dead end states with the help of a `token` predicate. This predicate is necessary for the execution of the action that needs to be reversed, i.e., it is added to the precondition of action a (`del-all` in Listing 4). The introduction of a new action, `consume`, results in the irreversible removal of this `token` predicate upon execution. As there exists no way of generating the `token` predicate, this action imminently results in a dead end as it prohibits the application of the final action a. This, in a sense malicious, action has the potential to be executed at any point during the search, but the effect may only be noted at the very end. Note that this is a variation from the original notion of dead ends presented in [15], where the `token` predicate was required by all actions and thus the execution of action `consume` would not permit any further action application afterwards. We believe that the notion presented here better resembles practical application domains, while being more challenging for any search procedure.

3.4 Search space characteristics

The three elementary domain generators discussed above produce PDDL domains with varying search space characteristics. Figure 1 offers an overview of these characteristics, serving as a sufficient basis for an initial conceptual performance analysis. This analysis aims to predict the relative performance of the search strategies guess-and-check, DFS, and BFS in anticipation of the experiments following below.

In domains generated by the single path domain generator, there exists only a single reverse plan, denoted by a single path in the figure. This path is the only viable path in the domain. Accordingly, both search-based strategies, DFS and BFS, are expected to perform similarly, as they both traverse the same single path. Additionally, we would expect that the performance does not significantly decrease as the input parameter i increases. While the length of the reverse plan grows linearly with i, the search space remains structurally similar. In contrast, the guess-and-check strategy is expected to perform worse with increasing i, as the number of possible action sequences, ignoring the viability check, grows.

In the multiple paths domains, there exist multiple reverse plans, denoted by the multiple paths that lead from n_0 to n_g in Fig. 1. Note that all viable paths are valid reverse plans, but may vary in length. Accordingly, the DFS strategy is expected to perform similarly to the single path domain generator, as it still reaches n_g by only traversing a single path. However, the BFS strategy is expected to perform worse than the DFS strategy, as it explores all viable paths step-by-step. In contrast to the single path domains, the length of a reverse plan no longer increases linearly with i, but quadratic due to the need of reinstating all preceding predicates. A comparison with the guess-and-check strategy is more difficult, because the domains produced by the single and multiple paths domain generators for the same input i share a similar number of actions but vary in reverse plan length. Accordingly, a guess-and-check strategy is expected to perform worse on domains from the multiple paths domain generator compared to the single path domain generator when using the same input i. Compared to the BFS strategy, we expect the guess-and-check strategy to perform worse for the multiple paths domains, as it also explores paths that are not valid. Overall, we expect the DFS strategy to perform best, followed by the BFS strategy, and finally the guess-and-check strategy.

The dead ends domain generator is only a slight structural variation of the multiple paths domain generator. As such, the search space is conceptually very similar. Multiple valid reverse paths exist that may vary in length, but generally grow quadratically with the input parameter i. The key difference is that at any point during the search, a path leading to a dead end may be entered. Lengthwise, these dead end paths are indistinguishable from a valid path reaching the goal, such that an algorithm might not be able to distinguish between valid actions and actions leading to a dead end. This is illustrated in Fig. 1, where all branches leading to dead ends are at least as long as valid branches. In terms of search strategy performance, we expect the guess-and-check strategy to perform like on the multiple path domains

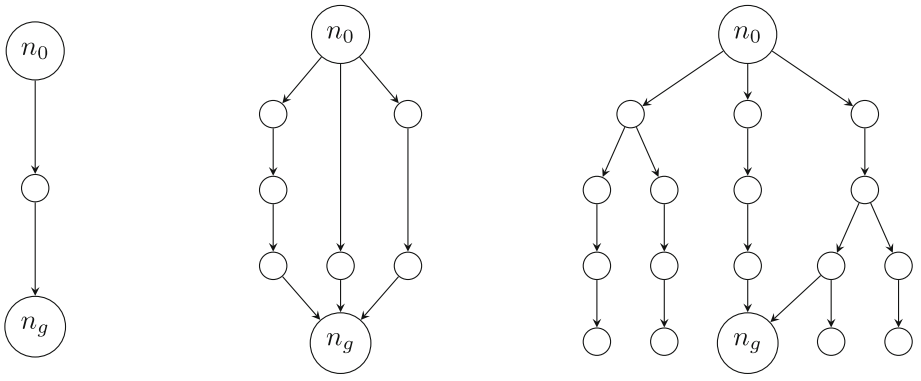

Fig. 1 Key search space characteristics of PDDL domains generated using single path (left), multiple paths (center), and dead ends (right) domain generators

because the number of predicates and the length of the reverse plan increases similarly. However, we expect the DFS strategy to perform worse than on the multiple path domains, because it has to backtrack whenever invoking action `consume`. Note that a dead end is not reached immediately after invoking action `consume`, but eventually after establishing all other predicates. If the dead end were immediately discovered, we would expect DFS strategies to perform like on the multiple path domains. In terms of BFS strategies, we expect a similar performance on the multiple path domains, because effectively a single additional action, namely `consume`, is being introduced. We compare these conceptual performance expectations with the results of our exploratory experiments in the following section.

4 Exploratory experiments with elementary domain generators

In what follows, we briefly describe our implementation of Algorithm 1, and primarily discuss the findings of our conducted performance evaluation for the BFS, DFS, and ASP implementation with respect to the aforementioned single path, multiple paths, and dead ends domain generators.

Our implementation[2] of Algorithm 1 uses PDDL as input format. The non-deterministic choice of the next action a in line 6, which requires a suitable selection strategy in a deterministic implementation, is implemented as follows: We successively traverse the search space where paths correspond to action sequences and explore it either in a BFS or DFS manner. We use these search strategies to provide a baseline. We consider two nodes in our search space equivalent if they share the same values for the sets F^+, F^-, and F^0, but ignore the value of π. As an optimization, we use this notion to introduce a cycle detection and do not consider a candidate action if its application yields an equivalent node, which has already been or is to be traversed.

For ASP, we use the current best performing encodings for both universal uniform reversibility (ASP-Simple) and uniform φ-reversibility (ASP-General) from [8], as demonstrated in a recent comparison [9]. The ASP(Q) [9] encoding is also considered. Note, however, that the focus of these experiments here is not primarily on evaluating the performance of action reversibility systems, but rather on testing our proposed domain generators and the characteristics of the domains they generate. Therefore, considering implementations that utilize different solving strategies is essential for our evaluation.

4.1 Results & discussion

Figure 2 depicts the runtime and memory usage of the four implementations, i.e., our DFS and BFS implementation, and both the simple and general ASP encoding of [8]. We use domains generated from the aforementioned three domain generators, i.e., single path, multiple path, and dead ends. In accordance with [6, 8, 9], we use the single path domain generator starting from input value $i = 10$ to $i = 500$ with step size 10. For our multiple paths and dead ends domain generators, we start at input value $i = 1$ and reduce the step size to 1.

[2] We implemented the DFS and BFS approaches as well as the domain generators in Python 3.9. Our implementation, the domain generators, and the results of our experiments presented in this paper are available online at https://github.com/TobiasSchwartz/strips-reversibility-benchmarks (accessed 2024-01-19). For persistence, the repository is also indexed in the Software Heritage Project's archive, see https://archive.softwareheritage.org/swh:1:rev:883df0b2683f8db058617aa1e8a783918b9ccb3e (accessed 2024-01-19). We conducted all experiments on a PC with a stock AMD Ryzen™ 7 5700X processor and 32 GB of RAM.

