Decomposition of planning problems into subtasks by learning description logic formulas

Draft

September 22, 2023

Abstract

1 Introduction

The greedy best-first search (GBFS) algorithm is a classic search algorithm that uses a heuristic function to determine the most promising path. Although simple, fast, and efficient, this algorithm is not always accurate and sometimes converges to local maxima. The paper by Heusner *et al.* [2017] showed that the search space of GBFS with a given heuristic h induces a *bench transition system* (BTS) in which *benches* are connected via *progress stats* and *bench entry states*, which are all components of the search space that one may carefully use to improve planning techniques. Figures 1 and 2 illustrate the idea with an example from Heusner et al. on a toy domain. In Figure 1, a search space is presented with the heuristic values needed to determine which of the states are considered to be progress states (progress in the sense that, if such a state is encountered during the search, it might be best to continue from that point and not skip to another state). In Figure 2, a search space is presented with the heuristic values needed to split it into benches according to the progress states such that for each bench, we move forward to another bench via an "exit" progress state. Before our work [Ferber *et al.*, 2022], other methods could identify progress states only for a single task, and only *after* a solution for the task has been found. We surpassed this limitation and are able to learn progress states and use this knowledge *during the search*. Learning the entire BTS is a huge step toward decomposing planning tasks into subtasks. In this study, we plan to achieve the following two goals:

1. learn a generalized representation of the BTS for a given domain and heuristic based on data from small instances;

2. exploit the learned BST by performing a sequence of searches from a bench entry state to a progress state.

In Heusner *et al.* [2017] and other studies, the BTS is not learned but deduced afterwards when the solution (the plan) has already been found. Our main goal is to use those benches, break down the search into intermediate subsearches, and ultimately achieve subplans such that, if combined sequentially, produce one plan. In Drexler *et al.* [2022], there is a similar approach; however, it uses a different technique that is limited to some specific domains.

To decompose planning problems into subtasks, the suggested research will use a technique based on decision trees to learn the BTS and represent them using description logic formulas. The learning process is performed for each domain from small examples (small instances), and then the learned BTS is used to solve large instances of the same domain.

A high-level overview of our approach is as follows: First, all benches for each task are generated, and then the description logic features are generated. For each bench, a formula is learned that describes its exit states, and bench A is merged into bench B if the goal formula of B is also suitable for A. Lastly, for each bench, a formula is learned that describes a state in the bench. To use this framework, bench walking, i.e., an intermediate search between the learned benches, is performed.

This line of research is an extension of our successful study presented at IJCAI 2022 [Ferber *et al.*, 2022], in which the main goal was to introduce a novel approach that learns a description logic formula characterizing all progress states in a classical planning domain. Using the learned formulas in a GBFS to break ties in favor of progress states often significantly reduces the search effort. Our previous work showed that learning progress states is feasible and efficient. The next step is to not only resolve cases of tie-breaking but to take the power of learning progress states to learn the complete BTS and improve the search itself. The immediate implication of the results attained and knowledge gained is the ability to decompose a planning problem into subproblems and solved it sequentially. Other work that is based on Sketches showed that the decomposition technique is efficient and revolutionary. An additional secondary implication is related to the implementation; we aim to represent PDDL goals as description logic and vice versa, and this itself is a contribution.

Fields likely to benefit from our results are learning and automated planning, including but not limited to applications such as path and motion planning for autonomous robots, unmanned aerial vehicles, and autonomous driving. As to our latest publication [Ferber *et al.*, 2022], we guarantee that progress states can be learned and improve the expansion rate during search. In this proposal, we move to our next questions: Can a complete BTS be learned? Would it improve search? Would it be possible to perform a sequence of searches from a bench entry state to a progress state? Positive answers to these questions would mean a more efficient search, smaller expansion rate, and shorter run time. Of course, trade-offs should be considered and presented as part of the empirical analysis.

2 Background

In this section, we first present four subareas that are relevant to the suggested research and then describe our last paper, "Learning and Exploiting Progress States in Greedy Best-First Search" [Ferber *et al.*, 2022], presented at IJCAI 2022. This paper proposed a fundamental building block needed for the current research and serves as an indication of its feasibility. We start by reminding the reader of basic planning concepts and notations. Moving to description logic, we describe a state or set of states using a learned formula. Next, we review again the concept of progress states and bench transition system, and finally, we describe decision trees, which are the learning technique used in this research for identifying bench states

2.1 Classical Planning

Throughout this proposal, we work with planning domains and tasks defined in the *Planning Domain Description Language* (PDDL) [McDermott *et al.*, 1998]. A *domain* is a tuple $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$, where $\mathcal{P}$

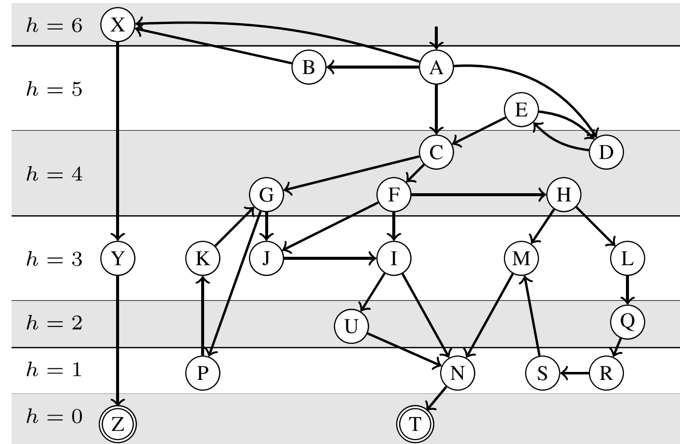

Figure 1: Search space topology from Heusner *et al.* [2017] for a toy domain, where the goal states are indicated by double lines.

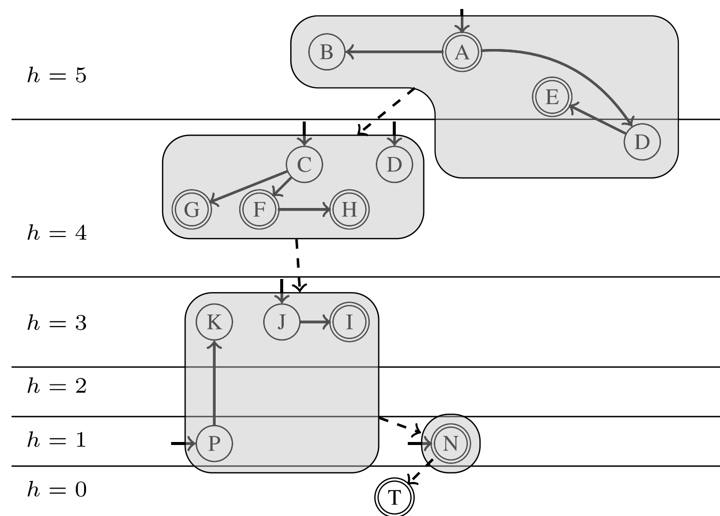

Figure 2: Search space topology from Heusner *et al.* [2017] for a toy domain, divided into benches connected via progress states. The progress states are indicated by double lines. Note that the progress states are actually the goal states for each bench.

is a set of predicate symbols (along with an arity); $\mathcal{A}$ is a set of action schemata...

