Modelling Git Operations as Planning Problems

Bachelor Thesis

Faculty of Science of the University of Basel
Department Mathematics and Computer Science
Artificial Intelligence Research Group
ai.dmi.unibas.ch

Examiner: Prof. Dr. Malte Helmert
Supervisor: Florian Pommerening

Tim Bachmann
tim.bachmann@stud.unibas.ch
15-916-299

Januar 20, 2021

Acknowledgments

First and foremost, I want to thank my supervisor Florian Pommerening for providing a great deal of guidance and feedback during the four months of writing this thesis. I would also like to thank Prof. Dr. Malte Helmert for giving me the opportunity to work on such an interesting topic. Additionally, I would like to thank my mother and partner for encouraging me along the way.

Abstract

Version control systems use a graph data structure to track revisions of files. Those graphs are mutated with various commands by the respective version control system. The goal of this thesis is to formally define a model of a subset of Git commands which mutate the revision graph, and to model those mutations as a planning task in the Planning Domain Definition Language. Multiple ways to model those graphs will be explored and those models will be compared by testing them using a set of planners.

Table of Contents

1 Introduction

Git is a version control system (VCS) that uses a graph to track file changes. Every time the user commits their file changes a new node gets added to the graph. A node represents the state of all files and their content at one point in time. The edges connect the nodes to their parent nodes. Many Git commands will mutate the graph in various ways: For example, the command `git commit` will add a new node to the graph and add an edge between it and the previously active node. Using the whole graph as a single state in a state-space and the mutations of the graph as transitions from one state to the next, finding a series of commands to mutate one graph into another can be modeled as a state space search problem. Planners are used to solve problems like this, and are the subject of active scientific research. To test and further optimize planners, sets of problems, so-called benchmarks are used. Models of the Git graph could prove to be an interesting benchmark for further planning research.
Understanding the many commands of Git can be an overwhelming task for new users. Allowing them to find a sequence of commands to transition from the current graph to a different graph can make it easier to use and learn. Multiple ways from a source graph to a destination graph can be found, where some ways are longer than others. Especially working with branching and rebasing can be very difficult for a novice Git user. In this thesis we will model the graph structure of Git in the Planning Domain Definition Language. Using this model any compatible planners such as the planners from the Fast Downward planning system [7] can be used to find more efficient paths to the desired result graph.
Git is a very complex piece of software, dealing not only with the graph but also with the state of the file system, synchronization between different computers, and more. Modeling all Git functionality would therefore go beyond the scope of this thesis and only a subset of Git will be discussed in this work. Namely, commands that modify the Git revision graph. One of the difficulties for new users of Git is keeping track of what Git commands are used to make certain changes to the graph. This is made even more complicated due to different behaviors of the same Git commands under varying circumstances.
We will formally define the Git behavior that modifies the Git graph, namely the commit, branch, checkout, merge and rebase commands which already cover a wide set of Git functionality. Other functionality of Git that does not relate to the Git graph like the contents of the nodes, the file state, merge conflicts, and similar will not be covered here.

We will explore how to model a complex graph as well as mutations on that graph as a domain in the Planning Domain Definition Language. Additionally to a reference model, multiple variants of this domain will be created and compared on how problems on those domains perform on a set of planners. The reference domain will mimic the internal workings of Git as close as possible. Then we will create a modified version of this domain without derived predicates and a variant of each of those domains with where all objects are ordered. To compare the domains, a suite of equivalent problems for each of the domains will be created to test and compare the performance of the domains on a set of planners. The time improvement, amount of generated axioms, and plan costs of the equivalent problems in each domain will be used as criteria for the comparison of the domain variants.

2 Background

It is important to explain the concepts and terminology used in this thesis. An in-depth overview of the used Git mechanics, as well as a short overview of planning and the Planning Domain Definition Language follows.

2.1 Overview of Git

The Git version control system stores file system revisions in a graph data structure. The user can submit changes on the file system to Git as a new revision. This creates a so-called commit, which is stored as a node in the Git revision graph. In the rest of this document, the term node will be used to refer to a commit in the graph. Exactly one node is always the HEAD node. When the graph is initialized and no commits have been made yet, we assume the HEAD points to a root node. The HEAD node represents the one revision that the user is currently working on. If the user decides to switch to a different node and work on that, then the other node will become the HEAD node. A node has references to its parents, which are the nodes it is dependent on. Usually, this will be a single node, the node that was the HEAD when the current node was committed. In the case of a merge, the node can also have two parents: the previous HEAD node and the node that was merged (see 2.1.4). A Git graph can contain copies of nodes, which are a result of the rebase command. Git does not keep track of a copy relation explicitly, but our model introduces this relation to reason about graphs that contain copies. The Git graph also has a set of pointers: the HEAD pointer and branches. A branch always points to one node. Most Git commands will move the branch pointer if a new node is created. In contrast to the branches there exists exactly one HEAD pointer. It can either point to a branch or a node. If the HEAD pointer points to a node, this node is called the HEAD node and the graph is in a "detached HEAD state". If it points to a branch this branch is called the HEAD branch and the node it points to is the HEAD node. Most Git actions will modify the graph based on the HEAD node. There also exists another type of pointer: the tag. But since tags are mostly static and meant as a human-readable name for a node this will not be discussed further in this thesis.

We define a *Git state* as a tuple $\mathcal{G} = \langle G, B, C, b, h \rangle$. The tuple $G = \langle N, E \rangle$ is a rooted directed acyclic graph with nodes N and edges E. The set B contains all branch names. The set C is the copy relation $\langle n_1, n_2 \rangle \in C$ where the node n_1 is a copy of the node n_2. The function $b : B \to N$ maps each branch name to a node. The value $h \in N \cup B$ is the node or branch where the HEAD pointer is currently pointing to. For convenience we define $n_h \in N$ as the HEAD node:

$$n_h = \begin{cases} h & \text{if} \quad h \in N \\ b(h) & \text{if} \quad h \in B. \end{cases}$$

An action maps from one Git state to a next one.
We will visualize Git states with a picture as shown in Figure 2.1. The round graph nodes correspond to commits (nodes in N), while the rectangle graph nodes are branches. A black arrow between two graph nodes n_i, n_j denotes the edge $\langle n_i, n_j \rangle \in E$ between the nodes, where the arrow points from the child node to the parent. In this example the newest node `n7` is at the bottom. The arrow from a branch to a node signifies that the branch points to the node. The HEAD pointer is signified by the colored background, in the example of the branch `b0`. A copy of a node is visualized by a gray dashed arrow from the copy to the source node. In this example the nodes and branches are numbered in sequence of their creation.

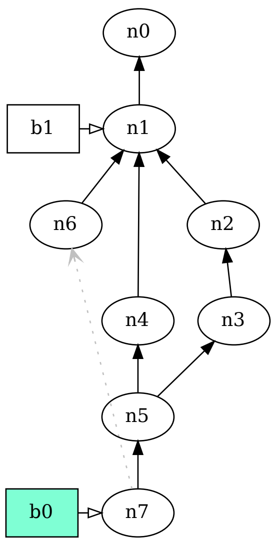


Figure 2.1: An example Git state.

2.1.1 Git Commit

Git commit is one of the most used Git commands that modifies the revision graph. It appends a new node to the HEAD node. The HEAD pointer is moved to the new node if the HEAD pointer points to the HEAD node directly. If it points to a branch, the branch will be moved to the new node. An example of the second case is shown in Figure 2.2. The git commit command can modify the HEAD node to change or update the files without

adding a new node with the `--amend` flag. This behavior is not modeled, since this does not result in any change in the graph.
We define the git commit action $\mathit{commit} : \mathcal{G} \to \mathcal{G}$ as

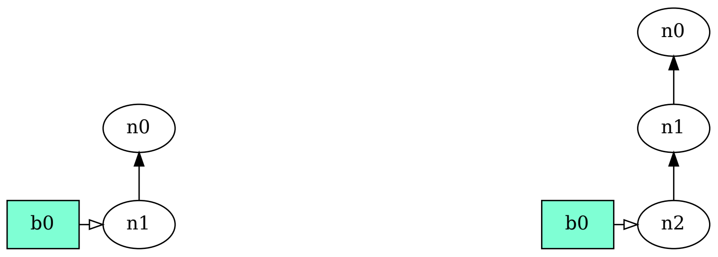
$$\mathit{commit}(\langle G, B, C, b, h\rangle) = \langle G', B, C, b', h'\rangle.$$

The graph $G' = \langle N', E'\rangle$ contains the nodes $N' = N \cup \{n_*\}$ and edges $E' = E \cup \{\langle n_*, n_h\rangle\}$, where $n_* \notin N$ is the new node. The function b' is defined as

$$b'(x) = \begin{cases} n_* & \text{if} \quad x = h \\ b(x) & \text{otherwise.} \end{cases}$$

This implies that if the HEAD is a node then b' is the same as b. We define the new HEAD element as

$$h' = \begin{cases} n_* & \text{if} \quad h \in N \\ h & \text{otherwise.} \end{cases}$$



(a) The initial Git state $\mathcal{G}$.

(b) The Git state $\mathcal{G}' = \mathit{commit}(\mathcal{G})$.

Figure 2.2: An example of the git commit command.

2.1.2 Git Branch

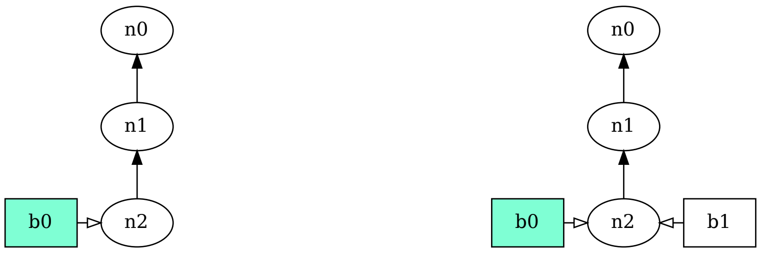
The command git branch creates a new branch. By default, the new branch will point to the HEAD node. It is also possible to specify a different node where the branch should point to. Git also allows the user to move a branch to point to an arbitrary node or to delete a branch. We only model the default behavior of creating a new branch pointing to a node, because it is the idiomatic way to use Git. Instead of moving the branch to an arbitrary node, a new branch should be created instead. We also will not model any deletion of branches or nodes from the graph to keep the model as simple as possible. Likewise, functionality that alters the branch metadata like the human-readable name, upstream branch, or description are not covered here, because they do not impact the Git state. Figure 2.3 shows an example of the branch action.
The default branch action is defined as

$$\mathit{branch}_n(\langle G, B, C, b, h\rangle) = \langle G, B', C, b', h\rangle$$

with $n \in N$, where the set of branches $B' = B \cup \{b_*\}$ contains the additional branch $b_* \notin B$, and where the function b' is defined as

$$b'(x) = \begin{cases} n & \text{if} \quad x = b_* \\ b(x) & \text{otherwise} \end{cases}$$

which points the new branch to the node n while keeping all other branch pointers as-is. By writing *branch*$(\mathcal{G})$ without the parameter n we assume that $n = n_h$.



