Planning as Theorem Proving with Heuristics

Mikhail Soutchanski[1], Ryan Young[1]

[1] *Toronto Metropolitan University, 245 Church St, ENG281, Toronto, ON, M5B 2K3, Canada*
https://www.cs.torontomu.ca/mes

Abstract

We explore a deductive approach to planning. We have developed a Theorem Proving Lifted Heuristic (TPLH) planner that searches for a plan in a tree of situations using the A* search algorithm. It is controlled by a delete relaxation-based domain independent heuristic. First, we compare a baseline version of TPLH with Fast Downward (FD) and Best First Width Search (BFWS) planners over several standard benchmarks. Since our implementation is not optimized, TPLH is slower than FD and BFWS. But it explores fewer states, sometimes it computes shorter plans, and this results in a comparable IPC scores for several domains. Next, we consider another version of TPLH that discards previously visited states, and that can do greedy search and/or use two priority queues. We determine experimentally the best configuration of the second version of TPLH that outperforms the baseline version. The IPC scores based on plan length and the number of visits are used as metrics for comparison. Thus, we show that the study of deductive lifted heuristic planning is a productive research direction.

1. Introduction

In modern automated (common-sense) AI planning, the instances of the planning problem are usually solved with domain-independent heuristics [6, 7] in a single model of a discrete transition system using model-based approaches. But there are potential advantages to theorem-proving based planning over the single model approach, e.g., the former can operate even if there are many different infinite logical models [5] for an application domain, or if an initial theory is incomplete. The single model approach rely on both the Closed World Assumption (CWA) and the Domain Closure Assumption (DCA) [30, 31, 32]. The main reason for relying on the (unrealistic) DCA in the single model approach is the need for instantiating the transition system before search starts. But the DCA can be (potentially) avoided in a theorem-proving based planner using progression in local effect action theories [22] with an incomplete initial theory (no CWA) [14].

In this paper, we revisit the deductive approach to planning despite a common incorrect belief that efficient deductive planning in situation calculus (SC) is impossible. In particular, we explore how to design a new SC-based Theorem Proving Lifted Heuristic (TPLH) planner that builds on [37]. It is “lifted" in the sense that it works with action schemata at run-time, rather than with actions instantiated before planning starts. Our planner does forward search over a situation tree, in contrast to other planners that usually search over a state space. Moreover, it makes use of a domain independent heuristic. To the best of our knowledge this is the first ever deductive planner to incorporate both of these features. This is our main contribution. Essentially, TPLH planner

IPS-RCRA-SPIRIT 2023: 11th Italian Workshop on Planning and Scheduling, 30th RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy. November 7-9th, 2023, Rome, Italy [2].

CEUR Workshop Proceedings (CEUR-WS.org)

provides heuristic control over resolution, but in [8, 9] control was not anticipated. The current version of TPLH works with a domain independent delete relaxation heuristic inspired by the FF planner [13, 3], but any other domain independent heuristics can be implemented as well.

We start with a review of SC, then we explain how our TPLH planner can be developed from the first principles as a search over the situation tree. To facilitate an implementation, our TPLH planner is implemented in PROLOG under the usual CWA and DCA. A more general implementation is left to future work. We present an experimental comparison of the baseline version of TPLH with the recent version of FastDownward (FD) planner [11, 36, 12] and Best First Width Search (BFWS) planner [18, 19, 20] on a set of the usual PDDL [10] benchmarks. We show our new improved implementation of FF is more informative than the original version from [13], since our version guides search better. We also explore a few variations of TPLH that do greedy best first search, and use extra priority queues. We determine experimentally the most promising configuration that outperforms the baseline version. Finally, we discuss future research directions and then conclude.

2. Background

We assume that the readers are familiar with PDDL [10]. Appendix 1 includes the well-known BlocksWorld domain formulated in PDDL. We note that PDDL and the situation calculus representations of the planning domains are somewhat complementary in the sense that PDDL formulations are action-centric, while situation calculus representations are fluent-centric.

The situation calculus (SC) is a logical approach to representation and reasoning about actions and their effects. It was introduced in [25, 26] to capture common sense reasoning about the actions and events that can change properties of the world and mental states of the agents. SC was refined by Reiter [33, 35] who introduced the Basic Action Theory (BAT). Unlike the notion of state that is common in model-based planning, SC relies on situation, namely a sequence of actions, which is a concise symbolic representation and a convenient proxy for the state in the cases when all actions are deterministic [15, 16]. We use variables s, s', s_1, s_2 for situations, variables a, a' for actions, and $\bar{x}, \bar{y}$ for tuples of object variables. The constant S_0 represents the initial situation, and the successor function $do : action \times situation \mapsto situation$, e.g., $do(a, s)$, denotes situation that results from doing action a in previous situation s. The terms σ, σ' denote situation terms, and $A_i(\bar{x})$, or $\alpha, \alpha_1, \alpha_2, \alpha'$, represent action functions and action terms, respectively. The shorthand $do([\alpha_1, \cdots, \alpha_n], S_0))$ represents situation $do(\alpha_n, do(\cdots, do(\alpha_1, S_0) \cdots))$ resulting from execution of actions $\alpha_1, \cdots, \alpha_n$ in S_0. The relation $\sigma \sqsubset \sigma'$ between situation terms σ and σ' means that σ is an initial sub-sequence of σ'. Any predicate symbol $F(\bar{x}, s)$ with exactly one situation argument s and possibly a tuple of object arguments $\bar{x}$ is called a (relational) fluent. Without loss of generality, we consider only relational fluents in this paper, but the language of SC can also include functional fluents. A first order logic (FO) formula $\psi(s)$ composed from fluents, equalities and situation independent predicates is called *uniform in* s if all fluents in ψ mention only s as their situation argument, and there are no quantifiers over s in the formula.

The basic action theory (BAT) $\mathcal{D}$ is the conjunction of the following classes of axioms: $\mathcal{D} = \Sigma \wedge \mathcal{D}_{ss} \wedge \mathcal{D}_{ap} \wedge \mathcal{D}_{una} \wedge \mathcal{D}_{S_0}$. We use examples from the BlocksWorld (BW) domain [35, 4]. For brevity, all $\bar{x}, a, s$ variables are implicitly assumed $\forall$-quantified at the outer level.

$\mathcal{D}_{\mathbf{ap}}$ is a set of action precondition axioms of the form $\forall s \forall \bar{x}.\ poss(A(\bar{x}), s) \leftrightarrow \Pi_A(\bar{x}, s)$, where $poss(a, s)$ is a special predicate meaning that an action a is possible in situation s, $\Pi_A(\bar{x}, s)$ is a formula uniform in s, and A is an n-ary action function. In most planning benchmarks, the formula Π_A is simply a conjunction of fluent literals and possibly negations of equality. We consider a version of BW, where there are three actions: *move-b-to-b*(x, y, z), move a block x from a block y to another block z, *move-b-to-t*(x, y), move a block x from a block y to the table, *move-t-to-b*(x, z), move a block x from the table to a block z.