A *task* Π of domain $\mathcal{D}$ is a tuple $\langle \mathcal{O}, \mathcal{P}, A, s_{\mathrm{I}}, \delta \rangle$, where $\mathcal{O}$ is a set of objects and $\mathcal{P}$ is a set of first-order predicates. A *fact* refers to a predicate $p \in \mathcal{P}$ with arity k grounded to $p(o_1, o_2, \ldots, o_k)$ with $o_i \in \mathcal{O}$. Let F be the set of all facts. Then, any $s \subseteq F$ is called a state, and the set of all states $S(\Pi)$ is called a *state space*. Moreover, $s_{\mathrm{I}} \in S(\Pi)$ is the *initial state* and δ is the *goal condition*, a first-order logical formula over $\mathcal{P}, \mathcal{C}$ and $\mathcal{O}$. All states $s \supseteq \delta$ are *goal states*, and the set of all goal states is denoted $S_G(\Pi)$. A is a set of *action schemas* that can be grounded using $\mathcal{O}$. We call grounded action schemas *actions*. An action a is a tuple $\langle pre, add, del \rangle$ with $pre, add, del \subseteq F$ and is associated with a cost $cost(a) \in \mathbb{R}_0^+$. Action a is applicable in state s if $pre \subseteq s$. Applying a in s, written as $s[\![a]\!]$, leads to the successor state $(s \setminus del) \cup add$. An action sequence $\pi = \langle a_1, a_2, \ldots, a_n \rangle$ is applicable in state s if every action a_i is applicable in the state $s[\![a_1]\!][\![a_2]\!][\![\ldots]\!][\![a_{i-1}]\!]$. The cost of an action sequence is the summed-up cost of its actions. A state s' is *reachable* from s if there is an applicable action sequence starting in s and ending in s'. The *reachable state space* $S_R \subseteq S$ is the set of all states reachable from s_{I}. An applicable action sequence starting in state s and ending in a goal state is called an s-*plan*. The objective in classical planning is to find an s_{I}-plan, i.e., a *plan* for the given task.

2.2 Description Logic

Description logic (DL) is a family of knowledge representation formalisms [Baader *et al.*, 2003] that use the notions of *concepts*, which are classes of objects that share some property, and *roles*, which are the relations between these objects. Interpreting the concepts and roles for a planning state yields a *denotation*, i.e., a set of objects $O \subseteq \mathcal{O}$ for a concept, and a set of object pairs $\{\langle o_1, o_2 \rangle, \langle o_3, o_4 \rangle, \ldots\} \subseteq \mathcal{O} \times \mathcal{O}$ for a role.

Concepts and roles are recursively defined and interpreted for a state $s \in S$. At its base are the *universal concept* $\top$ and the *bottom concept* $\bot$ with semantics

$$\top(s) = \mathcal{O} \qquad \text{and} \qquad \bot(s) = \emptyset,$$

as well as *atomic* concepts and roles. A atomic concept $C_{p,i}$ for a k-ary predicate $p \in \mathcal{P}$ and its i-th argument is interpreted in s as

$$C_{p,i}(s) = \{o_i \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}.$$

Accordingly, an atomic role $R_{p,i,j}$ for a k-ary predicate $p \in \mathcal{P}$ and its i-th and j-th arguments is interpreted as

$$R_{p,i,j}(s) = \{\langle o_i, o_j \rangle \mid \exists o_1, \ldots, o_k \text{ s.t. } p(o_1, \ldots, o_k) \in s\}$$

in s. Let X and X' be two concepts (respectively, two roles). They can be combined to form new concepts and roles via grammar rules. Examples are negation, union, and intersection, which are interpreted in a state s as

$$\begin{gathered}(\neg X)(s) = \mathcal{O} \setminus X(s) \quad \text{resp.} \quad (\neg X)(s) = \mathcal{O} \times \mathcal{O} \setminus X(s),\\ (X \sqcup X')(s) = X(s) \cup X'(s), \text{ and}\\ (X \sqcap X')(s) = X(s) \cap X'(s).\end{gathered}$$

We use the same grammar as Francès *et al.* [2021a]. For details, we refer to their extended paper [Francès *et al.*, 2021b].

We use two functions to convert denotations to integers. For a concept or role X, $|X(s)|$ is the *size* of the set $X(s)$. The concept distance $conceptdistance$ between concepts C_1 and C_2 over role R is the smallest $n \in \mathbb{N}_0$ with $x_0 \in C_1(s)$, $x_n \in C_2(s)$, and all $(x_i, x_{i+1}) \in R(s)$.

The complexity $\mathcal{K}(X)$ of the universal concept, the bottom concept, any atomic concept, and any atomic role is 1. The complexity of composed concepts and roles is defined as

$$\begin{aligned}\mathcal{K}(\neg X) &:= 1 + \mathcal{K}(X)\\ \mathcal{K}(X \sqcup X') = \mathcal{K}(X \sqcap X') &:= 1 + \mathcal{K}(X) + \mathcal{K}(X')\\ \mathcal{K}(|X|) &:= 1 + \mathcal{K}(X)\\ \mathcal{K}(conceptdistance(C_1, R, C_2)) &:= 1 + \mathcal{K}(C_1) + \mathcal{K}(R) + \mathcal{K}(C_2).\end{aligned}$$

2.3 State Space Topology/Progress States

Let Π be a planning problem with a state s. A heuristic $h : S \to \mathbb{R}_0^+ \cup \{\infty\}$ estimates the cost of an optimal s-plan. Let P be the set of all acyclic s-plans. The *high-water mark* of s is

$$hwm(s) = \min_{\pi \in P} \max_{a_i \in \pi} h(sa_1[\![\ldots]\!]a_i[\![])]\!].$$

Heusner *et al.* [2018] defined a state s as progress state iff its high-water mark is higher than the high-water mark of at least one of its successor states. Counterintuitively, this definition excludes goal states for goal-aware heuristics. We therefore adapt the definition as follows: a state s is a *progress state* iff $\delta \subseteq s$ or $h(s) > \min_{s' \in succ(()s)} hwm(s')$, where $succ(()s)$ is the set of all successor states of s and δ is the goal condition.

Let $\mathcal{X}()$ denote a state space. A bench b is a set of states $s \in \mathcal{X}()$. Let $\mathcal{B}$ denote the set of all benches of $\mathcal{X}()$. Intuitively, we would expect $\mathcal{B}$ to be a partitioning of $\mathcal{X}(,)$ but this is not the case. By the original definition, states can be in multiple benches.