(a) The initial Git state $\mathcal{G}$. (b) The Git state $\mathcal{G}' = \mathit{branch}_{\mathrm{n2}}(\mathcal{G})$.

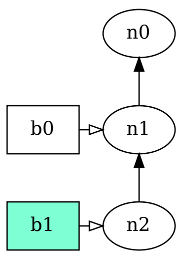
Figure 2.3: An example of the git branch command.

2.1.3 Git Checkout

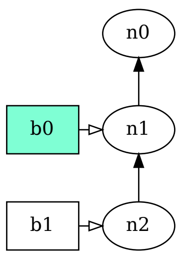
The git checkout command will move the HEAD pointer to a branch or commit. It is also possible to check out a branch that does not yet exist. In that case, Git will create the branch either pointing to a specified node or to the HEAD node if none is specified. This is equivalent to the *branch* action with an additional checkout and is therefore not modeled. It's also possible to check out a new branch that does not point to an existing node. Committing while the HEAD points at a so-called orphan branch results in a node with no parents. This allows for multiple independent histories in the same Git graph. Requiring the use of this orphan branch is rare and not modeled because it is most often avoided in practice.
The checkout action is modeled as

$$\mathit{checkout}_x(\langle G, B, C, b, h\rangle) = \langle G, B, C, b, x\rangle$$

where $x \in N \cup B$ is the new HEAD node or branch. Figure 2.4 shows an example of the checkout action.



(a) The initial Git state $\mathcal{G}$.



(b) The Git state $\mathcal{G}' = \mathit{checkout}_{\mathtt{b0}}(\mathcal{G})$.

Figure 2.4: An example of the git checkout command.

2.1.4 Git Merge

Git merge incorporates the changes made to the file system in one branch back into another branch. This makes working with Git branches powerful because independent work can be made in parallel on separate branches and merged back together. The merge command behaves differently depending on if the two branches are in an ancestor relation. A node n_1 is an ancestor of another node n_2 if the node n_1 is reachable in the graph starting from n_2. Likewise for two branches pointing to two nodes, the same ancestor relation holds between the branches as between the nodes. Figure 2.5(a) shows an example where the branch `b0` is an ancestor of `b2`.

In the first case, when merging two branches b_h, b_2 which point to n_1, n_2 where b_h is the HEAD branch and neither branch is in an ancestor relation with the other branch, then a new node will be added to the graph. This new node has two parents n_1, n_2 and can contain changes made in both parent nodes. An example for this behavior can be seen in Figure 2.5(b).

In the second case, if the branch b_h is an ancestor of the other branch b_2, the merge command will move the branch b_h to point to the same node as b_2 instead of creating a new node, because the node n_2 already contains changes made in n_1 and n_2. An example of this can be seen in Figure 2.5(c).

In the final case, when the branch b_2 is an ancestor of b_h, the merge command will not modify the graph at all. This is the case because the node $n_2 = b(b_2)$ already contains all the changes.

All three cases of the merge action are defined as

$$\mathit{merge}_y(\langle G, B, C, b, h \rangle) = \begin{cases} \mathit{merge\_a}_y(\langle G, B, C, b, h \rangle) & \text{if} \quad \langle h, y \rangle \in A_G \\ \langle G, B, C, b, h \rangle & \text{if} \quad \langle y, h \rangle \in A_G \\ \mathit{merge\_n}_y(\langle G, B, C, b, h \rangle) & \text{otherwise.} \end{cases}$$

The branch $y \in B$ is the branch to merge and the set A_G is the ancestor relation in the graph G as defined above. Git requires that $b(y) \neq n_h$ which means it is not allowed to merge a branch with itself. The sub-action $\mathit{merge\_a}$ is defined as

$$merge\_a_y(\langle G, B, C, b, h\rangle) = \langle G, B, C, b', h\rangle$$

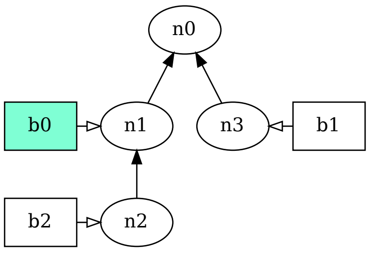
with

$$b'(x) = \begin{cases} n_h & \text{if} \quad x = y \\ b(x) & \text{if} \quad x \neq y. \end{cases}$$

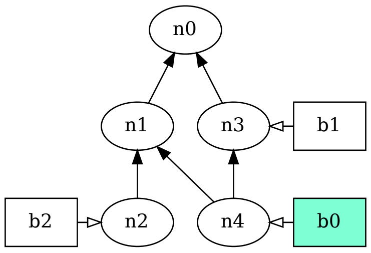
The other sub-action *merge_n* is defined as

$$merge\_n_y(\langle G, B, C, b, h\rangle) = \langle G', B, C, b', h\rangle$$

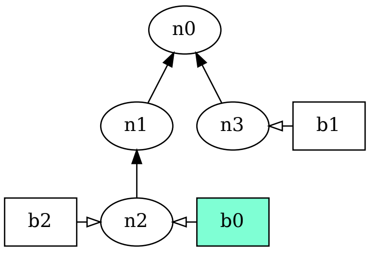
with $G' = \langle N', E'\rangle$, where $N' = N \cup \{n_*\}$ and $E' = E \cup \{\langle n_*, n_h\rangle, \langle n_*, b(y)\rangle\}$ with $n_* \notin N$ being the new node and b' being the same function as defined above for the sub-action *merge_a*.



(a) The initial Git state $\mathcal{G}$ where $\langle \texttt{b0}, \texttt{b2}\rangle \in A_G$, and `b1` and `b2` are not in the ancestor relation.



(b) The Git state $\mathcal{G}' = merge_{b_1}(\mathcal{G}) = merge\_n_{b_1}(\mathcal{G})$.



(c) The Git state $\mathcal{G}'' = merge_{b_2}(\mathcal{G}) = merge\_a_{b_2}(\mathcal{G})$.

Figure 2.5: Examples of the git merge command where (b) the HEAD branch is not in an ancestor relationship with the merged branch and where (c) the HEAD branch is an ancestor of the merged branch.

2.1.5 Git Rebase

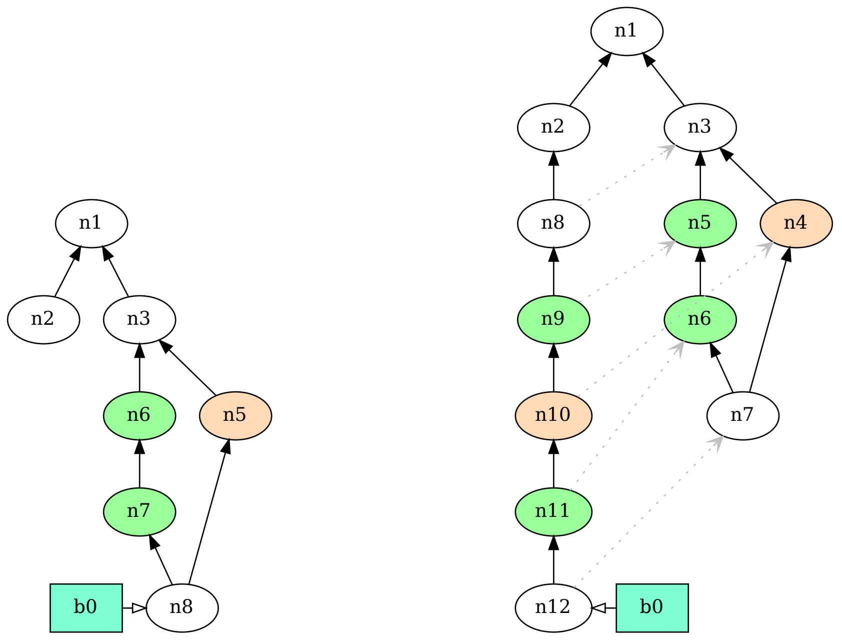
The Git rebase command serves a similar purpose as the merge command of integrating changes from one branch to another branch. Although the way this is accomplished is different. Git rebase copies nodes of one branch and inserts them as children of another branch. This means that the Git graph now contains copies of the same node in multiple

places, but no merge commit is created. An example the rebase command this is shown in Figure 2.6. A copy of a node is denoted by a gray dotted arrow from the copy to the source node. Git will occasionally garbage-collect all nodes that are no longer referenced by a branch or by other nodes, which essentially transforms the copy operation to a move operation. In Figure 2.6, the subgraph with the root `n3` will eventually be deleted by garbage-collection. We will not model garbage-collection. The rebase command is invoked on the destination node n_d. The lowest common ancestor [1] of the HEAD node n_h and n_d will be called n_{lca}. For this document the lowest common ancestor of the nodes n_i, n_{ii} is not defined if $\langle n_i, n_{ii}\rangle \in A_G$ or $\langle n_{ii}, n_i\rangle \in A_G$. The subgraph G_s of G with the root node n_{lca} and with all nodes reachable from n_h is copied over as a child of n_d with the exception of the node n_{lca} which is the node `n1` in Figure 2.6. However by default all merge commits in G_s are ignored and the subgraph is converted to a linear path from n_h to n_{lca}. For this thesis, we assume that the order of the nodes during linearization is only constrained by the fact that all ancestor relations between any nodes in G_s must still hold. Git lets the user choose from two different rebase backends, which are two rebase implementations with slightly different behaviors in edge cases such as when dealing with empty commits and merge commits. In this model of Git, all commits including the merge commits are copied. Furthermore, we only model the rebase action for nodes that are not ancestors of each other. The behavior of Git in the case that n_h and n_d are ancestors of each other can be replicated using the merge command.
We define the rebase command as

$$\mathit{rebase}_{n_d}(\langle G, B, C, b, h\rangle) = \langle G', B, C', b', h\rangle.$$