$poss($*move-b-to-b*$(x,y,z), s) \leftrightarrow clear(x,s) \wedge clear(z,s) \wedge on(x, y, s) \wedge x \neq z.$
$poss($*move-b-to-t*$(x, y), s) \leftrightarrow clear(x, s) \wedge on(x, y, s).$
$poss($*move-t-to-b*$(x, z), s) \leftrightarrow ontable(x, s) \wedge clear(x, s) \wedge clear(z, s).$

Let $\mathcal{D}_{\mathbf{ss}}$ be a set of the successor state axioms (SSA):

$$F(\bar{x}, do(a,s)) \leftrightarrow \gamma_F^+(\bar{x}, a,s) \vee\ F(\bar{x}, s) \wedge \neg\gamma_F^-(\bar{x}, a,s),$$

where $\bar{x}$ is a tuple of object arguments of the fluent F, and each of the γ_F's is a disjunction of uniform formulas $[\exists \bar{z}].a = A(\bar{u}) \wedge \phi(\bar{x}, \bar{z}, s)$,where $A(\bar{u})$ is an action with a tuple $\bar{u}$ of object arguments, $\phi(\bar{x}, \bar{z}, s)$ is a context condition, and $\bar{z} \subseteq \bar{u}$ are optional object arguments. It may be that $\bar{x} \subset \bar{u}$.

If $\bar{u}$ in an action function $A(\bar{u})$ does not include any z variables, then there is no optional $\exists \bar{z}$ quantifier. If not all variables from $\bar{x}$ are included in $\bar{u}$, then it is said that $A(\bar{u})$ has a *global effect*, since the fluent F has at least one $\forall$-quantified object argument x not included in $\bar{u}$. Therefore, F experiences changes beyond the objects explicitly named in $A(\bar{u})$. For example, if a truck drives from one location to another, and driving action does not mention any boxes loaded on the truck, then the location of *all* loaded boxes change. When the tuple of action arguments $\bar{u}$ contains all fluent arguments $\bar{x}$, and possibly contains $\bar{z}$, we say that the action $A(\bar{u})$ has a *local effect*. A BAT is called a *local-effect* BAT if all of its actions have only local effects. In a local-effect action theory, each action can change values of fluents only for objects explicitly named as arguments of the action. In our implementation, we focus on a simple class of local-effect BAT, where SSAs have no context conditions. However, since [27], it is common to consider a broader class of SSAs with conditional effects that depend on contexts $\phi(\bar{x}, \bar{z}, s)$. Often, contexts are quantifier-free formulas, and then SSA is called *essentially quantifier-free*. In BW, we consider fluents $clear(x, s)$, meaning block x has no blocks on top of it in situation s, $on(x, y, s)$, meaning block x is on block y in situation s, $ontable(x, s)$, meaning block x is on the table in situation s. The following SSAs are local-effect (with implicit $\forall x, \forall y, \forall a, \forall s$):

$clear(x, do(a, s)) \leftrightarrow \exists y,z(a=$*move-b-to-b*$(y,x,z)) \vee \exists y(a=$*move-b-to-t*$(y,x)) \vee$
$\quad clear(x, s) \wedge \neg\exists y, z(a=$*move-b-to-b*$(y, z, x)) \wedge \neg\exists y(a=$*move-t-to-b*$(y, x)),$
$on(x, y, do(a, s)) \leftrightarrow \exists z(a=$*move-b-to-b*$(x, z, y)) \vee \exists y(a=$*move-t-to-b*$(x, y) \vee$
$\quad on(x, y, s) \wedge \neg\exists z(a=$*move-b-to-b*$(x, y, z)) \wedge \neg\exists y(a=$*move-b-to-t*$(x, y)),$
$ontable(x, do(a, s)) \leftrightarrow \exists y(a=$*move-b-to-t*$(x, y)) \vee$
$\quad ontable(x, s) \wedge \neg\exists y(a=$*move-t-to-b*$(x, y)).$

Note that each SSA mentions which actions have a (positive) add-effect (i.e., make the fluent true in the resulting situation), and which actions have a (negative) delete-effect (i.e., make the fluent false in the resulting situation). Heuristics based on the so-called delete relaxation can ignore those parts of the SSA which are related to delete effects.

$\mathcal{D}_{una}$ is a finite set of unique name axioms (UNA) for actions and named objects. For example,

$$\textit{move-b-to-b}(x,y,z) \neq \textit{move-b-to-t}(x,y),$$
$$\textit{move-b-to-t}(x,y) = \textit{move-b-to-b}(x',y') \rightarrow x = x' \wedge y = y',$$

and other similar axioms.

$\mathcal{D}_{S_0}$ is a set of FO formulas whose only situation term is S_0. It specifies the values of fluents in the initial state. It describes all the (static) facts that are not changeable by actions. Also, it includes *domain closure for actions* such as

$$\forall a.\ \exists x,y,z(a = \textit{move-b-to-b}(x,y,z)) \vee \exists x,y(a = \textit{move-b-to-t}(x,y)) \vee \exists x,y(a = \textit{move-t-to-b}(x,y)).$$

In particular, it may include axioms for domain specific constraints (state axioms), e.g.,

$$\forall x \forall y(on(x,y,S_0) \rightarrow \neg on(y,x,S_0)) \wedge$$
$$\forall x \forall y \forall z(on(y,x,S_0) \wedge on(z,x,S_0) \rightarrow y = z) \wedge$$
$$\forall x \forall y \forall z(on(x,y,S_0) \wedge on(x,z,S_0) \rightarrow y = z).$$

Notice we did not include any state constraint (axioms uniform in s) into BAT, e.g.,

As stated in [35], they are entailed from the similar sentences about S_0 for any situation that includes only consecutively possible actions.

Finally, the foundational axioms Σ are generalization of axioms for a single successor function (see Section 3.1 in [5]) since SC has a family of successor functions $do(\cdot, s)$, and each situation may have multiple successors. As argued in [5], the complete FO theory of single successor has countably many axioms, but it has non-standard models. To eliminate undesirable non-standard models for situations, by analogy with Peano second-order (SO) axioms for non-negative integers, where the number 0 is similar to S_0, [34] proposed the following axioms for situations:

$$do(a_1,s_1) = do(a_2,s_2) \rightarrow a_1 = a_2 \wedge s_1 = s_2,$$
$$\neg(s \sqsubset S_0),$$
$$s \sqsubset do(a,s') \leftrightarrow s \sqsubseteq s', \text{ where } s \sqsubseteq s' \stackrel{def}{=} (s \sqsubset s' \vee s = s'),$$
$$\forall P.\ \big(P(S_0) \wedge \forall a \forall s(P(s) \rightarrow P(do(a,s)))\big) \rightarrow \forall s(P(s)).$$

The last SO axiom limits the sort *situation* to the smallest set containing S_0 that is closed under the application of do to an action and a situation.

These axioms say that the set of situations is really a tree; there are no cycles, and no merging. These foundational axioms Σ are domain independent. Since situations are finite sequences of actions, they can be implemented as lists in PROLOG, e.g., S_0 is like the empty list [ ], and $do(A,S)$ adds an action A at the front of a list representing S, i.e. $[A \mid S]$. Therefore, in PROLOG, all situation terms satisfy the foundational axioms [35]. Appendix 2 includes BW implemented in Prolog.

It is often convenient to consider only executable (legal) situations: these are action histories in which it is actually possible to perform the actions one after the other.

$$s < s' \stackrel{def}{=} s \sqsubset s' \wedge \forall a \forall s^*(s \sqsubset do(a,s^*) \sqsubseteq s' \rightarrow Poss(a,s^*))$$

where $s < s'$ means that s is an initial sub-sequence of s' and all intermediate actions are possible. Subsequently, we use the following abbreviations: $s \leq s' \stackrel{def}{=} (s < s') \vee s = s'$. Also, $executable(s) \stackrel{def}{=} S_0 \leq s$. [35] formulates

Theorem 1. $executable(do([\alpha_1, \cdots, \alpha_n], S_0)) \leftrightarrow$
$poss(\alpha_1, S_0) \wedge \bigwedge_{i=2}^{n} poss(\alpha_i, do([\alpha_1, \cdots, \alpha_{i-1}], S_0))$.

