

Reducing Memory Usage in Fast Downward by Working with a Delta Representation of States

Bachelor Thesis

Natural Science Faculty of the University of Basel
Department of Mathematics and Computer Science
Research Group Artificial Intelligence
<https://ai.dmi.unibas.ch>

Examiner: Prof. Dr. Malte Helmert
Supervisor: Simon Dold

Istref Uka
i.uka@stud.unibas.ch
23-051-428

21.07.2026

Acknowledgments

I would like to thank my family and friends for their continuous support throughout the writing of this bachelor thesis. Their help and encouragement allowed me to dedicate more time and attention to my work.

I am especially grateful to my supervisor, Simon Dold, for the many helpful meetings, his quick responses to my questions, and his continued interest in the progress of this thesis. His guidance was very valuable during the development of this work.

Finally, I would like to thank Prof. Dr. Malte Helmert for giving me the opportunity to write my thesis in the Artificial Intelligence research group.

Use of AI Tools

I used OpenAI's ChatGPT¹ as a supporting tool during the writing of this thesis. In particular, I used it to reformulate sentences and phrases when I was unsure how to express them in a more formal academic style. I also used ChatGPT to support the development of the Python scripts that generated the plots used in the experimental results section.

¹ <https://chatgpt.com/>

Abstract

This thesis investigates the reduction of memory usage in Fast Downward by using a delta representation of states. In this work, these states are referred to as delta-states. Instead of storing a complete state, a delta-state stores a reference to the state from which it was generated and the differences between the new state and its predecessor.

The motivation for this approach is that states may contain many variables, while often only a small number of variables change between consecutive states. The results show that, with an efficient implementation, delta-states can reduce memory usage when states are large and only a few variables change between transitions. However, this memory reduction comes at the cost of increased search time, because full states must be reconstructed by following predecessor references and applying the accumulated changes whenever the complete state representation is required.

Table of Contents

Acknowledgments	ii
Use of AI Tools	iii
Abstract	iv
1 Introduction	1
2 Overall Idea	3
2.1 SAS ⁺	3
2.1.1 States	3
2.1.2 Operators and Plans	3
2.1.3 Precondition of Operators and Goal Conditions	4
2.1.4 Effects of Operators	4
2.2 Delta-States	5
2.2.1 Mathematical Definition	5
2.2.2 Reconstructing Complete States from Delta States	5
2.2.3 Example	6
3 Existing State Storage	7
3.1 StateRegistry	7
3.1.1 SegmentedArrayVector	8
3.1.2 IntPacker	8
3.1.3 Example	9
3.1.4 StateIDSet	9
4 Delta-State Storage	11
4.1 StateRegistry Extensions for delta-states	11
4.1.1 DeltaStateInfo	11
4.1.2 DeltaPacker	12
4.1.2.1 Example	13
4.1.3 DeltaStateIDSet	15
5 Important Algorithms	17
5.1 Important Algorithms for Using Delta-States	17

5.1.1	Delta-State Eager Search Algorithm	17
5.1.2	Reconstructing Complete States from Delta-States	18
6	Experimental Results	20
6.1	Experimental Results and Analysis of Different Problems	20
6.1.1	PDDL	20
6.1.2	First Run	22
6.1.2.1	Problems Used	22
6.1.2.2	Heuristic and Search Configuration	22
6.1.2.3	Metrics Explained	23
6.1.2.4	Results	24
6.1.2.5	Analysis	25
6.1.3	Problems Used for the Second Run	26
6.1.3.1	Heuristic and Search Configuration	26
6.1.3.2	Results	27
6.1.3.3	Analysis	33
7	Conclusion and Outlook	34
7.1	Conclusion	34
7.2	Outlook	35
	Bibliography	36
	Appendix A Appendix	37
.1	PDDL Files	37
.1.1	Low-Change Problem	37
.1.2	High-Change Problem	42

1

Introduction

Automated planning is concerned with finding sequences of actions that transform an initial state into a goal state. A planning task defines the initial state, the available actions, and a goal condition. The planner then has to determine whether there exists a sequence of applicable actions that reaches a goal state. Depending on the search algorithm and the cost model, the objective may be to find any valid plan, to find an optimal plan with minimal cost, or to prove that no plan exists.

To solve a planning task, a planner has to explore the state space induced by the actions. Starting from the initial state, it generates successor states by applying all applicable actions. In a simple tree-search view, every generated successor is treated as a new search node, even if the same state has already been reached before through a different sequence of actions. This can lead to repeated exploration of equivalent states, especially in planning tasks where actions can undo each other or where different action orders lead to the same result.

For this reason, modern planners usually use a graph-search variant instead of pure tree search. In graph search, generated states are stored so that the planner can detect duplicates. If a newly generated state is equivalent to a state that has already been seen, the planner can reuse the existing state information instead of inserting and exploring the same state again. This duplicate detection is important for both correctness and efficiency: it avoids unnecessary repeated work and, in optimal planning, helps ensure that the best known path to a state can be tracked correctly.

Modern planning systems such as Fast Downward [8] often explore large search spaces, where many states have to be generated and stored. As a result, memory consumption can become a limiting factor, especially if the problem requires many states to be generated and saved before a solution is found or before the absence of a solution can be shown.

In Fast Downward, states are typically represented explicitly. This means that each stored state contains values for all state variables, even if only a small number of variables changed compared to its predecessor. For planning tasks with large states, this can lead to high memory usage.

Reducing memory usage can allow the planner to handle larger search spaces or solve tasks that would otherwise run out of memory. Since consecutive states often differ only in a few variables, storing only these differences may be more memory-efficient than storing

complete states. This idea is related to differential encoding in data compression, where data is represented by differences to previously known values instead of storing each value independently [10].

The goal of this thesis is to investigate whether a delta representation of states can reduce memory usage in Fast Downward. In this representation, a state stores a reference to its predecessor and the changes that distinguish it from that predecessor.

To evaluate this idea, we design and implement a delta-state representation in Fast Downward and compare it to the existing state representation. The comparison focuses on memory usage and search time across planning tasks with different state sizes and transition characteristics. We implement and evaluate delta-states in the context of eager search. To avoid confusion, we refer to the eager search variant that uses delta-states as the *delta-state eager search*. Eager search is a commonly used search algorithm in Fast Downward and therefore provides a suitable setting for studying the effects of the new state representation. The main idea of delta-states is independent of the specific heuristic used during search: different heuristics may influence which states are expanded and in which order, but the trade-off introduced by delta-states remains the same. They can reduce memory usage by storing only differences between states, while potentially increasing search time whenever complete states have to be reconstructed.

Focusing on eager search therefore allows us to evaluate the impact of delta-states in a controlled and practically relevant setting, without having to adapt every search algorithm in Fast Downward.

The results show that delta-states can reduce memory usage in settings where states are large and only a few variables change between transitions, but they also introduce additional computational overhead when full states need to be reconstructed.

The remainder of this thesis is structured as follows. First, Chapter 2 introduces the overall idea of delta states in a more formal way. It explains the relevant parts of the SAS⁺ representation and defines what is meant by storing only the difference between a state and its predecessor. Chapter 3 then describes how states are stored in the original Fast Downward implementation. This provides the necessary background for Chapter 4, where the delta-state storage implementation is presented. Chapter 5 explains the most important algorithms that use delta states, especially the modified eager search and the reconstruction of complete states from delta states. Chapter 6 presents the experimental setup and analyzes the results. Finally, Chapter 7 summarizes the conclusions of this thesis and discusses possible directions for future work.

2

Overall Idea

The overall idea is to reduce the amount of memory required for storing states by representing newly generated states as the previously introduced delta states. Fast Downward represents planning tasks in the SAS⁺ formalism [1]. Therefore, we first describe the relevant parts of this representation.

2.1 SAS⁺

A SAS⁺ planning task can be described as a tuple $\Pi = \langle V, I, O, \gamma \rangle$, where V is a finite set of state variables, I is the initial state, O is a finite set of operators, and γ is the goal condition. Each variable $v \in V$ has a finite domain D_v of possible values. A state s assigns exactly one value from D_v to every variable v . Therefore, a state can be viewed as a vector of variable values.

2.1.1 States

Let $V = \{v_1, v_2, \dots, v_n\}$.

Then a state s can be written as $s = (s[v_1], s[v_2], \dots, s[v_n])$, where $s[v_i]$ denotes the value assigned to variable v_i in state s .

2.1.2 Operators and Plans

An operator is defined by its precondition and its effects. The precondition specifies when the operator is applicable, while the effects describe how the state changes after applying the operator. In cost-based planning settings, operators may additionally be assigned a cost, which is used to compute the total cost of a plan. Such an operator can be viewed as consisting of a precondition, a set of effects, and an operator cost.

A plan is a finite sequence of operators

$$\pi = \langle o_1, o_2, \dots, o_m \rangle$$

with $o_i \in O$ for all $i \in \{1, \dots, m\}$. The length of the plan is the number of operators in the sequence. If operators have costs, the cost of the plan is the sum of the costs of all operators

in the sequence.

Semantically, a plan describes how to transform the initial state step by step. Starting from the initial state I , the first operator o_1 is applied if its precondition is satisfied in I . This produces a successor state, in which the next operator o_2 is applied, and so on. A plan is valid if every operator in the sequence is applicable in the state reached before it is applied. A valid plan solves the planning task if the final state reached after applying all operators satisfies the goal condition γ .

Operator costs are not relevant for the delta-state representation considered in this thesis, because delta-states only depend on which variable values are changed by an operator, not on the cost of applying it.

2.1.3 Precondition of Operators and Goal Conditions

In SAS^+ , operator preconditions and goal conditions are conjunctions of atoms. An atom has the form $v = d$, where $v \in V$ is a variable and $d \in D_v$ is one of its possible values. Thus, a precondition may require several variables to have specific values, for example

$$v_1 = d_1 \wedge v_3 = d_3.$$

This means that the operator can only be applied in states where variable v_1 currently has value d_1 and variable v_3 currently has value d_3 . Since the two atoms are connected by a conjunction, both requirements must be satisfied at the same time. The values of variables that do not occur in the precondition are irrelevant for the applicability of this operator.

Semantically, an operator is applicable in a state s if all atoms in its precondition are satisfied by s . Similarly, the goal condition describes the requirements that a final state must satisfy. It is usually only a partial assignment: variables that do not occur in the goal condition can have arbitrary values. A state s is therefore called a goal state if every atom in γ is true in s .

2.1.4 Effects of Operators

The effects of an operator are conflict-free assignments of new values to variables. Formally, we can view the effect of an operator o as a partial function

$$\text{eff}_o : V \rightharpoonup \bigcup_{v \in V} D_v,$$

where $\text{eff}_o(v)$ is defined only for variables whose value is changed by the operator. Conflict-free means that each variable receives at most one new value.

Let s be a state and let o be an operator that is applicable in s . The successor state $s' = \text{succ}(s, o)$ is defined variable-wise by

$$s'[v] = \begin{cases} \text{eff}_o(v) & \text{if } \text{eff}_o(v) \text{ is defined,} \\ s[v] & \text{otherwise.} \end{cases}$$

Thus, applying an operator overwrites exactly the variables mentioned in its effect, while all other variable values are copied from the predecessor state.

