University of Basel

Encoding Delete-Free Planning Tasks in Domain-Independent Dynamic Programming

Bachelor Thesis

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

Examiner: Prof. Malte Helmert
Supervisor: Florian Pommerening

Maria Desteffani
maria.desteffani@unibas.ch
2020-054-524

16. 02. 2025

Acknowledgments

Most of all, I would like to thank my supervisor Florian Pommerening for the great feedback and suggestions, and for always patiently trying to help me understand the topics I just could not grasp. I do not know how this thesis would have turned out without him. I would also like to thank Prof. Malte Helmert and the Artificial Intelligence Group for giving me the opportunity to write this thesis and dive into this topic. I have learned an incredible amount from writing this thesis, but also realized that I still have a long way to go. I would also like to thank my friends and family who never seemed to doubt me for a second. A special thank you also goes to all the people in the ZG at the Department, especially those who have spent these last few weeks there with me and have always shown me tremendous support, both in general and during some particularly stressful times.

Abstract

Automated planning is a branch of Artificial Intelligence that focuses on finding sequences of actions that achieve a given goal. Delete-free planning is a subset of automated planning which plays an important role in the computation of many heuristics. Domain-Independent Dynamic Programming (DIDP) is a framework for dynamic programming problems that provides a general approach to solving combinatorial optimization problems. It is particularly interesting because it allows for a separation of problem modeling and solving. In this thesis we explore the encoding of delete-free STRIPS planning problems within the DIDP framework using Dynamic Programming Description Language (DyPDL). We propose two different encodings. In the variable-to-variable encoding, each planning variable is mapped to an individual DyPDL variable. In contrast, the variable-to-set encoding uses a single DyPDL set variable to represent all planning variables. We also introduce several optimization techniques, including forcing transitions, filtering unproductive transitions and using heuristic functions to guide the search. For our experiments, we measured the number of completed runs, the number of runs terminated by memory errors or timeouts, the average memory usage, a time score and the average number of expanded nodes. We show that the performance of both encodings improved in all metrics when heuristics were added. However, the structural modifications resulted in no significant improvement and, in some cases, even a decline in performance. Exploring other variants of structural modifications, and especially the implementation of more powerful heuristics, is a promising topic for further research to help optimize the current models.

Table of Contents

1 Introduction

Automated Planning is a branch of Artificial Intelligence that focuses on reasoning about actions and their consequences to achieve a specific goal [4]. It is applied in various fields such as logistics, robotics, puzzle solving or space exploration. A planning task includes an initial state, a set of actions and one or more goal states. Finding a solution means determining a sequence of actions that ends in a goal state. This sequence of actions is called a *plan*. An example of a planning task is a robot vacuum cleaner that must clean a house by navigating through each rooms as efficiently as possible while avoiding obstacles. The available actions include moving forward, turning left or right, cleaning and stopping cleaning.
The planning formalism used in this thesis is *delete-free STRIPS*, a simplified version of the STRIPS formalism introduced by Fikes and Nilsson in 1971 [2]. In STRIPS, applying an action transforms the current problem state into a new one by processing add and delete effects. Add effects specify which facts are added to the current state and delete effects specify which facts are removed. For example, when our vacuum robot moves from the living room to the kitchen, it loses the fact that it is in the living room but gains the fact that it is now in the kitchen. As the name implies, delete-free STRIPS ignores delete effects. There are inherently delete-free problems, such as the minimal seed-set problem [3], where an organism accumulates nutrients by having them added to it or by synthesizing existing ones, never losing any previously acquired nutrients. While our vacuum robot problem is not inherently delete-free, the delete-free version of the problem can still be useful as many heuristics use the delete-free version of a planning task to estimate goal distance in the original problem.

Dynamic Programming (DP) is an optimization method in which a complex problem is decomposed into simpler subproblems [9]. The basic idea behind DP is to avoid redundant computations by reusing solutions found from solving previous subproblems. An example of a DP problem is the Fibonacci sequence, where each number is computed using the two preceding ones, avoiding the recalculation of already known values. *Domain-Independent Dynamic Programming (DIDP)*, as introduced by Kuroiwa and Beck [7], creates a general framework for optimization problems that allows for a separation of problem modeling and solving, whereas in the past DP problems historically were solved using problem-specific algorithms. The formalism we use to model DIDP problems in this thesis is the *Dynamic Programming Description Language* (*DyPDL*).

The goal of this thesis is to encode delete-free planning tasks within the DIDP framework, using delete-free STRIPS and DyPDL as formalisms. The encodings must ensure that the solution of the generated DyPDL task is equivalent to the solution of the original delete-free STRIPS task.

In the following chapters,we first introduce delete-free planning and DIDP in more detail and formally define the delete-free STRIPS and DyPDL formalisms. We then introduce two different encodings, as well as two structural modifications and two heuristics. Finally, we compare the encodings with and without the modifications and heuristics against each other to determine which improve performance and which do not.

2 Delete-Free Planning Tasks

In this chapter we introduce fundamental concepts of delete-free planning. We start by defining state spaces as a foundation for planning tasks. We then formally introduce delete-free STRIPS as a planning formalism and illustrate its use with an example. Finally, we also introduce heuristics, which play an important part in efficiently solving planning tasks.

A state space $\mathcal{S}$ is a directed, labeled, weighted graph. The definition of a state space is a tuple $\mathcal{S} = \langle S,\ L,\ cost,\ T,\ \alpha,\ \Gamma\rangle$, where:

- S is the set of possible states.
- L is the set of labels.
- $cost : L \mapsto \mathbb{N}_0$ assigns each label $l \in L$ a cost.
- $T \subseteq S \times L \times S$ is the deterministic transition relation. It contains tuples $\langle s,\ l,\ s'\rangle$ where $s,\ s' \in S$ are states and $l \in L$ is a label.
- α is the initial state.
- Γ is the set of goal states.

Each node in the graph correspond to a state and each edge represents a labeled transition with a weight given by the label's cost. A solution corresponds to a path in the graph, starting from the initial state α and ending in a goal state $\gamma \in \Gamma$. The sum of the cost of all traversed edges used is the cost of the path. An optimal solution is a path of minimal cost. If we have uniform cost, this corresponds to the shortest path in the graph. [11]

Here we introduce *delete-free STRIPS* as a description language for state spaces. A delete-free STRIPS planning task is defined as a tuple $\Pi = \langle V,\ I,\ G,\ A\rangle$ and contains the following [10]:

- A finite set of binary *state variables* V. States are represented as sets $s \subseteq V$.
- An *initial state* $I \subseteq V$.

- The *goals* $G \subseteq V$. For a state s to be a *goal state*, $G \subseteq s$ must hold.
- A finite set of *actions* A where each $a \in A$ is defined as $a = \langle pre(a),\ add(a),\ cost(a)\rangle$.
  - $pre(a) \subseteq V$: These are the *preconditions* of a and consist of a subset of variables of V. An action is *applicable* in a state s if $pre(a) \subseteq s$ holds. Otherwise the action is not applicable and can therefore not be applied.
  - $add(a) \subseteq V$: These are the *add effects* of a and are a subset of variables of V. Applying action a in a state s results in all variables of $add(a)$ getting added to s. After doing so, we reach the successor state of s, which we denote as $s[\![a]\!] = s \cup add(a)$.
  - $cost(a) \in \mathbb{N}_0$: the immediate *cost* of applying action a.

The state space induced by a delete-free STRIPS task $\Pi = \langle V,\ I,\ G,\ A\rangle$ is:

$$\mathcal{S}(\Pi) = \langle S_\Pi,\ L_\Pi,\ cost_\Pi,\ T_\Pi,\ \alpha_\Pi,\ \Gamma_\Pi\rangle \tag{2.1}$$

Each component is defined as follows:

- $S_\Pi = 2^V$ is the set of all possible states, each state s is represented by a subset of variables.
- $L_\Pi = A$ is the set of labels, corresponding to the set of actions A.
- $cost_\Pi : L \mapsto \mathbb{N}_0$ assigns a cost to each label such that $cost(l) = cost(a)$ for all $l = a \in A$.
- T_Π contains tuples $\langle s,\ a,\ s[\![a]\!]\rangle$ for all $s,\ s[\![a]\!] \in S_\Pi$ and $a \in A$ where a is applicable in s and $s[\![a]\!]$ is the successor state of s reached after applying a.
- $\alpha_\Pi = I$ is the initial state.
- $\Gamma_\Pi = \{\gamma \in S_\Pi \mid G \subseteq \gamma\}$ represents the goal states.

An example of a delete-free planning task is the *minimal seed-set problem* [3]. In the minimal seed-set problem we have an organism with a set of required nutrients C. We start off with no nutrients and can either add them for a cost or the organism can use chemical or metabolic reactions $r = (X,\ Y)$ to synthesize more nutrients for free. Here X is called the *substrate*, these are the nutrients which need to be present for the reaction to happen. Y is the *product set*, these are the nutrients which get produced by the reaction. Both are subsets of the required nutrients: $X,\ Y \subseteq C$. The set of all possible reactions is the *metabolic network* R. We further define a set of nutrients N as *reachable* from a subset Z of N if there is a finite sequence of reactions to synthesize all the missing nutrients of N. It is important to note here that nutrients do not get lost during reactions, each nutrient from the subset Z already counts towards completing N, we do not need to synthesize them too. This planning task is therefore inherently delete-free. A *seed-set* of a metabolic network is a subset of the nutrients C from which the entirety of C is reachable. The *minimal seed-set* is the minimal subset of C from which all nutrients C are reachable.

We can formulate the *minimal seed-set problem* as a delete-free STRIPS planning task $\Pi_{seed} = \langle V_{seed},\ I_{seed},\ G_{seed},\ A_{seed}\rangle$ where:

- $V_{seed} = C$, each nutrient is a variable $c \in C$.
- $I_{seed} = \emptyset$ as we start with no nutrients present.
- $G_{seed} = C$ as we want to accumulate all required nutrients C.
- For each $c \in C$ we have an action a_c and for each $r \in R$ an action a_r. Actions a_c add the nutrient c and actions a_r use reaction r. They are defined as:

$$\begin{aligned} pre(a_c) &= \emptyset & \qquad pre(a_r) &= X \\ add(a_c) &= \{c\} & \qquad add(a_r) &= Y \\ cost(a_c) &= 1 & \qquad cost(a_r) &= 0 \end{aligned}$$

A solution to a delete-free STRIPS task consists of a sequence of applicable actions, meaning each action must be applicable in the state it is applied in. Such a solution is also called a *plan*. The cost of a plan is the sum of the costs of all actions used. An optimal plan is a plan of minimal cost.

Lemma 1. *An optimal plan of a delete-free STRIPS task* Π *corresponds to an optimal solution in its induced state space* $\mathcal{S}(\Pi)$.