Observe that DFS and BFS perform similarly and outperform ASP for the single path domain in most cases. As outlined above, only a single action is applicable in each iteration when computing uniform φ-reversibility. Hence, the behavior of BFS and DFS are identical for this problem domain. Recall that there exists only a single reverse plan for the single path domain, such that computing uniform φ-reversibility is identical to the ASP approach computing universal uniform reversibility. However, the guess-and-check pattern employed by the ASP implementation only validates action sequences as a whole, leading to a weaker runtime performance. By contrast, the multiple paths domains exhibit numerous possible reverse plans. Hence, as expected, the DFS approach performs best for these domains, because every admissible action sequence eventually yields an applicable reverse plan. However, this poses a problem for the BFS approach due to the large number of potential states, which is also reflected by the increased memory usage. Despite the naive use of a guess-and-check pattern, the simple ASP encoding outperforms the BFS approach. We assume that this observation can be ascribed to the fact that our BFS implementation is not optimized regarding performance and memory usage in contrast to the solver underlying the ASP approach. Finally, in the dead ends domain we again find a large number of potential states. But, in this domain, not every action sequence necessarily has a valid reverse plan. Accordingly, both DFS and BFS require more time and space than the ASP approach.

In summary, our results show that minor changes regarding the domain generation and thus the resulting domains can have a major impact on the performance of different action reversibility algorithms. In particular, evaluating search based algorithms, e.g., the aforementioned DFS and BFS, using existing domain generators can easily lead to a biased result.

5 Advanced domain generators

Building upon the insights gained from the preceding experiments, we have developed two additional domain generators, which are examined in this section. The generalized domains generator emerges from a discussion on a general approach to domain generation, offering four adjustable parameters to facilitate the creation of domains that exhibit a predictable yet versatile search space structure. The parameters enable the quantification of the three previously discussed characteristics, i.e., single paths, multiple paths, and dead ends, that a domain may display. The Barabási-Albert domains generator uses the BA model to generate scale-free networks, which are transformed into PDDL domains with search spaces mimicking the networks' structure. This approach results in domains that exhibit less rigid search space structures than the domains generated by the generalized domains generator.

5.1 Generalized domains

The three domain generators discussed earlier, single paths, multiple paths, and dead ends, are derivatives of each other with slight syntactical modifications in the generator code. The thereby obtained domains each exhibit a different, but still very specific search space structure among the domains derived from the same generator for different values of input parameter i. However, our exploratory experiments showed that these three domain generators are not sufficient to derive a representative set of domains for evaluating action reversibility systems. In particular, we lack domains where a breadth-first strategy outperforms a depth-first strategy. Achieving this requires a new domain generator that allows for the construction of domains that favor breadth-first strategies. While one could design a new domain generator for this

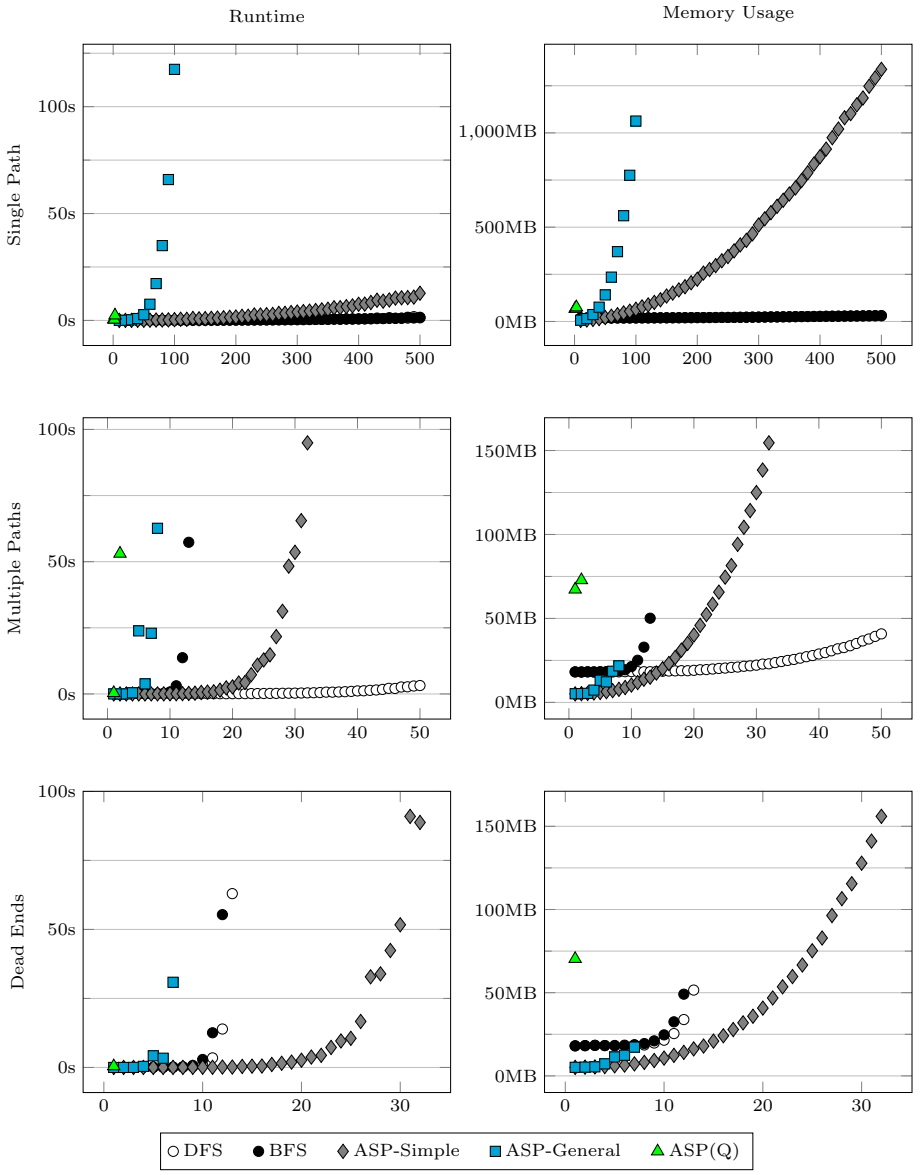

Fig. 2 Runtime (left column, y-axis) and memory usage (right column, y-axis) in relation to the domain generation parameter i (x-axis) of our BFS/DFS and the different ASP implementations for the single path (top), multiple paths (center), and dead ends (bottom) domains. A timeout of two minutes was applied. In the single path plots, the DFS data points are hidden behind the BFS data points

specific purpose in the same manner as the three domain generators discussed earlier, we opt for a more general approach that allows for the construction of a variety of domains with different search space characteristics. Depending on the supplied parameter values, the resulting domains favor different search strategies.

```
(:action del-all
:precondition (and  (f0) (f1) (f2) (f3))
:effect (and (f0) (not (f1)) (not (f2)) (not (f3))))

(:action edge-f0-f1
:precondition (f0)
:effect (f1))

(:action edge-f0-f2
:precondition (f0)
:effect (f2)))

(:action edge-f2-f3
:precondition (f2)
:effect (f3)))
```

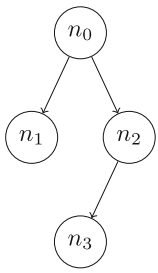

Fig. 3 Excerpt of an example domain (left) constructed from the search space (right), such that reversing action `del-all` yields the desired search space behavior

A closer look at the dead ends domain generator reveals that the length of a successful reverse plan is always at least as long as a path leading to a dead end (see Fig. 1). Therefore, increasing the only available input parameter i of the dead ends domain generator will increase the length of both path types. Controlling the lengths of these two path types individually can help generate domains exposing various search space characteristics. For example, one could construct domains that favor a breadth-first strategy using a different amount and length of valid and dead end path. Accordingly, our *generalized domain generator* receives four inputs: the number and length of valid paths, as well as the number and length of paths resulting in dead ends. With those four parameters, the generator is capable of generating domains that exhibit a similar search space structure as the domains generated by the three domain generators discussed earlier, and thus can be considered a generalization of these generators.