Equivalently, an implementation of a function `get_succ(s, o)` first copies the current state s and then updates every variable v for which the operator has an effect:

$$s'[v] := \text{eff}_o(v).$$

If the operator is not applicable in s , then no successor state is generated.

This observation is directly useful for saving memory. If a complete state stores one value for each of the n variables, then storing a successor state normally requires space for all n variable values. However, if an operator changes only k variables, where usually $k \ll n$, then the difference between the predecessor and successor can be represented by only these k changed variable-value pairs together with a reference to the predecessor state. In this case, the delta-state avoids storing the unchanged part of the state again. This can be beneficial in state-space search, where many successor states are generated and stored, because most operators typically affect only a small part of the complete SAS^+ state.

2.2 Delta-States

Instead of storing every generated state as a complete vector, only the differences to a previously stored parent state are stored in a delta-state together with a reference to the aforementioned parent. Suppose an operator o is applied to a state s and produces a successor state s' . Since a SAS^+ operator changes only the variables that occur in its effects, the difference between s and s' is usually limited to a small subset of variables.

2.2.1 Mathematical Definition

Mathematically, the delta between s and s' can be described as $\Delta(s, s') = \{(v, s'[v]) \mid v \in V \text{ and } s[v] \neq s'[v]\}$.

2.2.2 Reconstructing Complete States from Delta States

A delta-state therefore consists of a reference to its parent state s together with the set of changed variable-value pairs $\Delta(s, s')$. The complete successor state s' can later be reconstructed by starting from the root state, which is not a delta-state but rather a complete state, and then applying the stored changes along the path to the delta-state. This subsection describes the idea only on a conceptual level; the concrete reconstruction algorithm used in the implementation is presented later in Section 5.1.2.

This reconstruction introduces additional runtime overhead, because accessing the full state requires following parent references and applying several deltas. In exchange, the representation can reduce memory consumption, since unchanged variable values do not have to be stored repeatedly for every generated successor state. This can allow the planner to keep more search states in memory, which is especially useful in large state spaces where memory is a limiting factor. By reducing the memory required per stored state, the planner will be able to explore a larger part of the state space before running out of memory. In practice, this can increase the chance of reaching a goal state within the available memory limit.

2.2.3 Example

Assume that a state consists of 10 SAS⁺ variables $V = \{v_0, v_1, \dots, v_9\}$. Each variable has the same finite domain $D_{v_i} = \{0, 1, \dots, 9\}$ for all $i \in \{0, \dots, 9\}$. Thus, each variable can take one of the values from 0 to 9. At this point in the thesis, the value domains are not important, but we already define them because they become relevant later when discussing the implementation. Assume that the current state is represented as $s = (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$. This notation means that v_0 has value 0, v_1 has value 1, and so on. Now suppose an operator o is applied whose precondition is $v_1 = 1 \wedge v_2 = 2$. This operator is applicable in the current state s , because $s[v_1] = 1$ and $s[v_2] = 2$. Assume that the operator has the single effect $\text{eff}_o(v_2) = 1$. According to the definition of successor states, the value of v_2 is overwritten by this effect, while all variables that do not occur in the effect keep their previous values. Therefore, the successor state is $s' = (0, 1, 1, 3, 4, 5, 6, 7, 8, 9)$.

All variable values remain unchanged except for the value of v_2 which changes from 2 to 1. Using zero-based indexing, this change occurs at position 2. Therefore, instead of storing the complete successor state $(0, 1, 1, 3, 4, 5, 6, 7, 8, 9)$, we can store a delta-state consisting of a reference to the parent state $(0, 1, 2, 3, 4, 5, 6, 7, 8, 9)$ and the single change $\langle 2, 1 \rangle$. Here, the first component denotes the affected variable position, and the second component denotes the new value assigned to that variable.

This representation can save memory when operators modify only a small number of variables compared to the total number of variables. If a full state contains n variable values, but an operator changes only k variables with $k \ll n$, then storing only the k changed values can be substantially smaller than storing the complete successor state. In SAS⁺, this optimization is particularly natural because operator effects explicitly specify which variables receive new values, making the delta directly derivable from the applied operator.

3

Existing State Storage

In this chapter, we examine how states are currently stored in Fast Downward. The goal is to describe the existing state representation before introducing the delta representation in the following chapters. This is necessary because the proposed approach changes how registered states are stored and accessed, while other parts of the search infrastructure still rely on concepts such as `StateID`s, duplicate detection, and state lookup.

Figure 3.1 shows a simplified UML representation of the `StateRegistry`. The diagram does not list every implementation detail of the class, but focuses on the fields that are relevant for the state storage mechanism discussed in this chapter. The highlighted fields correspond to the following subsections: `SegmentedArrayVector` is the container in which the packed state data is stored, `IntPacker` describes how variable values are packed into this representation, and `StateIDSet` is used to detect whether a generated state has already been registered. Together, these fields form the core of the current state storage mechanism that is relevant for this thesis.

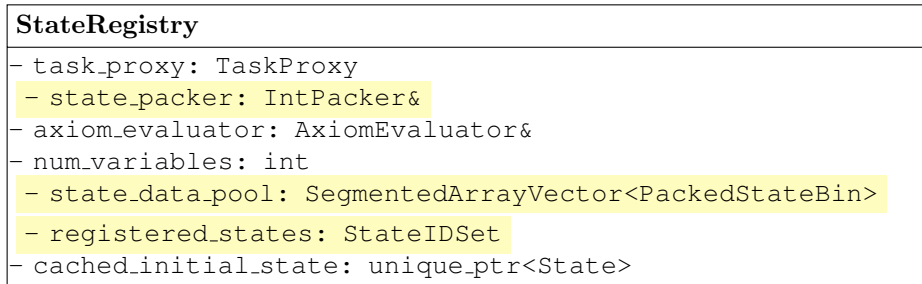


Figure 3.1: Simplified UML representation of the `StateRegistry`. The highlighted fields are discussed in the following subsections. All methods have been intentionally left out.

3.1 StateRegistry

The central component for storing states is the `StateRegistry`. It is responsible for creating, storing, and looking up states during the search. A state is called *registered* once it has been inserted into the registry and assigned a unique `StateID`. This identifier can

then be used to refer to the state without storing or copying the full state representation again. In contrast, an *unregistered* state is a temporary state object that has not yet been inserted into the registry. Such states occur when a successor state is generated from the current state. Before the search can use this successor as a registered state, the registry checks whether an equivalent state has already been stored. If it is new, the state is inserted and receives a new `StateID`; otherwise, the existing registered state and its `StateID` are reused, avoiding the storage of a duplicate state.

In contrast, unregistered states are temporary `State` objects that are not stored in the `StateRegistry`. In Fast Downward, such states store their values in an unpacked representation, which is held by a `std::shared_ptr` to a `std::vector`. Thus, while the `State` object itself may only be a temporary object, the vector containing the state values is dynamically allocated. However, this memory is not part of the long-term state storage in the registry and can be released once the temporary state is no longer used.

3.1.1 SegmentedArrayVector

As more states are registered, the memory usage of the `StateRegistry` grows accordingly. The long-term storage for registered states is allocated dynamically, i.e., on the heap, because the number of states that will be discovered during search is not known in advance and can become very large. Each newly registered state requires space for its packed state representation. These packed representations are stored in a `SegmentedArrayVector`. Unlike a regular contiguous vector, a `SegmentedArrayVector` stores its elements in several separate memory segments. This allows the storage to grow as new states are registered without moving all previously stored data to a new memory location. This property is important because registered states are referred to by their `StateID`, and the position of the packed data in the storage is derived from this identifier.

3.1.2 IntPacker

The stored state data consists of objects of type `PackedStateBin`. A `PackedStateBin` is an alias for `unsigned int`; therefore, each bin typically provides 32 bits of storage. Instead of storing one full integer for every planning variable, Fast Downward uses bitpacking: each variable value is represented only with as many bits as are necessary for its domain. For example, a binary variable can be represented with one bit, while a variable with four possible values needs two bits.

The `IntPacker` stores the information needed to pack and unpack these variable values. Based on the number of bits required for each variable, it knows in which `PackedStateBin` a variable is stored and at which bit offset within that bin its value begins. In this way, several variable values can be stored compactly in the same bin while still allowing each individual value to be retrieved when needed. The number of bins required for each state is fixed, because every state contains the same set of planning variables, and the domain size of each variable is known in advance.

This packed representation reduces the amount of memory required per state compared to

an unpacked representation such as a `std::vector<int>`. Nevertheless, each registered state still requires its own sequence of packed bins in the `SegmentedArrayVector`. Since the number of bins per state is fixed for a given planning task, the total memory required for registered state data grows linearly with the number of registered states.

3.1.3 Example

To illustrate the effect of the `IntPacker`, we consider a small concrete example using the SAS^+ notation introduced in Section 2.1. Let the set of variables be $V = \{a, b, c\}$. Assume that all three variables have the same domain $D_a = D_b = D_c = \{0, 1, \dots, 7\}$. Thus, each variable can take one of eight possible values.

Let the current state be $s = (6, 3, 2)$. This means that variable a has value 6, variable b has value 3, and variable c has value 2. In an unpacked representation, this state could be stored as a vector containing the three integer values $(6, 3, 2)$.

Before packing the state, the `IntPacker` knows how many bits are required for each variable. Since each variable has eight possible values, each value can be represented using $\log_2(8) = 3$ bits. Since the state contains three variables, at most $3 \cdot 3 = 9$ bits are needed to store the complete state. Therefore, the whole state fits into a single `PackedStateBin`, which typically provides 32 bits.

The `IntPacker` does not treat the state as one indivisible value. Instead, the state values are packed variable by variable. The `StateRegistry` iterates over the variables of the state and uses the `IntPacker` to insert each variable value into the packed representation. For each variable, the `IntPacker` already stores the information needed to determine where the corresponding value has to be placed, namely the target bin and the bit offset inside that bin. The same information is later used in the opposite direction when the state is unpacked. The packing of the state can then be described as follows. The value of variable a is 6, which is represented as 110 in binary. The value of variable b is 3, which is represented as 011 in binary. Finally, the value of variable c is 2, which is represented as 010 in binary. Written schematically in variable order, the packed bin therefore starts with these nine bits: 110011010. The remaining bits of the `PackedStateBin` are unused in this example. Thus, the bin can be illustrated as 11001101000000000000000000000000.

When the state is unpacked again, the `IntPacker` uses the stored packing information to recover the original variable values. Again, this happens variable by variable. For each variable, the `IntPacker` knows from which bin and from which bit offset the value has to be read. Therefore, it reads the first three bits, 110, and obtains the value 6 for variable a . It then reads the next three bits, 011, and obtains the value 3 for variable b . Finally, it reads the next three bits, 010, and obtains the value 2 for variable c . The unpacked result is again $(6, 3, 2)$, which represents the complete state.