Each node in the state space $\mathcal{S}(\Pi)$ represents a state of the delete-free STRIPS task, and each edge corresponds to an action. All outgoing edges of a node represent the applicable actions in the state that corresponds to the node. The *initial state* I in Π directly corresponds to the *starting node* α_Π in $\mathcal{S}(\Pi)$, while the *goal states* G in Π correspond to the set of *goal nodes* Γ_Π in $\mathcal{S}(\Pi)$. Each action in Π has a cost that matches the weight of its corresponding edge in $\mathcal{S}(\Pi)$, and applying an action in Π is equivalent to traversing the corresponding edge in $\mathcal{S}(\Pi)$.
A plan for a delete-free STRIPS task Π is a sequence of actions leading from the initial state I to a goal state $g \in G$. In its induced state space $\mathcal{S}(\Pi)$, this corresponds precisely to a path from α_Π to a goal state $\gamma \in \Gamma_\Pi$. Since the cost of a plan is the sum of the costs of its actions, and the cost of a path in $\mathcal{S}(\Pi)$ is the sum of its edge weights, an optimal plan in Π corresponds directly to the cheapest path in $\mathcal{S}(\Pi)$.
Thus, finding the cheapest plan in the delete-free STRIPS task Π is equivalent to finding the cheapest path in its induced state space $\mathcal{S}(\Pi)$, and vice versa.

Finally, we will also introduce the concept of *heuristics* [11]. We define a *heuristic* h for a state space $\mathcal{S}$ with states $s \in S$ as a function:

$$h : s \mapsto \mathbb{R}_0^+ \cup \{\infty\} \tag{2.2}$$

A heuristic h maps each state to a nonnegative number or to infinity. The idea behind heuristics is to estimate the distance, i.e., the cheapest path, to the nearest goal. The goal of using heuristics is to help guide the search to focus on exploring promising paths.
A heuristic is called *admissible* if it never overestimates the true cost to the goal:

$$h(s) \leq h^*(s) \textit{ for all } s \in S$$

where h^* is the perfect heuristic which maps all states to the cost of the optimal solution or ∞ if there is none.

3 Dynamic Programming

In this chapter we introduce *Dynamic Programming (DP)* as a method for combinatorial optimization where a complex problem is decomposed into a set of smaller and easier subproblems. Solutions to these subproblems are computed and used to determine a solution to the original problem [9].
For illustration purposes we introduce the *min 0-1 knapsack problem.* Given are n items $X = \{x_0,\ x_1,\ ...,\ x_{n-1}\}$ with a *weight* $w_i \in \mathbb{R}_+$ and a *utility* $u_i \in \mathbb{R}_+$ for $i \in \{0,\ 1,\ ...,\ n-1\}$. We also have an empty bag and a *target utility* U_{target}. Our task is to fill the bag with items such that we reach or exceed the target utility while minimizing the weight. Each item can only be used once, hence the *0-1* in the problem name.

When solving a DP problem s we take *decisions* from a set of applicable decisions $\mathcal{D}(s)$[7]. Applying a decision d creates one or more subproblems $s[\![d]\!]$. Taking another decision based on one subproblem $s[\![d]\!]$ creates subproblems of the subproblem and so on. [9]

Each (sub-)problem s in a DP problem is assigned a value $J(s)$ using the *value function* J. The value of the original problem is the solution to the DP problem. For a minimization problem $J(s)$ is defined using a function F as [7]:

$$J(s) = \min_{d \in \mathcal{D}(s)} F(d,\ \{J(s') \mid s' \in s[\![d]\!]\}) \tag{3.1}$$

We can also look at maximization problems using *max* instead of *min*, but we will focus solely on minimization problems from here on out.

Transitions $\mathcal{T}$ are a special case of decisions where only one subproblem is generated [7]. The set of applicable transitions for a problem s are denoted as $\mathcal{T}(s)$. In the following we will only consider transitions. The value function for DP problems using transitions is:

$$J(s) = \min_{\tau \in \mathcal{T}(s)} F(\tau,\ J(s[\![\tau]\!])) \tag{3.2}$$

We define a variable $B \subseteq X$ in the min 0-1 knapsack problem which represents the already 'bagged' items and denote the current total utility in the bag as $U \in \mathbb{R}_+$. A (sub-)problem is represented by a tuple $s = \langle B,\ U\rangle$. We initially start with an empty bag, i.e., $s = \langle \emptyset,\ 0\rangle$

and can take a transition τ to add a specific item x_i into it, creating the subproblem $s[\![\tau]\!] = \langle B \cup \{x_i\}, v_i\rangle$. In the min 0-1 knapsack problem $J(s)$ describes the minimal weight needed to reach or exceed the target utility from state s. We define $J(s)$ as :

$$J(\langle B, U\rangle) = \begin{cases} 0 & \textit{if } U \geq U_{target}, \\ \infty & \textit{if } B = X \textit{ and } U < U_{target}, \\ \min\limits_{x_i \in X \setminus B} (J(B \cup \{x_i\}, U + u_i) + w_i) & \textit{otherwise.} \end{cases}$$

The recursive value function in the knapsack problem differentiates between three cases. First, if the target utility is reached the cost is zero as no other items need to be added. Second, if all items where added and the target utility is still not reached, $J(s)$ will be infinity. Lastly, if neither of this is the case, $J(s)$ will be recursively computed using the value of the successor state.

3.1 Domain-Independent Dynamic Programming

Historically, solving DP-problems includes using problem-specific algorithms. *Domain-Independent Dynamic Programming (DIDP)* [7] creates a more general framework for combinatorial optimization problems and can model a wide range of problems independently from the solver. This removes the need to tailor the algorithm to the problem and we can instead solve several or even all DIDP problems with the same algorithm.

The modeling formalism used in DIDP is *Dynamic Programming Description Language (DyPDL)* [7] and is, as mentioned before, independent of the final solver used.

A DyPDL model is a tuple $D = \langle \mathcal{V},\ s^0,\ \mathcal{K},\ \mathcal{T},\ \mathcal{B},\ \mathcal{C},\ h\rangle$ which consists of state variables $\mathcal{V}$, a target state s^0, a set of constants $\mathcal{K}$, a set of transitions $\mathcal{T}$, a set of base cases $\mathcal{B}$, a set of state constraints $\mathcal{C}$ and a dual bound h. We now describe the components in detail and create a DyPDL model for the min 0-1 knapsack problem $D_k = \langle \mathcal{V}_k,\ s^0_k,\ \mathcal{K}_k,\ \mathcal{T}_k,\ \mathcal{B}_k,\ \mathcal{C}_k,\ h_k\rangle$:

State Variables $\mathcal{V}$: State variables are used to describe and define *states*. A full value assignment to all state variables results in a *state s*. There are three different types of state variables:

- *Element variables* reference *objects*. DIDP assumes a fixed amount of objects which can then be referenced using their indices. The indices are assigned to element variables as nonnegative integers.
- *Set variables* are assigned sets of nonnegative integers which reference several objects.
- *Numerical variables* are assigned real numbers. A numerical variable can further be classified as a *resource variable* if it has a preference for a smaller or greater value.

In the min 0-1 knapsack problem B is a set variable and U is a numerical variable. A state s_k is described by a variable assignment to B and U.

Target State s^0: This is the target state of the problem. Solving a DIDP problem involves determining its value.
The target state of the knapsack problem is $s_k^0 = \langle \emptyset,\ 0 \rangle$, this is the state where the bag is empty.

Constants $\mathcal{K}$: *Constants* are state-independent values which never change. There are four types of constants:

- *Element constants* contain nonnegative integers and represent the index of an object.
- *Set constants* contain sets of nonnegative integers and represent the indices of several objects.
- *Numeric constants* contain real numbers.
- *Boolean constants* contain a boolean value.

It is also possible to have multidimensional *tables of constants*.
In the min 0-1 knapsack problem we have X as a set constant, U_{target} as a numerical constant and the multidimensional numerical constants w_i and u_i.

Expressions: To explain transitions, base cases, state constraints and the dual bound we first introduce expressions. State variables and constants as well as computations using them are called expressions. An expression e can be understood as a function which maps a state s to a value $e(s)$. Based on the value type of $e(s)$ we can differentiate between four different expressions:

- An *Element Expressions* maps to a nonnegative integer that represents the index of an object. Element expressions can consist of element variables and constants. We can also use arithmetic operations on them.
- A *Set Expression* maps to a set of nonnegative integers representing the indices of several objects. We can use set variables and constants to build set expressions and take the union, intersection or difference of several set expressions.
- A *Numeric Expression* maps to a real number. Numeric expressions can be built with numeric constants and numeric variables and we can use arithmetic operations. We can also take the cardinality of a set expression.
- *Conditions* map to a boolean value. They consist of boolean constants and comparisons of element, set or numeric expressions and can use set operations. We can further use the disjunction and conjunction of several conditions as a new condition. We can also use the indicator function $\mathbb{1}$ over a condition to get an integer value.

  For a condition c and a state s we say that state s *holds under* c if $c(s) = \top$, this can be denoted by $s \models c$. If $c(s) = \bot$, then c does not hold under s, denoted by $s \not\models c$. A state s holds under a set of conditions C if for all $c \in C$ we have $s \models c$. This can be denoted as $s \models C$. If there is a $c \in C$ for which $s \not\models c$ holds, then s does not hold under C, we denote this by $s \not\models C$.

Transition $\mathcal{T}$: *Transitions* define how one state evolves to another state. The DyPDL model only supports transitions, not decisions in general. A transition is defined as a tuple $\tau = \langle \mathit{eff}_\tau,\ \mathit{cost}_\tau,\ \mathit{pre}_\tau,\ \mathit{forced}_\tau \rangle$ containing *effects*, *preconditions*, *cost expressions* and a *forced transition* boolean value.

- eff_τ: Effects contain an expression e for each variable $v \in \mathcal{V}$. Applying a transition τ in a state s means to map each variable v to $s(e(v))$ using the corresponding expression defined in the effects. The variable values $s(e(v))$ for all $v \in \mathcal{V}$ represent their values in the successor state $s[\![\tau]\!]$.

- cost_τ: The cost expression for state s is a numerical expression defined by the function $F(\tau,\ J(s[\![\tau]\!])$ included in the value function $J(s)$ as defined in (3.2). It describes the computation of the value of state s using the current state as well as the value of a successor state $J(s[\![\tau]\!])$. From now on we will always assume that the costs defined by F are of the form $e_\tau + J(s[\![\tau]\!])$ where e_τ is an arbitrary numerical constant.

- pre_τ: Preconditions are a set of conditions C. A state s can apply transition τ if $s \models C$. In this case, τ is an *applicable* transition. The set of all applicable transitions in a state s is denoted by $s(\mathcal{T})$.

- forced_τ: Each transitions contains a boolean value which described whether a transition is forced or not. A *forced transition* must always take place if the preconditions $\mathit{pre}_\mathcal{T}$ are met and will be preferred over other non-forced transitions. If there are several forced transitions in $\mathcal{T}$ there must be a strict total order defined over them.