The construction procedure of our generalized domain generator works as follows. Given that we aim to create a domain that imposes certain search space characteristics, we may represent the search space as a graph $G = (N, E)$ where each node $n \in N$ denotes a valid state and each edge $(n_x, n_y) \in E$ denotes a choice leading from node n_x to node n_y. We can now construct a domain such that the search space upon searching for a reverse plan reflects this exact graph. For this, every node $n_i \in N$ is represented by an associated predicate `f<i>` and every edge $(n_x, n_y) \in E$ is represented by an associated action a_y, where $pre(a_y) = \{\texttt{f<x>}\}$ and $add(a_y) = \{\texttt{f<y>}\}$. Finally, we specify that action `del-all` requires all predicates, i.e., $pre(\texttt{del-all}) = \{\texttt{f<i>} \mid n_i \in N\}$, and deletes them, i.e., $del(\texttt{del-all}) = \{\texttt{f<i>} \mid n_i \in N\}$, upon execution. Reversing this action then yields a search space that is similar to G. Figure 3 uses a small example to illustrate such a constructed domain. While action `del-all` requires that the search traverses the entire search space in this case, we can easily alter the search by modifying action `del-all`. For example, in Fig. 3, we may only require to reach n_3. The set of actions remains unchanged, but the `del-all` action reflects this by changing the preconditions to $pre(\texttt{del-all}) = \{\texttt{f3}\}$, and deleting all predicates except `f0`, i.e., $del(\texttt{del-all}) = \{\texttt{f<i>} \mid n_i \in N, i > 0\}$ and $add(\texttt{del-all}) = \{\texttt{f0}\}$. Thus, the search has to start at node n_0 in order to reach n_3.

Similarly, for our generalized domain generator, all paths start at the same node n_0, and are then constructed independently. All valid paths eventually join again in the same goal node n_g, while paths resulting in dead ends simply stop after the same specified length. As every path is created independently, the generalized domain generator exposes an overall higher number of nodes, compared to the multiple paths domain generator, for creating domains that contain reverse plans of the same length. Additionally, paths leading into dead ends share the same length and thus always require the same amount of backtracking steps, whereas in our previous dead ends domains, a dead end may be just a single action that is easily corrected. A listing for this domain generator is omitted due to the dependency on multiple variables, but an example domain for parameters ($v_c = 1, v_l = 2, d_c = 1, d_l = 1$) is given in Listing 5, where v_c is the number of valid paths of length v_l, and d_c the number of paths leading to dead ends of length d_l. Essentially, the goal is to gain fact `f3`, for which there exists exactly one valid plan $\pi_v =$ `add-f0` $\rightarrow$ `add-f0-f1` $\rightarrow$ `add-f1-goal`, such that $|\pi_v| = 2 = v_l$. Likewise, there exists exactly one dead end path $\pi_d =$ `add-f0` $\rightarrow$ `add-f0-deadend` of length $|\pi_d| = d_l = 1$.

Listing 5 Example domain from our generalized domain generator.

```
(define (domain generalized-1-2-1-1)
(:requirements :strips)
(:predicates (f0) (f1) (f2) (f3) (f-init))

(:action del-all
:precondition
    (and (f3) (not (f0)) (not (f1)) (not (f2)) (not (f-init)))
:effect
    (and (f-init) (not (f0)) (not (f1)) (not (f2)) (not (f3))))

(:action add-f0
:precondition (f-init)
:effect (f0) (not (f-init)))

(:action add-f0-f1
:precondition (f0)
:effect (and (f1) (not (f0))))

(:action add-f1-goal
:precondition (f1)
:effect (and (f3) (not (f1))))

(:action add-f0-deadend
:precondition (f0)
:effect (and (f2) (not (f0))))
```

5.2 Barabási-Albert domains

The generalized domain generator we previously introduced enables the creation of domains showcasing intriguing corner cases of search space characteristics, favoring different search strategies. Moreover, the parameters specifying the desired count and length of paths leading to valid reverse plans and dead ends facilitate the construction of domains whose search space characteristics lie between the extremes of the archetypal structure of the three domain generators discussed earlier. However, the structure of all domains generated by the generalized domain generator remains quite rigid, a property that sophisticated action reversibility

systems might exploit to their benefit. To ensure a representative collection of domains, we suggest exploring domain generators that yield domains with less rigid structure.

To derive less rigid domains, we suggest utilizing the well-known BA model, commonly employed to model networks found in real-world scenarios [18]. In technical terms, BA networks are scale-free, signifying that they display a power-law degree distribution. Given these attributes, we consider this model worthwhile to investigate for enhancing the diversity of domains used to evaluate action reversibility systems. Hence, we propose a BA domain generator that employs two parameters n and m to first generate BA networks. In a second step, the resulting BA networks are transformed into PDDL domains. The following paragraphs outline the domain generation process and reveal the role the parameters n and m play in the context of BA networks.

To construct PDDL domains from BA networks, our BA domain generator initially generates a BA network G comprising n nodes that are incrementally added to G and then connected with bidirectional edges to m existing nodes with probability $\Pi(k)$. The probability $\Pi(k)$ that a new node links to an existing node n_i depends on the node's degree k_i, as $\Pi(k_i) = k_i / \sum_j k_j$ [26, Chapter 5.3]. The interested reader is kindly referred to the referenced work for more details and illustrations of the network creation process.

We then make the graph directed by only permitting edges (n_x, n_y) where $x < y$. The first added node n_0 denotes the start node. The combination of these two choices creates a directed graph where n_0 has an in-degree of 0 and all other nodes are reachable from there in only one direction. From n_0, we determine the shortest paths to all other nodes in the network. From these shortest paths, we choose the one that exhibits the longest path and select the last node of this path as the designated goal node n_g. The constructed domain then resembles a path search from n_0 to n_g in G. Note that by increasing the parameter m, and thus increasing the number of edges in the network, the length of this path naturally decreases. So, typically, a value of $m = 1$ exposes the longest paths, while the maximum value of $m = n - 1$ makes the length of all paths close to 1. In our testing, due to the power-law degree distribution, moderate values for m usually do not yield drastic path length differences.