3.1.4 StateIDSet

In addition to the packed representation of states, the implementation also uses a `StateIDSet`. This data structure supports duplicate detection in the `StateRegistry`. During search, newly generated states have to be compared with states that were already

registered. If an equivalent state already exists, the planner can reuse the existing `StateID` instead of storing the same state again.

The `StateIDSet` is implemented using an integer hash set. It does not store complete state objects directly. Instead, it stores integer keys, namely the `StateIDs` of registered states. The actual state data is stored separately in the `SegmentedArrayVector`, and the `StateID` identifies the position at which the corresponding packed state representation can be found.

Internally, the hash set stores its data in an array of buckets. Each bucket can contain at most one `StateID` together with its hash value. Collisions are handled by placing entries in nearby buckets using open addressing, not by storing multiple keys in one bucket [3, Section 11.4]. Therefore, the set does not compare states by the numeric value of the `StateID` alone. Instead, membership is determined semantically: two entries are considered equal if the packed state data associated with their `StateIDs` represents the same state.

For regular states, both hashing and equality testing can be performed directly on the packed state representation. The hash function obtains the packed data belonging to a `StateID` from the `SegmentedArrayVector` and computes a hash value from the corresponding `PackedStateBins`. Equality is checked by comparing the packed bins of the two states. Thus, the hash set stores compact integer identifiers, while duplicate detection is still based on the actual state contents.

When a new state is inserted, the hash set first computes the hash value for the corresponding `StateID`. This hash value determines the ideal bucket in the bucket array. If the ideal bucket is already occupied, the implementation uses open addressing and considers nearby buckets instead. The implementation uses ideas from hopscotch hashing to keep each key close to its ideal bucket, which helps preserve efficient lookup. Since each bucket stores one key and one hash value, the hash set adds auxiliary memory overhead in addition to the packed state data itself.

If the hash set finds an existing entry with the same hash value, it uses the semantic equality test to check whether the newly generated state is equivalent to the already registered one. If an equivalent state is found, the existing `StateID` is returned and no new state has to be stored. Otherwise, a new entry is inserted into the hash set, and the newly registered state receives its own `StateID`.

This is relevant for memory usage because the registry does not only store the packed states themselves. It also maintains auxiliary data structures such as the `StateIDSet` to make lookup and duplicate detection efficient. The memory required for registered states therefore consists of both the compact packed state representation in the `SegmentedArrayVector` and the additional hash-table structure needed to find and manage already registered states during search.

4

Delta-State Storage

StateRegistry
-delta_packer:DeltaPacker& -delta_state_data_pool:vector<DeltaStateInfo> -registered_delta_states:DeltaStateIDSet

Figure 4.1: UML representation of fields of the StateRegistry that are important for the delta-state implementation.

4.1 StateRegistry Extensions for delta-states

Figure 4.1 shows the fields of the StateRegistry that are important for the delta-state implementation. This section describes the additional components that were introduced into the StateRegistry to support delta-states. While regular states are stored as complete packed representations, delta-states require different data structures: one component for storing the delta information itself, one component for encoding and decoding the changed variables, and one component for duplicate detection. The following subsections explain these components and show how they correspond to the existing mechanisms for regular state storage.

4.1.1 DeltaStateInfo

For delta-states, the long-term state data is stored as objects of type DeltaStateInfo. These objects are kept in delta_state_data_pool, which is a `std::vector<DeltaStateInfo>`. Similar to the regular state data pool, this container grows as more states are registered. The position of an entry in the vector corresponds to the StateID of the registered delta-state. Therefore, a StateID can be used to access the corresponding DeltaStateInfo object.

Each DeltaStateInfo object stores only the information needed to reconstruct a state from another state. It contains a parent_state, which is the StateID of the predecessor state, and the effects that are bitpacked, which identify the changes that distinguish the state

from its predecessor. The `StateID` itself is represented by an integer. Figure 4.2 shows the UML representation of the `DeltaStateInfo` struct.

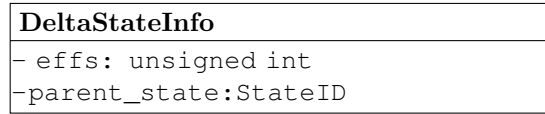


Figure 4.2: UML representation of the `DeltaStateInfo` struct.

Unlike the regular state data, which is stored in a `SegmentedArrayVector`, the `delta_state_data_pool` is implemented as a regular `std::vector`. This means that its elements are stored in one contiguous array, and the array may be reallocated when the vector grows. However, the logical mapping from `StateIDs` to vector positions remains unchanged: the delta information for the state with identifier (*i*) is stored at position (*i*) in the vector. As long as states are accessed through their `StateID` instead of through long-lived pointers to vector elements, this preserves the required mapping between registered states and their stored delta information.

The potential memory advantage of this representation comes from the fact that each registered delta-state does not store the complete values of all planning variables. Instead, it stores a reference to a predecessor state and a compact description of the changes. Therefore, the memory required per registered state can be smaller than in the regular representation, especially when full states require several packed bins (large states) but each transition changes only a small number of variables (small changes). However, in the current implementation on the platform used for the experiments, a delta-state has a minimum size of 8 bytes: 4 bytes for the `StateID` of the predecessor state and 4 bytes for the stored effects. In contrast, a regular state can require as little as 4 bytes if its complete packed representation fits into a single `PackedStateBin`, i.e., one `unsigned int`. These concrete byte values depend on the sizes of the underlying C++ integer types on the target platform. Nevertheless, the important point remains: a delta-state always has to store both a predecessor reference and the encoded effects, so it has a fixed overhead and is not automatically more memory-efficient.

4.1.2 DeltaPacker

The `DeltaPacker` is used to encode the changes between a state and its predecessor. While the regular `IntPacker` packs the complete values of all planning variables, the `DeltaPacker` only stores the variables that changed and their new values. Therefore, the packed delta representation consists of pairs of the form *position of changed variable* and *new value*.

This idea is similar to sparse vector representations. A dense vector stores an entry for every position, whereas a sparse vector stores only the positions whose values are relevant, usually the non-zero entries. Delta-states follow a similar principle, but the stored entries are not necessarily non-zero values. Instead, they are the variables whose values changed compared to the predecessor state. Thus, the delta representation can be seen as a sparse representation of the difference between two states: unchanged variables are omitted, while changed variables are stored together with their new values.

Similar to the regular state representation, the delta representation also uses bitpacking. For each changed variable, the `DeltaPacker` has to store two pieces of information: the position of the changed variable and the value assigned to that variable. Using the stored variable position, the `DeltaPacker` can access its internal information for the corresponding planning variable, including the variable's domain size. The number of bits needed for the variable position depends on the number of state variables there are in total, while the number of bits needed for the value depends on the domain size of the corresponding planning variable. For example, if a changed variable is binary, its new value can be stored using one bit. In a regular packed state, the values of all variables are stored in a fixed order. Therefore, the position of each value is already known and does not have to be stored separately. A delta-state only stores the variables that changed. For this reason, each stored value must also say which variable it belongs to, which is why we store the variable position.

The encoder constructs the packed representation of a list of effects. It first sorts the effects by the number of bits needed for their values, placing effects with larger value encodings first. The effects are then written into one or more bins of type `DeltaPacker::Bin`. The first bit of each bin is used as a continuation bit: if it is set to (1), the effect information continues in the next bin; if it is (0), the current bin is the last one. The remaining bits are used to store alternating entries for the changed variable and its new value. A variable position with value (0) marks the end of the encoded effects. Therefore, variable positions are stored with an offset of one: a variable with index (i) is encoded as ($i + 1$). As a result, the encoded range of possible variable positions is one larger than the actual number of state variables.

To read the encoded changes again, the decoder performs the inverse operation. Starting from the bin associated with a state, it reads the position of the changed variable with respect to its offset and then uses the domain information of that variable to determine how many bits have to be read for the corresponding value. This process continues until a bin is reached whose continuation bit is set to 0. In that bin, the decoder stops when it encounters the reserved variable position with value 0. The result is again a list of pairs describing which variables changed and which values they were changed to.

In contrast to the regular `IntPacker`, the number of bins required by the `DeltaPacker` is not fixed for every state. It depends on the number of changed variables and on the number of bits required to store their values. This is the main source of the potential memory advantage of delta-states: if only a few variables change between two consecutive states, the delta representation may require less memory than storing the complete packed state. However, this also means that accessing the full state is more expensive, because the complete vector of variable values has to be reconstructed from the predecessor state and the stored changes.

4.1.2.1 Example

To understand the `DeltaPacker` better, we consider a small example of encoding and decoding a delta-state, similar to the bitpacking example in Section 3.1.3. Assume that a state consists of seven SAS^+ variables $V = \{v_0, v_1, \dots, v_6\}$. Each variable has the same finite domain $D_{v_i} = \{0, 1, \dots, 7\}$ for all $i \in \{0, 1, \dots, 6\}$. Thus, each variable can take one of

eight possible values. Assume that the current state during search is $s = (2, 3, 5, 4, 0, 7, 5)$. The algorithm determines that this state satisfies the precondition of an operator o , for example $v_0 = 2$. Assume that this operator has the single effect $\text{eff}_o(v_0) = 1$. Applying o to s therefore produces the successor state $s' = (1, 3, 5, 4, 0, 7, 5)$.

For the delta-state corresponding to s' , we do not store the complete successor state. Instead, we store a reference to the parent state s , represented by its `StateID`, together with the single effect $\langle 0, 1 \rangle$ in a `DeltaStateInfo` object. This effect means that the variable at position 0, namely v_0 , is changed to the value 1.

During encoding, the first bit of the bin is reserved as the continuation bit. Since the effect fits into this bin and no additional bin is needed, this bit is set to 0. The actual effect information starts after this continuation bit.

The `DeltaPacker` first encodes the position of the changed variable. Since there are seven variables and the value 0 is reserved as an end marker, the stored variable positions are shifted by one. Thus, the actual variable position 0 is encoded as 1. This shifted position can be represented using three bits: 001.

After reading this shifted position, the `DeltaPacker` knows that the effect refers to variable v_0 . Since v_0 has eight possible values, its value also requires three bits. The new value is 1, which is represented as 001. Thus, the encoded information consists of the continuation bit, the shifted variable position, and the new value. In this example, the beginning of the bin is therefore 0001001. The remaining bits of the bin are unused. Written as a complete 32-bit bin, this can be illustrated as 00010010000000000000000000000000.

Only after encoding the effects can the `DeltaPacker` determine whether another bin is needed. This depends on the number of effects of the operator and on the number of bits required for the affected variables. In this example, one effect requires six data bits: three bits for the shifted variable position and three bits for the new value. Since one bin provides only 31 usable bits for effect data, because the first bit is reserved as the continuation bit, an operator with six such effects would require $6 \cdot 6 = 36$ bits. In that case, the effects would not fit into a single bin, and the continuation bit of the first bin would be set to 1 to indicate that the encoding continues in the next bin.