A transition $\tau_k \in \mathcal{T}_k$ of the min 0-1 knapsack problem has the following components:

- eff_{τ_k}: The set variable B gets updates by the expression $B \cup \{x_i\}$ and the numerical variable U by $U + u_i$.

- cost_{τ_k}: The cost expression of a transition τ_k is $w_i + J(s[\![\tau]\!])$.

- pre_{τ_k}: The preconditions for transition τ_k are $U < U_{target}$ and $B \neq X$.

- forced_τ: *False* as we do not have any forced transitions.

Base Cases $\mathcal{B}$: A *base case* is a tuple containing a set of conditions B and a numeric expression representing a cost associated with the base case. A *base state* is a state s where $s \models \beta$ for all conditions $\beta \in B$ and therefore $s \models B$. Base states are the most trivial subproblems of s and cannot be decomposed further.
In the min 0-1 Knapsack Problem we can define two base cases:

$$J(\langle B, U \rangle) = \begin{cases} 0, & \mathit{if}\, U \geq U_{target} \\ \infty, & \mathit{if}\, B = X\ \mathit{and}\, U < U_{target} \end{cases}$$

The base cases describe either reaching the target utility by the condition $U \geq U_{target}$ or adding all items and still not reaching target utility by $B = X$ and $U < U_{target}$.

State Constraints $\mathcal{C}$: *State constraint* are conditions which need to be satisfied by all states in the model. States which do not satisfy one or several constraints are discarded.
We add a simple state constraint to the min 0-1 knapsack problem. We take the first j items whose accumulated utility is equal or larger than U_{target}: $\sum_{i=0}^{j} u_i \geq U_{target}$. The sum of the weight of these items is $w_{max} = \sum_{i=0}^{j} w_i$. We now add a state constraint which states that we do not allow any state where there is an item in the bag whose weight is larger than w_{max}, so $w_{max} < w_k$ for $k \in \{j+1,\ j+2,\ ...,\ n-1\}$.

Dual Bound $\boldsymbol{h}$: *Dual Bounds* are optional lower or upper bounds for the value function J in a given state s. For minimization problems, lower bounds $h(s) \leq J(s)$ for all s are used and for maximization problems upper bounds $h(s) \geq J(s)$ for all s. Dual bounds are optional to include but can be helpful during solving.
In the min 0-1 Knapsack Problem we introduce a simple lower bound which estimates the proximity to the goal using the lowest weight of any item not yet added to the bag as long as the target utility U_{target} is not reached. So $h(s) = \min\limits_{x_i \in X \setminus B}(w_i)$.

3.2 DyPDL Problems as State Spaces

A DyPDL problem $D = \langle \mathcal{V},\ s^0,\ \mathcal{K},\ \mathcal{B},\ \mathcal{T},\ \mathcal{C},\ h\rangle$ induces a state space:

$$\mathcal{S}(D) = \langle S_D,\ L_D,\ cost_D,\ T_D,\ \alpha_D,\ \Gamma_D\rangle \tag{3.3}$$

The components are defined as follows:

- Each full variable assignment s_D of the set of variables $\mathcal{V}$ is a state. All possible states, i.e., all possible variable assignments build S_D.
- $L_D = \mathcal{T}$ is the set of labels.
- $cost_D : L_D \mapsto \mathbb{R}_+$ assigns a cost to each label. We always assume positive cost in this thesis. As previously defined it holds for all $\tau \in \mathcal{T}$ that $cost_\tau = e_\tau + J(s[\![\tau]\!])$. We define $cost_D(\tau) = cost_D(l) = e_\tau$ for all $\tau \in L_D$.
- T_D contains tuples $\langle s,\ l,\ s[\![l]\!]\rangle$ for all labels $l \in L_D$ and states $s,\ s[\![l]\!] \in S_D$ where it holds that $\tau = l$ is applicable, i.e., the preconditions of τ hold and $s[\![l]\!]$ is the successor state of s.
- $\alpha_D = s^0$ is the initial state
- Γ_D contains a goal state γ for every state s_D that is a base state.

Lemma 2. *Given a DIDP task $D = \langle \mathcal{V},\ s^0,\ \mathcal{K},\ \mathcal{T},\ \mathcal{B},\ \mathcal{C},\ h\rangle$ and its induced state space $\mathcal{S}(D) = \langle S_D,\ L_D,\ cost_D,\ T_D,\ \alpha_D,\ \Gamma_D\rangle$, then an optimal solution $J(s^0)$ of D is equivalent to the cost of an optimal solution in $\mathcal{S}(D)$.*

Each node in the induced state space $\mathcal{S}(D)$ corresponds to a unique state of the DyPDL task D, and each edge represents an applicable transition of that state. The edge weight in $\mathcal{S}(D)$

is determined by the immediate cost of a transition e_τ. Unlike in the DyPDL formulation, where the cost of a transition τ in a state s is recursively defined as $e_\tau + J(s[\![\tau]\!])$, the state space representation does not have to incorporate the values of successor states. Instead, the cost of a solution in $\mathcal{S}(D)$ is obtained by summing up edge weights.
A solution to the DyPDL problem is the minimal cost required to reach a base state starting from the target state. Computing this value involves finding a sequence of transitions of minimal cost leading from the initial state s^0 to a base state. In the corresponding state space $\mathcal{S}(D)$, this corresponds exactly to finding a path from α_D to a goal state $\gamma \in \Gamma_D$. The solution value $J(s^0)$ is given by the sum of transition costs used, which matches the sum of edge weights along the corresponding path in $\mathcal{S}(D)$.
Thus, finding an optimal solution to the DyPDL problem is equivalent to finding an optimal solution in its induced state space $\mathcal{S}(D)$ and vice versa.

4 Delete-free Planning Tasks in DyPDL

Delete-free planning tasks are well-suited to be expressed as DIDP problems using the DyPDL formalism as they already have some inherent structural similarities. Modeling them in DyPDL allows us to make use of the DIDP framework and therefore also using the corresponding solvers. In this chapter we define how we can convert delete-free STRIPS tasks into DyPDL models by defining the resulting DyPDL tuple using the original delete-free STRIPS tuple. We will define two different encodings. The rough outline of the first encoding is that we will represent the delete-free STRIPS variables V as DyPDL state variables $\mathcal{V}$, the initial state I as the target state s^0, the goals G as base cases $\mathcal{B}$ and actions A as transitions $\mathcal{T}$. The second encoding is similar, except that we only have a single set variable that will be representing all delete-free STRIPS variables. We will also illustrate these encodings using the *minimal seed-set problem*, which was introduced as a delete-free STRIPS planning task in chapter 2, and prove that solving the DyPDL problems is equivalent to solving the delete-free STRIPS problems. Finally, we will also present two structural modifications and two heuristics which may be used for optimization purposes. We evaluate their effectiveness in chapter 5.

4.1 Variable to Variable Encoding

Let $\Pi = \langle V,\ I,\ G,\ A\rangle$ be a delete-free STRIPS task. We define a function $\textit{d-state}_{VAR}$ which maps a delete-free STRIPS state s_Π to a DyPDL state s_{VAR}:

$$\textit{d-state}_{VAR}(s_\Pi)(v) = \begin{cases} 1 & \textit{if } v \in s_\Pi \\ 0 & \textit{otherwise} \end{cases} \tag{4.1}$$

Further, we define the DyPDL task $D_{VAR}(\Pi) = \langle \mathcal{V}_{VAR},\ s^0_{VAR},\ \mathcal{K}_{VAR},\ \mathcal{B}_{VAR},\ \mathcal{T}_{VAR},\ \mathcal{C}_{VAR},\ h_{VAR}\rangle$ corresponding to Π where:

- $\mathcal{V}_{VAR} = V$ where all $v \in \mathcal{V}_{VAR}$ are element variables.
- $s^0_{VAR} = \textit{d-state}_{VAR}(I)$
- $\mathcal{K}_{VAR} = \{0,\ 1\} \cup \{\textit{cost}(a) \mid a \in A\}$

- $\mathcal{B}_{VAR} = \{\langle b_G,\ 0\rangle\}$ with $b_G = \{(v = 1) \mid v \in G\}$
- $\mathcal{T}_{VAR} = \{\tau_a \mid a \in A\}$ where $\tau_a = \langle pre_{\tau_a},\ eff_{\tau_a},\ cost_{\tau_a},\ False\rangle$
  $pre_{\tau_a} = \{(p = 1) \mid p \in pre(a)\}$
  $eff_{\tau_a}(v) = \begin{cases} 1 & if\ v \in add(a), \\ v & otherwise \end{cases}$
  $cost_{\tau_a} = cost(a) + J(s[\![\tau_a]\!])$ where s is the state τ_a is applied in.

We do not have any state constraints $\mathcal{C}_{VAR}$ or dual bounds h_{VAR}. We will now explain this encoding and the function $d\text{-}state_{VAR}$ in more detail.

In this encoding, we encode each variable $v \in V$ as an element variable $v \in \mathcal{V}_{VAR}$. For function $d\text{-}state_{VAR}$ we need to take into account that a delete-free STRIPS state s_Π is represented as a set of variables $s_\Pi \subseteq V$. In $D_{VAR}(\Pi)$ however, we have an element variable $v \in \mathcal{V}_{VAR}$ for each $v \in V$ and a state is a full variable assignment of all state variables. We therefore model state s_Π by assigning 1 for all variables $v \in s_\Pi$ and assign 0 to all variables not included in s_Π.

Lemma 3. *The function $d\text{-}state_{VAR}$ is a bijection.*

Proof. Suppose $d\text{-}state_{VAR}$ is not injective, then there exist states s_1 and s_2 such that $s_1 \neq s_2$ but $d\text{-}state_{VAR}(s_1)(v) = d\text{-}state_{VAR}(s_2)(v)$ for all $v \in V$. From $s_1 \neq s_2$ it follows that there must be at least one variable v' in which they differ. Without loss of generality, assume that $v' \in s_1$ but $v' \notin s_2$. Therefore $d\text{-}state_{VAR}(s_1)(v') = 1$ and $d\text{-}state_{VAR}(s_2)(v') = 0$, which contradicts our assumption of $d\text{-}state_{VAR}(s_1)(v) = d\text{-}state_{VAR}(s_2)(v)$ for all $v \in V$. The function $d\text{-}state_{VAR}$ is therefore injective.
The function $d\text{-}state_{VAR}$ is surjective as for each DyPDL state s_{VAR} we can define the corresponding delete-free STRIPS state s_Π as $s_\Pi = \{v \in V \mid s_{VAR}(v) = 1\}$.
As the function $d\text{-}state_{VAR}$ is surjective and injective it follows that it is also bijective. □

We use $d\text{-}state_{VAR}$ to encode the initial state I as the target state s^0_{VAR}.