For a bench $b \in \mathcal{B}$, *states*(b) denotes the states of b; *entry*(b) denotes the entry states of b; and *exit*(b) denotes the exit states of b. The level of a bench is denoted by *level*(b).

For each bench b, we calculate a formula ϕ_b, which identifies all the states of the bench. We exclude the exit states here, because they are also part of another bench as entry state and the correct subgoal is determined by the other bench. ϕ_b evaluates to true on a state $s \in \mathcal{X}()$ iff $s \in$ *states*$(b) \setminus$ *exit*(b). To learn ϕ_b, we take all states $s \in \mathcal{X}()$ and label them as true iff $s \in b$. Now, we can use any approach – e.g., our description logic and decision tree technique – to learn the formula. Unfortunately, the learned formula can contain errors.

In addition, we associate each bench b with a formula δ_b that evaluates to true on all states $s \in$ *exit*(b) and to false on all states $s \in$ *states*$(b) \setminus$ *exit*(b). To learn δ_b we use only the states $s \in$ *states*(b) and label the results as true iff $s \in$ *exit*(b).

2.4 Decision Trees

A binary decision tree is a machine learning model with a binary tree structure [Breiman *et al.*, 1984]. Let $\mathcal{C}$ be a set of classes and let F be a list of features. A decision tree assigns a class $c \in \mathcal{C}$ to a vector $v \in \mathbb{R}^F$. Each internal tree node n_I is associated with a feature $f(n_I) \in \{1, \ldots, F\}$ and threshold

$\tau(n_I) \in \mathbb{R}$. Each leaf node n_L is associated with a class $c(n_L) \in \mathcal{C}$. To assign a class to an input vector v, the decision tree is traversed from the root node to a leaf node. At every internal node n_I, if $v[f(n_I)] \leq \tau(n_I)$, then the traversal continues to the first child node, otherwise it continues to the second one. When a leaf node n_L is reached, the input is labeled as $c(n_L)$.

Decision trees are greedily constructed given some training data $\langle D, L \rangle$ with feature matrix $D \in \mathbb{R}^{M \times F}$ and the label vector $L \in \mathcal{C}^M$, where M is the number of training samples. Each node n is associated with a non-exclusive submatrix $D_n \in \mathbb{R}^{M' \times F}$ and $L_n \in \mathbb{R}^{M'}$. The root node is associated with the whole training data D and L, and is initially a leaf node. Leaf node n_L is associated with the most frequent class in L_{n_L}. During training, the algorithm chooses a leaf node n_L and searches through combinations of features f' and thresholds τ', which are used to group data points $\langle D_{n_L}[i], L_{n_L}[i] \rangle$ for $i \in \{1, ..., M'\}$ into two sets using test $D_{n_L}[i][f'] \leq \tau'$. The quality of the groups is evaluated using a metric (e.g., the Gini impurity Breiman *et al.* [1984]). The leaf is associated with a combination of the best split ($f(n_L) = f'$ and $\tau(n_L) = \tau'$), and two child leaves are added to it, one per data set split. This transforms n_L into an internal node.

The algorithm continues until all leaves contain only labels from the same class or a maximum tree depth is reached.

2.5 Learning and Exploiting Progress States in GBFS

Theoretical properties of optimal state-space search algorithms like A* or IDA* have been extensively studied and are comparatively well understood [Martelli, 1977; Pearl, 1984; Dechter and Pearl, 1985; Korf *et al.*, 2001; Helmert and Röger, 2008; Holte, 2010]. A corresponding theory for suboptimal search algorithms such as GBFS [Doran and Michie, 1966] has received growing attention only in the last few years [Wilt and Ruml, 2014, 2015, 2016; Heusner *et al.*, 2017, 2018].

The main insight of Heusner *et al.* [2017] is that every run of a GBFS can be partitioned into different episodes defined by so-called *high-water mark benches*, and the *state-space topology* can be partitioned in the same way. All states s on a bench share the same *high-water mark* value, which is the largest heuristic value that needs to be considered to reach a goal state from s. *Progress states* are states that must be expanded to reach the next high-water mark bench. Exploiting knowledge of high-water mark benches or progress states during search gives rise to many applications. The only known algorithm that computes high-water mark benches does so *a posteriori*, i.e., it computes the benches of a problem after a plan has been found [Heusner *et al.*, 2018]. At this point, the high-water mark information is not needed.

Inspired by Ståhlberg *et al.* [2021], who successfully learned to characterize unsolvable states using description logic Baader *et al.* [2003], we presented an approach that learns to characterize progress states. First, we considered the GRIPPER and MICONIC planning domains and verified that that the set of progress states for the h^+ search algorithm [Hoffmann and Nebel, 2001; Imai and Fukunaga, 2015] can be compactly represented with a description logic formula. Then, we presented a method to learn such formulas automatically for any path-independent heuristic.

In Heusner *et al.* [2017] we proposed the following pipeline: For a given domain and heuristic, we fully expand the reachable state spaces of several small tasks and annotate all states with their heuristic value. Using the heuristic values, we determine whether each state is a progress state. Next, we compute a set of description logic features and evaluate each on a subset of states. Then, we adopt a decision tree Breiman *et al.* [1984] learning algorithm to learn simple formulas over the description logic features in disjunctive normal form (DNF), which predicts whether a state is a progress state. Finally, we use our

formulas to break ties in a greedy best-first search, demonstrating a use case for the trained progress state classifier.

We evaluated our method using the h^+ and h^{FF} heuristics Hoffmann and Nebel [2001] and showed that our approach successfully learns useful formulas for identifing progress states. We observed a trade-off between the quality of the formulas and the time required to evaluate them. Most importantly, we showed that exploiting progress states is beneficial: it significantly reduces the number of expansions required to find a plan.

3 Road map

In this section, we present the planned road map of this research, including formal definitions, technical algorithmic steps, and learning steps. We start by formally defining *generalized subtask systems* and then describing our current relevant model of the *generalized BTS*. Next, we present the *learning model* and all necessary components needed for implementation as *formula to PDDL translation*. Finally, we present the search itself and how it works in practice after all steps have been properly implemented and joined together. Practically, our research plan must achieve each and every one of the following intermediate steps:

1. Learning Subtask Systems
   (a) Generate all benches for each task.
   (b) Generate description logic features that are able to (perfectly) describe all states on all benches and all exit states of all benches.
   (c) For each bench, learn a formula that describes its exit states and separate the exit states from all other states reachable from that bench.
   (d) Merge bench A into bench B if the goal formula of B is also perfect for A.
   (e) For each bench, learn a formula that describes a state in the bench to separate the member states from all other states.
2. Bench Walking: Perform an intermediate search between the learned benches.

3.1 Generalized Subtask Systems

This proposal's main idea is to decompose planning problems into subtasks. To do so, we need to formally define subtasks, generalized subtasks, and a system of subtasks. In the described case, instantiating a **generalized** subtask with a given state of a given instance yields a subtask, which is a proper PDDL task.