For decoding, the `DeltaPacker` receives the `StateID` of the delta-state that should be decoded. Using this `StateID`, it accesses the corresponding entry in the `delta_state_data_pool` and reads the `effs` field of the first `DeltaStateInfo` object. The first bit is the continuation bit. In this example, it is 0, so the decoder knows that the encoded effect information does not continue in another bin. The decoder then reads the next three bits as the shifted variable position. It obtains the value 001, which represents the shifted position 1. After subtracting the offset, this corresponds to the actual variable position 0, i.e., variable v_0 . The decoder then uses the domain information for v_0 to determine that the following value requires three bits. It reads the next three bits, again 001, and obtains the value 1. The decoder then attempts to read the next variable position. Since the next three bits are 000, which corresponds to the reserved end marker, it concludes that there are no further effects to decode. The decoded effect is therefore again $\langle 0, 1 \rangle$.

4.1.3 DeltaStateIDSet

For delta-states, the implementation uses a `DeltaStateIDSet`, which is implemented by the class `IntHashSetDelta`. Its purpose is analogous to the regular `StateIDSet`: it supports duplicate detection for registered states. When a newly generated delta-state is inserted into the registry, the planner has to check whether an equivalent state has already been registered. If such a state exists, the existing `StateID` can be reused instead of storing the same state again.

The main difference to the regular `StateIDSet` is how equality between states is checked. For regular states, the packed state data can be compared directly in the `SegmentedArray Vector`. For delta-states, however, the stored representation does not contain the complete state explicitly. Instead, a delta-state stores information such as a reference to a predecessor state and the changes relative to that predecessor. Therefore, `IntHashSetDelta` cannot simply compare the stored delta information directly to decide whether two states are semantically equal. To compare two delta-states, `IntHashSetDelta` looks up both states through the `StateRegistry` using their `StateIDs`. The states are then reconstructed and unpacked into their full vector representation using the algorithm explained in 5.1.2. After this reconstruction, the resulting vectors of variable values are compared. Thus, duplicate detection for delta-states is still based on the semantic state contents, but checking equality is more expensive than for regular packed states because it requires reconstructing complete states from their delta representation. This is necessary because two delta-states may store different changes and refer to different predecessor states, but still reconstruct to the same complete state.

The hashing behavior also differs from the regular `StateIDSet`. The regular `StateIDSet` owns a semantic hasher and computes the hash value internally when a key is inserted. In contrast, `IntHashSetDelta` does not compute the hash value itself in its public insertion method. Instead, the hash value is passed as an argument to the insertion method. The hash table then uses this externally provided hash value for bucket placement and only performs the expensive semantic equality check when another entry with the same hash is found.

Internally, `IntHashSetDelta` still follows the same general hash-table principle as the regular `IntHashSet`: it stores integer keys that correspond to `StateIDs`, and each bucket stores such a key together with its hash value. However, for delta-states it is especially important that the hash value is consistent with the reconstructed semantic state, not merely with the stored delta information. Otherwise, two delta-states that reconstruct to the same complete state but have different delta representations could receive different hash values and would not be detected as duplicates by the hash set.

A small example illustrates this problem. Assume that a task has two variables, v_0 and v_1 , and consider the complete state $s = (1, 1)$. This state can be reached in different ways. For example, one predecessor state could be $s_1 = (1, 0)$, with the stored delta effect $\langle v_1, 1 \rangle$. Another predecessor state could be $s_2 = (0, 1)$, with the stored delta effect $\langle v_0, 1 \rangle$. Both delta-states reconstruct to the same complete state $(1, 1)$, but their stored representations are different because they refer to different parent states and store different effects. Therefore, hashing only the stored delta information would be incorrect. The two representations would likely receive different hash values, even though they describe the same semantic state. For

duplicate detection, however, these two states must be treated as equal. This is why the hash value used by `IntHashSetDelta` has to be based on the reconstructed complete state rather than on the parent reference and the stored effects.

As more delta-states are registered, the number of entries in this hash set grows, and the bucket array may have to be resized. Therefore, the delta representation also requires auxiliary memory for duplicate detection. This memory is separate from the memory used for storing the delta information itself in `delta_state_data_pool`. As a result, the memory required for registered delta-states consists not only of the delta-state data, but also of the additional hash-table structure needed to find and manage already registered states. Since `DeltaStateIDSet` and `StateIDSet` follow the same general hash-table principle and store the same kind of information, namely `StateIDs` together with hash values, this auxiliary data structure does not by itself explain any memory reduction of the delta representation. Any memory savings must therefore come from the representation of the stored state data, not from the duplicate-detection set.

5

Important Algorithms

5.1 Important Algorithms for Using Delta-States

This chapter puts the previously introduced data structures into a broader algorithmic context. Until now, the delta-state representation has mainly been described in terms of how states are stored. However, to understand how this representation is used during search, it is also important to look at the algorithms that interact with it.

This section therefore presents two central algorithms. The first algorithm describes one step of the modified eager search, which we call the delta-state eager search. This gives an overview of the search control flow and shows where states are selected, expanded, and where successor states are generated. The second algorithm describes how a delta-state is reconstructed into its complete state representation. Together, these algorithms show both sides of the delta-state approach: how delta-states are created and used during search, and how the full state can be recovered when it is needed.

5.1.1 Delta-State Eager Search Algorithm

The delta-state eager search follows the same general control flow as eager search in Fast Downward, but successor states are created using the delta-state representation. The search itself is performed step by step. Each call to the step method either expands one search node, reports that a solution has been found, or reports that the reachable state space has been completely explored. The search algorithm repeatedly performs such steps until the returned status is no longer INPROGRESS.

Algorithm 1 shows the high-level structure of one search step. First, the algorithm calls `GETNEXTNODETOEXPAND`. This procedure removes entries from the open list until it finds a search node that is still valid for expansion. In the delta-state version, the state belonging to the selected `StateID` is looked up through the state registry using the delta-state lookup mechanism.

If no node can be selected, the open list is empty. In this case, all reachable states have been explored without finding a goal state, and the search terminates with failure. Otherwise, the selected node is passed to `EXPAND`. The expansion procedure first checks whether the node's state satisfies the goal condition. If this is the case, the plan is extracted and the search

terminates successfully. If the state is not a goal state, its successors are generated. This is the point where new successor states are created. In the modified search, these successors are registered as delta-states by storing the changes relative to their parent state instead of storing a complete state representation. The remaining node information is handled in the same way as in the original eager search algorithm.

Algorithm 1 One step of the delta-state eager search

Ensure: Search status

```

1: node ← GETNEXTNODETOEXPAND()
2: if node =  $\emptyset$  then
3:   assert that the open list is empty
4:   report that the state space has been completely explored
5:   return FAILED
6: end if
7: return EXPAND(node)

```

5.1.2 Reconstructing Complete States from Delta-States

As we now know, a delta-state does not store the complete values of all variables. To obtain the full state, the chain of predecessor states has to be followed until a non-delta-state is reached, which in our implementation is the initial state. During this traversal, the effects stored in the delta-states are collected. If the same variable is changed multiple times along the path, only the first encountered value is relevant, because the traversal starts at the state that should be reconstructed and then moves backwards to its predecessors.

The complete state must be reconstructed whenever an operation needs access to the actual values of the variables. One important example is the goal test during search, as mentioned in Section 5.1.1. To check whether a state satisfies the goal condition, the planner has to know the values of the variables that occur in the goal condition. Since these values may not be stored directly in the current delta-state, but instead may be inherited from one of its predecessor states, the full state representation has to be reconstructed before the goal condition can be evaluated.

The same applies to other operations that inspect the variable values of a state. For example, heuristic evaluation usually requires access to the complete state in order to estimate the distance to the goal. In addition, checking operator preconditions and generating successors require the planner to determine which operators are applicable in the current state. These operations therefore also need access to the full state representation, even though the state is stored internally as a delta-state.

This reconstruction introduces additional runtime overhead. The deeper a delta-state is in the predecessor chain, the more parent references may have to be followed before a complete base state is reached. Therefore, the time needed for reconstruction increases with the depth of the delta-state chain and with the number of effects that have to be processed along the way. This is the main trade-off of the delta-state representation: memory consumption can be reduced because unchanged variable values are not stored repeatedly, but accessing the complete state becomes more expensive.

Algorithm 2 summarizes the reconstruction procedure described above.

Algorithm 2 Reconstructing a full state from a delta-state

Require: delta-state s

Ensure: Complete vector of variable values for s

```

1:  $current \leftarrow s$ 
2:  $effects \leftarrow \emptyset$ 
3:  $seen \leftarrow \emptyset$ 
4: while  $current$  is a delta-state do
5:   for all effects  $(variable, value)$  stored in  $current$  do
6:     if  $variable \notin seen$  then
7:       add  $(variable, value)$  to  $effects$ 
8:       add  $variable$  to  $seen$ 
9:     end if
10:  end for
11:   $current \leftarrow parent(current)$ 
12: end while
13:  $values \leftarrow$  complete values stored in  $current$ 
14: for all  $(variable, value) \in effects$  do
15:    $values[variable] \leftarrow value$ 
16: end for
17: return  $values$ 

```

6

Experimental Results

In this chapter, we present and analyze the experimental results. The main goal is to compare the original eager search with the delta-state eager search.

6.1 Experimental Results and Analysis of Different Problems

This section describes the problems, configurations, and metrics used in the experiments. We first introduce PDDL, since PDDL files are the input format used for the planning tasks in Fast Downward. We then describe the two experimental runs. The first run uses two specifically designed problems that allow a detailed comparison of different transition characteristics. The second run evaluates the implementation on a broader set of benchmark instances. Both runs were executed using Downward Lab by Seipp et al. [11]. The results are analyzed at the end of the corresponding subsections.

6.1.1 PDDL

Although the state representation used in this thesis is described in terms of SAS⁺ variables and operators, planning tasks are usually given to Fast Downward in PDDL, the Planning Domain Definition Language [6]. PDDL is a symbolic input language for planning problems. A PDDL task is typically split into two files: a domain file and a problem file.

The domain file describes the general planning domain. It defines the predicates that can be used to describe states and the actions that can be applied. Each action has a precondition, which specifies when the action is applicable, and an effect, which describes how the state changes after applying the action. The problem file describes one concrete instance of this domain. It specifies the objects, the initial state, and the goal condition.

A very small example is a counting domain. The domain file could look as follows:

```
(define (domain counting)
  (:requirements :strips :equality)

  (:predicates
    (counted-1 ?x)
    (counted-2 ?x))
```

```

)

(:action count-1
  :parameters (?x)
  :precondition ()
  :effect (counted-1 ?x)
)

(:action count-2
  :parameters (?x)
  :precondition (counted-1 ?x)
  :effect (counted-2 ?x)
)
)

```

The corresponding problem file describes one concrete object and the goal of reaching counted-2 for this object:

```

(define (problem count-to-2)
  (:domain counting)

  (:objects
    obj1
  )

  (:init
  )

  (:goal
    (counted-2 obj1)
  )
)

```

In this example, the initial state is empty. The action count-1 has no precondition and can therefore be applied immediately to obj1. Its effect adds the fact (counted-1 obj1). After that, the action count-2 becomes applicable, because its precondition (counted-1 obj1) is satisfied. Applying count-2 adds the fact (counted-2 obj1), which satisfies the goal condition.