We define a subgraph $G_s = \langle N_s, E_s\rangle$ of G with the root node n_{lca} which is the lowest common ancestor of n_h and n_d. The nodes of the subgraph are defined as $N_s = \{n_{lca}\} \cup \{n \in N : \langle n_{lca}, n\rangle \in A_G \text{ and not } \langle h, n\rangle \in A_G\}$, meaning all nodes in N that have n_{lca} as an ancestor but do not have h as an ancestor. The set of edges E_s contains all edges $\langle n_1, n_2\rangle \in E$ where $n_1, n_2 \in N_s$. The copy relation is defined as $C' = C \cup C_*$ where $C_* = \{\langle n'_s, n_s\rangle\}$ where all n'_s are different nodes with $n'_s \notin N$ for all $n_s \in N_s \setminus \{n_{lca}\}$. The graph $G' = \langle N', E'\rangle$ contains the nodes $N' = N \cup N_*$, where $N_* = \{n_* : \langle n_*, n\rangle \in C_*\}$ is the set of nodes containing all copies of the nodes in N_s, and edges $E' = E \cup E_*$, where E_* is a set of edges which forms a path $p = \langle n_1^*, n_2^*, \dots\rangle$ from the last copied node n_{lc}, where n_{lc} is a copy of n_h, to the node n_c^* which is the copy of the root node of the subgraph. The path p maintains all ancestor relations between node copies from the subgraph.
The function b' is defined as

$$b'(x) = \begin{cases} y \text{ where } \langle y, n_h\rangle \in C_* & \text{if} \quad x = h \\ b(x) & \text{otherwise.} \end{cases}$$



(a) The initial Git state $\mathcal{G}$. (b) The Git state $\mathcal{G}' = rebase_{n_2}(\mathcal{G})$.

Figure 2.6: An example of the git rebase command with a merge commit `n7` in the copied subgraph.

2.2 Planning

A planning task is defined as a tuple $\Pi = \langle \mathcal{V}, \mathcal{O}, s_I, s_*, cost\rangle$. $\mathcal{V}$ is a finite set of variables $V \in \mathcal{V}$. Every variable V has a domain $dom(V)$, a finite non-empty set of values that can be assigned to this variable. This is done with a partial variable assignment p. A partial variable assignment maps a value $v \in dom(V)$ to some or all variables $V \in \mathcal{V}$. The tuple $\langle V, v\rangle$ is called a fact. It is considered true for a partial assignment p if the assignment assigns a value $v = p[V]$ to the variable V. A variable assignment of all variables is a state. The set of all possible states is denoted by $\mathcal{S}$. A partial variable assignment p is consistent with a state s if all variables assigned in p have the same value as the respective variable assigned in s. The finite set $\mathcal{O}$ contains operators $o \in \mathcal{O}$ with a precondition $pre(o)$ and an effect $eff(o)$. The precondition and the effect are both a partial variable assignments. An operator o is applicable in a given state s if the precondition $pre(o)$ is consistent with s. Applying an operator o to a state s is only possible if the precondition is met. This results in the next state $s' = s[\![o]\!]$. The state s' assigns all variables that are assigned by $eff(o)$ to the same values as $eff(o)$, and any other variables to the values of s. Operators can be chained together to form a sequence $\pi = \langle o_1, \dots, o_n\rangle$ of operators. For a sequence of states $\langle s_0, \dots, s_n\rangle$, the operator sequence is applicable on the state s_0 if every operator o_i is applicable on the state s_{i-1}, resulting in the state s_i. The state s_I is the initial state, and the goal description s_* is a partial variable assignment. If the goal description is consistent with a state s, this state is called a goal state. The function $cost : O \rightarrow \mathcal{R}$ determines the cost of an operation. The cost of a sequence of operators $\langle o_1, o_2, \ldots, o_n \rangle$ is the sum $\sum_{i=1}^{n} cost(o_i)$.

A planning problem induces a state space $\langle \mathcal{S}, A, cost, T, s_I, s_* \rangle$, where $\mathcal{S}$ is a finite set of states as defined above, A is a finite set of actions corresponding to set of operators $\mathcal{O}$ as described above, the function $cost : A \rightarrow \mathbb{R}$ determines the cost of every action equal to the cost of the corresponding operator, $T \subseteq S \times A \times S$ is the transition relation, where $T = \{\langle s, s[\![o]\!] \rangle : s \in \mathcal{S}$ and $o \in \mathcal{O}$ is applicable in $s\}$, the states $s_I, s_* \in S$ are the initial and goal states as defined above. The states and the transition relation induce a labeled directed graph, with every state being a node and every transition being an edge between nodes. Finding a series of actions from the initial state to one of the goal states is equivalent to finding a path in the graph from the initial state to one of the goal states. A shortest path between the initial state and one of the goal states is also an optimal plan of the planning task. Many algorithms exist for finding paths in a graph, an example of one of the most popular ones is the A* algorithm [5].

2.3 Introduction to the Problem Domain Definition Language

The Problem Domain Definition Language (PDDL) is a language specifically designed to model planning tasks as defined above. PDDL splits the definition of a task into two parts, the domain file, and the problem file. The domain is modeled in PDDL by a set of predicates and a set of actions. To make defining larger state spaces less verbose, PDDL has the notion of objects and parameters. Predicates can have parameters. A predicate with an object assigned to each of the parameters is an atom which is equivalent to a variable with the domain $\{true, false\}$. Therefore each assignment of those values to the predicates is a state. Using predicates with parameters it is possible to write a large number of atoms compactly. As an example of how predicates with parameters can be used, we assume we have a problem domain with a list of predicates for the position of a vacuum cleaner like `(in_room_A) (in_room_B) ...`. Instead of manually specifying each predicate for every room, it is enough to define the predicate with a parameter `(in_room ?room)`, where `in_room` is the predicate and `?room` is the parameter. The problem file has a section to define objects, in this case: `A B C ...` representing all rooms. The planner will then create the set of atoms `(in_room A) (in_room B) ...` by assigning each object to every parameter.

PDDL has the notion of types of objects to restrict what objects will be assigned to predicate parameters. The default type is `object`. Types behave in a hierarchical manner, similar to classes in object-oriented programming. Every type implicitly inherits from the type `object`. A type can only inherit from one other type, but multiple types can inherit from the same parent type. Types allow the author of a domain to classify the objects defined in the problem file without having to introduce a predicate for each type which in turn has to be set for every object of that type in the problem file. An example of possible types for a logistics simulation domain would be the type `vehicle - object` representing any kind of vehicle and the types `car train - vehicle` representing the two types for car

objects and train objects which in turn are also vehicle objects.
To transition from one state to the next state, PDDL lets the domain define a set of actions. Every action has a precondition and an effect. The precondition describes a valid partial assignment of the atoms for the action to be applicable. Actions can have parameters, which allows them to dynamically specify what atoms are relevant in the precondition. The effect describes the partial assignment of the atoms as a result of the action. The preconditions and effects of an action can be mapped to the preconditions and effects of operators as defined in Section 2.2. Instead of directly specifying partial atom assignments as preconditions and effects, PDDL permits the use of predicates as well as logical operators. Preconditions are predicate logic formulas with the most used operators being `(and (expr) ...)`, `(or (expr) ...)` and `(not (expr))`. PDDL also supports both quantifiers: The existential quantifier `(exists (?o - type) (condition))` is evaluated to `true` if at least one object exists, which inserted in the condition results in the condition being evaluated to true. This is equal to the to the existential quantifier in predicate logic

$$\exists o \in \texttt{type}, \text{ where } (\textit{condition}).$$

The universal quantifier `(forall (?o - type) (condition))` is evaluated to `true` if all objects of type `type` result in the condition being evaluated to true. This is equal to the universal quantifier in predicate logic

$$\forall o \in \texttt{type}, (\textit{condition}).$$

Notably, just like in predicate logic, the following expression with the universal quantifier `(forall (?o) (not(pred ?o)))` and the expression with the existential quantifier `(not (exists (?o) (pred ?o)))` are equivalent and can be used interchangeably. An example for the use of the universal quantifier would be to make sure every person has left an elevator before it closes the door: `(forall (?p - person) (not (in_elevator ?p)))`.
An example of a simple domain with just a single predicate would be a light switch [6]. The predicate in this domain could be `(switch_on ?room)` and the actions would be `turn_switch_on` and `turn_switch_off` with the parameters `?room`. The first action `turn_switch_on` would have the precondition `(not (switch_on ?room))`, and the effect `(switch_on ?room)`. In this example the check that the light switch is off in the room is redundant because in case the light is already on, the state does not change. The planner would never apply this action. It is best practice to set this precondition anyway as it makes it easier for humans to read and interpret the action, especially in complex domains.
Some predicates are not necessary to describe a state, and are derivable from other predicates. PDDL has a construct called derived predicates. In contrast to normal predicates, they can not be explicitly set in an effect. Instead, they are derived from the assignment of the other predicates. Derived predicates are defined the same way as a precondition, using logical operators as well as predicates and even other derived predicates. A derived predicate can be recursively defined: An example for such a derived predicate is the notion of an ancestor relation in a rooted directed acyclic graph. Without using a derived predicate,

any action that modifies this graph has to update the predicate that models this ancestor relation. But since the ancestor relation of a graph is a function of this very graph it is redundant for the state-space and therefore it is not necessary to keep track of it manually. Instead, this relation could be derived from an `is_parent` predicate, which could be a predicate with two parameters `?parent` and `?child`, and which induces a directed graph. The ancestor relation could be modeled in the following way: The derived predicate `(is_ancestor ?n1 ?n2)` is true when either `?n1` is a parent of `?n2` or when another node `?n3` exists which is the parent of `?n2` and has `?n1` as an ancestor. This recursive definition is solved by the planner by assuming the default assignment of all atoms of the derived predicate is false and iteratively applying the rules of the derived predicates. Derived predicates do not expand the set of possible domains that can be modeled using PDDL [6] but they make some domains more compact and convenient.