Definition 1 (Generalized Subtask)**.** *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ be a planning domain and let $\mathcal{P}_G = \{p_G \mid p \in \mathcal{P}\}$ and $\mathcal{P}_I = \{p_I \mid p \in \mathcal{P}\}$. A generalized subtask $\mathcal{G}$ for $\mathcal{D}$ is a tuple $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle$, where*

- *$\phi_{membership}$ is a first-order logic formula over $\mathcal{P}, \mathcal{P}_G, \mathcal{C}$;*
- *ϕ_{goal} is a first-order logic formula over $\mathcal{P}, \mathcal{P}_I, \mathcal{P}_G, \mathcal{C}$.*

Definition 2 (Subtask)**.** *Let s be a state of planning task $\langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_{\mathrm{I}}, \delta \rangle$ of domain $\mathcal{D}$, where δ is a conjunction over F, and let $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle$ be a generalized subtask for $\mathcal{D}$. The (instantiated)* subtask *of s is the planning task $\Pi_{\mathcal{G}}(s) = \langle \mathcal{O}, \mathcal{P} \cup \mathcal{P}_I \cup \mathcal{P}_G, \mathcal{A}, s'_{\mathrm{I}}, \phi_{goal} \rangle$, where*

$$s'_{\mathrm{I}} = s \cup \{f_I \mid f \in s\} \cup \{f_G \mid f \in \delta\}.$$

Definition 3 (GeneralizedSubtask System)**.** *A set of generalized subtasks $\mathcal{S}$ for domain $\mathcal{D}$ is a* subtask system *for a set of tasks $\mathcal{T} = \{\Pi_1, \dots, \Pi_n\}$ of $\mathcal{D}$ if for each $\Pi \in \mathcal{T}$, the following hold:*

- *For a state $s \in S(\Pi) \setminus S_G(\Pi)$, there is exactly one generalized subtask $\mathcal{G} = \langle \phi_{membership}, \phi_{goal} \rangle \in \mathcal{S}$ such that $s \models \phi_{membership}$, which we denote $\mathcal{S}(s)$.*
- *Let $s \in S(\Pi) \setminus S_G(\Pi)$ be a state and $\pi = \pi_0 \circ \dots \circ \pi_n$ ($\circ$ is the concatentation of plans) be any sequence of operators such that π_0 is a plan for $\Pi_{\mathcal{S}(s)}(s)$ and π_i is a plan for $\Pi_{\mathcal{S}(s[\![\pi_0 \circ \dots \circ \pi_{i-1}]\!])}(s[\![\pi_0 \circ \dots \circ \pi_{i-1}]\!])$. Then, π is a (global) s-plan.*

If these properties hold for each task of $\mathcal{D}$, we say that $\mathcal{S}$ is a generalized subtask system.

3.2 Generalized BTS

Definition 4 (Generalized High-Water Mark Bench)**.** *Let $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ be a planning domain. A generalized high-water mark bench $\mathcal{G}$ for $\mathcal{D}$ is a tuple $\mathcal{G} = \langle \phi_{membership}, \phi_{goal}, \phi_{avoid} \rangle$, where*

- *$\phi_{membership}$ is a first-order logic formula over the set of predicate schemata and the set of constants of $\mathcal{D}$;*
- *ϕ_{goal} and ϕ_{avoid} are first-order logic formulas over $\mathcal{P}, \mathcal{P}_I, \mathcal{C}$, where $\mathcal{P}_I = \{p_I \mid p \in \mathcal{P}\}$.*

Let $f = p(o_1, ..., o_n)$ be a fact. We then define f' as $p'(o_1, ..., o_n)$ (necessary for init below)

Definition 5 (Bench Task)**.** *Let s be a state of planning task $\langle \mathcal{O}, \mathcal{P}, \mathcal{A}, s_{\mathrm{I}}, \delta \rangle$ and $\mathcal{G} = \langle \phi_{membership}, \phi_{goal}, \phi_{avoid} \rangle$ be a generalized bench. The* bench task *of s is the planning task $\Pi(s) = \langle \mathcal{O}, \mathcal{P}', \mathcal{A}', s'_{\mathrm{I}}, \delta' \rangle$, where*

$$\begin{aligned} \mathcal{P}' &= \mathcal{P} \cup \{p_I \mid p \in \mathcal{P}\} \\ \mathcal{A}' &= \{a' \mid a \in \mathcal{A}\} \\ s'_{\mathrm{I}} &= s \cup \{f_I \mid f \in s\} \\ pre(a') &= \neg\phi_{avoid} \wedge pre(a) \\ \delta' &= \phi_{goal} \wedge \neg\phi_{avoid} \end{aligned}$$

Definition 6 (Generalized BTS)**.** *A graph $\mathfrak{G} = \langle V, E \rangle$ with a set of generalized benches V and a set of edges between those benches E is a generalized bench transition system for a set of tasks $\mathcal{T} = \{\Pi_1, \dots, \Pi_n\}$ if for each $\Pi \in \mathcal{T}$, it holds that*

- *for all $s \in S \setminus S_G$, there is exactly one generalized bench $\mathcal{G} = \langle \phi_{membership}, \phi_{goal}, \phi_{avoid} \rangle$ such that $s \models \phi_{membership}$; let $\mathcal{G}(s)$ be that unique bench;*
- *for all $s \in S \setminus S_G$, all sequences of operators $\pi = \pi_0 \circ \cdots \circ \pi_n$ ($\circ$ is the concatentation of plans) are (global) s-plans, where π_0 is a plan for $\Pi(s)$ and π_i is a plan for $\Pi(s[\![\pi_0 \circ \cdots \circ \pi_{i-1}]\!])$.*

If these properties hold for each task of $\mathcal{D}$, we say that $\mathfrak{G}$ is a generalized BTS.

A generalized BTS can be used to decompose a planning task by identifying the bench for the current state s (starting with s_{I}), constructing and solving the corresponding bench task, and continuing with the resulting goal state until a (global) goal is reached.

Theorem 1. *A generalized bench transition (plus some restrictions, e.g., perfect classification of data) is a subtask system.*

Note that if the input is such that all domain features are included (make more formal and correct!), a generalized BTS is a generalized subtask system. (in our experiments, it generalized to all instances of a domain)

3.3 Learning Subtask Systems

Heusner *et al.* [2017] introduced high-water mark benches, which enable a run of GBFSs to be decomposed into several subsearches. So far, this decomposition has not been applied in practical tasks, as high-water mark benches can only be generated *a posteriori*, i.e., after a planning task has been solved. Here, we exploit the idea by learning a generalized subtask from benches that are generated on a set of tasks $\{\Pi_1, \ldots, \Pi_n\}$ from domain $\mathcal{D} = \langle \mathcal{P}, \mathcal{A}, \mathcal{C} \rangle$ that can be solved. We hope to show experimentally that the learned subtask systems generalize to large instances of the same domain in the empirical evaluation of the research. For now, we formally define such systems. Figure 3 is an example of a generalization over small instances; we plan to add more details as the research progresses.