PDDL is closely related to STRIPS-style planning [4]. In a STRIPS-style task, states are described by sets of facts, and actions are described by preconditions and effects. The example above follows this idea: the state contains facts such as (counted-1 obj1) or (counted-2 obj1), and actions add new facts when their preconditions are satisfied.

Fast Downward does not perform the search directly on this symbolic PDDL or STRIPS-style representation. Instead, the translator component of Fast Downward converts the input task into an internal finite-domain representation based on SAS⁺. In this representation, states are represented as vectors of variable values, and operators change some of these values. The delta-state representation considered in this thesis is therefore not applied directly to

the textual PDDL input, but to the internal SAS⁺ state representation produced by the translation step.

6.1.2 First Run

6.1.2.1 Problems Used

For the first run, we used two problems with different characteristics. The purpose of this run was to compare the behavior of the original eager search and the delta-state eager search in settings where successor states differ from their parent states by different amounts.

The first problem was designed so that each operator changes only one variable. Therefore, the difference between a parent state and its successor state is small. We refer to this problem as the *low-change* problem. Conceptually, it follows the same idea as the counting example used in the PDDL subsection, but it contains more objects and variables. As a result, the generated states are larger, while each individual transition still changes only a small part of the state. The PDDL files for the low-change problem are shown in Listings 1 and 2.

The second problem was designed to have many changes between parent and successor states. We refer to this problem as the *high-change* problem. It is based on a modified instance of the Blocks World domain from the downward-benchmarks² collection. More precisely, we started from the existing `probBLOCKS-7-0.pddl` instance and added an additional bookkeeping variable. This variable does not change the intended structure of the planning task, but it increases the number of effects of the operators. As a result, applying an operator changes more variables, which makes the problem less favorable for the delta-state representation. The PDDL files for the high-change problem are shown in Listings 3 and 4. Both problems, low-change and high-change, were created specifically for this thesis.

Using these two problems allows us to compare two opposite cases: one in which delta-states should be beneficial because only a few values change, and one in which delta-states are expected to be less effective because many values have to be stored for each transition.

6.1.2.2 Heuristic and Search Configuration

For the low-change problem, we used the `hmax` heuristic introduced by Bonet and Geffner [2]. The corresponding search configuration was the following:

```
eager(tiebreaking([sum([g(), hmax()]), hmax()]), hmax()),
unsafe_pruning=false),
reopen_closed=true,
f_eval=sum([g(), hmax()])))
```

This configuration uses eager best-first search with an A*-style evaluation function, where the path cost so far is combined with a heuristic estimate [7]. If two states have the same primary value, the second criterion `hmax()` is used as a tie breaker. Thus, among states with the same combined value, the search prefers states with a smaller heuristic estimate.

The option `unsafe_pruning=false` means that unsafe pruning is disabled. Therefore, the search does not discard states using pruning rules that could remove states needed for

² <https://github.com/aibasael/downward-benchmarks>

finding a solution. The option `reopen_closed=true` allows the search to reopen a closed state if a cheaper path to that state is found later. Finally, the option `f_eval=sum([g(), hmax()])` specifies the evaluator used for reporting and tracking the current search progress. The same configuration was used for both the original eager search and the delta-state eager search. The only difference between the two runs was the search algorithm itself: one run used `eager`, while the other used `delta_eager`. This makes the comparison more meaningful, because differences in runtime or memory usage are then mainly caused by the different state representation rather than by different heuristic settings.

For the high-change problem, we used the `blind` heuristic. The corresponding search configuration was the following:

```
eager(tiebreaking([sum([g(), blind()]), blind()],
unsafe_pruning=false),
reopen_closed=true,
f_eval=sum([g(), blind()])))
```

This configuration follows the same structure as the one used for the low-change problem, but replaces `hmax` with the `blind` heuristic. The choice of heuristic was made separately for the two problems. For the low-change problem, the `blind` heuristic led to very long search times, so we used `hmax` instead. This reduced the runtime while still generating enough states for a meaningful comparison. Since the `blind` heuristic does not provide an informed estimate of the remaining distance to the goal, this configuration mainly orders states according to the accumulated path cost. The second criterion, `blind()`, is used as a tie breaker.

As in the low-change problem, the option `unsafe_pruning=false` disables unsafe pruning. Therefore, the search does not discard states using pruning rules that could remove states needed for finding a solution. The option `reopen_closed=true` allows the search to reopen a closed state if a cheaper path to that state is found later. Finally, `f_eval=sum([g(), blind()])` specifies the evaluator used for reporting and tracking search progress.

The same configuration was used for both the original eager search and the delta-state eager search. Again, the only difference between the two runs was the search algorithm itself: one run used `eager`, while the other used `delta_eager`. This ensures that the comparison focuses on the different state representations rather than on different heuristic or search settings.

6.1.2.3 Metrics Explained

Table 6.1 compares the original eager search implementation with the delta-state eager search implementation using several metrics. The goal of these metrics is to measure both the runtime impact and the memory impact of replacing the regular state representation with delta-states.

The column *Search time* reports the time needed by the search algorithm to solve the corresponding problem. This value is important because delta-states can introduce additional runtime overhead. Whenever the full state representation is needed, for example for evaluating heuristics, checking goal conditions, or generating successors, a delta-state may first have to

be reconstructed from its predecessor chain.

The column *Peak memory* reports the maximum memory usage of the planner process during the run. This value is an operating-system-level measurement. On Linux, peak memory is commonly reported as the maximum resident set size, i.e., the largest amount of physical memory that was assigned to the process at any point during execution. Therefore, this value should not be interpreted as only the memory used by the state registry. It also includes other memory used by Fast Downward, such as search nodes, open lists, heuristic data, duplicate-detection structures, temporary allocations, and general allocator overhead.

This metric is nevertheless important because it reflects the practical memory pressure of the whole planner run. If the operating system has to provide more resident memory to one implementation than to another, then that implementation is more memory demanding in practice.

The column *Bytes per State* gives a more focused view of the memory required for the state representation itself. For the original eager search, this value is determined by the number of `PackedStateBins` required for one complete state. Since each `PackedStateBin` is based on `unsigned int`, one bin typically requires four bytes. Thus, the value describes how many bytes are needed to store one packed complete state.

For the delta-state eager search, the value describes the average memory required for the stored delta representation of one state. This includes the information stored for a delta-state, such as the reference to the parent state and the encoded effects. In contrast to regular states, the size of a delta-state can depend on the number of changed variables, because more changes may require more encoded effect information.

The column *Registered States* reports how many states were inserted into the state registry during the search. This number is identical for both implementations in the table. This is important for the comparison, because it means that both implementations explored the same number of semantic states. Therefore, differences in memory usage are not caused by one implementation storing more states than the other, but by the different way in which these states are represented.

Finally, the column *Delta Size* describes how many SAS^+ variables actually change their value between a parent state and the corresponding successor state. This is related to the effects of the applied operator, but it is not necessarily identical to simply counting all effects syntactically. An effect only contributes to the delta-state representation if the value of the affected variable in the successor state differs from its value in the parent state.

For the low-change problem, each transition changes exactly one variable. For the high-change problem, the number of actual changes is given as a range, because different operators can lead to different numbers of changed SAS^+ variables. This metric is especially relevant for delta-states, since the potential memory advantage depends on how many variable-value pairs have to be stored in the delta state, rather than on the size of the complete state representation.

6.1.2.4 Results

To make the results reproducible, all experiments of the first run were executed on the same system, namely an HP Pavilion x360 Convertible running Ubuntu 24.04.4 LTS. The

exact size of `unsigned int` is implementation-dependent. On the platform used for the experiments, `sizeof(unsigned int)` is four bytes. Therefore, each `PackedStateBin` requires four bytes in our measurements. Table 6.1 shows the main results. For readability, we refer to the eager search variant using delta states as `delta_eager`.

Table 6.1: Comparison of eager search and `delta_eager` search. The number of registered states is identical for both implementations.

Problem	Search time [s]		Peak memory [KB]		Bytes per State		Registered States	Delta Size
	Eager	Delta_Eager	Eager	Delta_Eager	Eager	Delta_Eager		
low-change	3.42	252.64	30528	28044	16	8	430'032	1
high-change	7.28	573.49	24780	28956	8	20	84'137	10–11

6.1.2.5 Analysis

The results in Table 6.1 show that the effect of delta-states depends strongly on the structure of the problem. For the `low-change` problem, the delta-state eager search uses less memory than the original eager search. The bytes per state are reduced from 16 bytes to 8 bytes. This is expected, because the problem was designed such that each transition changes only one SAS^+ variable. Therefore, storing only the difference to the parent state is more compact than storing the complete state representation.

This reduction is also visible in the peak memory usage. For the `low-change` problem, peak memory decreases from 30528 KB for eager search to 28044 KB for `delta_eager`. This shows that delta-states can reduce the practical memory consumption of the planner when the complete state representation is larger than the minimum delta-state representation and when the number of changes per transition is small. In our implementation, a delta state requires at least 8 bytes, as explained in Subsection 4.1.1. However, the reduction in peak memory is smaller than the reduction in bytes per state. This is because peak memory measures the memory usage of the whole planner process, not only the memory used for storing registered states. Other data structures, such as search nodes, open lists, heuristic data, and duplicate-detection structures, also contribute to the total memory usage.

For the `high-change` problem, the situation is different. Here, the delta-state eager search requires more memory than the original eager search. The bytes per state increase from 8 bytes to 20 bytes. This is also expected, because the operators in this problem lead to many actual variable changes between parent and successor states. As a result, each delta-state has to store more effect information. In this case, storing the differences is no longer more compact than storing the complete packed state. The peak memory usage also increases from 24780 KB to 28956 KB, which confirms that the delta representation is not beneficial for this type of problem.

The number of registered states is identical for both implementations in each problem. This is important for the interpretation of the results, because it shows that the compared runs explored the same number of semantic states. Therefore, the differences in memory usage are not caused by one implementation generating or storing more states, but by the different internal state representations.

The search time increases substantially for the delta-state eager search in both problems.

For the `low-change` problem, the runtime increases from 3.42 seconds to 252.64 seconds, which is a slowdown of about 74 times. For the `high-change` problem, it increases from 7.28 seconds to 573.49 seconds, which is a slowdown of about 79 times. Thus, the runtime overhead is close to two orders of magnitude in both cases.

This shows that the memory savings of the delta-state representation come with a very large time cost in the current implementation. The slowdown is caused by the reconstruction overhead of delta states. Whenever the planner needs access to the complete state, it may have to follow parent references and apply the stored changes before the state can be used. Overall, the first run confirms the expected trade-off of the delta-state representation. Delta-states can reduce memory usage when only a few variables change between parent and successor states, as shown by the `low-change` problem. However, they become less effective or even worse when many variables change, as shown by the `high-change` problem. At the same time, both problems show a significant increase in runtime. Thus, the first run indicates that delta-states can be useful for memory reduction in suitable problem domains, but only if the additional reconstruction cost is reduced or controlled.