Theorem 2. *[28] A basic action theory $\mathcal{D} = \Sigma \wedge \mathcal{D}_{ss} \wedge \mathcal{D}_{ap} \wedge \mathcal{D}_{una} \wedge \mathcal{D}_{S_0}$ is satisfiable iff $\mathcal{D}_{una} \wedge \mathcal{D}_{S_0}$ is satisfiable.*

Theorem 2 states that no SO axioms Σ are needed to check for satisfiability of BAT $\mathcal{D}$. This result is the key to tractability of $\mathcal{D}$, since $\mathcal{D}_{una} \wedge \mathcal{D}_{S_0}$ are sentences in FOL.

There are two main reasoning mechanisms in SC. One of them relies on the regression operator [39, 33] that reduces reasoning about a query formula uniform in a given situation σ to reasoning about regression of the formula wrt $\mathcal{D}_{S_0}$. Another mechanism called progression [17] is responsible for reasoning forward, where after each action α, the initial theory $\mathcal{D}_{S_0}$ is updated to a new theory $\mathcal{D}_{S_\alpha}$. In this paper, we focus on simplified progression in a local effect BAT [22], where SSAs are essentially quantifier free, as defined before.

The *Domain Closure Assumption* (DCA) for objects [30, 32] means that the domain of interest is finite, the names of all objects in $\mathcal{D}_{S_0}$ are explicitly given as a set of constants $C_1, C_2, \ldots, C_K$, and for any object variable x it holds that $\forall x(x = C_1 \vee x = C_2 \vee \ldots \vee x = C_K)$. According to the *Closed World Assumption* (CWA), an initial theory $\mathcal{D}_{S_0}$ is conjunction of ground fluents, and all fluents not mentioned in $\mathcal{D}_{S_0}$ are assumed by default to be false [31, 32]. According to an opposite *Open World Assumption* (OWA), an initial theory $\mathcal{D}_{S_0}$ can have a more general form, e.g., it can be in a $proper^+$ form [14].

As proved in Theorem 4.1 in [32], in the case of a database augmented with the axioms of equality, the queries that include only $\exists$-quantifiers over object variables can be answered without the DCA. Similar results can be proved for a $\mathcal{D}_{S_0}$ in a $proper^+$ form, assuming there are no object function symbols other than constants. From this fact, the above mentioned results, and the results from [21], it follows that in the case of a BAT where $\mathcal{D}_{S_0}$ is in a $proper^+$ form, the context conditions in SSAs are essentially quantifier free, where the preconditions $\Pi_A(\bar{x}, s)$ in $\mathcal{D}_{\mathbf{ap}}$ include only $\exists$-quantifiers over object variables, the goal formula includes only $\exists$-quantifiers over object variables, and all sets of axioms use only a bounded number of variables, the length-bounded planning problem can be solved without DCA over the object variables (and without CWA). In the next section, we formulate the (bounded) planning problem for BATs and show a planner can be developed from the first principles.

3. Bounded Lifted Planning with BATs

Let $G(s)$ be a goal formula that is uniform in s and has no other free variables. Let $Length(s)$ be a number of actions in situation s, i.e., $Length(do([\alpha_1, \cdots, \alpha_N], S_0)) = N$ and $Length(S_0) = 0$. Following [35], the bounded planning problem can be formulated in SC as

$$\mathcal{D} \models \exists s.\ Length(s) \leq N \wedge executable(s) \wedge G(s), \tag{1}$$

where $N \geq 0$ is an upper bound. From the Theorem 1, definition of $executable(s)$, the foundational axioms Σ, it follows that this can be equivalently reformulated for $N > 2$ as

$$\mathcal{D} \models G(S_0) \vee \exists a_1 \big(poss(a_1, S_0) \wedge G(do(a_1, S_0))\big) \vee$$
$$\exists a_1 \exists a_2 \big(S_0 < do([a_1, a_2], S_0) \wedge \big(G(do([a_1, a_2], S_0)) \vee$$
$$\exists s (do([a_1, a_2], S_0) \leq s \wedge Length(s) \leq N \wedge G(s)) \big) \big)$$

This simply means that if there exists a situation term that solves the planning problem (1), then either it is S_0, or for some action a_1 that is possible in S_0, it is $do(a_1, S_0)$, or for some actions a_1 and a_2 that are consecutively possible from S_0, either $G(do([a_1, a_2], S_0))$ holds, or there exists situation s that is executable from $do([a_1, a_2], S_0)$ such that its total length is less than or equal to N and the formula $G(s)$ holds in s. Suppose that a BAT $\mathcal{D}$ has k different action functions $A_1(\bar{x_1}), \ldots, A_k(\bar{x_k})$. Then, according to the DCA for actions, the formulas $\exists a\, \psi(do(a, S_0))$ and $\exists a \exists a'\, \psi(do(a', do(a, S_0)))$ are equivalent to $\bigvee_{i=1}^{k} \exists \bar{x_i} \psi\big(do(A_i(\bar{x_i}), S_0)\big)$ and $\bigvee_{j=1}^{k} \exists \bar{x_j} \bigvee_{i=1}^{k} \exists \bar{x_i} \psi\big(do(A_j(\bar{x_j}), do(A_i(\bar{x_i}), S_0))\big)$. Thus,

Theorem 3. *A ground situation term $do([\alpha_1, \cdots, \alpha_n], S_0)$, $n \leq N$ is a solution to problem (1) iff for some sequence $(i_1, \cdots, i_n)$ of action indices, $1 \leq i_j \leq k$, there are ground substitutions for action arguments that unify $A_{i_1}(\bar{x_{i_1}})$ with α_1,..., $A_{i_n}(\bar{x_{i_n}})$ with α_n, and for these substitutions both $\exists \bar{x_{i_n}} \cdots \exists \bar{x_{i_1}} G\big(do([A_{i_1}(\bar{x_{i_1}}), \cdots, A_{i_n}(\bar{x_{i_n}})], S_0)\big)$ and the formula $\exists \bar{x_{i_n}} \cdots \exists \bar{x_{i_1}} S_0 \leq do([A_{i_1}(\bar{x_{i_1}}), \cdots, A_{i_n}(\bar{x_{i_n}})], S_0)$ are entailed from a BAT $\mathcal{D}$.*

This theorem is the first key observation that helps design a lifted planner based on SC. The planner has to search over executable sequences of actions on a situation tree. Note that the state space and states themselves remain implicit, since situations serve as symbolic proxies to states. (For a given situation, state is a set of fluents that are true in this situation in a model of $\mathcal{D}$). Whenever a sequence of i ground actions determined by a search results in a situation $do([\alpha_1, \cdots, \alpha_i], S_0))$, to find the next action the planner must check among the actions $A_1(\bar{x_1}), \ldots, A_k(\bar{x_k})$ for which of the values of their object arguments these actions are possible in $do([\alpha_1, \cdots, \alpha_i], S_0))$. Since this computation is done at run-time, but not before the planner starts searching for actions, the SC-based planner is naturally lifted, no extra efforts are required.