For our base cases $\mathcal{B}_{VAR}$ we need tuples containing conditions and values. In delete-free STRIPS we have goals G which consist of subsets of variables of V. These variables need to be included in a state s for s to be a goal state. We create a condition $(v = 1)$ for each $v \in G$ and use them as the conditions of our base case. The corresponding value is 0 since we have reached the end of recursion, i.e., there is nothing more to solve left, therefore our cost is 0.

To build the transitions $\mathcal{T}_{VAR}$ we add one transitions τ_a for each action $a \in A$. To define the specific transitions we need to define the tuple $\langle pre_{\tau_a},\ eff_{\tau_a},\ cost_{\tau_a},\ False\rangle$.
To define the preconditions pre_{τ_a} we need to specify a set of conditions. Looking at the preconditions $pre(a)$ of the corresponding action a we have a set of variables given. Similar to what we did for the base cases, we will now add a condition $(p = 1)$ for each $p \in pre(a)$. The set of all those conditions defines pre_{τ_a}.

For the effects eff_{τ_a} we need to specify expressions that update the state variables $v \in \mathcal{V}_{VAR}$ while the add effects $\mathit{add}(a)$ of an action specify which variables will be added to the state it was applied in. We can define an expression where we set each variable v to 1 if it is included in $\mathit{add}(a)$ or set it to its old value if it is not included in $\mathit{add}(a)$. We do not have to consider a case where a variable is being set to 0 since delete-free STRIPS tasks inherently do not allow this.
To define the cost expression cost_{τ_a} we need to define a numerical expression $e_{\tau_a} + J(s[\![\tau_a]\!])$ for a state s where e_{τ_a} is a numerical constant and $J(s[\![\tau_a]\!])$ is the successor state of state s. We can interpret e_{τ_a} as the immediate cost of applying transition τ_a, which corresponds to $\mathit{cost}(a)$ of the corresponding action a.
We do not want the transitions to be forced, therefore we set the boolean flag to *False*.

Finally we will also define the constants $\mathcal{K}_{VAR}$ by having a look at which constants were used in the previous definitions. The only constants used are 0 and 1 as well as the costs of actions $\mathit{cost}(a)$ for all $a \in A$.

As an example, we will take a concrete instance of the minimal seed-set problem Π_m and use the previously defined encoding to create the corresponding DyPDL task $D_{VAR}(\Pi_m)$. Our minimal seed-set problem instance is defined as a tuple

$$\Pi_m = \langle V_m,\ I_m,\ G_m,\ A_m \rangle \tag{4.2}$$

Where we have:

- we have four nutrients: $V_m = \{c_1,\ c_2,\ c_3,\ c_4\}$
- our initial state does not have any nutrients: $I_m = \emptyset$
- our goal is to accumulate all nutrients: $G_m = \{c_1,\ c_2,\ c_3,\ c_4\}$
- we have actions and reactions $A_m = \{a_{c_1},\ a_{c_2},\ a_{c_3},\ a_{c_4},\ a_{r_1},\ a_{r_2}\}$ consisting of actions a_{c_i} for $i \in \{1,\ 2,\ 3,\ 4\}$ to add a nutrient c_i:
  $\mathit{pre}(a_{c_i}) = \emptyset \quad \mathit{add}(a_{c_i}) = \{c_i\} \quad \mathit{cost}(a_{c_i}) = 1$
  and two reactions a_{r_1} and a_{r_2} defined as:
  $\mathit{pre}(a_{r_1}) = \{c_1,\ c_2\} \quad \mathit{add}(a_{r_1}) = \{c_3\} \quad \mathit{cost}(a_{r_1}) = 0$
  $\mathit{pre}(a_{r_2}) = \{c_3\} \quad \mathit{add}(a_{r_2}) = \{c_1,\ c_4\} \quad \mathit{cost}(a_{r_2}) = 0$

We now build the DyPDL task $D_{VAR}(\Pi_m) = \langle \mathcal{V}_m,\ s^0_m,\ \mathcal{K}_m,\ \mathcal{B}_m,\ \mathcal{T}_m,\ \mathcal{C}_m,\ h_m \rangle$:

- As specified in the encoding, we use the variables $V_m = C$ for our new state variables: $\mathcal{V}_m = V_m = C$ with variables $c \in C$.
- Our target state s^0_m corresponds to the initial state I_m where all nutrients are still missing:
  $s^0_m = \{c \mapsto 0 \mid c \in C\}$
- For our constants we have 0 and 1. We do not have to add any more since the costs of all our actions are always either 0 or 1 anyways:
  $\mathcal{K}_m = \{0,\ 1\}$

- We have one base case which is reached when all nutrients are present, i.e., all state variables are set to 1:
  $\mathcal{B}_m = \{\langle\{(c_1 = 1),\ (c_2 = 1),\ (c_3 = 1),\ (c_4 = 1)\},\ 0\rangle\}$
- For each $a \in A_m$ we get a transition $\tau \in \mathcal{T}_m$. We first map the actions of adding a nutrient to the transitions $\tau_{c_i} = \langle pre_{c_i},\ \mathit{eff}_{c_i},\ cost_{c_i},\ \mathit{False}\rangle$ for $i \in \{1,\ 2,\ 3,\ 4\}$ and define the preconditions, effects and cost expressions as:
  $pre_{c_i} = \emptyset$
  $\mathit{eff}_{c_i}(v) = \begin{cases} 1 & \mathit{if}\ v = c_i \\ v & \mathit{otherwise} \end{cases}$
  $cost_{c_i} = 1 + J(s[\![\tau]\!])$
  Further we define two transitions $\tau_{r_1} = \langle pre_{r_1},\ \mathit{eff}_{r_1},\ cost_{r_1},\ \mathit{False}\rangle$ and $\tau_{r_2} = \langle pre_{r_2},\ \mathit{eff}_{r_2},\ cost_{r_2},\ \mathit{False}\rangle$ which represent the reactions a_{r_1} and a_{r_2}:
  $pre_{r_1} = \{(c_1 = 1),\ (c_2 = 1)\}$ $\quad \mathit{eff}_{r_1}(v) = \begin{cases} 1 & \mathit{if}\ v = c_3 \\ v & \mathit{otherwise} \end{cases}$ $\quad cost_{r_1} = 0 + J(s[\![\tau_{r_1}]\!])$
  $pre_{r_2} = \{(c_3 = 1)\}$ $\quad \mathit{eff}_{r_2}(v) = \begin{cases} 1 & \mathit{if}\ v \in \{c_1,\ c_4\} \\ v & \mathit{otherwise} \end{cases}$ $\quad cost_{r_2} = 0 + J(s[\![\tau_{r_2}]\!])$

We do not have any state constraints and dual bounds. With this we have fully defined the DyPDL task $D_{VAR}(\Pi_m)$.

Finally, we prove that solving the DyPDL task $D_{VAR}(\Pi)$ is equivalent to solving the original delete-free STRIPS task Π.

Theorem 1. *Given a delete-free STRIPS task* $\Pi = \langle V_\Pi,\ I_\Pi,\ G_\Pi,\ A_\Pi\rangle$ *and a DyPDL task* $D_{VAR}(\Pi) = \langle \mathcal{V}_{VAR},\ s^0_{VAR},\ \mathcal{K}_{VAR},\ \mathcal{B}_{VAR},\ \mathcal{T}_{VAR},\ \mathcal{C}_{VAR},\ h_{VAR}\rangle$*, then an optimal solution of the DyPDL task* $D_{VAR}(\Pi)$*, i.e.,* $J(s^0_{VAR})$*, is equal to the cost of an optimal plan of the delete-free STRIPS task* Π.

Proof. In the following we will use the induced state space of the delete-free STRIPS task $\mathcal{S}(\Pi) = \langle S_{\mathcal{S}\Pi},\ L_{\mathcal{S}\Pi},\ cost_{\mathcal{S}\Pi},\ T_{\mathcal{S}\Pi},\ \alpha_{\mathcal{S}\Pi},\ \Gamma_{\mathcal{S}\Pi}\rangle$ introduced in (2.1) and the induced state space of the DyPDL task $\mathcal{S}(D_{VAR}(\Pi)) = \langle S_{\mathcal{S}VAR},\ L_{\mathcal{S}VAR},\ cost_{\mathcal{S}VAR},\ T_{\mathcal{S}VAR},\ \alpha_{\mathcal{S}VAR},\ \Gamma_{\mathcal{S}VAR}\rangle$ as defined in (3.3). We show that $\mathcal{S}(\Pi)$ is isomorphic to $\mathcal{S}(D_{VAR}(\Pi))$.

In Lemma 3 we established that the function $\mathit{d\text{-}state}_{VAR}$ which maps delete-free STRIPS states to DyPDL states is bijective. For readability we use $d = \mathit{d\text{-}state}_{VAR}$ in this proof. With d we have a valid bijection for all delete-free STRIPS and DyPDL states.
The initial states directly correspond to each other as by definition $\alpha_{\mathcal{S}VAR} = s^0_{VAR} = d(I_\Pi) = d(\alpha_{\mathcal{S}\Pi})$ holds. The same argumentation also holds for all other states, which is why we can conclude $\Gamma_{\mathcal{S}VAR} = \{d(\gamma) \mid \gamma \in \Gamma_{\mathcal{S}\Pi}\}$ for the goal states and $S_{\mathcal{S}VAR} = \{d(s) \mid s \in S_{\mathcal{S}\Pi}\}$ for states in general.

Next, we introduce a bijection $\lambda : A_\Pi \mapsto \mathcal{T}_{VAR}$ which maps an action $a \in A_\Pi$ to a transition $\tau_a \in \mathcal{T}_{VAR}$. It follows that $L_{\mathcal{S}VAR} = \{\tau_a \mid a \in A_\Pi\} = \{\lambda(a) \mid a \in A_\Pi\} = \{\lambda(l) \mid l \in L_{\mathcal{S}\Pi}\}$.

Further, we get a correspondence between the costs as it holds that $cost_{\mathcal{S}VAR}(\lambda(a)) = cost_{\mathcal{S}VAR}(\tau_a) = e_{\tau_a} = cost_{\Pi}(a) = cost_{\mathcal{S}\Pi}(a)$.

Finally, we need to ensure that $\langle d(s),\ \lambda(l),\ d(s[\![\lambda(l)]\!])\rangle \in T_{\mathcal{S}VAR}$ iff $\langle s,\ l,\ s[\![l]\!]\rangle \in T_{\mathcal{S}\Pi}$ holds.
We first show that $\lambda(l) = \lambda(a)$ is applicable in $d(s)$ iff $l = a$ is applicable in s by showing both directions:

1. If action a is applicable in s, $pre(a) \subseteq s$ holds. By definition $d(s)(v) = \begin{cases} 1 & \textit{if } v \in s \\ 0 & \textit{otherwise} \end{cases}$.
It follows that in state $d(s)$ it holds that $(p = 1)$ for all $p \in pre(a)$, which makes $\lambda(s) = \tau_a$ applicable in $s(d)$.
2. Transition $\lambda(a) = \tau_a$ is applicable in s, if $p = 1$ for all $p \in pre(a)$. Looking at bijection d this can only be the case if $p \in s$ for all $p \in pre(a)$, meaning $pre(a) \subseteq s$ and therefore making a applicable in s.