6.1.3 Problems Used for the Second Run

For the second run, we extended the evaluation from the two specifically designed problems of the first run to a larger set of benchmark instances. For this purpose, we selected several domains from the Fast Downward benchmark collection: `agricola-opt18-strips`, `agricola-sat18-strips`, `barman-mcol4-strips`, `blocks`, `elevators-opt08-strips`, `gripper`, and `logistics00`. The goal of this second run was to evaluate how the delta-state representation behaves on a broader collection of planning tasks, rather than only on the two artificial problems from the first run. The metrics used are the same as in the first run, except that the *Delta Size* and the number of registered states are not reported separately.

6.1.3.1 Heuristic and Search Configuration

For the second run, we used the `lmcut` heuristic introduced by Helmert and Domshlak [9] for all benchmark tasks. In contrast to the first run, where different heuristics were used for the `low-change` and `high-change` problems, the second run uses one fixed heuristic configuration for all tasks. This makes the results easier to compare across the selected benchmark problems.

The search configuration was generated by the experiment script and has the following structure:

```
let (h, lmcut()),
    let (f, sum([g(), h])),
    eager(tiebreaking([f, h], unsafe_pruning=false),
    reopen_closed=true,
    f_eval=f)))
```

For the delta-state eager search, the same configuration was used, except that `eager` was replaced by `delta_eager`:

```

let (h, lmcut()),
let (f, sum([g(), h])),
delta_eager(tiebreaking([f, h], unsafe_pruning=false),
reopen_closed=true,
f_eval=f)))

```

The overall time limit for each run was set to twenty minutes using the driver option `--overall-time-limit 20m`. The calculations were performed on sciCORE³ with up to six processes in parallel. For each benchmark task, two runs were created: one using the original Fast Downward implementation with `eager`, and one using the modified implementation with `delta_eager`. Apart from this difference in the search algorithm and the underlying state representation, both runs used the same task, heuristic, open-list configuration, reopening setting, and time limit. This allows the results to focus on the effect of the delta-state representation rather than on differences in the search configuration.

6.1.3.2 Results

Table 6.2 shows which runs were successful and which runs ended in a timeout because the available search time was not sufficient.

Domain	Timeout	Success
agricola-opt18-strips (20)	20	0
agricola-sat18-strips (20)	20	0
barman-mco14-strips (20)	20	0
blocks (35)	7	28
elevators-opt08-strips (30)	8	22
gripper (20)	13	7
logistics00 (28)	8	20
Sum (173)	96	77
Percentage (100%)	55.49%	44.51%

Table 6.2: Outcome of the second run on the selected benchmark domains. The outcome counts are identical for `eager-original` and `delta_eager`. The table layout is inspired by Gadzhiev [5].

Figure 6.1 gives a more detailed view of the memory behavior of the successful runs. For each solved problem instance, the figure shows the difference between the size of the regular packed state representation and the average size of the delta-state representation. The value on the y -axis is computed as bytes saved per state = bytes per state of `eager` – average delta-state memory of `delta_eager`. A positive value therefore means that the delta-state representation uses fewer bytes per state than the regular representation. A value of zero means that both representations require the same amount of memory per state, while a negative value means that the delta-state representation is larger. The problem instances are grouped by benchmark domain. The marker shape and color identify the domain, and the labels next to the points identify the concrete problem instances.

³ <http://scicore.unibas.ch/>

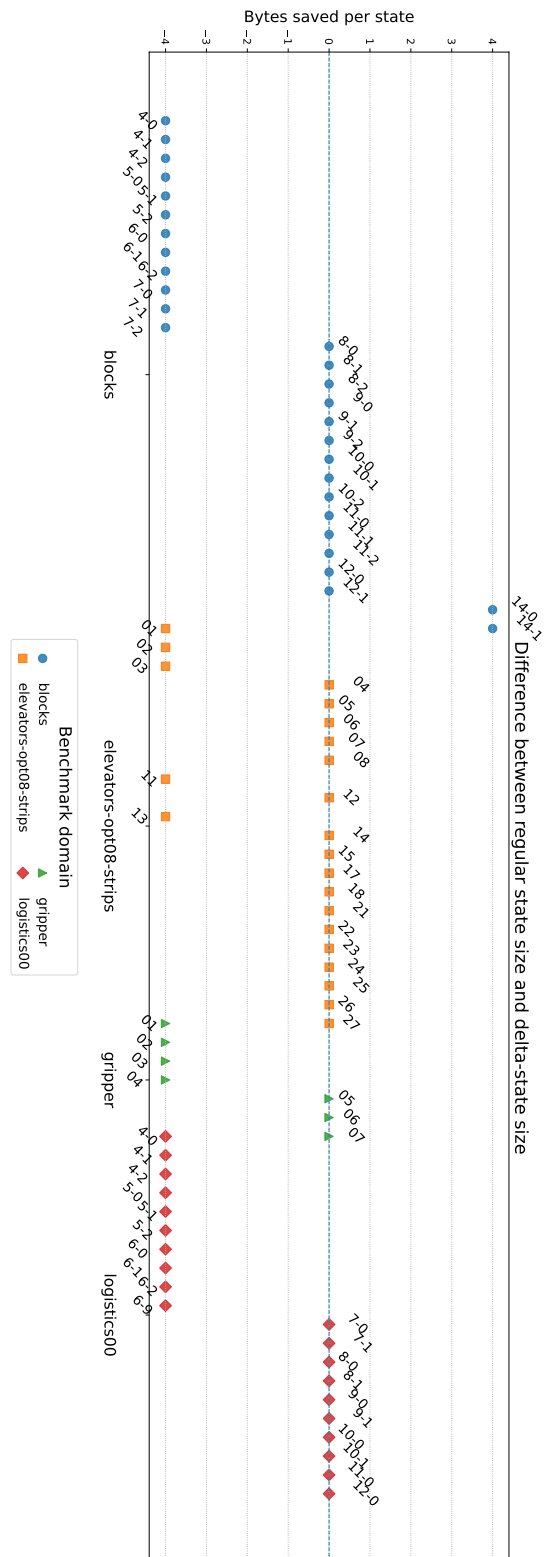


Figure 6.1: Difference between the regular packed state size and the average delta-state size in the second run using the `lmcut` heuristic. Each point represents one concrete problem instance. Positive values mean that `delta-eager` uses fewer bytes per state than the original eager search, while negative values mean that it uses more.

Figure 6.2 shows the same comparison from the perspective of the overall peak memory usage. While Figure 6.1 focuses only on the memory required for storing one state, Figure 6.2 considers the peak memory usage of the complete planner process. For each solved problem instance, the value on the y -axis is computed as $\text{peak memory saved} = \text{peak memory of eager} - \text{peak memory of delta.eager}$. A positive value therefore means that `delta.eager` used less peak memory than the original eager search. A value of zero means that both implementations used the same amount of peak memory, while a negative value means that `delta.eager` used more peak memory.

This figure is important because peak memory measures the practical memory consumption of the whole planner run. In contrast to the bytes-per-state metric, it does not only include the stored state representation. It also includes other data structures used during search, such as search nodes, open lists, heuristic data, duplicate-detection structures, temporary allocations, and allocator overhead. Therefore, the peak-memory difference does not always follow the bytes-per-state difference exactly. Even if the delta-state representation is smaller for individual states, the total peak memory can still be influenced by other parts of the planner.

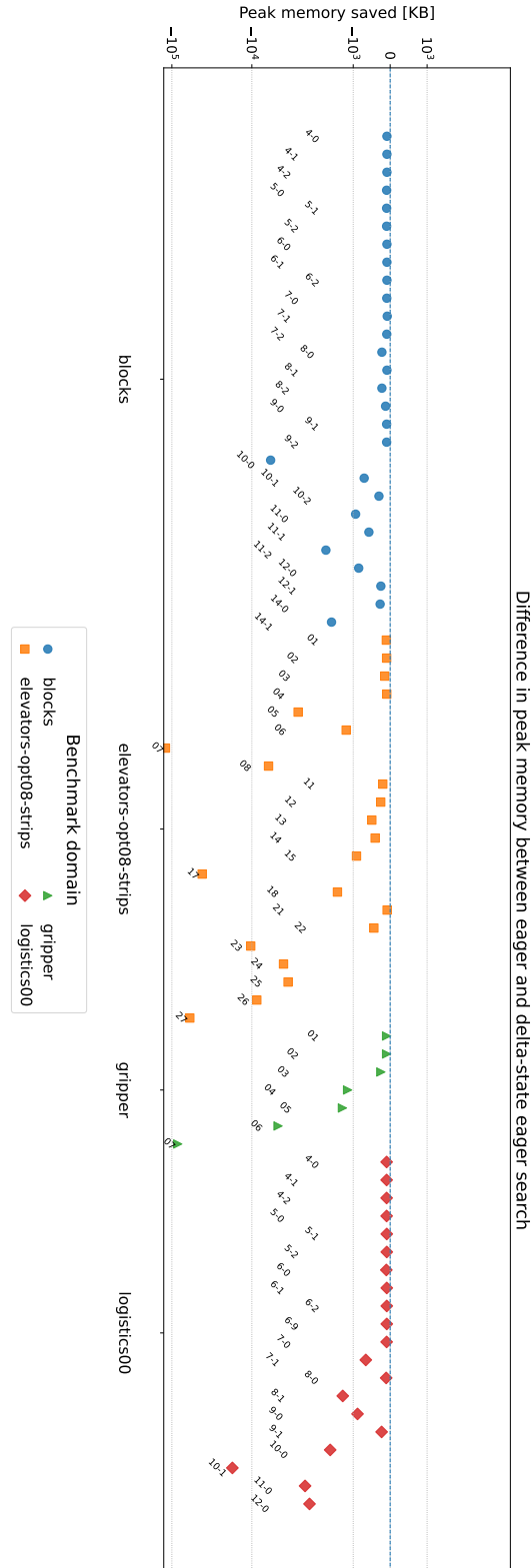


Figure 6.2: Difference in peak memory usage between the original eager search and the delta-state eager search in the second run using the `lmcut` heuristic. Each point represents one concrete problem instance. Positive values mean that `delta_eager` used less peak memory than the original eager search, while negative values mean that it used more. The y -axis uses a symmetric logarithmic scale.