The translation of the network $G = (N, E)$ into a benchmark domain follows the same technique as employed by the generalized domain generator introduced earlier. Specifically, each node $n_i \in N$ gets represented by a corresponding predicate `f<i>`, and every edge $(n_x, n_y) \in E$ gets represented by an action a_y, where $pre(a_y) = \{$`(f<x>)`$\}$ and $add(a_y) = \{$`(f<y>)`$\}$. As a final step, we introduce the action `del-all` with precondition $pre($`del-all`$) = \{$`(f<g>)`$\}$, delete effects $del($`del-all`$) = \{$`(f<i>)` $\mid n_i \in N, i > 0\}$, and add effect $add($`del-all`$) = \{$`(f0)`$\}$. As previously, this action serves as the target for computing reverse plans.

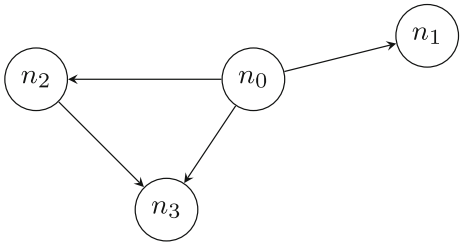

Fig. 4 A directed Barabási-Albert network with $n = 4$ nodes and parameter $m = 2$

An example of such a domain produced by our *BA domain generator* for $n = 4, m = 2$ is depicted in Listing 6. The domain represents a path search from node n_0 to n_1 in the network shown in Fig. 4. To guarantee reproducible results, our BA domain generator is seeded. This means that a network with $n = 4, m = 2$ is identical to a network with $n = 3, m = 2$ generated at some other point in time, except for one additional node and its edges according to the parameter $m = 2$.

Listing 6 Example domain from our Barabási-Albert domain generator.

```
(define (domain barabasiAlbert-4-2)
(:requirements :strips)
(:predicates (f0) (f1) (f2) (f3) (f-init)

(:action del-all
:precondition
    (and (f1) (not (f0)) (not (f2)) (not (f3)) (not (f-init)))
:effect
    (and (f-init) (not (f0)) (not (f1)) (not (f2)) (not (f3))))

(:action add-f0
:precondition (f-init)
:effect (f0) (not (f-init)))

(:action add-f0-f1
:precondition (f0)
:effect (and (f1) (not (f0))))

(:action add-f0-f2
:precondition (f0)
:effect (and (f2) (not (f0))))

(:action add-f0-f3
:precondition (f0)
:effect (and (f3) (not (f0))))

(:action add-f2-f3
:precondition (f2)
:effect (and (f3) (not (f2)))))
```

5.3 Search space characteristics

The two advanced domain generators produce domains that explore search spaces with distinct characteristics. As done before, this section examines these characteristics and draws conclusions for the expected performance of algorithms addressing the related problems. Figure 5 provides a schematic depiction of the search spaces produced by the generalized and BA domain generators.

The domains produced by the generalized domain generator display a rigid search space structure, similar to the three domain generators: single paths, multiple paths, and dead ends. While the generalized domain generator introduces more new actions to achieve the desired search space characteristic, the multiple paths domain generator in particular modifies the precondition and effect of actions to attain the desired search space characteristic. Note that this encoding style of the generalized domain generator is due to the graph-based translation, which is also leveraged by the BA domain generator. The four parameters of the generalized

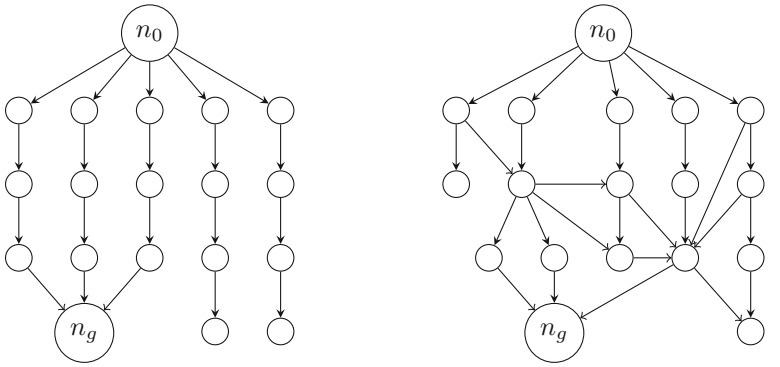

Fig. 5 On the left: search space of a domain from the generalized domain generator with 2 dead end and 3 valid paths leading from n_0 to n_g, all of length four. On the right: search space of a domain from the Barabási-Albert domain generator

domain generator can be utilized not only to enhance the complexity of the generated domains, but also to influence their search space characteristics. This flexibility facilitates the creation of domains that mimic the characteristics of the three basic domain generators. For instance, setting the number of valid paths to one and the number and length of paths leading to dead ends to zero results in domains akin to those of the single path domain generator. As before, we anticipate BFS and DFS to perform comparably in these domains. To mimic the multiple paths domain generator using the generalized domain generator, the length of paths leading to dead ends must remain zero, but the number of valid paths must be greater than one. However, unlike traditional multiple path domains, all valid paths generated by this method have equal length. Nevertheless, we still expect DFS to outperform BFS in this scenario, because all paths eventually lead to the goal state. Moreover, we expect a guess-and-check strategy to perform worse than for the multiple paths domain generator due to the action heavy domain encoding of the generalized domain generator. To emulate the dead ends domain generator, the generalized domain generator is configured with identical non-zero path lengths for both valid and dead-end paths, while ensuring a positive number of paths. For domains with few short valid paths but many long paths leading to dead ends, we predict BFS to outperform strategies employing a DFS approach. Such a situation cannot be achieved using the single path, multiple paths, or dead ends domain generators.

Regarding the domains generated by the BA generator, the search space structure is less rigid compared to the domains produced by the other generators, primarily due to its probabilistic generation procedure. While the domains generated by the single path, multiple paths, and dead ends generator display a rigid search space characteristic, the four parameters of the generalized domain generator can be leveraged to obtain domains exhibiting similar rigid characteristics. In contrast, the domains generated by the BA model are less rigid, which, however, complicates conceptual performance estimation. Similar to the generalized domain generator, estimating performance is particularly challenging for domains that do not display explicit corner cases. In this regard, we expect that primarily extreme values for m, such as 1 or $n - 1$ provide special cases. For $m = 1$ the network has relatively few edges and the reverse paths in the generated domains thus tend to be the longest. Without many choices deviating from the correct path, we especially expect DFS to perform well here, but recon that also BFS might show similar performance based on the same reason. While this observation holds for almost any number of nodes, the performance of the ASP encodings is

expected to heavily depend on the network size, as every node introduces a new predicate. In contrast, a network with maximum number of edges, produced by setting $m = n - 1$, likely yields only very short reverse plans. Consequently, we expect that the BFS strategy performs exceptionally well, whereas the DFS strategy may struggle with higher values of n due the numerous options of paths. Meanwhile, the ASP encodings should not be affected by the additional edges. Finally, a moderate value for m yields domains with multiple edges that provide choice, while slightly increasing the reverse plan length. We adopt $m = 5$ for subsequent experiments, as this moderate value is commonly used in related work, including [18]. As the maximum path length is limited in any given BA network, we again expect BFS to perform well. DFS may sometimes get lucky and get to the goal fast, in other instances it may run into irrelevant parts of the network or even dead ends, where backtracking can be time-consuming. This is the area where we believe that search can greatly benefit from additional guidance, for instance through heuristics.