Next, we ensure that applying an action and then translating it is equal to translating it into a transition first and then applying it: $d(s[\![l]\!]) = d(s[\![a]\!]) = d(s)[\![\lambda(a)]\!] = d(s)[\![\lambda(l)]\!]$. We start with $d(s[\![a]\!])$. Using the definition of applying action a we get $d(s[\![a]\!]) = d(s \cup add(a))$ and by definition of d we end up with:

$$d(s[\![a]\!])(v) = \begin{cases} 1 & \textit{if } v \in (s \cup add(a)) \\ 0 & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if } v \in s \ \textit{or } v \in add(a) \\ 0 & \textit{otherwise} \end{cases}$$

We now consider $d(s)[\![\lambda(a)]\!]$. As $d(s[\![\lambda(a)]\!]) = d(s[\![\tau_a]\!])$, with the definition d and the definition of applying τ_a we get:

$$d(s)[\![\tau_a]\!](v) = \begin{cases} 1 & \textit{if } v \in add(a) \\ d(s)(v) & \textit{otherwise} \end{cases} = \begin{cases} 1 & \textit{if } v \in s \ \textit{or } v \in add(a) \\ 0 & \textit{otherwise} \end{cases}$$

Therefore $d(s[\![a]\!]) = d(s)[\![\lambda(a)]\!]$.

It follows that the delete-free STRIPS state space $\mathcal{S}(\Pi)$ is isomorphic to the the DyPDL state space $\mathcal{S}(D_{VAR}(\Pi))$.

An optimal plan to a delete-free STRIPS problem Π corresponds to an optimal solution in its induced state space $\mathcal{S}(\Pi)$, as stated in Lemma 1. As we have shown that the state space graphs of Π and $D_{VAR}(\Pi)$ are isomorphic, this is also an optimal solution in the induced state space of the DyPDL problem $\mathcal{S}(D_{VAR}(\Pi))$. Using Lemma 2 we follow that this also represents an optimal solution in $D_{VAR}(\Pi)$. □

4.2 Variable to Set Encoding

Let $\Pi = \langle V,\ I,\ G,\ A\rangle$ be a delete-free STRIPS task. We define a trivially bijective function $\textit{d-state}_{SET}$ which maps a delete-free STRIPS state s_Π to a DyPDL state s_{SET}:

$$\textit{d-state}_{SET}(s) = \{\mathcal{X} \mapsto s\} \tag{4.3}$$

We define a DyPDL task $D_{SET}(\Pi) = \langle \mathcal{V}_{SET},\ s^0_{SET},\ \mathcal{K}_{SET},\ \mathcal{B}_{SET},\ \mathcal{T}_{SET},\ \mathcal{C}_{SET},\ h_{SET}\rangle$ corresponding to Π where:

- $\mathcal{V}_{SET} = \{\mathcal{X}\}$ where $\mathcal{X}$ is a set variable.
- $s^0_{SET} = \textit{d-state}_{VAR}(I)$
- $\mathcal{K}_{SET} = \{\textit{cost}(a) \mid a \in A\} \cup \{\textit{pre}(a) \mid a \in A\} \cup \{\textit{add}(a) \mid a \in A\} \cup G$
- $\mathcal{B}_{SET} = \{\langle b_G,\ 0\rangle\}$ with $b_G = \{(G \subseteq \mathcal{X})\}$
- $\mathcal{T}_{SET} = \{\tau_a \mid a \in A\}$ where $\tau_a = \langle \textit{pre}_{\tau_a},\ \textit{eff}_{\tau_a},\ \textit{cost}_{\tau_a},\ \textit{False}\rangle$ with:
  $\textit{pre}_{\tau_a} = \{(\textit{pre}(a) \subseteq \mathcal{X})\}$
  $\textit{eff}_{\tau_a}(\mathcal{X}) = \mathcal{X} \cup \textit{add}(a)$
  $\textit{cost}_{\tau_a} = \textit{cost}(a) + J(s[\![\tau_a]\!])$ where s is the state τ_a is applied in.

We do not have state constraints or dual bounds. We will now explain this encoding and the function $\textit{d-state}_{SET}$ in more detail.

In this encoding we use a single set variable to represent all delete-free STRIPS variables. In function $\textit{d-state}_{SET}$ we specify how a delete-free STRIPS state is being encoded as a DyPDL state. Similar to how we represent delete-free STRIPS states, we also represent states as a set. We use the set variable $\mathcal{X}$ which saves the indices of all variables $v \in s_\Pi$ of a delete-free STRIPS state s_Π for this purpose. In general, this encoding is very straightforward due to this similarity.

We use the function $\textit{d-state}_{SET}$ to encode the initial state I as the target state s^0_{SET}.

We introduce a base case with value 0 which occurs when the goals G are a subset or equal to the set variable $\mathcal{X}$.

For the transitions $\mathcal{T}_{SET}$ we add one transition $\tau_a = \langle \textit{pre}_{\tau_a},\ \textit{eff}_{\tau_a},\ \textit{cost}_{\tau_a},\ \textit{False}\rangle$ for each action $a \in A$. The new preconditions $\textit{pre}_{\tau_a}$ contains one conditions stating that $\textit{pre}(a)$ needs to be a subset or equal to $\mathcal{X}$.
For the effects we need to define expressions to update the state variables. In delete-free STRIPS, $\textit{add}(a)$ defines all variables which are added to state s when applying a in s. Similarly, we add all variables from $\textit{add}(a)$ to the set variable $\mathcal{X}$ in $\textit{eff}_{\tau_a}$.
The cost of a transition is defined the same way as in the variable-to-variable encoding.

Finally, we define the constants $\mathcal{K}_{SET}$. Looking at all previous definitions, we see that we need the costs, preconditions and add effects as well as the goals of Π as constants.

As an example, we take the instance of the minimal-seed set problem Π_m defined in (4.2) and create the DyPDL task $D_{SET}(\Pi_m) = \langle \mathcal{V}_m,\ s^0_m,\ \mathcal{K}_m,\ \mathcal{B}_m,\ \mathcal{T}_m,\ \mathcal{C}_m,\ h_m \rangle$ which looks as follows:

- We have a set variable which will store the nutrients:
  $\mathcal{V}_m = \{\mathcal{X}\}$
- In the initial state, we do not have any nutrients yet:
  $s^0_m = \{\mathcal{X} \mapsto \emptyset\}$
- The constants consist of all possible action costs, preconditions and effects as well as the goal:
  $\mathcal{K}_m = \{0,\ 1,\ \{c_1,\ c_2\},\ \{c_3\},\ \{c_1,\ c_4\},\ \{c_1,\ c_2,\ c_3,\ c_4\}\}$
- We have one base case which is fulfilled when all nutrients are contained in $\mathcal{X}$:
  $\mathcal{B}_m = \{\langle \{c_1,\ c_2,\ c_3,\ c_4\} \subseteq \mathcal{X},\ 0 \rangle\}$
- We introduce a transition for each action. Adding a nutrient results in the transitions $\tau_{c_i} = \langle \mathit{pre}_{c_i}, \mathit{eff}_{c_i}, \mathit{cost}_{c_i},\ \mathit{False} \rangle$ for $i \in \{1,\ 2,\ 3,\ 4\}$ where:
  $\mathit{pre}_{c_i} = \emptyset$
  $\mathit{eff}_{c_i}(\mathcal{X}) = \mathcal{X} \cup \{c_i\}$
  $\mathit{cost}_{c_i} = 1 + J(s[\![\tau_{c_i}]\!])$ where s is the state τ_{c_i} is applied in
  Next we define a transition $\tau_{r_1} = \langle \mathit{pre}_{r_1},\ \mathit{eff}_{r_1},\ \mathit{cost}_{r_1},\ \mathit{False} \rangle$ and a transition $\tau_{r_2} = \langle \mathit{pre}_{r_2},\ \mathit{eff}_{r_2},\ \mathit{cost}_{r_2},\ \mathit{False} \rangle$ which represent the reactions a_{r_1} and a_{r_2}:
  $\mathit{pre}_{r_1} = \{\{c_1,\ c_2\} \subseteq \mathcal{X}\}$ $\mathit{eff}_{r_1}(\mathcal{X}) = \mathcal{X} \cup \{c_3\}$ $\mathit{cost}_{r_1} = 0 + J(s[\![\tau_{r_1}]\!])$
  $\mathit{pre}_{r_2} = \{\{c_3\} \subseteq \mathcal{X})\}$ $\mathit{eff}_{r_2}(\mathcal{X}) = \mathcal{X} \cup \{c_1,\ c_4\}$ $\mathit{cost}_{r_2} = 0 + J(s[\![\tau_{r_2}]\!])$
  Here s is the state the transition is applied in.

Finally, we prove that solving the DyPDL task $D_{SET}(\Pi)$ is equivalent to solving the original delete-free STRIPS task Π.

Theorem 2. *Given a delete-free STRIPS task* $\Pi = \langle V_\Pi,\ I_\Pi,\ G_\Pi,\ A_\Pi \rangle$ *and a DyPDL task* $D_{SET}(\Pi) = \langle \mathcal{V}_{SET},\ s^0_{SET},\ \mathcal{K}_{SET},\ \mathcal{B}_{SET},\ \mathcal{T}_{SET},\ \mathcal{C}_{SET},\ h_{SET} \rangle$*, then an optimal plan of the DyPDL task* $D_{SET}(\Pi)$*, i.e.,* $J(s^0_{SET})$*, is equal to an optimal solution of the delete-free STRIPS task* Π*.*

Proof. In the following we will use the induced state spae of the delete-free STRIPS task $\mathcal{S}(\Pi) = \langle S_{\mathcal{S}\Pi},\ L_{\mathcal{S}\Pi},\ \mathit{cost}_{\mathcal{S}\Pi},\ T_{\mathcal{S}\Pi},\ \alpha_{\mathcal{S}\Pi},\ \Gamma_{\mathcal{S}\Pi} \rangle$ as introduced in (2.1) and the induced state space of the DyPDL task $\mathcal{S}(D_{SET}(\Pi)) = \langle S_{\mathcal{S}SET},\ L_{\mathcal{S}SET},\ \mathit{cost}_{\mathcal{S}SET},\ T_{\mathcal{S}SET},\ \alpha_{\mathcal{S}SET},\ \Gamma_{\mathcal{S}SET} \rangle$ as defined in (3.3).