The equality operator `(= ?x ?y)` behaves like a built-in derived predicate which is true for any two parameters that are the same object. Like real derived predicates it can only be used in preconditions and other derived predicates.

Contrary to preconditions, effects can only consist of a subset of predicate logic. Specifically disjunctions and the existential quantifier can't be used in effects. The universally quantified effect `(forall (?o - type) (effects))` uses the same keyword as the universal quantifier. All predicates in the universal quantified effect that use the specified parameter are assigned to all objects of the specified type. Those atoms will then be assigned to a specified value. The conditional effect `(when (condition) (effects))` allows an action to only apply some effects in case the condition is evaluated as true. It can either be used by itself or can be especially useful if used as an effect of the universal quantified effect.

By default actions in PDDL have a cost of one. It is possible to define a domain with custom action costs by defining a function called `total-cost`. There are many ways to define action costs based on the state and the action, but in this thesis we only work with zero cost actions and actions of cost one. To indicate the cost of an action being one we add the statement `(increase (total-cost) 1)` to the effect of an action. To indicate a zero-cost action we will not add this statement.

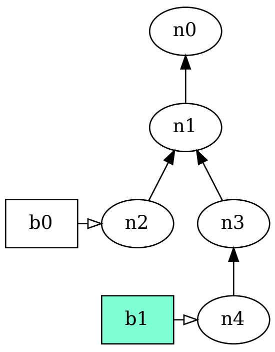
3 Modelling Git Operations

The model of Git as defined in Section 2.1 can be translated to PDDL in multiple ways. This first model aims to encode the semantics in the most intuitive way possible, disregarding any compatibility constraints of planners. We opted to use as many PDDL features as necessary to create the PDDL domain as legible and understandable as possible. We will also implement a modification of the model without derived predicates. Thereafter we will modify both models to include an ordering of PDDL objects.

3.1 Modeling Git States in PDDL

To model the mutation of the Git state $\langle G, B, C, b, h \rangle$ as a state space problem in PDDL, Git states must be modeled with predicates. Since graphs G are directed, they can be modeled with a predicate `(n_is_parent ?n_parent ?n_child - node)`. This predicate represents the edge between any two nodes in our graph, in this case PDDL objects of type `node`. This is equivalent to our formal definition of the Git graph in Section 2.1, where the objects of type node are the elements of the set N and the tuple of objects for which the predicate is true are elements of the set E. We use PDDL objects to represent nodes and branches. The PDDL types `node` and `branch` extend the type `graphobj` which is used for predicates and action parameters that accept both nodes as well as branches.
Every node, including the ones that will be used during the solution, has to be specified as an object in the problem file. Therefore only a finite, predefined amount of nodes can be used to find a plan. Unlike in the formal model, all nodes are predefined and exist from the start. The predicate `(is_in_graph ?nb - graphobj)` is needed to flag all nodes and branches that are already in the graph (i.e. they are elements of the set $N \cup B$).
As discussed in Section 2.1, a Git state consists of more than just the graph. It also contains branch pointers and the HEAD pointer. Because it is possible to have an arbitrary amount of branch pointers, they must be defined in the problem file as objects of type `branch`. The function b of the Git state is represented by the predicate `(b_points_to ?branch - branch ?node - node)`. The HEAD pointer is modeled as the predicate `(is_head ?bn - graphobj)`, which is set for exactly one node or branch. Figure 3.1 shows a Git state and the equivalent atoms that model it. The copy relation C is modeled with the

predicate (n_is_copy_of ?n1 ?n2).



```
(:init
  (n_is_parent n0 n1)
  (n_is_parent n1 n2)
  (n_is_parent n1 n3)
  (n_is_parent n3 n4)
  (b_points_to b0 n2)
  (b_points_to b1 n4)
  (is_head b1)
  (is_in_graph b0)
  (is_in_graph b1)
  (is_in_graph n0)
  (is_in_graph n1)
  (is_in_graph n2)
  (is_in_graph n3)
  (is_in_graph n4)
)
```

(a) Visual representation of Git state $\mathcal{G}$. (b) List of atoms describing Git state $\mathcal{G}$.

Figure 3.1: An example Git state.

The graph structure of the Git graph also exhibits some properties which are not directly accessible from the predicates defined above, but that are necessary for the implementation of the Git commands as PDDL actions. We model them as derived predicates.
The easiest one of those properties is the HEAD node. We could introduce a separate predicate to flag a node as the HEAD node but then every action that modifies the HEAD has to keep track of it manually. Instead, this flag is implemented as the derived predicate (n_is_head ?n - node). It is set for any node ?n where the property (is_head ?n) is set or where the branch ?b exists with the properties is_head ?b and (b_points_to ?b ?n). This directly corresponds to the node n_h as defined in Section 2.1.
The ancestor relation is a very important property of the Git graph. Depending on the ancestor relation some Git commands exhibit completely different behaviors. It is defined as the derived predicate (n_is_ancestor ?ancestor ?base - node). It is set for any pair of nodes with at least one of the following two properties: The node ?ancestor is a parent of the node ?base, or a node ?n exists where ?n is the parent of the node ?base and the node ?ancestor is an ancestor of ?n.
The last derived predicate represents the lowest common ancestor of two nodes. It is defined as the derived predicate (n_is_first_common_ancestor ?n_common ?n1 ?n2). It is set for the three nodes ?n_common, ?n1 and ?n2 if the following conditions are met: The node ?n_common must be an ancestor of both ?n1 and ?n2. The nodes ?n1 and ?n2 can not be the same node. No node ?n_child can exist that is a child of ?n_common and an ancestor of both ?n1 and n2. This derived predicate depends on the previously defined derived predicate (n_is_ancestor) which is valid in PDDL.

3.2 Modeling Git Actions

Using the predicates defined in Section 3.1 it is now possible to model Git commands as defined in Section 2.1 as actions in PDDL. Not all Git commands can be implemented as a single action. Some commands like merge need more than one action to be able to model the behavior of the command entirely. The rebase command is modeled in such a way that the multiple rebase actions have to be applied in a certain order. To make sure no other actions can be applied while the rebase is taking place, the predicate `mode_rebase` is set to signify that a rebase is in progress. Every action has a precondition that checks if this predicate is set.

3.2.1 Git Commit

Recall from Section 2.1.1 that the commit action is defined as the function $commit(\mathcal{G}) = \langle G', B, C, b', h' \rangle$, where G' contains a new node n_* with the edge $\langle n_*, n_h \rangle$, the branch pointer $b'(h) = n_*$, and the HEAD pointer $h' = n_*$ if $h \in N$. As seen in Figure 3.2, the `commit` action in PDDL gets the following parameters: The HEAD node n_h as `?n_h`, the HEAD branch h as `?h` and a free node n_* as `?n_star`. The preconditions check that the parameters are correct: The free node must not be in the graph, and the branch `?h` must be the HEAD branch and point to the HEAD node. The effect of the action flags the node `?n_star` as being in the graph, which is equivalent to $n_* \in N'$ in the model. The effect also marks the node `?n_h` as a parent of the node `?n_star` and moves the HEAD branch pointer to point to`?n_star`, as defined in the model, when $h \in B$. A second Git action `commit-no-branch` allows the planner to make a commit in the detached HEAD state where $h \in N$. It is implemented the same as the `commit` action as just described, except it has no parameter for the HEAD branch, the precondition makes sure the HEAD pointer points to the node `?n_h` directly. The effect directly moves the HEAD pointer to the node `?n_star` instead of moving the branch pointer. Only one of the actions `commit` and `commit-no-branch` is applicable in all situations because either the precondition $h \in B$ of the `commit` action is met or the precondition $h \in N$ of the `commit-no-branch` is met.

```
(:action commit :parameters (?n_h ?n_star - node ?h - branch)
    :precondition (and
        (not (is_in_graph ?n_star))
        (is_head ?h)
        (b_points_to ?h ?n_h )
        (not (mode_rebase)))
    :effect (and
        (is_in_graph ?n_star)
        (n_is_parent ?n_h ?n_star)
        (b_points_to ?h ?n_star )
        (not (b_points_to ?h ?n_h ))
        (increase (total-cost) 1)))
```

Figure 3.2: The PDDL commit action.

3.2.2 Git Branch

As in Section 2.1.2, the branch action is defined as $branch_n(\mathcal{G}) = \langle G, B', C, b', h \rangle$ with B' containing the new branch b_*, and $b'(b_*) = n$. As shown in Figure 3.3 the action `branch` has the parameters n as `?n` and b_* as `?b_star`. The precondition makes sure that the branch object `?b_star` is not in the graph. The effect of the action flags the branch `?b_star` as being in the graph, which corresponds to $b_* \in B'$, and points it to the node `?n`, which corresponds to $b'(b_*) = n$ in the model. This action is always applicable and is equivalent to the action $branch_n(\mathcal{G})$.

```
(:action branch :parameters (?n - node ?b_star - branch)
    :precondition ( and
        (not (is_in_graph ?b_star))
        (not (mode_rebase)))
    :effect ( and
        (is_in_graph ?b_star)
        (b_points_to ?b_star ?n)
        (increase (total-cost) 1)))
```

Figure 3.3: The PDDL branch action.

3.2.3 Git Checkout

The checkout action, as shown in Section 2.1.3, is defined as $checkout_x(\mathcal{G}) = \langle G, B, C, b, x\rangle$ with x being the new HEAD node or new HEAD branch. Figure 3.4 shows the PDDL code of the checkout action. The action `checkout` has the parameters h as `?h` and x as `?x`. Both parameters are of type `graphobj`, which allows them to be both node- as well as branch objects. The precondition checks that `?h` is the current HEAD node or HEAD branch and that `?x` is a node or branch that is already in the graph. The effects of the action specify that the predicate `(is_head ?nb)` is assigned to false for the current HEAD `?h`, and instead set to true for the next HEAD `?x`. This is equivalent to replacing h in the model with the new value x. The planner can potentially apply this action with `(= ?h ?x)`, In which case the state does not change, just like in the model. This is because of the so-called add-after-delete semantics, which ensures that the effect `(not (is_head ?h))` is applied before the effect `(is_head ?h)`. Since this results in the same state, the planner will never apply this action. This is consistent with the behavior of Git when checking out the current HEAD.

```
(:action checkout :parameters (?h ?x - graphobj)
    :precondition (and
        (is_head ?h)
        (is_in_graph ?x)
        (not(mode_rebase)))
    :effect (and
        (not (is_head ?h))
        (is_head ?x)
        (increase (total-cost) 1)))
```

Figure 3.4: The PDDL checkout action.