It is also worth noting that the reverse plan length was previously determined by supplied parameter values, whereas the reverse plan length of the BA domains is determined by the longest path found during network construction. If this information is expected to be available to the search algorithm during evaluation, it needs to be manually supplied. This information is often required when using a guess-and-check approach. However, when this information is not available, the assumed length of the reverse plan can be increased incrementally until the approach finds a valid path.

Figure 5 displays exemplary search spaces that can be created using the generalized domain generator and the BA domain generator respectively. Regarding the former, the three valid paths leading to the goal node n_g share all the same length, just like the two paths that lead into dead ends.

6 Experiments with advanced domain generators

This section covers a discussion of the experiments we conducted on domains produced with the advanced, i.e., the generalized and the BA, domain generators and presents the obtained results. To showcase the versatility of the generators regarding the characteristics of the generated domains, the experiments build upon different scenarios that translate to different parameter settings respectively.

For compatibility reasons, we introduced a syntactic modification to the advanced domain generators[3], which is necessary for the ASP-Simple encoding to work properly. In particular, we extend the precondition of the `del-all` action (cf. line 6, Listings 5 and 6) with all missing predicates (in negated form). This satisfies the observation in [8] that any action in the reverse plan π for a given action a only permits predicates contained in $pre(a)$. So, without the addition, intermediate actions would not be considered relevant and ignored by the ASP encoding. Note that this change does not affect the domains' semantics, as it merely explicitly enumerates facts already implicitly false in the STRIPS state representation. Furthermore, it does not impede the runtime of the BFS and DFS strategies on the created domains, since the algorithms compute the same set of actions, now with explicitly provided preconditions.

[3] Listings 5 and 6 already reflect these modifications.

6.1 Generalized domain generator experiments

The primary advantage of the generalized domain generator is its ability to produce domains with diverse search space characteristics. This ability is achieved by informing the generation procedure with four parameters that control the number of valid paths (v_c) and their length (v_l) as well as the number of paths leading to dead ends (d_c) and their length (d_l) exhibited by the generated domain. The following experiments examine the performance of our BFS/DFS implementations and the ASP implementations from [8] on a set of domains produced by the generalized domain generator. To be able to make a comparison between the new experimental results and the results from the exploratory experiments with the elementary domain generators, we examine three scenarios, adjustable via a single parameter that incrementally increases complexity. This parameter is used to derive values for the four internal parameters, ensuring that the resulting domains maintain comparable search space characteristics.

For our three scenarios, we use functions $f_1(x)$ to $f_3(x)$, where x is the parameter scaling the domains' complexity. Scenario 1 subsumes domains that contain exactly one short path to reach a desired goal and numerous paths that lead to dead ends of the same constant length, resulting in the following function[4]:

$$f_1(x) = (v_c, v_l, \lfloor x \cdot d_c \rceil, d_l), \text{ where } x \geq 1, v_c = 1, v_l = 4, d_c = 20, d_l = 4$$

In domains from Scenario 2, the numbers of valid paths and paths leading to dead ends scale, the former more pronounced than the latter. Furthermore, the length of the paths is again constant but higher in general. Hence, the function:

$$f_2(x) = (\lfloor x \cdot v_c \rceil, v_l, \lfloor x \cdot d_c \rceil, d_l), \text{ where } x \geq 1, v_c = 6, v_l = 10, d_c = 4, d_l = 10$$

Finally, domains from Scenario 3 exhibit a constant number of valid paths with a short constant length and a large number of dead end paths scaling in length. For this scenario, we use the function:

$$f_3(x) = (v_c, v_l, \lfloor x \cdot d_c \rceil, \lfloor x \cdot d_l \rceil), \text{ where } x \geq 1, v_c = 10, v_l = 4, d_c = 2, d_l = 2$$

Figure 6 shows the results of our experiments with the generalized domain generator with respect to the three scenarios. In contrast to Fig. 2, the x-axes of the individual plots now signify the scaling factor x used in f_1 to f_3 instead of the values of the parameter i used by the elementary domain generators. A discussion of the results follows after some general remarks on the BA domain generator experiments in the final part of this section.

6.2 Barabási-Albert domain generator experiments

The BA domain generator internally utilizes common network structures found in real world applications to derive PDDL domains for evaluating action reversibility. For this purpose, the two parameters n and m allow control over the number of nodes, and the approximate number and placement of edges in the network. Based on the preferential attachment, most nodes have only a few connecting edges, while a few become hubs, accumulating multiple edges. Consequently, most nodes in the network are either directly connected to a hub, or

[4] The $\lfloor \rceil$ brackets indicate the nearest integer function.

require only a few intermediary nodes to reach the next hub, yielding shortest paths with relatively low hop distances between all pairs of nodes.

Our experiments using the BA domain generator aim at studying the effect of this growth in network interconnectivity and how it impacts the performance of the action reversibility algorithms. To this end, we study three distinct values for m, with $m \in \{1, 5, (n-1)\}$ that represent the scenarios for this domain generator. We then use the number of nodes n for scaling the complexity of the domains. Figure 7 shows the results of our experiments with the BA domain generator in regards to the three values of m. The x-axes here display the number of nodes $n \in [2000, 6000]$.

6.3 Results & discussion

In this section, we discuss the results of our experiments with the advanced domain generators. We start with the results for the generalized domain generator and then map these insights into the less rigid domains generated by the BA domain generator. Note that we do not consider the ASP-General [8] or the ASP(Q) [9] encodings in this evaluation, as they consistently showed worse performance in the exploratory experiments compared to the ASP-Simple encoding.

In Scenario 1 of the generalized domain generator, each domain comprises one valid, and an increasing number of dead end paths of short constant length. As shown in Fig. 6, the runtime of both the DFS and BFS strategy increase similarly, as both strategies here have to traverse almost the entire search space. Meanwhile, the ASP-Simple encoding shows a more constant runtime that remains well below that of the others. We can see the similar exploration of the search space of the DFS and BFS strategies also manifest in a strong overlap in their memory usage.

In contrast to the ASP implementations, our BFS and DFS implementations are based on Python and excessively use set operations. Hence, we ascribe the salient jumps for BFS and DFS in the right column plots to Python's set implementation, which increases the allocated memory at certain thresholds based on the occupancy of a set. The exploratory experiments did not show such jumps since the number of facts remained below these thresholds.

As expected, the plots for Scenario 2 show that the BFS strategy suffers in domains where a lot of valid and dead end paths of greater length are present, since it has to explore all available paths incrementally. In contrast, the DFS strategy just has to select the starting node of an arbitrary valid path to terminate successfully soon after, hence its lower runtime. The ASP-Simple encoding shows a similar runtime as the DFS strategy for smaller domains, but again outperforms the others on larger instances.

The plots for Scenario 3, where both the amount and length of dead end paths increase, displays a sudden and drastic increase of the DFS strategy's runtime and memory usage. At the same time, the two other strategies remain mostly unaffected. The reason for this is that the DFS strategy gets lost in the increasingly long and numerous dead end paths, while the BFS strategy eagerly evaluates all nodes of the same depth before diving deeper into the graph.