This proof proceeds in the same way as the proof for Theorem 1, for a more detailed explanation of certain steps therefore please refer to the previous proof.

We already defined a bijective function $\textit{d-state}_{SET}$ in (4.3) which maps delete-free STRIPS states to DyPDL states. We use $d = \textit{d-state}_{SET}$ for readability in this proof. It holds that

$S_{\mathcal{S}SET} = \{d(s) \mid s \in S_{\mathcal{S}\Pi}\}$.
Further, by definition $\alpha_{\mathcal{S}SET} = d(\alpha_{\mathcal{S}\Pi})$ and $\Gamma_{\mathcal{S}SET} = \{d(\gamma) \mid \gamma \in \Gamma_{\mathcal{S}\Pi}\}$ hold.

We introduce a bijection $\lambda : A_\Pi \mapsto \mathcal{T}_{SET}$ which maps an action $a \in A_\Pi$ to a transitions $\tau_a \in \mathcal{T}_{SET}$. Therefore $L_{\mathcal{S}SET} = \{\lambda(l) \mid l \in L_{\mathcal{S}\Pi}\}$.

Further, it holds that $cost_{\mathcal{S}SET} = cost_{\mathcal{S}\Pi}$.

Finally, we need to ensure that $\langle d(s),\ \lambda(l),\ d(s[\![\lambda(l)]\!])\rangle \in T_{\mathcal{S}SET}$ iff $\langle s,\ l,\ s[\![l]\!]\rangle \in T_{\mathcal{S}\Pi}$. We show that $l = a$ is applicable in s iff $\lambda(l) = \lambda(a)$ is applicable in $d(s)$:

1. If a is applicable in s, then $pre(a) \subseteq s$ holds. As $d(s) = \{\mathcal{X} \mapsto s\}$, it also holds that $pre(a) \subseteq \mathcal{X}$, which makes $\tau_a = \lambda(a)$ applicable in $d(s)$.
2. If $\lambda(a) = \tau_a$ is applicable in $d(s)$, then $pre(a) \subseteq \mathcal{X}$ holds. Looking at d, this is only possible if $pre(a) \subseteq s$, which means a is also applicable in s.

Next, we show that $d(s[\![l]\!]) = d(s[\![a]\!]) = d(s)[\![\lambda(a)]\!] = d(s)[\![\lambda(l)]\!]$ holds. By the definition of applying an action and then the definition of translating we get $d(s[\![a]\!]) = d(s \cup add(a)) = \{\mathcal{X} \mapsto (s \cup add(a))\}$. When translating first and then applying the transition we get by definition: $d(s)[\![\lambda(a)]\!] = d(s)[\![\tau_a]\!] = \{\mathcal{X} \mapsto (s \cup add(a))\}$. Therefore $d(s[\![a]\!]) = d(s)[\![\lambda(a)]\!]$.

It follows that the delete-free STRIPS state space $\mathcal{S}(\Pi)$ is isomorphic to the the DyPDL state space $\mathcal{S}(D_{SET}(\Pi))$.

As defined in Lemma 1, an optimal plan to a delete-free STRIPS problem Π corresponds to an optimal solution in its induced state space $\mathcal{S}(\Pi)$. As we have shown that the state space graphs of Π and $D_{SET}(\Pi)$ are isomorphic, this is also an optimal solution in the induced state space of the DyPDL problem $\mathcal{S}(D_{SET}(\Pi))$. Using Lemma 2 we follow that this also represents an optimal solution in $D_{SET}(\Pi)$. □

4.3 Forcing Transitions

For our first modification we introduce a new mechanic where as soon as a transition is applicable it is forced to either be applied or ignored. Afterwards, a transition is marked as considered and cannot be forced again. We first formally define the modification and then explain it in more detail.
If we have a delete-free STRIPS task $\Pi = \langle V,\ I,\ G,\ A\rangle$ and one of the previously established encodings $D_\pi = \langle \mathcal{V}_\pi,\ s^0_\pi,\ \mathcal{K}_\pi,\ \mathcal{B}_\pi,\ \mathcal{T}_\pi,\ \mathcal{C}_\pi,\ h_\pi\rangle$ then we get a modified encoding
$D_{\pi_M} = \langle \mathcal{V}_{\pi_M},\ s^0_{\pi_M},\ \mathcal{K}_\pi,\ \mathcal{B}_{\pi_M},\ \mathcal{T}_{\pi_M},\ \mathcal{C}_\pi,\ h_\pi\rangle$ with:

- $\mathcal{V}_{\pi_M} = \mathcal{V}_\pi \cup \{\mathcal{A},\ f\}$ where $\mathcal{A}$ is a set variable and f an element variable.
- $s^0_{\pi_M} = s^0_\pi \cup \{\mathcal{A} \mapsto \emptyset,\ f \mapsto -1\}$.
- $\mathcal{T}_{\pi_M} = \{f_a \mid a \in A\}\ \cup\ \{t_a \mid a \in A\}\ \cup\ \{i_a \mid a \in A\}$ where:
  $f_a = \langle pre_{f_a},\ \mathit{eff}_{f_a},\ cost_{f_a},\ True\rangle$ with:

- $pre_{f_a} = (a \notin \mathcal{A}) \cup pre_{\tau_a}$
- $\mathit{eff}_{f_a}(f) = a \qquad \mathit{eff}_{f_a}(\mathcal{A}) = \mathcal{A} \qquad \mathit{eff}_{f_a}(v) = v$ for all $v \in \mathcal{V}_\pi$.
- $cost_{f_a} = 0$

$t_a = \langle pre_{t_a},\ \mathit{eff}_{t_a},\ cost_{\tau_a},\ \mathit{False}\rangle$ with:

- $pre_{t_a} = \{(f = a)\}$
- $\mathit{eff}_{t_a}(v) = \mathit{eff}_{\tau_a}$ for all $v \in \mathcal{V}_\pi \quad \mathit{eff}_{t_a}(\mathcal{A}) = (\mathcal{A} \cup \{a\}) \quad \mathit{eff}_{t_a}(f) = -1$

$i_a = \langle pre_{i_a},\ \mathit{eff}_{i_a},\ cost_{i_a},\ \mathit{False}\rangle$ with:

- $pre_{i_a} = \{(f = a)\}$
- $\mathit{eff}_{i_a}(v) = v$ for all $v \in \mathcal{V}_\pi \qquad \mathit{eff}_{i_a}(\mathcal{A}) = \mathcal{A} \cup \{a\} \qquad \mathit{eff}_{i_a}(f) = -1$
- $cost_{i_a} = 0$

There are no dual bounds or heuristics. We will now explain the modification in further detail.

In the set of variables $\mathcal{V}_{\pi_M}$ we add a new set variable $\mathcal{A}$. It will be used to track which action were already considered by storing their indices. We also add the element variable f which tracks which action is currently being forced. If there is none, f will have value -1.

In the target state $\mathcal{A}$ is empty as no action has been considered yet. Variable f is assigned value -1 as no action is forced at the moment.

For each action $a \in A$ we have three transitions: forcing an action f_a, applying an action t_a and ignoring an action i_a.
The forced transition f_a checks whether the preconditions of action a hold and $a \notin \mathcal{A}$. If this is the case, element variable f is set to a, this transition has no cost. The order over the forced transitions f_a for $a \in A$ is simply the order in which the actions are defined in the delete-free STRIPS task.
Transition t_a checks whether a is currently being forced, i.e., $f = a$. If this is the case, then the effects of τ_a can be applied as normal, a is marked as considered by adding it to set variable $\mathcal{A}$ and f is freed up by assigning it -1. The transition costs are the same as for $cost_{\tau_a}$.
Transition i_a also checks if $f = a$ holds, but it only adds a to $\mathcal{A}$ and frees f by assigning -1 to it. It will not update any other state variable but also does not cost anything.
Using these transitions it is ensured that an action gets considered as soon as it is applicable and we then decide whether to apply it for a cost or ignore it for free, afterwards it can never be forced again and can therefore also never be considered again.

4.4 Ignoring Unproductive Transitions

The second modification is to only consider transitions whose effects actually change state variables. Given are a delete-free STRIPS task $\Pi = \langle V,\ I,\ G,\ A\rangle$, as well as the DyPDL task

$D_{VAR} = \langle \mathcal{V}_{VAR},\ s^0_{VAR},\ \mathcal{K}_{VAR},\ \mathcal{B}_{VAR},\ \mathcal{T}_{VAR},\ \mathcal{C}_{VAR},\ h_{VAR} \rangle$ resulting from the variable-to-variable encoding and the DyPDL task $D_{SET} = \langle \mathcal{V}_{SET},\ s^0_{SET},\ \mathcal{K}_{SET},\ \mathcal{B}_{SET},\ \mathcal{T}_{SET},\ \mathcal{C}_{SET},\ h_{SET} \rangle$ resulting from the variable-to-set encoding.
Each transition $\tau_a = \langle pre_{\tau_a},\ \mathit{eff}_{\tau_a},\ cost_{\tau_a},\ forced_{\tau_a} \rangle \in \mathcal{T}_{VAR}$ of the DyPDL task $D_{VAR}(\Pi)$ is changed to a transition $\tau'_a = \langle pre'_{\tau_a},\ \mathit{eff}_{\tau_a},\ cost_{\tau_a},\ forced_{\tau_a} \rangle$ where:

$$pre'_{\tau_a} = pre_{\tau_a} \cup \sum_{v \in \mathcal{V}_{VAR}} \mathbb{1}(v \neq \mathit{eff}_{\tau_a}(v)) > 0$$

We keep all current preconditions but add a new one which states that the sum of variables getting assigned a new value when applying τ_a is larger than 0.
In the variable-to-set encoding each transition $\tau_a = \langle pre_{\tau_a},\ \mathit{eff}_{\tau_a},\ cost_{\tau_a},\ forced_{\tau_a} \rangle \in \mathcal{T}_{SET}$ of the DyPDL task D_{SET} gets replaced with a transition $\tau''_a = \langle pre''_{\tau_a},\ \mathit{eff}_{\tau_a},\ cost_{\tau_a},\ forced_{\tau_a} \rangle$ where:

$$pre''_{\tau_a} = pre_{\tau_a} \cup (add(a) \setminus \mathcal{X}) \neq \emptyset$$

This ensures that no transition is considered where the set variable $\mathcal{X}$ already contains all variables which it would acquire after applying the effects of the transition.