3.2.4 Git Merge

Recall from Section 2.1.4, the merge action is defined as

$$merge_y(\mathcal{G}) = \begin{cases} merge\_a_y(\mathcal{G}) & \text{if} \quad \langle h, y\rangle \in A_G \\ \mathcal{G} & \text{if} \quad \langle y, h\rangle \in A_G \\ merge\_n_y(\mathcal{G}) & \text{otherwise.} \end{cases}$$

The merge action has three different behaviors depending on the parameter and the Git state. We model the behaviors as separate actions in PDDL. The first case, where $\langle h, y\rangle \in A_G$, is modeled by the PDDL action `merge-ancestor` shown in Figure 3.5. The second case, where $\langle y, h\rangle \in A_G$, does not need to be modeled as a PDDL action because the Git state does not change. An action without any effects would never be applied by the planner because it adds to the plan cost without getting closer to a goal state. The third case is modeled by the action `merge` shown in Figure 3.6.

As described in Section 2.1.4, the merge action where h is an ancestor of y is defined as $merge\_a_y(\mathcal{G}) = \langle G, B, C, b', h \rangle$ where $b'(y) = n_h$. The corresponding PDDL action called `merge-ancestor` has the parameters n_h as `?n_h`, $b(y)$ as `?y`, and the branches y as `?y` and h as `?h`. The first preconditions make sure that `?h` is the HEAD branch and points to the node `?n_h` and that `?y` points to `?y_b`. The next precondition models the case condition of the definition: it makes sure that `?n_h` is an ancestor of `?y`. The effects of the action set the branch pointer for the branch `?h` to the node `?y` and removes it from the node `?n_h`. Just as in the definition, the action `merge-ancestor` is applicable exactly when `?n_h` is an ancestor of `?y` for the current HEAD node and any other branch `?y`.

```
(:action merge-ancestor :parameters (?n_h ?y_b - node ?y ?h - branch)
    :precondition (and
        (is_head ?h)
        (b_points_to ?h ?n_h)
        (b_points_to ?y ?y_b)
        (n_is_ancestor ?n_h ?y_b)
        (not(mode_rebase)))
    :effect (and
        (b_points_to ?h ?y_b)
        (not (b_points_to ?h ?n_h))
        (increase (total-cost) 1)))
```

Figure 3.5: The PDDL merge ancestor action.

As seen in Section 2.1.4 the merge action when y and h are not ancestors of each other is defined as $merge\_n_y(\mathcal{G}) = \langle G', B, C, b', h \rangle$ where the graph G' has a new node n_* with the parents n_h and $b(y)$, and where $b'(y) = n_h$. The PDDL action `merge` has the parameters: n_h as `?n_h`, $b(y)$ as `?y_b`, n_* as `?n_star` and the branches y as `?y` and h as `?h`. The first four preconditions make sure the parameters are as just defined: `?n_h` must be the HEAD node, the node `?n_star` must not be in the graph and the branches `?y` and `?h` must point to the nodes `?y_b` and `?n_h` respectively. The next two preconditions codify the case condition of the $merge_y$ action by checking that y and h are not the ancestors of each other. The next precondition makes sure that the HEAD node and $b(y)$ are not the same node. The effects of the `merge` action set the `is_in_graph` flag for the new node `?n_star`, and set its parents as `?n_h` and `?y`. It also moves the branch pointer for the HEAD branch to the new node `?n_star`.

Just as required by the definition, exactly one or none of the actions `merge` and `merge-ancestor` are applicable in any case.

```
(:action merge :parameters (?n_h ?y_b ?n_star - node ?y ?h -
branch)
    :precondition (and
        (n_is_head ?n_h)
        (not (is_in_graph ?n_star))
        (b_points_to ?h ?n_h )
        (b_points_to ?y ?y_b )
        (not (n_is_ancestor ?y_b ?n_h))
        (not (n_is_ancestor ?n_h ?y_b))
        (not (= ?n_h ?y_b))
        (not(mode_rebase)))
    :effect (and
        (is_in_graph ?n_star)
        (n_is_parent ?n_h ?n_star)
        (n_is_parent ?y_b ?n_star)
        (b_points_to ?h ?n_star)
        (not (b_points_to ?h ?n_h))
        (increase (total-cost) 1)))
```

Figure 3.6: The PDDL merge action.

3.2.5 Git Rebase

The rebase action is the most complex action covered in this work. This is because it does not operate on a fixed, predefined number of nodes. Recall from Section 2.1.5, the rebase action is defined as $rebase_{n_d}(\mathcal{G}) = \langle G', B, C', b', h \rangle$ where G' contains a copy of the subgraph G_s without the node n_{lca}, where G_s is the subgraph with the root n_{lca} and containing the nodes reachable from n_s. Copying a subgraph in PDDL can not be accomplished with a single action. To copy a subgraph we need to iterate over the nodes and copy the nodes sequentially, while always updating the pointers to the next source and target nodes as visualized in Figure 3.8. PDDL does have universally quantified effects, notably the `forall` clause shown in 2.3 that allows an action to have an effect that modifies an arbitrary amount of nodes. But effects of an action are applied to the state and not executed like code. This means it is impossible to maintain and update a destination pointer for each inserted node. For this reason, the implementation of the rebase command is split into multiple PDDL actions: The action `rebase`, shown in Figure 3.7(a) saves all parameters for the rebase and sets the `mode_rebase` flag, so no other actions can be applied until the rebase is done. The action `rebase__copy`, shown in Figure 3.7(b), copies a single node from the subgraph G_s to the target node. The action `rebase__end`, shown in Figure 3.7(c), will reset all predicates that are used to save parameters for the rebase, as well as the `mode_rebase` flag. All predicates that are only used for rebasing are prefixed with the string `rebase_`. The following predicates are used exclusively for the rebase actions: The predicate `(rebase_state ?head ?target - node)` saves the HEAD node and the current target node, which is the parent of the next node that will be copied. This predicate is only set for exactly one pair of nodes for the entire duration of the rebase. The predicate `(rebase_to_copy ?n)` represents the set of all nodes that are yet to be copied. This predicate is only set for a single node if the subgraph G_s consists of a single path. Otherwise, the predicate can temporarily be set for multiple nodes. If the predicate is not set for any nodes, all nodes have been copied and the rebase is done. The `rebase_copy` action receives one node from this predicate as a parameter and copies it. The last predicate is `(rebase_is_copied ?n - node)` which is set for all nodes that already have been copied. This is used to make sure no node can be copied more than once in a single rebase.

```
(:action rebase :parameters (?n_h ?n_d ?n_lca - node)
    :precondition (and
        (n_is_head ?n_h)
        (n_is_first_common_ancestor ?n_lca ?n_h ?n_d)
        (not (n_is_ancestor ?n_h ?n_d))
        (not (n_is_ancestor ?n_d ?n_h))
        (not(mode_rebase)))
    :effect (and
        (mode_rebase)
        (rebase_state ?n_h ?n_d)
        (forall (?n - node) (when (and
            (n_is_parent ?n_lca ?n)
            (or (n_is_ancestor ?n ?n_h)
                (= ?n ?n_h)))
            (rebase_to_copy ?n)))
        (increase (total-cost) 1)))
```

(a) The PDDL `rebase` action.

```
(:action rebase__copy :parameters (?n_h ?n_target ?n_next ?n_star - node)
    :precondition (and
        (mode_rebase)
        (rebase_state ?n_h ?n_target)
        (rebase_to_copy ?n_next)
        (not (is_in_graph ?n_star)))
    :effect (and
        (is_in_graph ?n_star)
        (n_is_parent ?n_target ?n_star)
        (n_is_copy_of ?n_star ?n_next)
        (rebase_is_copied ?n_next)
        (forall (?n - node) (when (and
            (n_is_parent ?n_next ?n)
            (not (rebase_is_copied ?n))
            (or (n_is_ancestor ?n ?n_h)
                (= ?n ?n_h)))
            (rebase_to_copy ?n)))
        (not (rebase_to_copy ?n_next))
        (not (rebase_state ?n_h ?n_target))
        (rebase_state ?n_h ?n_star)))
```

(b) The PDDL `rebase__copy` action.

```
(:action rebase__end :parameters (?n_h ?n_lc - node ?h - branch)
    :precondition (and
        (mode_rebase)
        (rebase_state ?n_h ?n_lc)
        (is_head ?h)
        (not (exists (?n - node)
          (rebase_to_copy ?n))))
    :effect (and
        (not (mode_rebase))
        (not (rebase_state ?n_h ?n_lc))
        (forall (?n - node) (not (rebase_is_copied ?n)))
        (not (b_points_to ?h ?n_h))
        (b_points_to ?h ?n_lc)))
```

(c) The PDDL `rebase__end` action.

Figure 3.7: The PDDL actions corresponding to the *rebase* action.