According the above discussion, if N is large, then the right hand side of (1) expands into the long disjunction of formulas, and it is not clear in what order the deductive planner has to search over these formulas. The second key observation is that an efficient deductive planner needs control that helps select for each situation the most promising next possible action to execute. This control can be provided by a search algorithm that relies on a domain independent heuristic.

4. Implementation

Our SC-based TPLH planner is implemented in PROLOG following the two key observations mentioned in the previous section. The planner is driven by theorem proving that is controlled by a version of A* search for a shortest sequence of actions that satisfies (1). The distinctive feature of TPLH is that it does forward *search over the situation tree from S_0*. Since each search node is a unique situation, the previously visited nodes cannot be reached again. Moreover, frontier nodes cannot be reached along different paths, since each situation represents a unique path. However, different situations can represent the same state, where same fluents are true. Therefore, we consider two versions of our planner. In this section and in 5.2, we discuss the baseline version that keeps no records of what situations have been already visited. In Section 5.3, we consider

another version that checks each visited state to ensure it has not been visited before. More specifically, the list of actions corresponding to each visited state is stored in a hash table. When search visits a new situation, we progress it to compute its state, then compute a hash value of this state and check whether the value maps to a hash table slot occupied by any of the previously visited situations. Upon encountering a collision, the state represented by a previous situation in the hash table is recomputed to verify whether it is equivalent to the current state. If so, the shortest situation is preserved, the longer situation is discarded, and then search continues. In both versions, for simplicity, the cost of every action is 1, and the cost of a path to a node is simply the length of the situation representing that node. This search terminates as soon as it finds a ground situation S that satisfies a goal $G(S)$. In Algorithm 1, a plan is a situation that is represented as a list of actions from S_0, while S_0 is represented as the empty list.

The main advantage of this design is that the frontier stored in a priority queue consists of situations and their f-values[1] computed as the sum of situation length and a heuristic estimate. Therefore, situations serve as convenient symbolic proxies for states. As usual, a state corresponding to situation s is a set of fluents that are true in s. However, in hard-to-ground domains, each state can be very large, and storing all intermediate states can exhaust all memory. This issue was demonstrated on realistic domains such as Organic Synthesis [24]. Moreover, in the case of planning in physical space and real time, the state space is infinite, but a deductive planner can still search (without ad-hoc discretizations of space and time) over finite sequences of actions according to semantics in [1].

In Algorithm 1, the sub-procedure InitialState($\mathcal{D}_{S_0}$) on Line 5 takes the initial theory as its input, and computes the initial state under the usual DCA and CWA. (Note this is a limitation of the current implementation, but not of the TPLH approach in general). We store this initial state $Init$ in a specialized data structure that facilitates computing progression efficiently. On Line 7, the algorithm extracts the next most promising situation S from the frontier. Then, on Line 8, it computes progression Now of the initial state using the actions mentioned in S. On Line 9, there is a check for whether the goal formula G is satisfied in the current state Now. If it is, then S is returned as a plan. If not, then on Line 12, the algorithm finds all actions that are possible from the current state using the precondition axioms. In fact, the sub-procedure $FindAllPossibleActions$ is using preconditions to ground all action functions from the given BAT in the current state. Since actions are grounded at run-time, TPLH is a lifted planner by design. If there are no actions possible from Now, then the algorithm proceeds to the next situation from the frontier. Otherwise, for each possible ground action A_i, it constructs the next situation $S_n = do(A_i, S)$, and if its length does not exceed the upper bound N, it computes the positive integer number d on Line 21 as $N - Length(S)$. This bound d is provided as an input to the heuristic function $H(\mathcal{D}, G, d, S_n, St)$ that does limited look-ahead up to depth d from St to evaluate situation S_n. On Line 24, S_n and its f-value $S_n.Val$ are inserted into the frontier, and then search continues until the algorithm finds a plan, or it explores all situations with at most N actions. The **for**-loop, Lines 16-24, makes sure that all possible successors of S are constructed, evaluated and inserted into the frontier. This is important to guarantee completeness of Algorithm 1.

[1]This is a term from the area of heuristic search, see [6, 7]. There are plan costs $g(s)$, the number of actions in s, and there are heuristic estimates (h values) of the number of actions remaining before the goal can be reached. The total priority of each search node (in our case it is a situation s) is estimated as $f(s) = g(s) + h(s)$. A smaller total effort $f(s)$ indicates a more promising successor situation s.

Algorithm 1: A^* search over situation tree to find a plan

Input: $(\mathcal{D}, G)$ - a BAT $\mathcal{D}$ and a goal formula G
Input: H - Heuristic function
Input: N - Upper-bound on plan length
Output: S that satisfies (1) ▷ Plan is the list of actions in S

1: **procedure** PLAN($\mathcal{D}, G, N, H, S$)
2: $PriorityQueue \leftarrow \varnothing$ ▷ Initialize PQ
3: $S_0.Val \leftarrow (N + 1)$
4: $PriorityQueue.insert(S_0, S_0.Val)$
5: $Init \leftarrow$ InitialState($\mathcal{D}_{S_0}$) ▷ Initialize state
6: **while not** $PriorityQueue.empty()$ **do**
7: $S \leftarrow PriorityQueue.remove()$
8: $Now \leftarrow$ Progress($Init, S$) ▷ Current state
9: **if** Satisfy(Now, G) **then**
10: **return** S ▷ Found a plan
11: **end if**
12: $Acts \leftarrow$ FindAllPossibleActions(Now)
13: **if** $Acts == \varnothing$ **then**
14: **continue** ▷ No actions are possible in S
15: **end if**
16: **for** $A_i \in Acts$ **do**
17: $S_n \leftarrow do(A_i, S)$ ▷ S_n is next situation
18: $St \leftarrow$ Progress(Now, A_i) ▷ Next state
19: **if** Length(S) $\geq N$ **then**
20: **continue** ▷ S_n exceeds upper bound
21: **else** $d \leftarrow N -$Length(S) ▷ d is depth bound
22: **end if**
23: $S_n.Val \leftarrow$ Length(S_n)$+H(\mathcal{D}$,G,d,S_n,$St)$
24: $PriorityQueue.insert(S_n, S_n.Val)$
25: **end for**
26: **end while**
27: **return** $False$ ▷ No plan for bound N
28: **end procedure**

The bound $N \geq 0$ makes sure that search will always terminate in a finite domain, since there are finitely many ground situations with length less than or equal to N, and in the worst case, all of them will be explored. However, due to this upper bound, search may terminate prematurely, i.e., without reaching a goal state, if the shortest plan includes more than N action. Consequently, this planner is complete only if the bound N is greater than or equal to the length of a shortest plan. Obviously, the planner is sound thanks to Lines 8 and 9.