4.5 Possible Heuristics

The third possible modification is to add a dual bound to the models. We can use heuristics for this purpose. We again consider a delete-free STRIPS task $\Pi = \langle V_\Pi,\ I_\Pi,\ G_\Pi,\ A_\Pi \rangle$ and the DyPDL encodings $D_{VAR}(\Pi) = \langle \mathcal{V}_{VAR},\ s^0_{VAR},\ \mathcal{K}_{VAR},\ \mathcal{B}_{VAR},\ \mathcal{T}_{VAR},\ \mathcal{C}_{VAR},\ h_{VAR} \rangle$ and $D_{SET}(\Pi) = \langle \mathcal{V}_{SET},\ s^0_{SET},\ \mathcal{K}_{SET},\ \mathcal{B}_{SET},\ \mathcal{T}_{SET},\ \mathcal{C}_{SET},\ h_{SET} \rangle$ as introduced in chapter 4.1 and chapter 4.2. Here we introduce two simple heuristics: The first heuristic is the trivial *zero heuristic*:

$$h^0(s) = 0$$

It assigns each state s the value 0. For both encodings the zero heuristic can be directly implemented as a dual bound $h^0(s) = 0$ for all states s.
The zero heuristic h^0 is trivially admissible.

The second heuristic we introduce is a modified version of the *goal count heuristic*:

$$h^g : s \mapsto \mathbb{R}_+$$

$$h^g(s) = \frac{\#\ \mathit{unsatisfied\ goal\ variables}}{\mathit{max}\ \{\#\ \mathit{affected\ variables}\}}$$

The heuristic h^g assigns each state s a value resulting from dividing the amount of not satisfied goal variables by the maximal number of affected variables any transition has.
The modified goal count heuristic h^g can be implemented as a dual bound h^g_{VAR} in the variable-to-variable encoding and a dual bound h^g_{SET} in the variable-to-set encoding:

$$h^g_{VAR}(s) = \frac{\sum_{v \in G_\Pi} \mathbb{1}(v \neq 1)}{max\{\lvert add(a) \rvert \mid a \in A_\Pi\}} \qquad \textit{for all states } s$$

$$h^g_{SET}(s) = \frac{\lvert G_\Pi \setminus \mathcal{X} \rvert}{max\{\lvert add(a) \rvert \mid a \in A_\Pi\}} \qquad \textit{for all states } s$$

Each transition can at most effect $max \ \{\# \ affected \ variables\} = k$ goal variables. If we have an optimal sequence of transitions, we still need at least $\frac{\# \ unfulfilled \ goal \ variables}{k}$ transition to reach the goal. Since h^g is exactly defined as this lower bound, it follows that we cannot overestimate the distance to the goal. Therefore, h^g is admissible.

5 Experiments

To evaluate the practical performance of the proposed encodings and modifications, we conduct several experiments on preexisting STRIPS benchmarks. In this chapter we first present the general experimental setup as well as the metrics chosen to measure performance. Afterwards, we present different hypotheses regarding the performance of different configurations of the variable-to-variable encoding introduced in chapter 4.1 and the variable-to-set encoding introduced in chapter 4.2 as well as the modifications introduced in chapters 4.3, 4.4 and the heuristics from chapter 4.5.

5.1 Experimental Setup

The experiments are based on existing benchmark problems formulated in *Planning Domain Definition Language (PDDL)* which serve as input. We used the optimal-strips suite of the downward-benchmark repository [1]. We use the Fast Downward translator [5] to preprocess the PDDL files into a suitable internal representation and remove the delete effects of all problems. We use the DIDPPy Python interface [8] to implement the variable-to-variable and variable-to-set encoding as well as all modifications. DIDPPy provides multiple solvers to choose from, and while the official website [8] recommends the *Complete Anytime Beam Search solver (CABS)*, we did not end up using it as CABS is not complete. Instead we use the *Cost-Algebraic A* Solver for DyPDL (CAASDy)*, which uses the dual bound as a heuristic. CAASDy is complete and, if we provide an admissible heuristic, also optimal. As both the zero heuristic and the modified goal count heuristic are admissible, we are guaranteed to get optimal solutions when using them.
We further use the Python package Lab [12] to run different configurations of the encodings and modifications on the benchmark problems.

We ran experiments on all domains of the suite except for the petri-net-alignment-opt18-strips, quantum-layout-opt23-strips and snake-opt18-strips domains as the Fast Downward translator classifies them as unsolvable when delete-effects are removed which requires further investigation.

For each problem run, we set a time limit of 15 minutes and a memory limit of 3.5GB and measured the following metrics:

- **Finished runs:** We measure the amount of problems successfully solved without running out of time or memory. A higher number implies a more stable and efficient configuration.
- **Memory errors:** We measure the amount of runs terminated by memory errors to identify configurations which are especially memory intense.
- **Timeouts:** We also measure the amount of runs which terminate due to exceeding the time limit. Combined with the previous metric of measuring memory errors we can find out which configuration are more memory and which more time intense. This can serve as a guideline as to what optimization methods could be interesting to look into in the future.
- **Time score:** Instead of measuring the absolute time a run needs we calculate a score for each run. We use the score which was originally used for measuring time in the agile tracks of the International Planning Competition in 2023 [6]:

  $$score : \mathbb{R}_+ \mapsto [0,\ 1]$$

  $$score(t) = \begin{cases} 1 & if\, t < 1 \\ 0 & if\, t > 900 \\ 1 - \frac{log(t)}{log(900)} & otherwise \end{cases}$$

  If a solution is found in less than a second, the run receives a score of 1. If it exceeds the time limit of 15 minutes (900 seconds), the run receives a score of 0. A run which takes anything in between gets a score logarithmically scaled between those values. We use the arithmetic mean of all individual runs in a domain to get a score for the entire domain. Using the arithmetic mean of all domain scores we get a score for the overall time performance.
- **Memory used:** We measure the total amount of memory used for each run in MB and use it to calculate the geometric mean of each domain. To get an overall score we use the geometric mean over all domains. The amount of memory used indicates how resource efficient a specific configuration is.
- **Nodes expanded:** We measure the amount of expanded nodes as a hardware-independent measure of search efficiency. Fewer expanded nodes indicate that the configuration focused on promising paths and avoided unnecessary exploration. Due to this, the amount of expanded nodes is also a measure of heuristic quality. We are also interested to see if other modifications also an influence too. We use the geometric mean over all runs as a measure for the entire domain. The geometric mean over all domains is used as a general score.

We only consider runs where all configurations finished for measuring average memory use, average number of nodes expanded and the time score.

5.2 Baseline Variable-to-Variable and Variable-to-Set Encoding

In our first experiment we compare the variable-to-variable (VAR) and variable-to-set encoding (SET) without any additional modifications.

We expect that the SET encoding uses less memory as we only have a single set variable which can be represented as a bitset internally. The SET encoding should therefore also lead to less memory errors and end up solving more problems. The time score may also be a bit better for the SET encoding as set comparisons should be faster than processing single variables. Further, we expect the amount of expanded nodes to be very similar with no encoding being significantly more efficient. Table 5.1 shows the result of the experiment

	finished	**mem. error**	**timeout**	**memory**	**time score**	**nodes exp.**
VAR	362	1407	78	43.05 MB	0.752	469.75
SET	346	1143	358	42.78 MB	0.735	469.75

Table 5.1: The performance of the variable-to-variable encoding (VAR) vs. the variable-to-set encoding (SET). The table shows the metrics described in chapter 5.1

In general, we can see that neither encoding performed great. Both where only able to solve roughly a fifth of all problems with the VAR encoding solving 362 and the SET encoding solving 346 of 1847 total problems. The amount of runs finished in the individual domains was very similar, but with one exception. While the VAR encoding managed to solve all 20 problems of the "spider-opt18-strips" domain, the SET encoding solved none due to 5 timeouts and 15 memory errors. We can therefore follow that for this domain in particular the VAR encoding is a better fit. It remains to be seen whether the SET encoding may manage to solve this domain if we add heuristics or other modifications.
The SET encoding results, as expected, in less memory errors, having runs terminated 1143 times due to them compared to 1407 times for the VAR encoding. However, the SET encoding experiences far more timeouts with 358 to 78 in the VAR encoding. This suggests that while SET reduces memory-related failures, it does not improve overall stability. Looking at individual domains, we gather that runs which failed due to memory errors in VAR also failed in SET, but due to a timeout instead. Tasks which ran out of memory before the time limit may have also ran into a timeout if they had had enough memory. The mean memory usage for SET was also lower than for VAR, but not to a significant degree with 42.78 MB vs. 43.05 MB. This could partially be explained by the fact that we have a lot more constants in the SET encoding which also take up memory and therefore equalize the advantage of using set variables. However, the constants alone should not outweigh the expected benefits of SET to this degree.
The time score of both encodings have a difference of 0.017, the VAR encoding ended up being better with 0.752 vs 0.735 in the SET encoding. Our initial assumption that computations using set-variables is faster than processing single variables is therefore not necessarily true.
The mean of expanded nodes is completely identical across all domains. The search therefore seems to proceed the exact same way for both encodings.

We conclude that both the baseline VAR and SET encoding do not perform well and would almost certainly benefit from heuristics or other modifications. With the exception of the "spider-opt18-strips" domain, they overall solve about the same amount of problems and do not have significant differences regarding memory usage and nodes expanded. The time score of VAR was however better than SET. Combined with the fact that SET could not solve the "spider-opt18-strips" domain, we would recommend using VAR over SET if no other modifications are added.

5.3 Heuristics

In this experiment we investigate the influence of the zero heuristic and the modified goal count heuristic. We combine both heuristics with both encodings, resulting in four different configurations. We also look at whether one encoding experiences a greater improvement using a certain heuristic than the other encoding.

We expect an improvement to all metrics with both heuristic, while the modified goal count heuristic should make a significantly greater improvement than the zero heuristic. While the zero heuristic does not seem useful in theory, the official DIDPPy website [8] mentions the importance of using any heuristic at all, even if it is the zero heuristic. Due to this reason, we should see at least a slight improvement to all metrics. We do not expect a certain heuristic to perform better on one encoding than on the other. Table 5.2 shows the results of this experiment.