Our approach consists of the following steps:

1. **Generate Benches:** generate all benches for each task;
2. **Generate Features:** generate the description logic features;
3. **Learn Goals:** for each bench, learn a formula that describes its exit states;
4. **Merge Benches:** merge bench A into bench B if the goal formula of B is also perfect for A;
5. **Learn Memberships:** for each bench, learn a formula that describes the states in the bench;
6. **Iterate:** if any learned formula is imperfect, return to description logic feature generation (step 2) with a higher complexity limit and repeat.

In the following, we describe these steps in more detail.

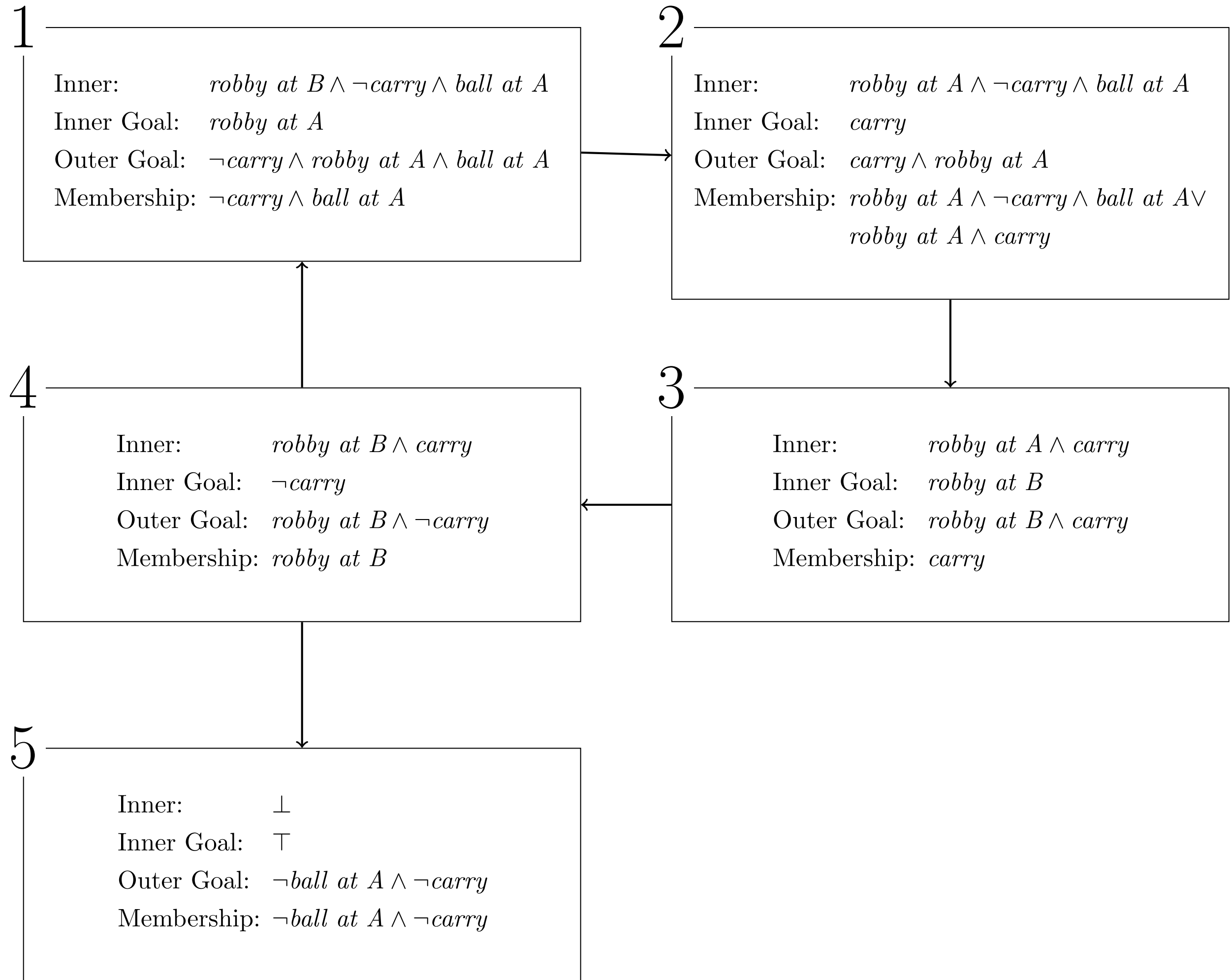

Figure 3: GBTS for a toy domain, a gripper with one arm that needs to move balls between rooms using the h^* heuristic. In each box, a formula representing the relevant states is presented.

Generating Benches: For each input task Π_i, we label the states of a planning instance with respect to a heuristic h as progress or non-progress states. To do this, we first expand the reachable state space. This includes states reachable only on paths through goal states. For each state s, we track its predecessors $pred(s)$, successors $succ(s)$, and heuristic estimate $h(s)$.

In a second iteration, we compute the high-water marks (hwm). Initially, we set $hwm(g) = h(g)$ for all goal states g and regard $hwm(s)$ as undefined for all other states s. We maintain an open list of states ordered by high-water mark values that initially contain all goal states. Upon retrieving a state s from the open list, we insert all its predecessors $p \in pred(s)$ with undefined high-water marks into the open list with a high-water mark value of $hwm(p) = \max(h(p), hwm(s))$. The algorithm guarantees that a state is only inserted once into the open list, namely after its successor with the lowest high-water mark value has been retrieved from the open list. When this process terminates, only unsolvable states have an undefined high-water mark, which we treat as ∞ from here on.

As a result, for each input task Π_i, we have constructed a BTS $\mathfrak{B}_i = \langle V_i, E_i \rangle$, where V_i is a partition over $S_i \backslash S_G$ and E_i are the edges between the benches. Let $\mathfrak{B} = \langle V, E \rangle$ such that $V = \bigcup_i V_i$, $E = \bigcup_i E_i$, and $S = \bigcup_i S_i$ are all goal state S. See Algorithm **??** for more details.

Generating Features: In the second step, we generate the features used in our formulas to describe the membership and goals of the subtasks. The feature generation is an iterative process and is parameterized with a complexity limit ℓ. In iteration j, only features of complexity j are generated. In the first iteration, we only consider atomic concepts and roles over the predicates $\mathcal{P}$. In every succeeding iteration j, we additionally consider all features obtained by combining features from iteration $j-1$ with a rule from Section 2.2 if their combined complexity is equal to j. We prune any new feature f if its denotation is the same on all states $s \in S$ as the denotation of a previously considered feature. We stop the feature generation once we reach the complexity limit. Let $\mathcal{F}_\ell$ denote the set of considered features.