Note that $Progress(Init, S)$ computes afresh the current state from the given initial state and the list of actions in S. If the computed state Now does not satisfy a goal formula, it is not

Algorithm 2: GraphPlan heuristic with delete relaxation

Input: $(\mathcal{D}, G)$ - BAT $\mathcal{D}$ and a goal formula G
Input: $d \geq 1$ - Look-ahead bound for the heuristic algorithm
Input: S_n, L - The current situation and its length
Input: St - The current state
Output: *Score* - A heuristic estimate for the given situation

procedure $H(\mathcal{D}, G, d, S_n, St)$
  $Depth \leftarrow 0$
  $PG \leftarrow \langle S_n, St \rangle$ ▷ Initialize Planning Graph
  while not Satisfy(St, G) **and** $Depth \leq d$ **do**
    $\{ActSet\} \leftarrow$ FindAllPossibleActions(St) ▷ Need only relevant actions
    $NewActs \leftarrow$ Select relevant actions from $ActSet$ ▷ that add new fluent(s)
    $St \leftarrow$ ProgressRelaxed$(St, NewActs)$
      ▷ Add all new positive effects $NewEffs$ to the state
    $NextLayer \leftarrow \langle NewEffs, NewActs, St \rangle$
      ▷ Record actions added, their effects, the current state
    $PG.extend(NextLayer)$
    $Depth \leftarrow Depth + 1$
  end while
  $Goal \leftarrow$ Convert G into a set of literals
  if $Depth > d$ **then return** $(L + d)$ ▷ Penalty
  else return $Reachability(\mathcal{D}, Goal, PG)$
  end if
end procedure

preserved after computing the heuristic value of Now. Only successor situations are retained in the frontier, but not their corresponding states. This is an important contribution of the TPLH approach. The previous planning algorithms usually retained states, but not situations in their frontiers; see [38] for a detailed discussion. In the TPLH approach, the initial state $Init$ remains in memory, but all other intermediate states are recomputed from $Init$ on demand. Therefore, TPLH trades speed for memory. Since the memory footprint of TPLH is smaller than it would be for alternative implementations, our approach is suitable for planning in hard to ground domains.

Computing the heuristic function is done in two stages, using the usual delete relaxation. First, a planning graph is built from the current state, layer-by-layer until all goal literals are satisfied. Best supporting actions are then found for the goal literals, going backwards through the graph; see Algorithm 2.

The planning graph is initialized to the current situation and state. At each step in building the planning graph, all possible actions for the current state are found, and then filtered so that only those actions with one or more new (positive) add effects not in the current state are kept. The state is updated using relaxed progression to incorporate their new add effects. These 'relevant actions', their new add effects and the updated state are inserted into the next layer of the planning graph, and the process is repeated.

Once all goal literals are satisfied, the most recent layer of the planning graph is examined. For

Algorithm 3: Reachability score for a set of goal literals

Input: $(\mathcal{D}, G)$ - A BAT $\mathcal{D}$ and a set G of goal literals
Input: PG - A planning graph, initialized to $\langle S_n, St \rangle$
Output: V - A heuristic estimate for achieving G

procedure $Reachability(\mathcal{D}, G, PG)$
  if $PG == \langle S_n, St \rangle$ **then return** 0
  else $\langle Effs, Acts, St \rangle \leftarrow PG.removeOuterLayer$
  end if
  $CurrGoals \leftarrow G \cap Effs$ ▷ The set of achieved goals
  $NewGoals \leftarrow \emptyset$ ▷ To collect preconditions
  $BestSupport \leftarrow \emptyset$ ▷ Easiest causes for CurrGoals
  for $g \in CurrGoals$ **do**
    $Relev \leftarrow$ {actions from $Acts$ with g as add effect}
    for $a \in Relev$ **do**
      $a.Pre \leftarrow$ {the set of preconditions of a}
      $a.Estimate \leftarrow Reachability(\mathcal{D}, Pre, PG)$ ▷ Notice recursive call
    end for
    $BestAct \leftarrow$ ArgMin $\{a.Estimate$ over $Relev\}$ ▷ This is different from FF
    ▷ Find the easiest action from $Relev$ with minimum estimate
    $NewGoals \leftarrow NewGoals \cup BestAct.Pre$
    $BestSupport \leftarrow BestSupport \cup BestAct$
  end for
  $RemainGoals \leftarrow G - CurrGoals$
  $NextGoals \leftarrow RemainGoals \cup NewGoals$
  $C_1 \leftarrow$ Count$(BestSupport)$ ▷ i.e. # of best actions
  $C_2 \leftarrow Reachability(\mathcal{D}, NextGoals, PG)$
  return $C_1 + C_2$
end procedure

each of the new add effects in this layer belonging to the set of goal literals, all relevant actions from the layer which achieve the effect are selected. These are referred to as the 'supporting actions' for the goal literal. For each supporting action, a 'reachability' score is recursively computed using its preconditions as the new goal literals. The easiest action whose preconditions have the lowest reachability is considered the 'best supporting action'. Thus, our reachability score represents the estimated cost of achieving a set of literals. If all literals are satisfied in the initial layer of the planning graph PG, then the set's reachability is 0. Otherwise, its reachability is equal to the reachability of the remaining goals and preconditions for the set of the easiest actions, plus the number of best support actions. The details are summarized in Algorithm 3. This heuristic is not admissible, but our experiments show it is informative in several applications.

5. Experimental Results

To evaluate our implementation experimentally, we run our planner on several STRIPS benchmarks, where preconditions of actions are conjunctions of fluents (though they can include negations of equality between variables or constants), the SSAs have no context conditions, and the goal formula is a conjunction of ground fluents.

Tests were run separately using the TPLH, FD, and BFWS planners. TPLH and FD used the A* algorithm to prioritize shorter plan lengths, whereas BFWS used a default greedy search algorithm based on a width heuristic [18, 19, 20]. Both TPLH and FD did eager search with FF heuristic. All testing was done on a desktop with an Intel(R) Core(TM) i7-3770 CPU running at 3.40GHz. Tests measured total time spent, plan length, and number of states (situations) visited. Comparisons are made based on International Planning Competition (IPC) scores for satisficing planning that are extensivly discussed in [23]. As defined there, each participating planner p gets a score S_i^p per planning task i "expressed as $S_i^p = C_i^*/C_i^p$, where C_i^p is the total cost of the best solution found by planner p for instance i, and C_i^* is the lowest total cost found so far by any planner for the same problem, that is, $C_i^* = min_p\{C_i^p\}$" [23]. Since unsolved problems are scored as 0, coverage is taken into account by the score function. As you can see, the highest possible score per instance is 1. Usually, the score is based on the total cost, i.e., the sum of the costs of all individual actions in a plan, the IPC score can be adapted to other metrics as well. In our research we conside both the IPS score based on plan length (since TPLH assigns cost 1 to each action), and on the number of situations or states visited, where situation is visited when TPLH evaluates whether it is a goal state. The TPLH planner, domain files and problem instances have been loaded, compiled and run within ECLiPSe Constraint Logic Programming System, Version 7.0 #63 (x86_64_linux), released on April 24, 2022. In comparison, the FD and BFWS were compiled into executable files.

5.1. Domains and Problem Generation

Testing was done over randomly generated problems for 8 different popular domains that represent well the variety of planning problems from the IPC competitions. These domains were Barman (BR), BlocksWorld (BW), ChildSnack (CS), Depot (D), FreeCell (FC), Grippers (GR), Logistics (L), and Miconic (M). In addition, testing was also done on 10 pre-existing problems belonging to the PipesWorld (PW) domain. All domains are in STRIPS, extended to include negated equalities and object typing. For simplicity, the Barman domain was modified to remove action costs.