Considering the first BA network scenario resulting from parameter $m = 1$, we observe a mostly linear increase in runtime across all strategies with growing network size. While the DFS strategy and the ASP-Simple encoding show a very similar performance, the BFS strategy performs slightly worse. In this scenario, the valid paths are typically the longest and only few dead end paths of shorter length exist. It is hence comparatively easy to find the correct path in a DFS manner.

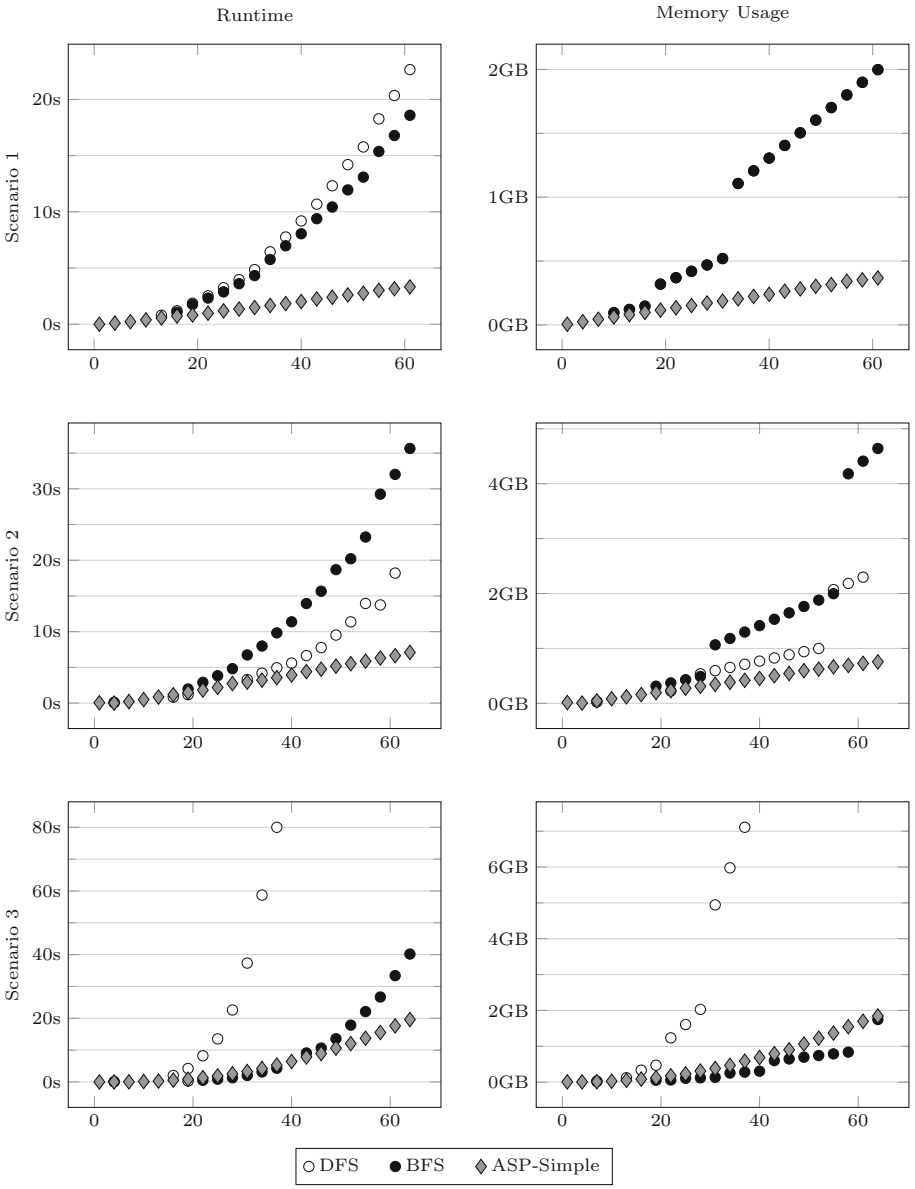

Fig. 6 Runtime (left column, y-axis) and memory usage (right column, y-axis) of our procedural BFS/DFS and the ASP-Simple/ASP-General implementations from [8] for the generalized domains. The domains originate from three different scenarios that are specified via functions f_1 to f_3. The x-axes signifies the value of the scaling factor x used in these functions. A timeout of two minutes was applied

These observations seem to also hold for the second BA network scenario with $m = 5$. While the plots indeed look very similar, note that the runtimes differ by almost an order of magnitude. As expected, the problem gets harder for all search-based approaches, the more edges the network comprises. Only the ASP encoding is hardly affected by this change, as

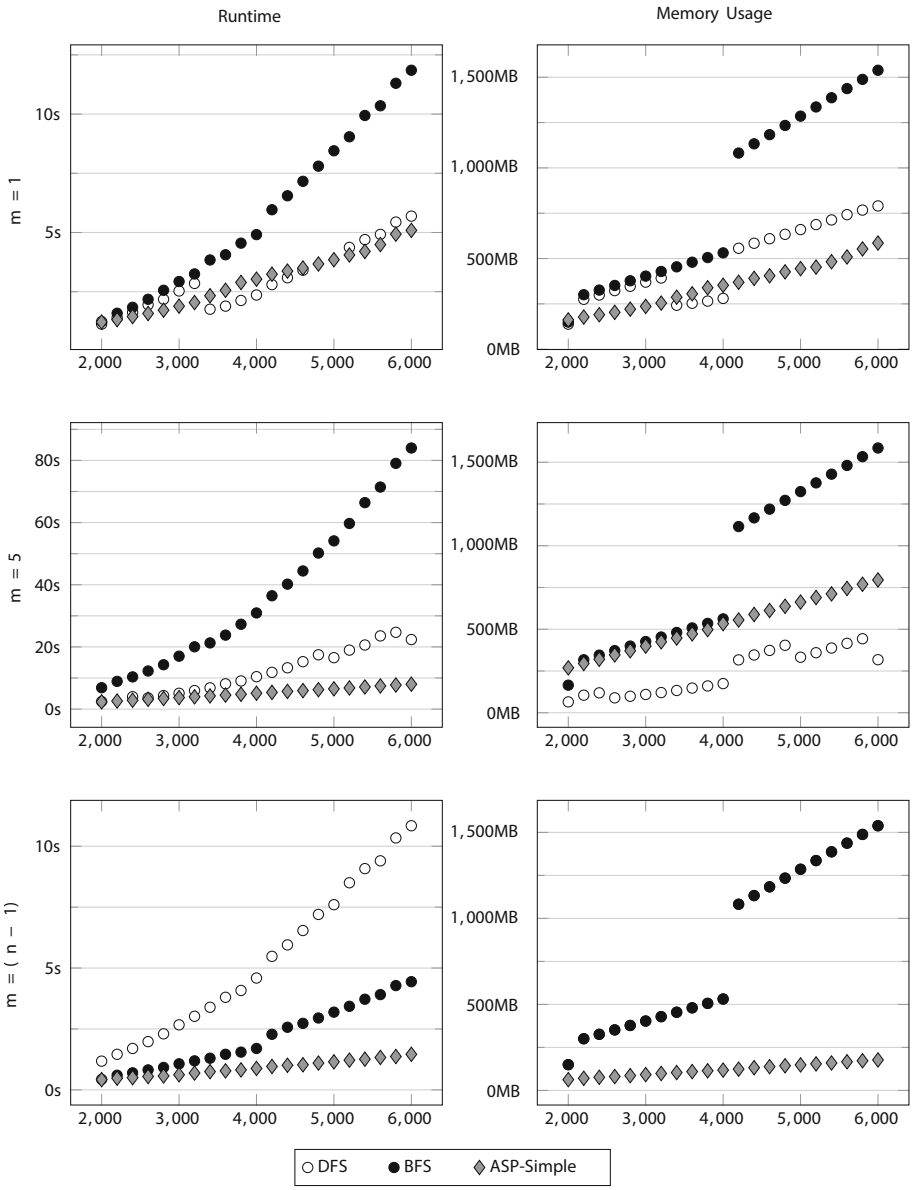

Fig. 7 Runtime (left column, y-axis) and memory usage (right column, y-axis) of our procedural BFS/DFS and the ASP-Simple/ASP-General implementations from [8] for Barabási-Albert domains with three different settings of the parameter m. The x-axis indicates the value of the n parameter. A timeout of two minutes was applied