The responsibility of the `rebase` action is to make sure a rebase is applicable in the current state and to save parameters to the predicates for the other actions `rebase__copy` and `rebase__end` to use. The action has the parameters n_h as `?n_h`, n_d as `?n_d` and n_{lca} as `?n_lca`. The first two preconditions make sure the parameters `?n_h` is the HEAD node and `?n_lca` is the first common ancestor of `?n_h` and `?n_d` using the derived predicate `n_is_first_common_ancestor` defined in Section 2.1. The next two preconditions make sure `?n_h` is not an ancestor of `?n_d` and vice versa, because in that case no lowest common ancestor exists as defined in Section 2.1.5. The last precondition makes sure we are not already rebasing. The effect of this action sets the `mode_rebase` flag to ensure only the `rebase__copy` and `rebase__end` actions are applicable until the rebase is completed. It

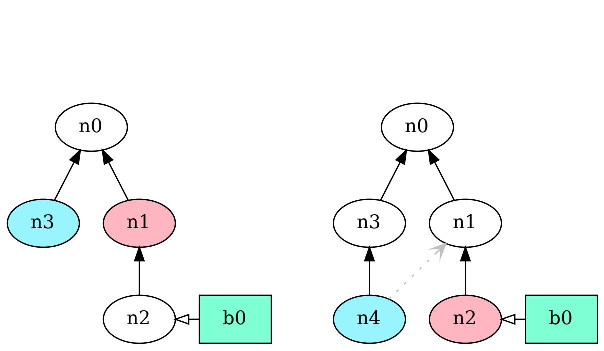


Figure 3.8: Example of steps to perform a rebase to the node `n3`. Nodes that are marked as `to_copy` are colored in pink, the current node `?target` for every step is colored light blue.

also stores the parameters `?n_h` and `?n_d` in the predicate `(rebase_state ?head ?end)`. Last, all children of the node `?n_lca`, which are in the subgraph that is being copied are flagged with the predicate `(rebase_to_copy ?n)`, which requires the use of the PDDL features “Universally Quantified Effects” and “Conditional Effects”. Using those features it is possible to formulate the following effect: We mark all nodes `?n` with the predicate `(rebase_to_copy ?n)` that meet the conditions, `?n` is a child of `?n_lca` and either an ancestor of, or the same node as `?n_h`. These conditions are the same as the conditions for the parameter `?n_next` in the action `rebase` and make sure only nodes are marked as `rebase_to_copy` if they are in the subgraph G_s.
The `rebase` action is the only action for the rebase command that has a cost. This makes sure the rebase command is considered with the same cost as any other command, regardless of how many nodes are copied.
The action `rebase__copy` copies a single node from the subgraph G_s to the target node and marks all children of the copied node that are in the subgraph G_s to be copied as well, by setting the predicate `(rebase_to_copy ?n)` for all child nodes that are not copied yet. The parameters of the action are n_h as `?n_h`, the node `?n_target` which will be the parent node of the copy, the node `?n_next` which is the node that will be copied by this action, as well as the node `?n_star` which is a new node that will be used as the copy. The first precondition checks that the current state is in rebase mode by checking the flag `mode_rebase`. The next preconditions check that the parameters `?n_h` and `?n_target` match the ones specified in the `rebase_state` predicate, that `?n_next` is one of the nodes that is flagged by the predicate `(rebase_to_copy ?n)`, and that the node `?n_star` is not yet in the graph. The effect of this action copies the the node `?n_next`, flags all children of `?n_next` which are in the subgraph G_s with the predicate `(rebase_to_copy ?n)` and updates the predicate `rebase_state`. This is accomplished in the following way: Copying the node `?n_next` is done similarly to inserting a new node in the commit action as shown in Section 3.2.1. The node `?n_star` is marked as being in the graph and its parent node is set to `?n_target`. Additionally the predicate `(n_is_copy_of ?n_star ?n_next)`

is set to signify that the new node is a copy of the node `?n_next`, and the predicate `(rebase_is_copied ?n_next)` is set to prevent the node `?n_next` to be copied multiple times in case the subgraph contains a merge. Flagging all children of the node `?n_next`, which are in the subgraph that is being copied, is done similarly as in the action `rebase`. Additionally to the requirement for any node `?n` being a child of `?n_next`, and being either an ancestor of, or the same node as `?n_h`, it must also not have been copied yet. The copied node `?n_next` is removed from the `rebase_to_copy` predicate. The final effect of the action is to update the predicate `(rebase_state ?head ?target)` by unsetting the old values and setting the predicates with the values `?n_h` and `?n_new`. This ensures that the predicate is set for always exactly one pair of nodes. The action is applicable exactly as long as there is at least one node marked as `rebase_to_copy`. By lazily marking nodes as `?to_copy` we implement a graph traversal algorithm which uses the predicate `rebase_to_copy` as an open list and the predicate `rebase_is_copied` as a closed list. Contrary to the definition of the rebase command, the action `rebase__copy` can be applied to a subgraph G_s in a way that the resulting copy G_* does not respect all ancestor relations from G_s. This means that the planner can find a plan for a problem that should be unsolvable. This could be resolved by ordering the nodes in the open list. This could be achieved using a predicate `(rebase_copy_after ?n1 ?n2)` orders the nodes according to this relation, and checking that a node `?n1` must be copied before the next node `n2`.
The action `rebase__end` is applied when all nodes are copied and will perform cleanup on the rebase predicates as well as move the HEAD branch to point to the last copied node. The parameters of the action are n_h as `?n_h`, the last copied node `?n_lc` as well as the HEAD branch as `?h`. The action has the following preconditions: The first precondition is checking if we are currently in rebase mode. The next two preconditions check that `?h` is the HEAD branch and that `?n_h` and `?n_lc` correspond are the nodes that are flagged by the predicate `rebase_state`. The `rebase_state` predicate was previously updated with the HEAD node and the target node by the action `rebase__copy`. Because `rebase__copy` builds a linear graph, the target node always corresponds to the last copied node in this graph. The last precondition makes sure that no node `?n` can exist with the flag `(rebase_to_copy ?n)`. This makes sure every node in the subgraph has been copied over. The effect of this action will reset the predicates `(mode_rebase)`, `(rebase_state ?head ?target)` and, using universally quantified effects, the predicate `(rebase_is_copied ?n)`. It will also move the branch pointer of the HEAD branch to the last copied node `?n_lc`. While in rebase mode either the action `rebase__copy` or `rebase__end` is always applicable, because either there are nodes flagged as `rebase_to_copy`, in which case the copy action is applicable, or there are none, in which case the end action is applicable.

3.3 Other Ways of Modelling the Domain

A PDDL domain can be modeled in multiple ways, for example with a different subset of PDDL features resulting in domains that model the same state-space. Some features of PDDL are computationally easier for planners to process and/or have better support across multiple planners. We will create domain variations based on the domain D_{ref} which is the reference domain and defined in Section 3.1. The first variation will be a domain D_{nd} without derived predicates. A second variation D_o will use new nodes and branches in a predefined order. The last variation $D_{nd,o}$ will combine both modifications and therefore remove the derived predicates as well as introduce ordered nodes and branches.

3.3.1 Without Derived Predicates

Most PDDL planners do not support the whole feature set of PDDL. One of the more rarely implemented features is axioms. While not explicitly a feature of PDDL, axioms are being used in the internals of planners to represent some PDDL constructs such as derived predicates and conditional effects. The goal of eliminating derived predicates is to possibly expand the set of planners that can use the domain, as well as to compare the performance of planners that support both domains with and without derived predicates. The domain without derived predicates will be called D_{nd}. Derived predicates are not the only PDDL features that require the planner to use axioms, but removing derived predicates and therefore potentially reducing the number of axioms is a necessary first step to creating an equivalent domain without axioms.

As shown in Section 3.1, the derived predicates used in the domain are `(n_is_head)`, `(n_is_ancestor)` and `(n_is_first_common_ancestor)`.

The derived predicate `(n_is_head ?n)` is trivially replaced in the preconditions of actions. The action `merge` already has the parameter `?h` of type branch, which is the HEAD branch. So to make sure the parameter `?n_h` is indeed the HEAD node, it suffices to replace the precondition `(n_is_head ?n_h)` with the check that either `(is_head ?n_h)` or `(is_head ?h)` is set. The action `rebase` does not have a parameter for the HEAD branch. One approach could be to add this parameter and check if either `?n_h` is the HEAD node or the branch `?h` is the HEAD branch and points to `?n_h`. The drawback of this approach is that due to the action with the additional parameter the plans are no longer compatible. To keep both domains as close as possible we used another approach of checking if either the parameter `?n_h` is the HEAD node, or using existential preconditions checking if a branch `?b` exists that is the HEAD branch and points to the node `?n_h`. This way no additional parameter has to be added to the action and plans from both domains are directly comparable. A third approach could be to split the action in two and have one accept the HEAD branch and the other accept the HEAD node.

The derived predicate `(n_is_ancestor ?ancestor ?base)` is not removed completely but instead turned into a normal, non-derived predicate. This means no modifications have to be made in the preconditions that check the ancestor relation but instead, every action that adds a node to the graph has to add the ancestor relation to all ancestors of that graph, which is the case for the actions `commit`, `commit-no-branch` and `rebase__copy`. Because no actions exist that remove nodes from the graph, the ancestor relation is monotonic, meaning tuples of nodes will only ever be added and never removed. All actions that need to modify the ancestor predicate already have a parameter for the parent `?n_h` of the newly inserted node `?n_star`. The first step is to mark the node `?n_h` as an ancestor of `?n_star`. For all nodes that are ancestors of the parent node, the ancestor predicate can be set for the

node `?n_star` using a universal quantified effect. The action `merge` does not only have one parent node, but two. In this case, both parent nodes are marked as ancestors, as well as any ancestors of either parent node. This approach has the drawback that the problem files of the domain D_{ref} are not compatible with problem files of the domain D_{nd}. The reason is that the problem files of the D_{nd} must contain all ancestor relations in the initial state, whereas this would result in an error in the domain D_{ref} because derived predicates can not be explicitly set in the initial state.
The derived predicate `(n_is_first_common_ancestor ?n_common ?n1 ?n2 - node)` is only used in the action `rebase`. Because it is not recursively defined, its usage can be replaced by its definition.