Roughly 100 problems with varying numbers of objects were generated for each of the specified domains, using publicly available PDDL generators. All PDDL domains and generated instances files were automatically translated from PDDL to PROLOG using our program that constructs a hash table based representation of an initial theory. The TPLH planner was run over every problem using a 15 minute time-out limit, and a 512M MB stack size limit, i.e., much less than typical memory cutoffs 6 GB. Problems for which the planner timed out were discarded, as were problems with 0-step solutions (i.e., where the initial state satisfied the goal state). The number of kept instances for each domain is shown in parentheses after the domain name in Table 1. The remaining instances are not trivial for several domains that we checked, i.e., when we run TPLH on them without heuristic (h set to 0), it could not solve most of them within the allocated time

and memory bounds. The TPLH planner was given the upper bound $N = 100$ for all planning instances that usually had short solutions, e.g., 20 steps or less. Miconic was the only domain where some of the computed plans were longer than 20 steps. Recall N is used to guarantee completeness of TPLH, but it had little effect in this set of experiments. Namely, when we tried different values $N = \{50, 75, 100, 125, 150\}$ over some domains, the total time varied within 1%, but plan length and the number of situations visited by TPLH did not change at all.

Before TPLH could be tested on a domain, the domain file was converted from PDDL to a BAT implemented in PROLOG, and initial state hash tables were built for each individual problem. More specifically, we implemented initial state as a list of fluent names, where each fluent is represented by a hash table which stores all instances which are true in the initial state $Init$. Hashing fluent arguments allows for efficient access and updating of state information to facilitate progression planning. Translating domains files themselves took very little time (under 0.1 seconds in all cases), and this cost was further amortized by the fact that it only needed to be done once, regardless of how many problems were tested. Building initial state hash tables however could take a non-negligible amount of time. This time was consistent across all problems belonging to a domain, and ranged from approximately 1.5 seconds per problem (for BlocksWorld) to nearly 10 seconds per problem (for Barman). The inefficiency here is tied to the current implementation of the script used, and is not inherent to the task of creating the hash tables themselves. Preprocessing time for each problem was added to the time spent by the planner itself to get the total time to solve a problem. (Performance of TPLH on easy instances was much better when preprocessing was factored out.)

In terms of CPU time, as expected, TPLH was much slower than FD and BFWS. More specifically, TPLH was on average about 10^2 times slower than FD. In BW, Grippers, Miconic and PipesWorld, TPLH was on average 10^3 times slower than BFWS, and on other domains TPLH was about 10^4 times slower than BFWS. TPLH timed out on several instances, but both FD and BFWS solved all the instances within allocated time and memory. Note that the number of objects in the generated instances was relatively small. The slow performance of TPLH was not surprising, but it is interesting to compare TLPH with FD and BFWS in terms of IPC scores. We do this in the next sub-sections, starting with a baseline version of TPLH, and then we proceed to more optimized versions of TPLH.

5.2. Plan Lengths and Number of Situations Visited

In this sub-section, we discuss only a baseline version of TPLH which does not check whether the current situation corresponds to a state that was previously visited. When testing the problems using TPLH, the number of situations visited was recorded, as was the length of the produced plan and the total time taken. A situation was considered as having been visited upon checking whether it satisfied a goal state. Thus, the minimum number of *situations* visited by TPLH is one greater than the length of the produced plan. The same data was gathered when testing using the FD and BFWS planners, with the distinction that the number of *states* visited by each planner was recorded, rather than situations. In this section, we consider only a mini-competition between a baseline TPLH, FD, and BFWS. We compare the IPC scores for plan length and the number of situations visited for TPLH to FD and BFWS, see Table 1. In the second column, we include for each domain the object counts used to generate the random instances.

		Plan Length			Situations Visited		
Domain	#Obj	**TPLH**	**FD**	**BFWS**	**TPLH**	**FD**	**BFWS**
BR (100)	11-31	100	100	90.28	100	10.12	20.42
BW (95)	8-11	90.14	91.54	57.82	88.87	10.92	21.60
CS (100)	9-17	100	100	95.78	51.89	5.01	60.03
D (76)	8-21	75.83	75.78	74.21	65.39	5.58	36.92
FC (95)	21-37	93.89	93.98	92.25	95	9.52	21.71
GR (98)	10-27	97.72	97.85	77.80	70.23	4.51	59.25
L (119)	9-32	119	119	103.87	93.31	13.10	61.16
M (93)	17-40	93	93	80.31	91.94	10.30	17.03
PW (6)	26-44	5.92	6	5.65	3.23	1.69	4.77

Table 1
IPC scores for plan length and situations/states visited when comparing the baseline TPLH planner with FD and BFWS. Higher score means better performance.

TPLH was competetive with FD when evaluating plan length across every domain; it matched FD in four of the domains, and outperformed it for problems belonging to the Depot domain. In addition, it only fell short in the BlocksWorld domain due to the fact that it failed to complete three problems within the allotted time. When comparing to BFWS, TPLH outperformed it for plan length across all nine domains. This is not surprising, since TPLH does A* search, but BFWS does greedy search guided by novelty, and this leads to increased exploration as explained in [19].

When using IPC scores based on the number of situations/states visited, TPLH greatly outperformed FD, and scored better than BFWS in seven of the nine domains, but was beaten in the ChildSnack and PipesWorld domains. A few inherent aspects of the ChildSnack domain lead to the heuristic performing poorly wrt BFWS. Firstly, plans in this domain are highly 'interleavable'; i.e. there are several permutations of the same actions which are all valid solutions. Secondly, ChildSnack problems have relatively few goal atoms, which are all achieved by the last few actions of an optimal plan. Third, no heuristic is perfect. As the heuristic is domain-independent, it is natural that there will be some domains where it excels, and some where it struggles. Apparently, the width-based heuristic in BFWS is better on this domain. Notice that in ChildSnack TPLH performs much better than FD (with a FF heuristic) in terms of the situations/states visited.

It is important to realize our heuristic is more informative than the implementation of FF used by the FD planner. In Algorithm 3, see Lines 10-14, when we evaluate support actions $Relev$ which achieve one of the fluents in $CurrGoals$, the cost of achieving the preconditions of each action is recursively estimated, and then the action with the lowest such cost is selected on Line 14. The cost is the number of the best supporting actions, see Line 21. This is in contrast to the FF heuristic, which does one top-down loop over the layers and selects the minimum difficulty supporting action for each fluent, see Figure (2) and Section 4.2.2 in [13]. There, the difficulty of an action is measured as the sum over its preconditions of the earliest layers where each precondition holds, and therefore it overcounts.

The heuristic used by TPLH performed remarkably well on certain problems, only ever visiting situations which were a subsequence of the final plan. In the FreeCell domain for example, this was true of every problem tested. This is likely due to the nature of the Planning Graph data structure and the process used for finding best supporting actions. When evaluating actions which

Domain	FD		BFWS	
BR	1	1	1	1
BW	0.86	0.95	0.96	0.93
CS	1	1	1	0.50
D	0.97	1	0.99	0.78
FC	0.91	1	0.94	1
GR	0.96	0.99	1	0.57
L	1	0.97	1	0.69
M	1	1	1	0.98
PW	0.83	0.5	0.83	0.5

Table 2
Percentage of problems for each domain where TPLH produced a plan of shorter or equal length to FD and BFWS (left), and visited fewer situations than FD and BFWS (right)

Domain	Avg r	% s.t. $r \geq 0.75$
BR	0.29	0
BW	0.59	0.45
CS	0.40	0.50
D	0.58	0.54
FC	0.89	1
GR	0.38	0.30
L	0.41	0.28
M	0.57	0.14
PW	0.24	0.16

Table 3
Average ratio r of plan length to number of situations visited by TPLH (left), as well as the fraction of problems from each domain for which r was ≥ 0.75 (right)

achieve the goal state for the relaxed problem, the cost of achieving the preconditions of each action is recursively computed, and the action with the lowest such cost is selected. This means that for highly sequential problems, where a specific chain of actions is necessary to allow a sub-goal to be achieved (e.g. in FreeCell, cards must be placed on the foundation pile in sequential order), the heuristic can identify situations which allow for shorter causal chains. As long as a given move completes a step in this chain, TPLH recognizes the resulting situation as more promising than the previous one, and pursues it. When the causal chain is complete for the final goal, the problem is solved.