Learning Goals: For each bench $v \in V$, we compute a formula ϕ_{goal} that describes all exit states of the bench. Those exit states become subgoals of subtask. We generate the training samples by enumerating all pairs of states $\langle s, r \rangle$ such that $s \in v$ and r is reachable from s without traversing over an exit state. We label a pair $\langle s, r \rangle$ positively if r is an exit state and negatively otherwise. Furthermore, for each feature $f \in \mathcal{F}_\ell$ we compute whether its denotations on s and r as well as the set differences between those denotations are empty, i.e., $|f(s)| > 0, |f(r)| > 0, |f(s) \backslash f(r)| > 0, |f(r) \backslash f(s)| > 0$. As Ferber *et al.* [2022], we train a decision tree on those samples, extract a formula in disjunctive normal form from the tree, and simplify it using SymPy citation.

There is no guarantee that the generated features are sufficient to perfectly separate exit states from non-exit states because we might need a feature with a complexity higher than the complexity limit ℓ. In this case, we have to increase ℓ and start again from step 2.

Merging Benches: Let $v, v' \in V$ be two benches with goal formulas $\phi_{goal}(v)$ and $\phi_{goal}(v')$, respectively.

If these two benches represent the same subtask, then we replace $\phi_{goal}(v)$ with $\phi_{goal}(v')$ iff $\phi_{goal}(v')$ is more general than $\phi_{goal}(v)$ (same for $\phi_{avoid}(v)$).

Let $\mathfrak{B}' = \langle V', E' \rangle$ such that

- V' is the minimal set for which for all $v' \in V'$ and $v_1, v_2 \in V$: $v_1 \subseteq v'$ and $v_2 \subseteq v'$ iff $\phi_{goal}(v_1) = \phi_{goal}(v_2)$ and $\phi_{avoid}(v_2) = \phi_{avoid}(v_2)$;
- $(v, v') \in E'$ iff exists $v_1 \subseteq v$, $v_2 \subseteq v'$ such that $(v_1, v_2) \in E$ (and E' is the minimal set).

Learning Membership: In this step, we learn formulas for $\phi_{membership}(v)$ for each $v \in V'$.

Theorem 2. *What we learn (under some assumptions) is a subtask system.*

Note that if the input is such that all domain features are included (make more formal and correct!), we learn a **generalized** subtask system. (in our experiments, it generalizes to all instances of a domain)

Iteration:

3.4 Bench Walking

Baader *et al.* [2003] show any description logic (DL) concept/role can be converted to first order logic (FOL). Any FOL is easily written in PDDL. To support this end, one of the side-effects of this research is to support the translation from DL to PDDL. This is especially necessary to switch from the learned "exit" progress states represented as DL formulas into subgoals represented as PDDL, which provide the syntax and semantics needed in Fast Downward planning systems Helmert [2006].

In Algorithm 1, we modify the given task Π for the subsearches. Thus, we store the original goal of task Π (line 2). Furthermore, we have a stack that keeps track of the plans of the previous subsearches; we also have a stack that keeps track of the alternative bench options for each subsearch so far; lastly, we have a stack for the initial state of the subsearches (lines 3–5). To detect whether we are traversing in a circle, for all subsearches, we remember their initial state and bench (line 6).

The first step of our bench walker is to identify the initial state and bench to use for the next subsearch. In the simplest case, we have the current initial state s'_I and identify exactly one bench b that has s'_I as an inner state (lines 14 & 22), and that we have not traversed in a circle (line 27). Then, we execute a subsearch for the pair $\langle s'_\mathrm{I}, b\rangle$ (line 28).

In the second step, we set up and execute the subsearch (lines 30–34). We first update the initial state of the task Π to s'_I (line 31). Then, we construct the next subgoal from the outer goal formula of the bench (line 32). The inner goal formula incorrectly assumes that the subsearch visits only member states of the current bench. Thus, it learns a simpler formula, which falsely identifies non-member states as the goal. Furthermore, the additional information in the outer goal formula improves the guidance of the heuristic.

Example 1 (Superiority of the Outer Goal Formula over the Inner Goal Formula)**.** *Figure 3 shows the GBTS for GripperOne under the h^+ heuristic. Let s be a state where there are balls in room A, the robot is in room B, and the robot carries a ball. This state belongs to bench A. The intention of this bench is to drop the ball first in room B and then to move to room A. The inner goal formula only requires that the robot be in room A. A GBFS on this goal ends in a state where the robot is in room A and is still carrying a ball. This plan produces the opposite of progress. The outer goal formula also says that the robot is* not *carrying a ball. As a consequence, the robot drops its ball either in the current room B and then moves to room A or it moves to room A and drops the ball there. Both resulting states are members of bench A. The first state is actual progress. The second state is again a step backwards.*

This example shows that even with the better guidance of the outer goal formula, a subsearch can reverse progress. This is because multiple concrete benches $\hat{b}_1, \ldots, \hat{b}_n$ are merged into the current generalized bench b. As long as the subsearch is allowed to visit *any* state, we could start in a state of the concrete bench $\hat{b}_n$ and end in a goal state of another concrete bench $\hat{b}_1$. To prevent this, we should use an *avoid condition* Steinmetz *et al.* [2022] using the negated membership formula of b (line 33). During the search, we prune all states that satisfy the avoid condition. In other words, the search cannot leave the member states of the current bench. *This issue still persists if a GBTS has generalized benches with self-loops.*

4 Experiments

As part of this research, we aim to evaluate the technique empirically. For that, we will generate several experiments to compare our technique to the state-of-the-art using several metrics such as expansion rate, run time, path size, and correctness. The main challenge is to learn the first order formulas for the benches, and hence we will develop a procedure for the learning phase similar to the one from Drexler *et al.* [2022]. We will add additional small instances to the training set to enable the construction of a comprehensive formula. Therefore, the first experiment will be to investigate the learning rate. Our theory allows arbitrary first-order logic formulas, and the set of DL rules used to generate features determines which fraction of the FOL our learned formulas can come from. It is also possible that we cannot learn a perfect formula if our DL is not sufficiently expressive.

Take the Gripper domain with four balls as an example. If the robot is in the origin room with a single ball left, then the goal formula could be *empty origin room*. If there are more balls left, the learned formula could be *one less ball in the origin room*. This happens because the decision tree greedily learns a formula. If two features split the data equally well, then less complex features are preferred. For some benches, a non-generalizing formula could be learned because that formula is cheaper. On a theoretical basis, this is no problem. If the number of balls increases, there is one point at which the overspecific formulas become more expensive than the generalized ones, and the system will learn the generalized one. Thus, for any domain, there will only be a finite number of overspecific benches, and afterward, the generalization will works. in practice, we do not want to work with such a large state space. Thus, we postprocess the formulas: after relearning the formulas, all formulas are evaluated on all benches. Whenever a perfect formula $f1$ also fits another bench b perfectly, then $f1$ becomes the formula for b. Therefore, after making sure we can really learn the formulas, the next experiment is to ensure the intermediate plans merge together into a correct plan.