3.3.2 With Ordered Nodes and Branches

In both previously discussed domains D_{ref} and D_{nd}, the nodes and branches have no particular order. This means that in every action that needs to use a new node or branch the planner has the option to use any one of the available ones. The planner can not reason that any one of those results in an equivalent state with just other names. Therefore the planner has to compute a large number of states which are, for our purpose, equivalent. To limit the number of states we introduce a branch order and a node order. This forces the planner to use the free node `n1` before `n2` and `n3` and so on, limiting the planner to exactly one branch or node that can be inserted to the graph at any time.
To model this in PDDL we introduce the predicate `(next_obj ?o1 ?o1)` which represents the order of objects and the predicate `(cur_obj ?o)` which represents the object that will be used next. Both predicates are not typed, which means they can be used with node- as well as branch objects. All actions that use a new node or branch have a precondition that checks if this new object is flagged with `(cur_object ?o)`. This limits the planner to exactly this object. The effect of these actions updates the current object to the next object according to the `next_obj` predicate. Using universally quantified effects we can write the effect as

```
                    (not (cur_obj ?cur))
    (forall (?o) (when (next_obj ?cur ?o) (cur_obj ?o)))
```

Where `?cur` is the node or branch that was inserted in the graph in the current action. To avoid the universally quantified effect it would also be possible to add an additional parameter to those actions. All problem files must specify the order for all nodes and branches, as well as the first node that is not used in the initial state.
This model has the drawback in that is no longer equivalent to the previous models. This is the case because problems that can be solved in the other domains may become unsolvable if the goal state uses nodes or branches in an arrangement that is incompatible with the order of those objects. An example of a problem that is unsolvable in this domain is the following: All nodes are ordered in ascending order of their names. The initial state consists of the graph with the single node `n0`. The goal state consists of the graph

$$\texttt{n0} \leftarrow \texttt{n2} \leftarrow \texttt{n1}.$$

This problem is unsolvable in this domain because the planner can not apply the action `commit` with the new node `n2` before the new node `n1`. To remedy this we would have to find a way to define the shape of the graph without using the names of nodes and branches directly. For example using existentially quantified goals.
It is also possible to apply the modification from the domain D_o to the domain D_{nd}. This will result in the domain $D_{nd,o}$, which does not have any derived predicates but has an order for nodes and branches.

4 Evaluation

After modelling a subset of Git in multiple domain variants it is time to compare the domains and see how the modified domains D_{nd}, D_o and $D_{nd,o}$, performed in comparison to the reference domain D_{ref}.

4.1 Problem generation

To analyze the performance and behavior of multiple planners on the previously described domains it is necessary to provide a big enough set of problem files. Some of the problem files are taken from the game Learn Git Branching [3] which inspired this work in the first place. But it was necessary to create a substantially bigger set of problem files to gather enough data for any meaningful comparison between the domain variants. For this purpose a Python script was developed which can generate an arbitrary amount of random problem files. To generate a random problem file it has to generate an initial and a goal state. Multiple initial states are predefined in the script and one is selected randomly. To generate a goal state it is not enough to randomly create any directed acyclic graph, because this resulting state has a high chance of not being reachable from the initial state. Therefore we opted to implement the previously described Git commands in Python. To generate a solvable problem we select one of the predefined initial states, apply a set amount of Git commands to it and serialize the resulting graph as a PDDL problem file for each domain variant. This way we ensure that our generated problems are solvable and that we create equivalent problems in the different domain variants. The problems in all domains must be equivalent for any meaningful comparison of the run-time characteristics of the domain variants. We generated problem files two up to fifteen PDDL nodes.

4.2 Benchmark setup

To compare the performance of the domains we let a set of planners solve each problem of each domain. This must be done in a reproducible manner on dedicated hardware to prevent other processes from interfering with the results. The run of a planner with a problem while collecting its run-time metrics is called a benchmark. The collected metrics let us compare

multiple aspects of the domains.
All benchmarks were run on the infai nodes of the SciCORE scientific computing cluster at the University of Basel. The nodes have 4 GiB of RAM per core and are running Intel Xeon E5-2660 2.2 GHz processors. No more than a single process was run on every processor core at any time.
We use the Downward Lab [9] Python package to inspect metrics of different planners on the domain variants. It simplifies the creation of experiments using multiple planners and benchmarks and directly interfaces with the slurm workload manager which runs on the SciCORE cluster. Downward Lab runs all specified planners on all problem files and collects run-time data from the generated log output. This allows the comparison of the performance and behavior of planners on a suite of benchmark problems. Downward Lab has features builtin to help with the comparison of different planners while developing and optimizing them but it lacks features for comparing domains. Due to the extendability of the library it was possible to implement the necessary missing features, which include support for multiple domain variants either from Git revisions or from multiple folders. Downward Lab has built-in mechanisms to generate various types of reports, notably scatter plots comparing two revisions of the same planner. To compare properties of the runs of multiple domains we extended the scatter plot functionality accordingly.
To analyze the differences in the domain variants we used a set of planners. The set consists of one optimal planner, four non-optimal planners, and a planner configuration that outputs increasingly better solutions over time. The optimal planner uses the A* algorithm [5] with a blind heuristic “astar-blind”. This planner will find the shortest possible plan if one exists and if enough time and memory is available. The next two planner use the A* algorithm with the goal count heuristic [4] “astar-goalcount” and the h^{max} heuristic [2] “astar-hmax”. The goalcount heuristic is neither admissible nor consistent, so this planner will not yield optimal plans. The h^{max} heuristic is not admissible for our domain because it contains axioms, and will therefore not yield optimal plans either. For our purpose of measuring differences in our domain, finding optimal plans is not required. We also use a planner that uses an eager greedy algorithm with the goal count heuristic “eager-goalcount”. This planner is not optimal. Lastly, we used the LAMA [8] planner “lama”. This planner consists of a list of progressively closer to optimal planners which are run in sequence until either an optimal solution has been found or the time runs out, in which case the best solution will be returned. This configuration is used to find the best solution in a given timeframe. The “lama-first” planner consists of the first iteration done by LAMA and is its fastest but least optimal planner.
To compare metrics of a domain with a base domain we use “relative” scatter plots, where every point (x, y) in the scatter plot represents the values of a single problem on both domains. The x-value is the absolute value of the metric for the base domain. The y-value is the relative value where $x * y$ yields the value of the metric of the compared domain. As a result, if the y value of a point is one, the problem performed the same in both domains. If the value is less than one the metric of this problem in the compared domain is smaller than in the base domain. This allows the visualization of even small differences between both domains.

	D_{ref}	D_{nd}	D_o	$D_{nd,o}$
lama	50.59	34.80	24.50	18.45
lama-first	8.35	3.54	1.18	0.73
astar-hmax	25.81	24.76	7.36	11.88
astar-blind	37.43	39.43	17.75	6.19
astar-goalcount	10.80	9.68	7.06	3.56
eager-goalcount	24.58	15.70	12.33	6.07

Table 4.1: Comparing the mean time it took the planners to solve problems in the domains. Problems that could not be solved by the planners are not taken into consideration for calculating the mean time.

4.3 Results

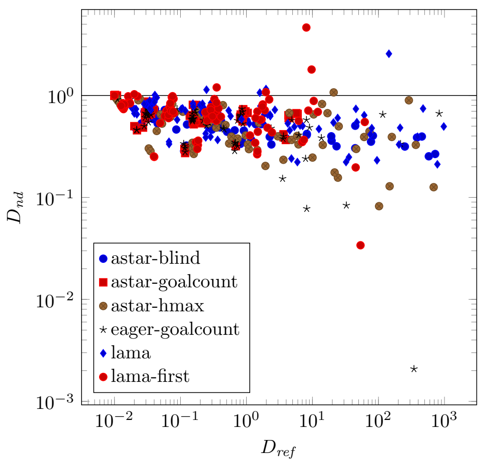
Using the benchmarks we compared metrics from the planners solving the problems for each domain. We will discuss the results.

4.3.1 Planning time

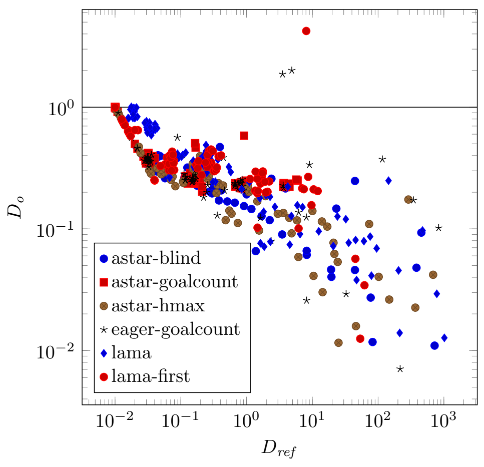
The goal of creating multiple domain variants was to analyze the performance characteristics of them. We measure the time it took every planner to find a plan for each problem. A domain performs better if the time to find a plan is smaller.
The domain D_{nd} performed generally better than the domain D_{ref} in terms of total time to find a problem solution. The big majority of problems were solved faster in the D_{nd} domain, which was expected. As shown in Table 4.1 the mean of the time it took to solve all problems in the domain D_{ref} is less than half compared to the domain D_{nd} for the planner "lama-first". All other planners except "astar-blind" were faster in the domain D_{nd} but not by this big of a factor. A likely reason for the improvement of most planners is that axioms, and in turn, derived predicates, have to be calculated for every expanded state. By reducing the number of axioms we also reduce the amount of work that needs to be done on each expanded state. By moving those calculations to the precondition and effects of the few actions that either depend on them, or change the state in such a way that an update of the predicates is necessary, this work has to be done only when one of those actions is applied. Another reason could be the difference in the quality of the heuristic values, which could depend on the presence of axioms. The Figure 4.1(a) shows the relative amount of time that was needed to solve each problem in the domain D_{nd} compared to the time in the domain D_{ref}. Throughout our experiment, some problems did not finish successfully in the given time and memory constraints. Table 4.2 shows a summary of how many problems were solved successfully in each domain. Runs that were aborted due to a timeout or reaching the memory limit are not included in the figures. None of the planners succeeded to solve a problem with more than eleven PDDL objects. Which shows how quickly the size of the state-space grows even for this relatively small amount of nodes and branches.

	D_{ref}	D_{nd}	D_o	$D_{nd,o}$
Success	616	613	633	629
Out of Time	143	126	40	39
Out of Memory	12	15	3	7
No Solution	46	63	138	144
fail	5	5	8	3

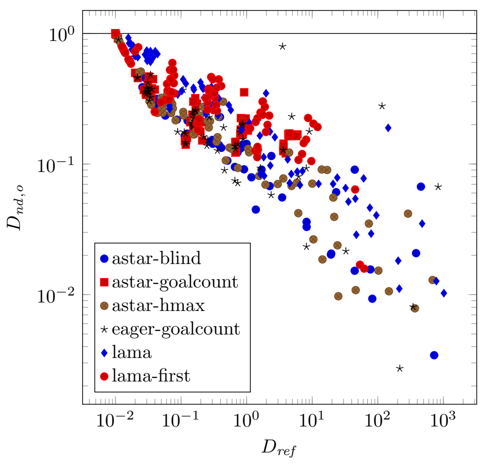
Table 4.2: Summary of results of all domains. A run is considered failed if an unexpected error happens during the solve. For example when the planner exits with a segmentation fault or if it outputs a log file bigger than 10 MiB, which is the maximum Lab will try to parse. In total 822 runs were conducted per domain.