TPLH was also competetive with FD and BFWS when comparing on a problem-by-problem basis. Refer to Table 2 for % of problems across each domain for which TPLH performed at least as well as its competitors on plan length (left column) and situations visited (right column).

Measuring the ratio r of the length of the plan produced to the number of situations visited by TPLH, we can evaluate the performance of our heuristic across each of the nine domains tested. We used a cutoff value of $r \geq 0.75$ to identify the percentage of problems that the heuristic guided effectively, see Table 3. As previously discussed, the heuristic was able to effectively guide 100% of problems in the FreeCell domain. It also performed well on the BlocksWorld domain (45%) and the Depot domain (54%). At the lowest end, none of the problems from the Barman domain met this threshold $r \geq 0.75$.

The recursive nature of finding the best supporting actions necessitates a lot of redundant computations. The current non-optimized implementation of the heuristic spends the vast majority (upwards of 95%) of total TPLH time. This is part of the reason why TPLH is orders of magnitude slower than FD and BFWS. Moreover, finding new possible actions inside heuristic takes time. More specifically, we found that Lines 5 and 6 consume significant time in Algorithm 2, due to the fact that they must recompute all actions which were possible in previous layers of the planning graph in addition to those new to the current layer. Computing only those new relevant actions which were not previously possible is non-trivial; this is future work.

5.3. Extensions to the Baseline Version of TPLH

This section will compare the performance of our TPLH planner with and without various extensions we have implemented, in order to examine trends across different domains and determine an optimal configuration. All tests were performed over the same set of problems from the previous subsections, using the same hardware, memory, and timeout limits. Table 4 contains IPC scores for each of the tested configurations of TPLH, along with FD and BFWS, across all problems for each domain. Note that in this section we computed IPC scores not only for FD and BFWS, but also for all different configurations and extensions of TPLH. Therefore, the data in Tables 4 and 5 are not directly comparable with the data in Table 1. We postpone our discussion of these tables to the end of this section.

Domain	**A*-U**	**A*-1**	**A*-2**	**G-1**	**G-2**	**FD**	**BFWS**
BR (100)	100	100	100	75.31	88.96	100	90.28
BW (95)	89.63	90.14	89.61	73.75	76.41	90.55	57.24
CS (100)	100	100	80	86.75	87.88	100	95.78
D (76)	75.83	75.71	75.94	65.11	67.57	75.77	74.21
FC (95)	93.89	93.89	94.00	93.89	93.53	93.98	92.24
GR (98)	97.31	97.31	97.77	87.88	88.90	97.44	77.48
L (119)	119	119	119	100.75	98.35	119	103.87
M (93)	93	93	93	67.52	69.63	93	80.31
PW (6)	5.81	5.89	5.92	5.01	5.33	6	5.55
Total (782)	774.46	774.86	755.24	655.97	676.58	775.76	676.97

Table 4
Comparisons of IPC scoress for plan length across FD, BFWS, and different configurations for TPLH: A* search without filtering of duplicated states (**A*-U**), A* search with filtering of duplicated states using both a single-queue (**A*-1**) and dual-queue (**A*-2**) configuration, and greedy search using single-queue (**G-1**) and dual-queue (**G-2**) configurations

Domain	**A*-U**	**A*-1**	**A*-2**	**G-1**	**G-2**	**FD**	**BFWS**
BR (100)	49.97	56.40	60.02	68.76	99.37	3.96	8.87
BW (95)	67.42	67.63	62.29	82.13	78.61	7.42	13.13
CS (100)	51.10	51.72	51.53	60.86	100	5.00	32.19
D (76)	56.01	56.08	52.82	64.29	70.44	5.04	24.49
FC (95)	94.92	94.92	83.06	94.92	90.47	9.51	21.70
GR (98)	42.73	43.89	36.53	95.10	89.50	2.81	19.92
L (119)	63.22	66.72	59.64	95.79	107.20	8.56	23.73
M (93)	75.21	76.42	32.72	87.93	71.36	7.70	11.03
PW (6)	1.68	1.76	1.71	5.95	5.27	0.58	1.61
Total (782)	502.27	515.54	440.33	655.73	712.22	50.58	156.68

Table 5
Comparisons of IPC scores for states/situations visited across FD, BFWS, and different configurations for TPLH: A* search without filtering of duplicated states (**A*-U**), A* search with filtering of duplicated states using both a single-queue (**A*-1**) and dual-queue (**A*-2**) configuration, and greedy search using single-queue (**G-1**) and dual-queue (**G-2**) configurations

The most important extension we have added to the baseline version of TPLH is the ability to detect and filter out repeated states while exploring different action sequences in the situation tree. For example, two different sequences of actions in the Gripper domain can pick up two balls in a different order and move them from one room to another so that the state resulting from these two sequences is exactly the same. The baseline TPLH discussed in the previous Section 5.2 did not have this check. Now, we can record all visited situations in a hash table. When the search algorithm takes a new situation, we compute the state it represents, compute from the state its hash function to check whether the corresponding hash table slot is occupied or not by any of the previously visited situations, and if yes, then verify whether their states are actually the same or not. If their states are the same, we pursue only the shorter situation and discard the longer. Otherwise, we insert a new visited situation in the hash table slot. Notice we still only store situation in memory, and recompute the corresponding states on demand.

In the sequel, when we directly compare two configurations of the TPLH planner, only those problems solved by both configurations were included while calculating average performances across different criteria.

With filtering enabled, the A* planner was able to find plans more quickly across every domain except for ChildSnack, where the large number of situations visited led to a greater number of hash collisions, and FreeCell, where the strong performance of the heuristic we used meant that repeated states were never encountered. The most marked improvements effected by filtering were found in the Depot, Grippers, and Logistics domains. Unsurprisingly, filtering duplicate states also resulted in fewer states being visited across almost every domain. Table 6 contains full information on the effects of filtering for each domain. (The bold font shows the best performance.)

Domain	Average Time	Average Length	Average Visited
BR (100)	11.0/**8.4**	**10.5**/**10.5**	52.3/**43.9**
BW (92)	73.8/**67.2**	**11.5**/11.5	**55.0**/59.3
CS (100)	**12.8**/19.8	**5.9**/**5.9**	2078.1/**1683.1**
D (76)	3.3/**0.7**	**8.7**/8.7	234.1/**65.6**
FC (95)	**76.9**/77.2	**8.6**/**8.6**	**9.6**/**9.6**
GR (98)	13.7/**5.2**	**13.8**/**13.8**	690.6/**265.7**
L (119)	82.1/**38.3**	**11.6**/**11.6**	457.7/**131.2**
M (93)	13.9/**11.6**	**26.8**/**26.8**	77.3/**58.3**
PW (6)	91.0/**65.5**	**9.2**/**9.2**	868.3/**217.8**

Table 6
Average performances of the A* planner both without (left) and with (right) filtering of repeated states across problems which both were able to solve, for solving time (seconds), plan length, and situations visited

Filtering of repeated states also allowed for the use of a greedy search strategy, as repeated action sequences cycling through states with the same heuristic value can be avoided. Table 7 contains a breakdown of its performance relative to the A* search method with filtering. The greedy search strategy was able to achieve a substantial improvement in solving time over the A* strategy across almost every domain. This is due to it visiting fewer overall situations, meaning that the greedy strategy shows a smaller overall improvement across domains where the heuristic guides the search more effectively, such as FreeCell and Miconic.