it mostly depends on the total number of states (nodes) and not necessarily the transition between those.

Finally, the last BA network scenario yields very short valid paths, but numerous dead ends. Accordingly, the BFS strategy here performs best out of all scenarios. The DFS strategy

requires the longest runtime, but still remains faster than on the second scenario, probably because the length of the dead end paths is more limited here. Again, the ASP-Simple encoding shows a very fast runtime here that is only slightly affected by network size.

In summary, with respect to this paper's overarching goal of identifying useful domain generators for the evaluation of action reversibility in STRIPS, these experimental results are promising. We show that the presented generalized domain generator is highly versatile, allowing to create benchmarks that tend to favor certain search characteristics by adjusting only few simple parameters. We demonstrate this using three scenarios within the generalized domain generator, where different results are observed for the considered strategies based on varying parameter settings.

Similar effects can also be observed in the BA network scenarios, indicating that the proposed BA network domain generator might be a useful resource for assembling a diverse benchmark suit, including less rigid domains. Note that currently these domain generators are adjusted such that they produce domains that can be solved using universal uniform reversibility. This small modification allows the usage of the ASP-Simple encoding, which employs distinct optimizations that do not easily translate to uniform φ-reversibility (cf. [9]). While this optimization effectively resulted in superior performance of the ASP-Simple encoding in our evaluation, this does not necessarily translate to encodings for the computationally more expensive uniform φ-reversibility that both search-based strategies implement.

6.4 Threats to validity

This paper focuses on creating benchmarks for the formalism of uniform φ-reversibility [5]. Multiple studies in this area [6, 7, 9] have employed similar benchmarks to experimentally validate and compare the performance of implementations. It is noteworthy, however, that the theoretical complexity of finding reverse plans has not been fully established. The algorithm proposed by [5] is proven to be sound but not complete, which may impact the interpretation of benchmark results. Current complexity results are as follows:

1. Finding reverse plans in STRIPS has an upper bound of Σ_2^P [5].
2. In more general planning contexts, checking for the existence of a reverse plan is Σ_3^P-complete [3].
3. When the reverse plan contains at most two actions, the problem is Σ_2^P-complete [3].

These results indicate that the exact complexity for the STRIPS setting remains open, and that reverse plan length influences problem complexity. Further research in this direction could strengthen the theoretical foundations in this domain.

Despite these existing theoretical uncertainties, we argue that empirical studies can provide valuable insights and guide further theoretical work. Our benchmarks are designed to cover a range of problem structures and sizes, aiming to maintain relevance as theoretical understanding advances. We acknowledge that future theoretical advancements may impact our results and interpret them with appropriate caution.

Future research should focus on strengthening the theoretical foundation, particularly for the STRIPS setting, proving the completeness of existing algorithms or developing new complete ones, and investigating how theoretical advancements might refine the interpretation of empirical results. Our current work provides a practical foundation for these future theoretical investigations.

7 Conclusion

In this paper, we implemented the notion of uniform φ-reversibility based on the nondeterministic algorithm proposed by [5]. Our work responds to the authors' call in [5] for a practical implementation of their theoretical algorithm. Our implementation includes a BFS and a DFS search strategy. Following the suggestion of [6], we evaluate our procedural implementation in close comparison to different variants of Answer Set Programming implementations [6, 8, 9]. Evaluating action reversibility, however, turns out to be a difficult challenge on its own. In contrast to classical planning problems, the complexity of action reversibility primarily depends on the planning domain and not on particular planning instances. However, the majority of existing planning benchmark problems, e.g., from the International Planning Competition, are complex at instance level, whereas actions in the domain are often, if reversible at all, trivially reversible using inverse actions. The domain generator introduced by [6] generates STRIPS domains of increasing complexity. We show that these contain a strong bias favoring particular search strategies. We have designed two domain generators that produce domains featuring multiple reverse plans and dead ends. All these elementary domain generators, albeit similar in construction, produce domains that drastically favor different search strategies. We thereby highlight the challenge of constructing and using domain generators with respect to evaluating action reversibility systems.

As a next step towards a benchmark for the evaluation of action reversibility in STRIPS, we introduced two advanced domain generators, the generalized domain generator and the BA domain generator. While the former provides granular control over the characteristics of the resulting domains through four parameters, the latter focuses on reducing the usage of contrived generation patterns by leveraging the BA model to generate less rigid networks used as a basis for the domains. Our experiments demonstrate the multifaceted impact of different scenarios on the performance of all considered search strategies. Hence, we conclude that the advanced domain generators enable the composition of versatile and diverse benchmarks for the evaluation of action reversibility in STRIPS, thus paving the way for research on topics such as automated algorithm selection [27].

For future work, establishing standard scenarios and parameter settings for the advanced domain generators is crucial to create a comprehensive benchmark. While the scenarios presented in this paper provide a foundation, we recognize that defining these parameters should be a collaborative community effort. The investigation of additional advanced domain generators, e.g., ones that are based on n-bit counters, might also be expedient to deepen our understanding of the intricacies domains with distinct structures exhibit in terms of reversibility. Moreover, further theoretical investigations that analyze the complexity of the decision problems and their fragments underlying action reversibility are crucial to assess the significance of benchmark-based evaluations. Additionally, algorithms addressing uniform reversibility need further study to establish their theoretical properties with greater certainty.

Acknowledgements We would like to thank Moritz Bayerkuhnlein for his help regarding the application of the ASP encodings. We also express our gratitude to the anonymous reviewers, whose feedback significantly improved the paper's quality.

Author Contributions The authors collaborated on all sections of the paper. T.S. was the primary contributor (60%), with J.B. and L.M. providing equal secondary contributions (20% each).

Funding Open Access funding enabled and organized by Projekt DEAL.

Data Availability A replication package is available at the Software Heritage Project's archive: https://archive.softwareheritage.org/swh:1:rev:883df0b2683f8db058617aa1e8a783918b9ccb3e

Declaration

Competing Interests The authors declare no competing interests.