Finally, after passing those two sanity tests, we would like to compare them to the state-of-the-art methods from Drexler *et al.* [2022] and test our framework on the domains presented in that paper. The performance metrics of interest are obviously related to the efficiency of the search, e.g., a smaller expansion rate and shorter run time. However, trade-offs should be considered and discussed as part of the analysis. In our previous work, we noted that the expansion rate decreased but the run time did not change. This phenomenon is due to the intensive computation power needed for evaluating large formulas, which in the end affected the run time and reduced the efficiency of the method for some domains. Addressing this issue will be a task for the proposed research.

We already have some preliminary code and successful preliminary results for some domains. However, learning the right formula for all domains could be challenging. Because the research is carefully structured, we will gain a deeper understanding of the problem, even if the results are not as hoped.

Algorithm 1: Bench Walker($\Pi, s_{\mathrm{I}}, \delta, \mathfrak{G}, h$). For a given planning task Π with an initial state s_{I} and a goal δ, a generalized bench transition system $\mathfrak{G}$, and a heuristic h returns a plan.

1 $\delta_{global} \leftarrow \Pi.\delta$;
2 $\pi_{global} \leftarrow Stack()$;
3 $choices \leftarrow Stack()$;
4 $inits \leftarrow Stack()$;
5 $closed \leftarrow \emptyset$;
6 $inits$.push($\Pi.s_{\mathrm{I}}$);
7 $identify \leftarrow True$;
8 **while** $inits$.*top()* $\not\supseteq \delta_{global}$ **do**
9     $s'_{\mathrm{I}} \leftarrow inits$.top();
10   // Find bench for next subsearch;
11   **if** $identify$ **then**
12         $B \leftarrow$identify_benches(s'_{I}, $\mathfrak{G}$);
13         $choices$.push(B);
14   **while** $choices$.*top().empty()* **do**
15         $choices$.pop();
16   **if** $choices.empty()$ **then**
17   **return** unsolvable;
18   π_{global}.pop();
19         $inits$.pop();
20         $s'_{\mathrm{I}} \leftarrow inits$.top();
21     $b \leftarrow choices$.top().pop();
22   **if** $\langle s'_{\mathrm{I}}, b\rangle \in closed$ **then**
23         $identify \leftarrow False$;
24   Continue;
25   **else**
26         $closed \leftarrow closed \cup \{\langle s'_{\mathrm{I}}, b\rangle\}$;
27   // Setup and execute subsearch;
28     $\Pi.s_{\mathrm{I}} \leftarrow s'_{\mathrm{I}}$;
29     $\Pi.\delta \leftarrow$ to_pddl(b.outer_goal);
30     $avoid \leftarrow$ to_pddl($\neg b$.membership);
31     $\pi \leftarrow$ GBFS(Π, $avoid$);
32   **if** π **then**
33   π_{global}.push(π);
34         $inits$.push($s'_{\mathrm{I}}[\![\pi]\!]$);
35         $identify \leftarrow True$;
36   **else**
37         $identify \leftarrow False$;
38 **return** concatenate(π_{global});

References

Franz Baader, Diego Calvanese, Deborah L. McGuinness, Daniele Nardi, and Peter F. Patel-Schneider, editors. *The Description Logic Handbook: Theory, Implementation and Applications*. Cambridge University Press, 2003.

Leo Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. *Classification and Regression Trees*. Wadsworth, 1984.

Rina Dechter and Judea Pearl. Generalized best-first search strategies and the optimality of A^*. *JACM*, 32(3):505–536, 1985.

James E. Doran and Donald Michie. Experiments with the graph traverser program. *Proceedings of the Royal Society A*, 294:235–259, 1966.

*Dominik Drexler, Jendrik Seipp, and Hector Geffner. Learning sketches for decomposing planning problems into subproblems of bounded width. In *Proc. ICAPS 2022*, pages 62–70, 2022.

Patrick Ferber, Liat Cohen, Jendrik Seipp, and Thomas Keller. Learning and exploiting progress states in greedy best-first search. In *Proc. IJCAI 2022*, pages 4740–4746, 2022.

Guillem Francès, Blai Bonet, and Hector Geffner. Learning general planning policies from small examples without supervision. In *Proc. AAAI 2021*, pages 11801–11808, 2021.

Guillem Francès, Blai Bonet, and Hector Geffner. Learning general policies from small examples without supervision. arXiv:2101.00692 [cs.AI], 2021.

Malte Helmert and Gabriele Röger. How good is almost perfect? In *Proc. AAAI 2008*, pages 944–949, 2008.

Malte Helmert. The Fast Downward planning system. *JAIR*, 26:191–246, 2006.

*Manuel Heusner, Thomas Keller, and Malte Helmert. Understanding the search behaviour of greedy best-first search. In *Proc. SoCS 2017*, pages 47–55, 2017.

Manuel Heusner, Thomas Keller, and Malte Helmert. Best-case and worst-case behavior of greedy best-first search. In *Proc. IJCAI 2018*, pages 1463–1470, 2018.

Jörg Hoffmann and Bernhard Nebel. The FF planning system: Fast plan generation through heuristic search. *JAIR*, 14:253–302, 2001.

Robert C. Holte. Common misconceptions concerning heuristic search. In *Proc. SoCS 2010*, pages 46–51, 2010.

Tatsuya Imai and Alex Fukunaga. On a practical, integer-linear programming model for delete-free tasks and its use as a heuristic for cost-optimal planning. *JAIR*, 54:631–677, 2015.

Richard E. Korf, Michael Reid, and Stefan Edelkamp. Time complexity of iterative-deepening A^*. *AIJ*, 129:199–218, 2001.

Alberto Martelli. On the complexity of admissible search algorithms. *AIJ*, 8:1–13, 1977.

Drew McDermott, Malik Ghallab, Adele Howe, Craig Knoblock, Ashwin Ram, Manuela Veloso, Daniel Weld, and David Wilkins. PDDL – The Planning Domain Definition Language – Version 1.2. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control, Yale University, 1998.

Judea Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley, 1984.

Simon Ståhlberg, Guillem Francès, and Jendrik Seipp. Learning generalized unsolvability heuristics for classical planning. In *Proc. IJCAI 2021*, pages 4175–4181, 2021.

Marcel Steinmetz, Jörg Hoffmann, Alisa Kovtunova, and Stefan Borgwardt. Classical planning with avoid conditions. In *Proc. AAAI 2022*, pages 9944–9952, 2022.

Christopher Wilt and Wheeler Ruml. Speedy versus greedy search. In *Proc. SoCS 2014*, pages 184–192, 2014.