Figure 6.3 shows the runtime behavior of the successful runs. Instead of plotting the absolute search time, the figure shows the search-time factor between the delta-state eager search and the original eager search. For each solved problem instance, the value on the y -axis is computed as

$$\text{search-time factor} = \frac{\text{search time of delta_eager}}{\text{search time of eager}}.$$

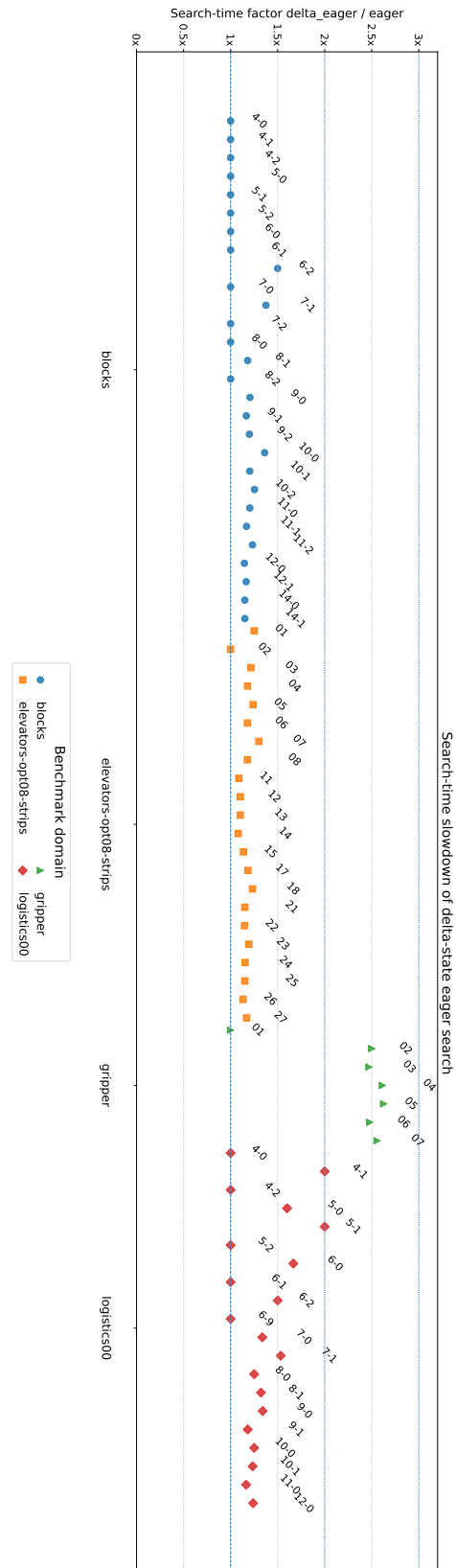


Figure 6.3: Search-time factor between the delta-state eager search and the original eager search in the second run using the `lmcut` heuristic. Each point represents one concrete problem instance. A value of 1 means that both implementations required the same search time. A value of 3 means that `delta_eager` took three times longer than the original eager search.

6.1.3.3 Analysis

As we can see in the results of Figure 6.1, only a small portion of the problems were able to save space using the delta-state representation. This can also be explained by the fact that larger complete states are better suited for saving memory, but these types of problems often need longer to be solved and could hence have been excluded by the timeout we implemented. For the memory saved we see in Figure 6.2 that delta-states never saved memory, but in fact were always worse than complete state representation. This can be explained by the fact that we used a vector to store the delta-states as explained in Subsection 4.1.1. This causes overallocation to happen, which we do not have for complete states since these are saved in a `SegmentedArrayVector`, which has less allocation overhead, as explained in Subsection 3.1.1. Also the rest of the memory needed does not differ, since the data structures all have been implemented very similarly, as can be seen in Chapters 3 and 4. Especially for the blocks problems 14-0 and 14-1 where we see we can save 4 bytes per state in Figure 6.1, we should also have less memory usage in total, which is not the case as shown in Figure 6.2. For the runtime we see exactly what we expected, namely that delta-states come with runtime overhead due to reconstruction needed, as shown in Figure 6.3.

7

Conclusion and Outlook

7.1 Conclusion

The goal of this thesis was to investigate whether memory usage in Fast Downward can be reduced by storing states as delta-states instead of always storing complete state representations. The main idea of this approach is to store only the differences between a state and its predecessor.

The results show that delta-states can be useful only under certain conditions. In particular, they are most promising for planning tasks with large state representations, for example tasks with many variables or variables with large domains, and with operators that affect only a small number of variables. In such cases, storing only the changed values can require less memory than storing the complete state vector for every registered state.

However, the experiments also show that the concrete implementation is crucial. Even if the delta-state representation reduces the amount of information stored per state, this does not automatically lead to lower overall memory usage. Additional overhead can be introduced by the chosen data structures. In the current implementation, the most important example is the use of a `std::vector` for storing the delta-state information. Since a vector may allocate more memory than is currently needed, unused capacity can increase the total memory consumption.

This effect was visible in the experimental results. In some problems, the delta-state representation required fewer bytes per state, but the measured peak memory usage of the whole planner did not improve. This shows that the conceptual advantage of delta-states can easily be lost if the implementation introduces too much additional memory overhead. The delta-state representation also introduces a runtime trade-off. While complete states can be accessed directly, delta-states often have to be reconstructed by following the chain of parent states and applying the stored changes. This makes operations such as goal testing, heuristic evaluation, and successor generation more expensive. Therefore, reducing memory usage with delta-states comes at the cost of potentially higher search time.

Overall, delta-states remain a promising idea for reducing memory usage in Fast Downward, but the implementation has to be carefully optimized. A compact representation of parent references and effects is necessary for the approach to become beneficial in practice. The results of this thesis therefore show both the potential of delta-states and the main challenges

that have to be addressed in future work.

7.2 Outlook

There are several possible directions for improving the current delta-state implementation. The first improvement concerns the layout of the stored delta-state data. In the current implementation, each `DeltaStateInfo` object stores both the encoded effects and a reference to the parent state. This is not optimal when the effects of a transition require more than one bin. In that case, additional bins are needed to store the encoded effects, but each additional `DeltaStateInfo` object still contains space for a parent `StateID`. This leads to redundant storage, because the parent state is the same for all bins belonging to one delta-state. A more compact representation would therefore separate the parent reference from the effect storage. The parent `StateID` would only have to be stored once per delta-state, while the encoded effects could be stored in a separate sequence of bins.

A second possible improvement concerns the container used for storing delta-state information. Currently, the `DeltaStateInfo` objects are stored in a `std::vector`. While this provides simple indexed access, a vector can allocate more memory than is currently needed because its capacity may be larger than its size. This unused capacity can contribute to additional memory overhead. A possible alternative would be to use a data structure similar to the `SegmentedArrayVector` used for regular packed states. Such a structure could allow the storage to grow in segments and may reduce the overhead caused by reallocations and unused capacity.

The experimental results show that these implementation details are important. Even though delta-states in some cases store less information than complete states, the current implementation can require more memory than the regular representation. This indicates that the potential memory savings of delta-states can only be achieved if the parent references and effects are stored compactly and with little per-state overhead.

A third improvement concerns the runtime of the delta-state eager search. Reconstructing a complete state from a delta-state can be expensive, because the algorithm has to follow the chain of parent states and apply the stored effects. One possible optimization would be to limit the maximum depth of such delta chains. If a fixed depth is exceeded, the next state could be stored as a regular complete state instead of as another delta-state. These complete states would then act as checkpoints. During reconstruction, the algorithm would only have to follow the chain back to the nearest checkpoint, rather than all the way back to the initial state.

Another possible runtime optimization would be to update the parent reference of a delta-state when a cheaper path to the same semantic state is found. A similar idea is already used for search nodes, where path information can be updated when a better path is discovered. Applying this idea to delta-states could shorten reconstruction chains in some cases, because a state could be linked to a parent that is closer or cheaper to reconstruct from. However, this would require careful handling to ensure that the stored delta information remains consistent with the new parent state.

Bibliography

- [1] Christer Bäckström and Bernhard Nebel. Complexity results for SAS+ planning. *Computational Intelligence*, 11(4):625–655, 1995.
- [2] Blai Bonet and Hector Geffner. Planning as heuristic search. *Artificial Intelligence*, 129(1–2):5–33, 2001. doi: 10.1016/S0004-3702(01)00108-4.
- [3] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 3 edition, 2009.
- [4] Richard E. Fikes and Nils J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3–4):189–208, 1971. doi: 10.1016/0004-3702(71)90010-5.
- [5] Khadzhimurad Gadzhiev. Computing planning task width via sat compilation. Bachelor’s thesis, University of Basel, June 2026. URL <https://ai.dmi.unibas.ch/papers/theses/gadzhiev-bachelor-26.pdf>.
- [6] Malik Ghallab, Adele Howe, Craig Knoblock, Drew McDermott, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL—the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, 1998.
- [7] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968. doi: 10.1109/TSSC.1968.300136.
- [8] Malte Helmert. The Fast Downward planning system. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.
- [9] Malte Helmert and Carmel Domshlak. Landmarks, critical paths and abstractions: What’s the difference anyway? In *Proceedings of the Nineteenth International Conference on Automated Planning and Scheduling (ICAPS 2009)*, pages 162–169. AAAI Press, 2009.
- [10] Khalid Sayood. *Introduction to Data Compression*. Morgan Kaufmann, 3 edition, 2006. ISBN 978-0-12-620862-7.
- [11] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward lab, 2017. URL <https://doi.org/10.5281/zenodo.399255>.



Appendix

.1 PDDL Files

.1.1 Low-Change Problem

```
1 (define (domain Artificial)
2   (:requirements :strips :equality :negative-preconditions)
3
4   (:predicates
5     (a ?x)
6     (b ?x)
7     (c ?x)
8     (d ?x)
9     (e ?x)
10    (f ?x)
11    (g ?x)
12    (h ?x)
13    (i ?x)
14    (j ?x)
15    (k ?x)
16    (l ?x)
17    (m ?x)
18    (n ?x)
19    (o ?x)
20    (p ?x)
21    (q ?x)
22    (r ?x)
23    (s ?x)
24    (t ?x)
25    (u ?x)
26    (v ?x)
27    (w ?x)
28    (x ?x)
29    (y ?x)
30    (z ?x)
31    (aa ?x)
32    (ab ?x)
33    (ac ?x)
34    (ad ?x)
```

```
35      (ae ?x)
36      (af ?x)
37      (ag ?x)
38    )
39
40    (:action true-a
41      :parameters (?x)
42      :precondition (not (a ?x))
43      :effect (a ?x)
44    )
45
46    (:action true-b
47      :parameters (?x)
48      :precondition (a ?x)
49      :effect (b ?x)
50    )
51
52    (:action true-c
53      :parameters (?x)
54      :precondition (b ?x)
55      :effect (c ?x)
56    )
57
58    (:action true-d
59      :parameters (?x)
60      :precondition (c ?x)
61      :effect (d ?x)
62    )
63
64    (:action true-e
65      :parameters (?x)
66      :precondition (d ?x)
67      :effect (e ?x)
68    )
69
70    (:action true-f
71      :parameters (?x)
72      :precondition (e ?x)
73      :effect (f ?x)
74    )
75
76    (:action true-g
77      :parameters (?x)
78      :precondition (f ?x)
79      :effect (g ?x)
80    )
81
82    (:action true-h
83      :parameters (?x)
84      :precondition (g ?x)
85      :effect (h ?x)
86    )
87
88    (:action true-i
```

```
89     :parameters (?x)
90     :precondition (h ?x)
91     :effect (i ?x)
92   )
93
94   (:action true-j
95     :parameters (?x)
96     :precondition (i ?x)
97     :effect (j ?x)
98   )
99
100  (:action true-k
101    :parameters (?x)
102    :precondition (j ?x)
103    :effect (k ?x)
104  )
105
106  (:action true-l
107    :parameters (?x)
108    :precondition (k ?x)
109    :effect (l ?x)
110  )
111
112  (:action true-m
113    :parameters (?x)
114    :precondition (l ?x)
115    :effect (m ?x)
116  )
117
118  (:action true-n
119    :parameters (?x)
120    :precondition (m ?x)
121    :effect (n ?x)
122  )
123
124  (:action true-o
125    :parameters (?x)
126    :precondition (n ?x)
127    :effect (o ?x)
128  )
129
130  (:action true-p
131    :parameters (?x)
132    :precondition (o ?x)
133    :effect (p ?x)
134  )
135
136  (:action true-q
137    :parameters (?x)
138    :precondition (p ?x)
139    :effect (q ?x)
140  )
141
142  (:action true-r
```

```
143     :parameters (?x)
144     :precondition (q ?x)
145     :effect (r ?x)
146   )
147
148   (:action true-s
149     :parameters (?x)
150     :precondition (r ?x)
151     :effect (s ?x)
152   )
153
154   (:action true-t
155     :parameters (?x)
156     :precondition (s ?x)
157     :effect (t ?x)
158   )
159
160   (:action true-u
161     :parameters (?x)
162     :precondition (t ?x)
163     :effect (u ?x)
164   )
165
166   (:action true-v
167     :parameters (?x)
168     :precondition (u ?x)
169     :effect (v ?x)
170   )
171
172   (:action true-w
173     :parameters (?x)
174     :precondition (v ?x)
175     :effect (w ?x)
176   )
177
178   (:action true-x
179     :parameters (?x)
180     :precondition (w ?x)
181     :effect (x ?x)
182   )
183
184   (:action true-y
185     :parameters (?x)
186     :precondition (x ?x)
187     :effect (y ?x)
188   )
189
190   (:action true-z
191     :parameters (?x)
192     :precondition (y ?x)
193     :effect (z ?x)
194   )
195
196   (:action true-aa
```

```

197     :parameters (?x)
198     :precondition (z ?x)
199     :effect (aa ?x)
200 )
201
202 (:action true-ab
203   :parameters (?x)
204   :precondition (aa ?x)
205   :effect (ab ?x)
206 )
207
208 (:action true-ac
209   :parameters (?x)
210   :precondition (ab ?x)
211   :effect (ac ?x)
212 )
213
214 (:action true-ad
215   :parameters (?x)
216   :precondition (ac ?x)
217   :effect (ad ?x)
218 )
219
220 (:action true-ae
221   :parameters (?x)
222   :precondition (ad ?x)
223   :effect (ae ?x)
224 )
225
226 (:action true-af
227   :parameters (?x)
228   :precondition (ae ?x)
229   :effect (af ?x)
230 )
231
232 (:action true-ag
233   :parameters (?x)
234   :precondition (af ?x)
235   :effect (ag ?x)
236 )
237 )