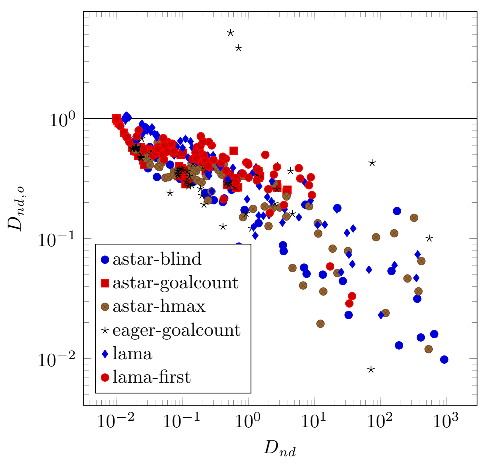
(a) Comparing the total time used to solve each problem in the domain D_{nd} compared to the time used in the domain D_{ref}.



(b) Comparing the total time used to solve each problem in the domain D_o compared to the time used in the domain D_{ref}.

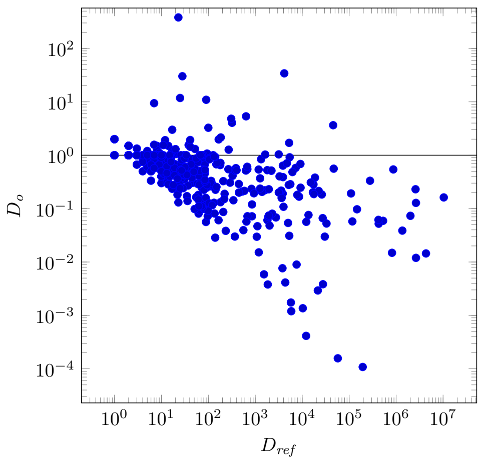


(c) Comparing the total time used to solve each problem in the domain $D_{nd,o}$ compared to the time used in the domain D_{ref}.

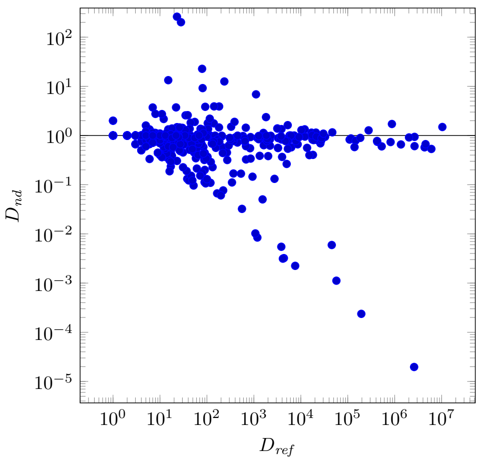


(d) Comparing the total time used to solve each problem in the domain $D_{nd,o}$ compared to the time used in the domain D_{nd}.

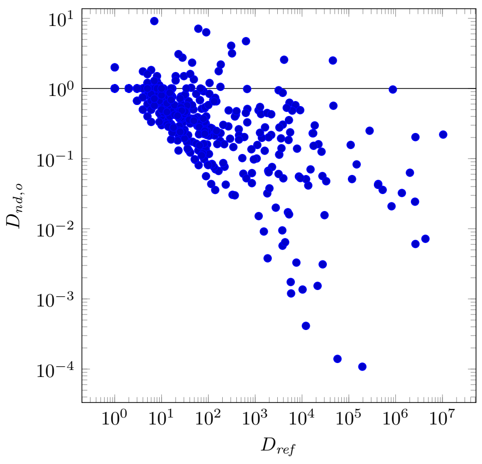
Figure 4.1: Relative scatter plots comparing the time a planner took to find a plan for a problem in one domain (y-axis) compared to a base domain (x-axis).



(a) Relative number of expansion in the domains D_o to those in the domain D_{ref}.



(b) Relative number of expansion in the domains D_o to those in the domain D_{ref}.



(c) Relative number of expansion in the domains D_o to those in the domain D_{ref}.

Figure 4.2: Relative scatter plots comparing the amount of states expanded by the planner for problems in one domain (y-axis) compared to a base domain (x-axis).

By comparing the total planning times of problems in the domains D_{ref} and D_o it is clear that reducing the search space significantly reduces the time to solve most of the problems. As seen in Figure 4.1(b), the problems of the domain D_o get solved significantly faster than those of domain D_{ref}, with a duration between slightly more than 14% for "lama-first" up to 70% for "astar-goalcount". The amount of states expanded was reduced significantly, as can be seen in Figure 4.2(b). The mean number of expansions for all problems is approximately 52471 in the domain D_{ref} and 6990 in the domain D_o, which is a reduction by a factor of 12.17%. We expect the combination of the optimizations used in the domains D_{nd} and D_o to yield even better performance metrics in the domain $D_{nd,o}$. As shown in Figure 4.1(c) this is indeed the case for all planners except "astar-hmax". The performance improvements by ordering the objects potentially orthogonal to the performance improvements by removing the derived predicates. This means that ordering the objects in the domain D_{ref} results in comparable time improvements as ordering the objects in the domain D_{nd} as shown in Figure 4.1(d). This can also be seen in the improvement of the mean time for each problem as seen in Table 4.1. Interestingly, the planner "astar-blind" which exhibited no improvement from D_{ref} to D_{nd}, had a run time of about 35% for the domain $D_{nd,o}$ compared to the domain D_o, which is the highest improvement over all planners.

The planner configuration "lama" is the only one of the planners used, that will deliver increasingly optimal plans, the longer it runs. We expect that the cost of the plans will generally be lower for the domains with lower problem solve duration. The mean cost of all plans for the domain D_{ref} is 4.02, whereas the mean cost for the domain $D_{nd,o}$ is marginally better with 3.88. Interestingly, the mean cost for the domain D_o is about the same 3.86. The probable cause of this is that the timeout of the planner was high enough to reach optimal or almost optimal plans for most problems of the domains D_o and $D_{nd,o}$. Therefore no improvements can be made, even for problems where the planner finishes before the timeout.

4.3.2 Axioms

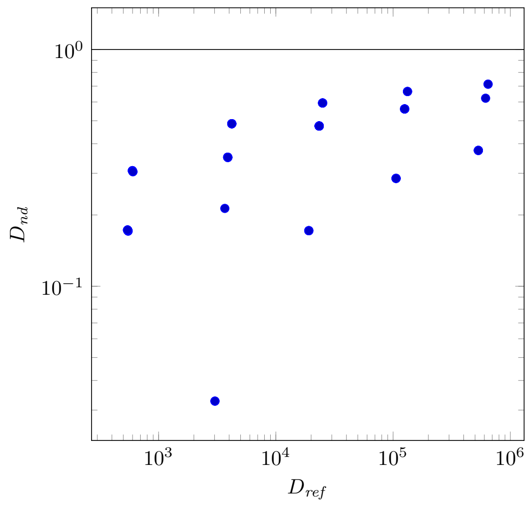


Figure 4.3: Relative number of axioms in the domains D_{nd} compared to those in the domain D_{ref}.

The goal of removing derived predicates from the domain was to expand the number of planners that support all features needed to solve problem files for the domain. Derived predicates are internally transformed to axioms in the translation step of the planner, which in turn are not fully supported by all planners. To remove the derived predicates we used PDDL features that also introduce axioms, such as conditional effects. Figure 4.3 shows by how much the number of axioms was reduced by removing derived predicates. Even though new axioms were introduced the total number of axioms was reduced significantly, for some problems even up to an order of magnitude. The number of axioms can potentially be further reduced by modifying the domain to remove other features that result in axioms. For example by replacing conditional effects by duplicating the action and introducing the condition as a precondition. Removing conditional effects inside of a universal quantified effect could be achieved in much of the same way the rebase command was implemented in Section 3.2.5, by introducing a predicate which marks the state as in a “forall-mode” and an action that is only applicable in this mode, which performs the same effect as the original quantified effect it replaces. Using those techniques it should be possible to create a domain that will not have any axioms and therefore be compatible with a bigger set of planners.

5 Conclusion

The goal of this thesis was to formally define Git states, and model them as a domain in PDDL. Variants of this domain were created, one without derived predicates, one with ordered nodes and branches, and one where both modifications were applied simultaneously. We compared the characteristics of those domains on a set of planners.
At the beginning of this thesis, we expected to be able to solve problems with git states the size encountered in real-world projects. It became clear quickly that the state space with hundreds or even just a few more than ten PDDL objects were not realistically solvable in our domain variants. However, we found out that removing derived predicates already has a substantial positive performance impact on most tested planners. An even greater performance boost can be achieved by manually restricting the state space as much as possible by ordering the nodes and branches.
Even though the derived predicates used in the reference domain were used for their convenience to keep the PDDL domain as close as possible to the formal definition, and could easily be removed by hand, they had a substantial impact on the search time of most tested planners. This begs the question if it were possible for planner authors to implement a feature to compile away derived predicates, just as we were able to by hand.
We found out that the modeled domain still uses a great number of axioms, even when removing the derived predicates. Further work could try to create other domain variants to remove all PDDL features that require axioms and compare those axiom-free domains to the variants introduced in this thesis.

Bibliography

[1] Michael A. Bender, Martín Farach-Colton, Giridhar Pemmasani, Steven Skiena, and Pavel Sumazin. Lowest common ancestors in trees and directed acyclic graphs. *Journal of Algorithms*, 57(2):75–94, 2005.

[2] Blai Bonet and Héctor Geffner. Planning as heuristic search. *Artificial Intelligence*, 129 (1):5–33, 2001.

[3] Peter Cottle. *Learn Git Branching.* 2020. https://github.com/pcottle/learnGitBranching, accessed 2020-10-05.

[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):189–208, 1971.

[5] 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.

[6] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language.* Morgan & Claypool Publishers, 2019.

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

[8] Silvia Richter and Matthias Westphal. The LAMA Planner: Guiding Cost-Based Anytime Planning with Landmarks. *Journal of Artificial Intelligence Research*, 39:127–177, 2010.

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