Encoding	**Heuristic**	**finished**	**mem. error**	**timeout**	**memory**	**time score**	**nodes exp.**
VAR	**baseline**	362	1407	78	43.05 MB	0.752	469.75
	zero	509	1316	22	34.55 MB	0.796	127.53
	goal	562	1266	19	33.64 MB	0.796	105.71
SET	**baseline**	346	1143	358	42.78 MB	0.735	469.75
	zero	513	1053	286	34.47 MB	0.769	127.53
	goal	566	995	281	33.43 MB	0.770	105.71

Table 5.2: The performance of the VAR and SET encodings without heuristics, with the zero heuristic and with the goal heuristic. The table shows the metrics described in chapter 5.1

Both the zero heuristic and the modified goal count heuristic improve performance over all metrics compared to the baseline for both VAR and SET. The modified goal heuristics achieves the best results across all metrics, which was expected. The VAR encoding could solve 147 more problems using the zero heuristic and 200 more using the modified goal count heuristic. While the zero heuristic solves 40.6% more problems, the modified goal count heuristic solves 55.2% more problems. This is similar in the SET encoding where we can solve 167 more problems using the zero heuristic and 220 more using the modified goal count heuristic. The zero heuristic offers an increase of 48.3% and the modified goal count heuristic one of 63.6%. We can observe that the SET encoding in general profits slightly more off of using heuristics than the VAR encoding, but there is no specific domain in which SET is significantly better than VAR now. Interestingly, the zero heuristic performs almost as well as the modified goal count heuristic as it only offers an additional increase of about

15% in both encodings. As the zero heuristic theoretically should not offer many benefits, the improvement is likely due to how the DIDPPy python package works internally. Knowing this, the modified goal count heuristics increase of 55.2% for VAR and 63.6% for SET is less due to the heuristic being inherently powerful and more due to how DIDPPy works. However, we also need to take into account that simply looking at percentages could be misleading here as the still unsolved problems in the optimal-strips suite will get more and more difficult, therefore possibly distorting how great of an increase 15% really is. Overall, the modified goal count heuristic definitely performs better than the zero heuristic. As the modified goal count heuristic is in essence a very simple heuristic best used for smaller problems, a more powerful heuristic may potentially lead to a greater improvement over all metrics.
The zero heuristic managed to reduce the amount of memory errors by 91 for VAR and by 90 for SET. The modified goal count heuristic resulted in a reduction of 141 for VAR and 148 for SET. There are no particular domains in which one encoding profited more off of the heuristics than the other.
The amount of timeouts is reduced by 56 for VAR and 72 for SET using the zero heuristic. The modified goal count heuristic reduces them by 59 for VAR and 77 for SET. Similarly to the memory errors, there was no specific domain in which one encoding got a significantly greater improvement in timeouts, but the SET encoding did overall get a greater reduction in timeouts for both heuristics. Despite using heuristics, the SET encoding is still not able to solve the ”spider-opt18-strips” domain.
The mean memory usage is almost identical for the VAR encoding and SET encoding using either heuristic. Once again, comparing using no heuristic to the zero heuristic offers a greater jump in performance than comparing the zero heuristic with the modified goal count heuristic. While the zero heuristic results in a decrease of about 19-20%, the modified goal count heuristic only offers an additional 1-2% increase. This again indicates that the modified goal count heuristic is not very powerful.
The time score increased for both heuristics in both encodings, as expected. Both heuristics result in an increase of 0.044 for VAR and in an increase of 0.034-0.035 for SET, so VAR does seem to profit more off of using these specific heuristics.
The nodes expanded are once again exactly the same in VAR and SET, no matter which heuristic is used. When heuristics are added we can see a great improvement, just as expected. Once again, the zero heuristic does not perform significantly worse than the goal heuristic offering a decrease of 72.6% while the modified goal count heuristic results in a decrease of 77.5%.

We conclude that adding heuristics, no matter how simple they may be, result in a significant performance boost to both encodings. The modified goal count heuristic improved performance more than the zero heuristic, therefore we recommend to always use it when running experiments with our models. Interestingly, the improvement from not using a heuristic to using the zero heuristic is greater than the improvement we get when switching from the zero heuristic to the modified goal count heuristic. However, as the difficulty of unsolved problems in the optimal-strips suite only increases the more problems are being solved, the modified goal count heuristic should not be underestimated. As these simple heuristics already lead to a great improvement, exploring and implementing more powerful heuristics is a very promising direction for future work.

5.4 Excluding Transitions

In the final experiment we want to see whether we can improve performance by excluding certain actions. We use the modification which forces transitions to be either applied or discarded forever once they become applicable from chapter 4.3 and the modification which ignores transitions which do not affect any state variables when applied from chapter 4.4. We refer to them as the *force modification* and *ignore modification* in this chapter. We test both modifications on both encodings and always use the modified goal count heuristic.

The base encodings only contain one transitions for each delete-free STRIPS action, which is to apply it. The force transition modification however contains three transitions per action, one to force it, one to apply it and one to discard it. Forcing an action allows it to either be applied or discarded. Discarding it results in a new state where the action is never applicable again. This leads to a higher branching factor. While this causes potential overhead, it is also more powerful than the ignore modification. The force modification will therefore lead to more nodes being expanded and more memory used but could also speed runs up and allow for more of them to finish. The ignore modification is not expected to result in a great improvement, ideally the solver will ignore unproductive transitions anyways. If this is not the case, then we could see a slight improvement in time needed to solve problems and nodes expanded, but overall we do not expect a great difference compared to not using a modification at all. The results from this experiment are shown in table 5.3.

Encoding	Heuristic	finished	mem. error	timeout	memory	time score	nodes exp.
VAR	**regular**	562	1266	19	33.64 MB	0.796	105.71
	ignore	562	1251	34	34.86 MB	0.807	105.71
	force	396	1074	377	55.29 MB	0.764	1537.11
SET	**regular**	566	995	286	33.43 MB	0.770	105.71
	ignore	566	981	300	33.72 MB	0.782	105.71
	force	383	738	726	53.62 MB	0.741	1549.47

Table 5.3: The performance of the VAR and SET encodings using no modification, the ignore modification and the force modification. All runs used the modified goal count heuristic. The table shows the metrics described in chapter 5.1

We immediately see that the ignore modification does not have a meaningful impact. The slight changes we do see are similar for both encodings. There were no changes in the amount of runs finished and while there were slightly less memory errors terminating runs, we did see more timeouts. The mean memory usage rose a tiny bit, but not significantly. However, the time score did actually improve by about 0.011-0.012. While this modification is definitely not harmful as it even improves the time score, it will also not help runs significantly enough that it is worth using.
Looking at the force modification we however see a clear deterioration to almost all metrics. In VAR, 166 less problems were solved, in SET 183. That is a deterioration of 29.5% resp.

32.3%. Neither the ignore modification nor the force modification enabled the SET encoding to solve the domain ”spider-opt18-strips” while the VAR encoding could still solve it with both modifications. This is still the only domain in which such a drastic difference could be found between the two encodings.
The memory errors did decrease by 192 in VAR and 257 in SET, but the timeouts increased drastically. Runs which would have ran out of memory usually ended up crashing due to a timeout instead. We have an increase of 358 timeouts in VAR and 440 in SET compared to not using a modification. Mean memory usage also increased by more than 20MB for both encodings and the time score sunk by about 0.03 from not using modifications. The amount of nodes expanded increased by more than 1450% in both encodings, going from 105.71 to 1537.11 in VAR and up to 1549.47 in SET. The expected advantaged of the force modification therefore do not outweigh the significant overhead it causes, resulting in more timeouts, more memory used, a worse time score and more nodes expanded.

Neither the ignore modification nor the force modification have therefore proven to be useful. The ignore modification did not change any metric significantly enough to be worth considering and the force modification lead to a clear deterioration, probably due to its overhead. It lead to a an increase in timeouts, mean memory used, nodes expanded and to a worse time score. While these structural modifications did not improve performance, we still expect researching further modifications to be fruitful as we have only looked at two different possibilities.

6 Conclusion

This thesis explored the encoding of delete-free planning tasks within the framework of Domain-Independent Dynamic Programming (DIDP). Our objective was to demonstrate how delete-free STRIPS problems can be formulated in Dynamic Programming Description Language (DyPDL). We introduced two different encoding methods, the variable-to-variable and the variable-to-set encoding. The variable-to-variable encoding maps each delete-free STRIPS variable to a DyPDL variable while the variable-to-set encoding maps all delete-free STRIPS variables to a single set variable. This allows us to use set comparisons during solving instead of having to compare single variables.
To improve the performance of our encodings, we introduced two structural modifications and two heuristics. The force modification forces a transition to either be applied or discarded once it becomes applicable. After this, the transition may never be forced again and can therefore not be applied at a later point in time. The ignore modification does not consider unproductive transitions. If a transitions does not change a single state variable when applied, we ignore it. The heuristics we introduced were the trivial zero heuristic, which assigns each state the value 0, and the modified goal count heuristic, where the amount of not satisfied goal variables is divided by the maximal number of state variables a transition affects.

We implemented our encodings and modifications using the DIDPPy Python interface [8] and tested them on the optimal-strips suite from the downward-benchmarks repository [1]. In our first experiment we compared the variable-to-variable mapping to the variable-to-set mapping without adding any modifications or heuristics. Our hypothesis was that the variable-to-set encoding would result in better performance as the set variable could be internally represented as a bitset and therefore saving memory. We concluded that this was not the case, instead both performed in a very similar manner. Neither of them performed well overall.
The second experiment showed how the zero heuristic and the modified goal count heuristic influence performance for both encodings. Both heuristics significantly improved the performance of both encodings. Interestingly, the performance jump between using no heuristic and using the zero heuristic was greater than the jump between the zero heuristic and the

modified goal count heuristic.
In our final experiment, we investigated how discarding transitions influences solving by comparing the ignore modification with the force modification. While the ignore modification did not have a significant impact on the performance, the force modification resulted in a clear degradation of overall performance.

In conclusion, while both encodings are viable, they do not perform well overall. In future work, introducing new and more powerful heuristics to the encodings is a promising way to further optimize the models. Finding new and more efficient encodings is also a possible direction for future research. While structural modifications did not end up fruitful in this thesis, looking into them further is also an interesting approach as we only scratched the surface of what is possible in DIDP.

Bibliography

[1] AI Group - University of Basel. Downward Benchmarks, 2025. URL https://github.com/aibasel/downward-benchmarks. Accessed: 2025-02-12.

[2] 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.

[3] Avitan Gefen and Ronen Brafman. The Minimal Seed Set Problem. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 21, pages 319–322, 2011.

[4] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: theory and practice*. Elsevier, 2004.

[5] Malte Helmert. The Fast Downward Planning System. *Journal of Artificial Intelligence Research*, 26:191–246, 2006.

[6] IPC 2023. IPC 2023: The International Planning Competition (Classical Track), 2025. URL https://ipc2023-classical.github.io/. Accessed: 2025-02-12.

[7] Ryo Kuroiwa and J Christopher Beck. Domain-Independent Dynamic Programming: Generic State Space Search for Combinatorial Optimization. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 33, pages 236–244, 2023.

[8] Kuroiwa, Chen, Beck. DIDPpy Documentation, 2025. URL https://didppy.readthedocs.io/en/stable/. Accessed: 2025-02-12.

[9] Art Lew and Holger Mauch. Dynamic Programming: A Computational Tool. volume 38 of *Studies in Computational Intelligence*. Springer Science & Business Media, 2006.

[10] Florian Pommerening and Malte Helmert. Optimal Planning for Delete-Free Tasks with Incremental LM-cut. In *Proceedings of the International Conference on Automated Planning and Scheduling*, volume 22, pages 363–367, 2012.

[11] Stuart J Russell and Peter Norvig. *Artificial Intelligence: a modern approach*. Pearson, 2016.

[12] Jendrik Seipp, Florian Pommerening, Silvan Sievers, and Malte Helmert. Downward Lab. https://doi.org/10.5281/zenodo.790461, 2017.