```

Listing 1: PDDL domain file for the low-change problem.

```

1 (define (problem Artificial-Problem-4)
2   (:domain Artificial)
3
4   (:objects
5     obj1 obj2 obj3 obj4
6   )
7
8   (:init
9   )
10

```

```

11 (:goal
12   (and (ag obj1) (ag obj2) (ad obj3) (k obj4))
13   )
14 )

```

Listing 2: PDDL problem file for the low-change problem.

.1.2 High-Change Problem

```

1  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
2  ;;; 4 Op-blocks world
3  ;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
4
5  (define (domain MYBLOCKS)
6    (:requirements :strips :equality)
7
8    (:predicates
9      (on ?x ?y)
10     (ontable ?x)
11     (clear ?x)
12     (handempty)
13     (holding ?x)
14     (bookkeepingvar ?x)
15   )
16
17   (:action pick-up
18     :parameters (?x ?y ?z ?a ?b ?c ?d)
19     :precondition (and
20       (clear ?x)
21       (ontable ?x)
22       (handempty)
23
24       (not (= ?x ?y))
25       (not (= ?x ?z))
26       (not (= ?x ?a))
27       (not (= ?x ?b))
28       (not (= ?x ?c))
29       (not (= ?x ?d))
30       (not (= ?y ?z))
31       (not (= ?y ?a))
32       (not (= ?y ?b))
33       (not (= ?y ?c))
34       (not (= ?y ?d))
35       (not (= ?z ?a))
36       (not (= ?z ?b))
37       (not (= ?z ?c))
38       (not (= ?z ?d))
39       (not (= ?a ?b))
40       (not (= ?a ?c))
41       (not (= ?a ?d))
42       (not (= ?b ?c))
43       (not (= ?b ?d))
44       (not (= ?c ?d))

```

```

45     )
46     :effect (and
47         (not (ontable ?x))
48         (not (clear ?x))
49         (not (handempty))
50         (holding ?x)
51         (bookkeepingvar ?x)
52         (not (bookkeepingvar ?y))
53         (not (bookkeepingvar ?z))
54         (not (bookkeepingvar ?a))
55         (not (bookkeepingvar ?b))
56         (not (bookkeepingvar ?c))
57         (not (bookkeepingvar ?d))
58     )
59 )
60
61 (:action put-down
62   :parameters (?x ?y ?z ?a ?b ?c ?d)
63   :precondition (and
64     (holding ?x)
65
66     (not (= ?x ?y))
67     (not (= ?x ?z))
68     (not (= ?x ?a))
69     (not (= ?x ?b))
70     (not (= ?x ?c))
71     (not (= ?x ?d))
72     (not (= ?y ?z))
73     (not (= ?y ?a))
74     (not (= ?y ?b))
75     (not (= ?y ?c))
76     (not (= ?y ?d))
77     (not (= ?z ?a))
78     (not (= ?z ?b))
79     (not (= ?z ?c))
80     (not (= ?z ?d))
81     (not (= ?a ?b))
82     (not (= ?a ?c))
83     (not (= ?a ?d))
84     (not (= ?b ?c))
85     (not (= ?b ?d))
86     (not (= ?c ?d))
87   )
88   :effect (and
89     (not (holding ?x))
90     (clear ?x)
91     (handempty)
92     (ontable ?x)
93     (not (bookkeepingvar ?x))
94     (bookkeepingvar ?y)
95     (bookkeepingvar ?z)
96     (bookkeepingvar ?a)
97     (bookkeepingvar ?b)
98     (bookkeepingvar ?c)

```

```

99      (bookkeepingvar ?d)
100    )
101  )
102
103  (:action stack
104    :parameters (?x ?y ?z ?a ?b ?c ?d)
105    :precondition (and
106      (holding ?x)
107      (clear ?y)
108
109      (not (= ?x ?y))
110      (not (= ?x ?z))
111      (not (= ?x ?a))
112      (not (= ?x ?b))
113      (not (= ?x ?c))
114      (not (= ?x ?d))
115      (not (= ?y ?z))
116      (not (= ?y ?a))
117      (not (= ?y ?b))
118      (not (= ?y ?c))
119      (not (= ?y ?d))
120      (not (= ?z ?a))
121      (not (= ?z ?b))
122      (not (= ?z ?c))
123      (not (= ?z ?d))
124      (not (= ?a ?b))
125      (not (= ?a ?c))
126      (not (= ?a ?d))
127      (not (= ?b ?c))
128      (not (= ?b ?d))
129      (not (= ?c ?d))
130    )
131    :effect (and
132      (not (holding ?x))
133      (not (clear ?y))
134      (clear ?x)
135      (handempty)
136      (on ?x ?y)
137      (not (bookkeepingvar ?x))
138      (bookkeepingvar ?y)
139      (bookkeepingvar ?z)
140      (bookkeepingvar ?a)
141      (bookkeepingvar ?b)
142      (bookkeepingvar ?c)
143      (bookkeepingvar ?d)
144    )
145  )
146
147  (:action unstack
148    :parameters (?x ?y ?z ?a ?b ?c ?d)
149    :precondition (and
150      (on ?x ?y)
151      (clear ?x)
152      (handempty)

```

```

153
154     (not (= ?x ?y))
155     (not (= ?x ?z))
156     (not (= ?x ?a))
157     (not (= ?x ?b))
158     (not (= ?x ?c))
159     (not (= ?x ?d))
160     (not (= ?y ?z))
161     (not (= ?y ?a))
162     (not (= ?y ?b))
163     (not (= ?y ?c))
164     (not (= ?y ?d))
165     (not (= ?z ?a))
166     (not (= ?z ?b))
167     (not (= ?z ?c))
168     (not (= ?z ?d))
169     (not (= ?a ?b))
170     (not (= ?a ?c))
171     (not (= ?a ?d))
172     (not (= ?b ?c))
173     (not (= ?b ?d))
174     (not (= ?c ?d))
175 )
176 :effect (and
177     (holding ?x)
178     (clear ?y)
179     (not (clear ?x))
180     (not (handempty))
181     (not (on ?x ?y))
182     (bookkeepingvar ?x)
183     (not (bookkeepingvar ?y))
184     (not (bookkeepingvar ?z))
185     (not (bookkeepingvar ?a))
186     (not (bookkeepingvar ?b))
187     (not (bookkeepingvar ?c))
188     (not (bookkeepingvar ?d))
189 )
190 )
191 )

```

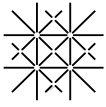
Listing 3: PDDL domain file for the high-change problem.

```

1 (define (problem MYBLOCKS-7-0)
2 (:domain MYBLOCKS)
3 (:objects C F A B G D E )
4 (:INIT (CLEAR E) (ONTABLE D) (ON E G) (ON G B) (ON B A) (ON A F) (ON F C)
5 (ON C D) (HANDEEMPTY))
6 (:goal (AND (ON A G) (ON G D) (ON D B) (ON B C) (ON C F) (ON F E) (NOT (
7     BOOKKEEPINGVAR A) (BOOKKEEPINGVAR B) (BOOKKEEPINGVAR C) (BOOKKEEPINGVAR D) (
8     BOOKKEEPINGVAR E) (BOOKKEEPINGVAR F) (BOOKKEEPINGVAR G)) )
9 )

```

Listing 4: PDDL problem file for the high-change problem.



Declaration on Scientific Integrity

(including a Declaration on Plagiarism and Fraud)

Translation from German original

Title of Thesis: Reducing Memory Usage in Fast Downward by Working with a Delta Representation of States

Name Assessor: _____

Name Student: Uka, Istref

Matriculation No.: 23-051-428

I attest with my signature that I have written this work independently and without outside help. I also attest that the information concerning the sources used in this work is true and complete in every respect. All sources that have been quoted or paraphrased have been marked accordingly.

Additionally, I affirm that any text passages written with the help of AI-supported technology are marked as such, including a reference to the AI-supported program used. This paper may be checked for plagiarism and use of AI-supported technology using the appropriate software. I understand that unethical conduct may lead to a grade of 1 or "fail" or expulsion from the study program.

Place, Date: Basel, 21.07.2026 Student: Istref Uka

Will this work, or parts of it, be published?

No

Yes. With my signature I confirm that I agree to a publication of the work (print/digital) in the library, on the research database of the University of Basel and/or on the document server of the department. Likewise, I agree to the bibliographic reference in the catalog SLSP (Swiss Library Service Platform). (cross out as applicable)

Publication as of: _____

Place, Date: Basel, 21.07.2026 Student: Istref Uka

Place, Date: _____ Assessor: _____

Please enclose a completed and signed copy of this declaration in your Bachelor's or Master's thesis.