Christopher Wilt and Wheeler Ruml. Building a heuristic for greedy search. In *Proc. SoCS 2015*, pages 131–139, 2015.

Christopher Wilt and Wheeler Ruml. Effective heuristics for suboptimal best-first search. *JAIR*, 57:273–306, 2016.

A Formula to PDDL

We use some formulas during a search. Thus, we must convert them to *PDDL*. Here, we show how to convert a description logic feature to a first-order logic (*FOL*). Rephrasing them from *FOL* to *PDDL* is then trivial. Let $C_{p,i}$ be an atomic concept for the predicate symbol p with $arity(p) = k$. Feature $|C^s_{p,i}| > 0$ expresses that the denotation of $C_{p,i}$ for state s contains at least one element.

$$\begin{aligned}|C^s_{p,i}| > 0 = &|\{x\ |\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : \\ &\quad p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k) \in s\}| > 0 \\ \rightarrow\ &\exists x \exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : \\ &\quad p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k)\end{aligned}$$

The condition to have at least one x is translated into the first existential quantifier. The quantifiers introduced by the atomic concept follow afterwards, and the actual condition comes last. For an atomic role, the schema is the same. Let $R_{p,i,j}$ be an atomic role with $i < j$.

$$\begin{aligned}|R^s_{p,i,j}| > 0 = &|\{\langle x, y\rangle\ | \\ &\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k : \\ &p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k) \in s\}| > 0 \\ \rightarrow &\exists x, y \exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k : \\ &p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k)\end{aligned}$$

The adaption for an atomic role with $i > j$ is trivial. For the translation in the general case, we introduce three recursive functions: f translates a feature, f_C translates a concept and receives as an additional input the name of the variable bound on the outside (the red part in the atomic concept example above). f_R translates a role and receives as additional input the names of the two variables bound on the outside (red part in the atomic role example above). For every rule that constructs a complex concept, f_C requires an overload. For every rule that constructs a complex role, f_R requires an overload. Let X be either a concept or a role, D, E be concepts, and S, T be roles. Let $C_{p,i}$ be an atomic concept, $R_{p,i,j}$ be an atomic role, and n be the number of objects in the universe. Then, we recursively define f, f_C, f_R as follows:

$$
\begin{aligned}
f(\neg|X^s| > 0) &= \neg f(|X^s| > 0) \\
f(|C^s| > 0) &= (\exists x : f_C(C^s, x)) \\
f(|R^s| > 0) &= (\exists x, y : f_R(R^s, x, y))
\end{aligned}
$$

$$
\begin{aligned}
f_C(C^s_{p,i}, x) &= (\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_k : \\
&\qquad p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_k)) \\
f_C(\neg D, x) &= \neg(f_C(D, x)) \\
f_C(D^s \sqcap E^s, x) &= (f_C(D, x) \wedge f_C(E, x)) \\
f_C(D^s \sqcup E^s, x) &= (f_C(D, x) \vee f_C(E, x)) \\
f_C((\exists S.D)^s, x) &= (\exists v : f_R(S^s, x, v) \wedge f_C(D^s, v)) \\
f_C((\forall S.D)^s, x) &= (\forall v : f_R(S^s, x, v) \implies f_C(D^s, v))
\end{aligned}
$$

$$
\begin{aligned}
f_R(R^s_{p,i,j}, x, y) &= (\exists v_1, \ldots, v_{i-1}, v_{i+1}, \ldots, v_{j-1}, v_{j+1}, \ldots, v_k : \\
&\qquad p(v_1, \ldots, v_{i-1}, x, v_{i+1}, \ldots, v_{j-1}, y, v_{j+1}, v_k)) \\
f_R(\neg S, x, y) &= \neg(f_R(S, x, y)) \\
f_R(S^+, x, y) &= (x, y) \in S \vee \exists v_1 : (x, v_1) \in S \wedge \\
&\qquad ((v_1, y) \in S \vee \exists v_2 : (v_1, v_2) \in S \wedge \\
&\qquad ((v_2, y) \in S \vee \exists v_3 : (v_2, v_3) \in S \wedge \\
&\qquad ((v_3, y) \in S \vee \exists v_4 : (v_3, v_4) \in S \wedge \\
&\qquad \ldots \\
&\qquad ((v_{n-2}, y) \in S \vee (v_{n-2}, v_{n-1}) \in S \wedge \\
&\qquad (v_{n-1}, y) \in S) \ldots))) \\
f_R(S^*, x, y) &= (x = y) \vee f_R(S^+, x, y)?
\end{aligned}
$$

A.1 Transitive Closure

Computing the transitive closure not easily achieved in PDDL, as the following suggests.

Naive approach.

$$
\begin{aligned}
TC(c) = \exists x, y :& (x, y) \in R \\
& \vee \exists v_1 : (x, v_1) \in R \wedge (v_1, y) \in R \\
& \vee \exists v_1, v_2 : (x, v_1) \in R \wedge (v_1, v_2) \in R \wedge (v2, y) \in R \\
& \ldots \\
& \vee \exists v_1, v_2, \ldots, v_{n-1} : (v, v_1) \in R \wedge \ldots, (v_{n-1}, y) \in R
\end{aligned}
$$

Number of Operators:

$$1 + \sum_{i=1}^{n-1} 2i + 1 = 1 + n * (n-1) + (n-1) = n^2$$

More sophisticated approach.

$$\begin{aligned}
TC(c) = \exists x, y :& \\
&(x, y) \in R \\
&\vee \exists v_1 : (x, v_1) \in R \wedge \\
&\quad ((v_1, y) \in R \\
&\quad \vee \exists v_2 : (v_1, v_2) \in R \wedge (\\
&\qquad (v2, y) \in R \\
&\qquad \vee \exists v_3 : ((v_2, v_3) \in R \wedge (\\
&\qquad\quad (v_3, y) \in R) \\
&\qquad\quad \vee \exists \ldots \wedge (\\
&\qquad\qquad (v_{n-2}, y) \in R \\
&\qquad\qquad \vee (v_{n-2}, v_{n-1}) \in R \wedge (v_{n-1}, y) \in R))))
\end{aligned}$$

Number of Operators:

$$2 * (n-1) + 1 = 2n - 1$$

If R is an atomic role (e.g., $r_{primitive(link,0,1)}$), then we can just replace $(x, y) \in R$ by $link(x, y)$. If it is a complex role, then we can easily construct the PDDL formula for it using the procedures described here. However, inserting this formula at every point where R is used means we compute the formulas from scratch every time. Instead, the formula should be constructed and stored in an axiom A. Then, the fast downward step computes role R once and stores it in axiom A. Afterwards, A can be used as an atomic role, i.e., $A(x, y)$.

A.2 Caching

If the formula is evaluated at every bottleneck **and some concepts appear mutliple times in the overall concept**, then we can construct an axiom for that subconcept and evaluate it only once.