References

1. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning –Ttheory and Practice. Morgan Kaufmann, San Francisco, CA (2004). https://doi.org/10.1016/B978-1-55860-856-6.X5000-5
2. Ghallab, M., Nau, D.S., Traverso, P.: Automated Planning and Acting. Cambridge University Press, Cambridge, UK (2016). https://doi.org/10.1017/CBO9781139583923
3. Eiter, T., Erdem, E., Faber, W.: Undoing the effects of action sequences. J. Appl. Log. **6**(3), 380–415 (2008). https://doi.org/10.1016/j.jal.2007.05.002
4. Daum, J., Torralba, Á., Hoffmann, J., Haslum, P., Weber, I.: Practical undoability checking via contingent planning. In: Coles, A.J., Coles, A., Edelkamp, S., Magazzeni, D., Sanner, S. (eds.) Proceedings of the 26th International Conference on Automated Planning and Scheduling (ICAPS 2016), pp. 106–114. AAAI Press, Washington, DC (2016). https://doi.org/10.1609/icaps.v26i1.13751
5. Morak, M., Chrpa, L., Faber, W., Fiser, D.: On the reversibility of actions in planning. In: Calvanese, D., Erdem, E., Thielscher, M. (eds.) Proceedings of the 17th International Conference on Principles of Knowledge Representation and Reasoning (KR 2020), pp. 652–661. IJCAI Organization, California (2020). https://doi.org/10.24963/kr.2020/65
6. Chrpa, L., Faber, W., Fiser, D., Morak, M.: Determining action reversibility in STRIPS using answer set programming. In: Dodaro, C., Elder, G.A., Faber, W., Fandinno, J., Gebser, M., Hecher, M., LeBlanc, E., Morak, M., Zangari, J. (eds.) International Conference on Logic Programming 2020 Workshop Proceedings Co-located with 36th International Conference on Logic Programming (ICLP 2020). CEUR Workshop Proceedings, vol. 2678. CEUR-WS.org, Aachen, Germany (2020). http://ceur-ws.org/Vol-2678/paper2.pdf
7. Chrpa, L., Faber, W., Morak, M.: Universal and uniform action reversibility. In: Bienvenu, M., Lakemeyer, G., Erdem, E. (eds.) Proceedings of the 18th International Conference on Principles of Knowledge Representation and Reasoning (KR 2021), pp. 651–654 (2021). https://doi.org/10.24963/kr.2021/63
8. Faber, W., Morak, M., Chrpa, L.: Determining action reversibility in STRIPS using answer set and epistemic logic programming. J. Theory Pract. Logic Program. **21**(5), 646–662 (2021). https://doi.org/10.1017/S1471068421000429
9. Faber, W., Morak, M., Chrpa, L.: Determining action reversibility in STRIPS using answer set programming with quantifiers. In: Cheney, J., Perri, S. (eds.) Proceedings of the 24th International Symposium on Practical Aspects of Declarative Languages (PADL 2022). Lecture Notes in Computer Science, vol. 13165, pp. 42–56. Springer, Cham, Switzerland (2022). https://doi.org/10.1007/978-3-030-94479-7_4
10. Williams, B.C., Nayak, P.P.: A reactive planner for a model-based executive. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 1178–1185. Morgan Kaufmann, San Francisco, CA (1997). http://ijcai.org/Proceedings/97-2/Papers/056.pdf
11. Weber, I., Wada, H., Fekete, A.D., Liu, A., Bass, L.: Automatic undo for cloud management via AI planning. In: Freedman, M.J., Suri, N. (eds.) Proceedings of the 8th Workshop on Hot Topics in System Dependability (HotDep 2012). USENIX Association, Berkeley, CA (2012). https://www.usenix.org/conference/hotdep12/workshop-program/presentation/weber

12. Cserna, B., Doyle, W.J., Ramsdell, J.S., Ruml, W.: Avoiding dead ends in real-time heuristic search. In: McIlraith, S.A., Weinberger, K.Q. (eds.) Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI 2018), pp. 1306–1313. AAAI Press, Washington, DC (2018). https://doi.org/10.1609/aaai.v32i1.11533
13. Bylander, T.: The computational complexity of propositional STRIPS planning. J. Artif. Intell. **69**(1–2), 165–204 (1994). https://doi.org/10.1016/0004-3702(94)90081-7
14. Celorrio, S.J., Segovia-Aguas, J., Jonsson, A.: A review of generalized planning. The Knowl. Eng. Rev. **34**, 5 (2019). https://doi.org/10.1017/S0269888918000231
15. Schwartz, T., Boockmann, J.H., Martin, L.: Towards the evaluation of action reversibility in STRIPS using domain generators. In: Varzinczak, I. (ed.) Proceedings of the 12th International Symposium on Foundations of Information and Knowledge Systems (FoIKS 2022). Lecture Notes in Computer Science, vol. 13388, pp. 226–236. Springer, Cham, Switzerland (2022). https://doi.org/10.1007/978-3-031-11321-5_13
16. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd Edition. MIT Press, Cambridge, MA (2009). http://mitpress.mit.edu/books/introduction-algorithms
17. Kurant, M., Markopoulou, A., Thiran, P.: On the bias of BFS (breadth first search). In: 22nd International Teletraffic Congress, ITC 2010, Amsterdam, The Netherlands, September 7-9, 2010, pp. 1–8. IEEE, Piscataway, NJ (2010). https://doi.org/10.1109/ITC.2010.5608727
18. Barabási, A.-L., Albert, R.: Emergence of scaling in random networks. Sci. **286**(5439), 509–512 (1999). https://doi.org/10.1126/science.286.5439.509
19. Haslum, P., Lipovetzky, N., Magazzeni, D., Muise, C.: An introduction to the planning domain definition language. In: Synthesis Lectures on Artificial Intelligence and Machine Learning. Springer, Cham, Switzerland (2019). https://doi.org/10.2200/S00900ED2V01Y201902AIM042
20. Vallati, M., Chrpa, L., Grzes, M., McCluskey, T.L., Roberts, M., Sanner, S.: The 2014 international planning competition: progress and trends. J. AI Mag. **36**(3), 90–98 (2015). https://doi.org/10.1609/aimag.v36i3.2571
21. Seipp, J., Torralba, Á., Hoffmann, J.: PDDL Generators. Zenodo, Geneva, Switzerland (2022). https://doi.org/10.5281/zenodo.6382174
22. Koehler, J., Hoffmann, J.: On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. J. Artif. Intell. Res. **12**, 338–386 (2000). https://doi.org/10.1613/jair.715
23. Gupta, N., Nau, D.S.: On the complexity of blocks-world planning. J. Artif. Intell. **56**(2–3), 223–254 (1992). https://doi.org/10.1016/0004-3702(92)90028-V
24. Hoffmann, J., Nebel, B.: The FF planning system: fast plan generation through heuristic search. J. Artif. Intell. Res. **14**, 253–302 (2001). https://doi.org/10.1613/jair.855
25. Shleyfman, A., Karpas, E.: Position paper: reasoning about domains with PDDL. In: Proceedings of the 26th AAAI Spring Symposium, pp. 582–587. AAAI Press, Washington, DC (2018). https://aaai.org/proceeding/01-spring-2018
26. Barabási, A.-L.: Network Science. Cambridge University Press, Cambridge, UK (2015). http://networksciencebook.com
27. Rice, J.R.: The algorithm selection problem. Adv. Comput. **15**, 65–118 (1976). https://doi.org/10.1016/S0065-2458(08)60520-3