Domain	Average Time	Average Length	Average Visited
BR (100)	8.4/**4.4**	**10.5**/14.5	43.9/**34.0**
BW (92)	67.2/**45.8**	**11.5**/16.9	59.3/**31.4**
CS (100)	19.8/**0.4**	**5.9**/7.7	1683.1/**142.0**
D (76)	0.7/**0.3**	**8.7**/10.8	65.6/**18.7**
FC (95)	77.2/**76.8**	**8.6**/**8.6**	**9.6**/**9.6**
GR (98)	5.2/**0.6**	**13.8**/16.1	265.7/**18.8**
L (119)	38.3/**18.7**	**11.6**/15.7	131.2/**42.7**
M (93)	11.6/**10.0**	**26.8**/38.1	58.3/**41.5**
PW (6)	65.5/**23.7**	**9.2**/11.2	217.8/**12.8**

Table 7
Average performances of the A* planner with filtering of repeated states (left) and the greedy planner (right), both planners are with a single queue configuration, across problems which both were able to solve, for solving time (seconds), plan length, and situations visited

In addition, we implemented a second queue in the frontier, to hold “useful" situations reached via helpful/preferred actions. These are defined recursively as actions which achieve a goal fluent, or actions which achieve a ground fluent which is a precondition for a previously found preferred action [13, 11, 6]. These are computed once at the beginning, from the planning graph for the initial state. When the planner selects a new situation from the frontier when using the dual queue configuration, it alternates between the queue containing all situations and the queue containing “useful" situations. Situations were ordered in both queues based on the same heuristic value. This strategy seemed to help keep the planner ‘on track’ while performing a greedy search, generally resulting in shorter plans, though often with more states being visited. See Table 8 for a full summary of the results. Similar patterns were observed when comparing single-queue and dual-queue configurations of TPLH using A* search. Notably however, the dual-queue A* planner completed twenty fewer problems in the ChildSnack domain than its single-queue counterpart due to exceeding the allotted memory. This is likely due to the overhead of maintaining two separate priority queues.

Domain	Average Time	Average Length	Average Visited
B (100)	4.4/**3.8**	14.5/**12.2**	34.0/**24.4**
BW (93)	46.5/**42.2**	17.0/**15.7**	**31.6**/41.9
CS (100)	0.4/**0.1**	7.7/**7.4**	142.0/**18.5**
D (76)	0.3/**0.2**	10.8/**9.8**	18.7/**12.6**
FC (95)	**76.8**/76.9	**8.6**/8.6	**9.6**/10.0
G (98)	0.6/**0.5**	16.1/**15.5**	**18.8**/19.5
L (119)	18.7/**15.2**	**15.7**/15.9	**42.7**/43.5
M (93)	**10.0**/11.1	38.1/**36.9**	**41.5**/52.5
PW (6)	**23.7**/26.5	11.2/**10.2**	**12.8**/15.2

Table 8
Average performances of the greedy planner with a single queue (left) and a dual queue (right) configuration across problems which both were able to solve.

Finally, we would like to sumamrize the results reported in the Tables 4 and 5. IPC scores for

plan length and situations visited can be best compared according to the search strategy used. Consider the Table 4 and note that TPLH with an A* search strategy was competitive with FD for plan length in the A*-U and A*-1 configurations, falling approximately one point short of it. Notably however, TPLH was unable to solve three of the problems within the BlocksWorld domain within the time/memory limits, and the A*-2 configuration could not solve an additional twenty problems from the ChildSnack domain. If these instances were removed from the comparison pool, then we could say that TPLH slightly outscores FD, meaning that TPLH actually finds shorter solutions on a problem-by-problem basis. Now, focusing on the Table 5, note that the G-2 configuration scores very similarly to BFWS despite solving two fewer problems overall. When considering the number of states/situations visited, TPLH greatly outshines FD and BFWS regardless of the search strategy being used. Therefore, we can conclude from these experimental results that deductive planning is a productive research direction.

6. Conclusion and Future Work

We have developed a sound and complete lifted planner based on theorem proving in the situation calculus. It searches for a plan in a tree of situations, but not in a state space, and therefore it has minimal memory footprint. It was tested using a heuristic inspired by FF. To the best of our knowledge, TPLH is the first deductive planner based on SC with a domain independent heuristic. The readers can find a detailed discussion of the previous work on deductive planning in [38].

It is easy to consider arbitrary action costs within TPLH. It is possible to develop a deductive planner that works not only with context-free domains, but also with more general BATs, where SSAs have context conditions. The bound N is not essential to the design of TPLH. It can be easily removed, but then TPLH will lose completeness guarantees over finite domains with DCA.

In future, we would like to develop lifted versions of several heuristics. In particular, the current implementation of the Plan Graph based heuristics that is described in this paper grounds both fluents and actions at run-time and builds a large data structure, but this is inefficient and consumes more memory as the number of objects grow. However, one can implement a lifted version of the same heuristic that can be more suitable to planning in the hard-to-ground domains. We noted that deductive planning in SC leads naturally to lifted planning with action schemas at run time. However, in this paper we do not compare our planner with other recent lifted single-model planners. This study remains an interesting and important future research direction.

Since we ground actions at run-time by evaluating their preconditions, and this is one of the computational bottlenecks, in particular, in the hard-to-ground domains with complex preconditions as in [24, 29], we need a better algorithm for finding possible actions. This is work in progress. It is important for research in deductive planning.

In the case of incomplete $\mathcal{D}_{S_0}$ (no CWA), and a local effect BAT, one would need a more sophisticated algorithm for progression. In addition, we would like to develop an implementation that does not rely on DCA for objects, e.g., an implementation for the planning problems where the actions can create or destroy objects. This is doable within our deductive approach to planning.

7. Acknowledgments

Thanks to the Natural Sciences and Engineering Research Council of Canada for partial funding of this research and to the reviewers of the preliminary version of this paper for useful comments.

Appendix 1: The Blocks World in PDDL

```
(define (domain blocksworld_3ops)
(:requirements :equality)
(:predicates (clear ?x)     (on-table ?x)     (on ?x ?y))
(:action move-b-to-b
  :parameters (?bm ?bf ?bt)
  :precondition (and (clear ?bm) (clear ?bt)
                     (on ?bm ?bf) (not (= ?bm ?bt)))
  :effect (and (not (clear ?bt)) (not (on ?bm ?bf))
               (on ?bm ?bt) (clear ?bf)))

(:action move-b-to-t
  :parameters (?bm ?bf)
  :precondition (and (clear ?bm) (on ?bm ?bf))
  :effect (and (not (on ?bm ?bf))
               (on-table ?bm) (clear ?bf)))

(:action move-t-to-b
  :parameters (?bm ?bt)
  :precondition (and (clear ?bm) (clear ?bt)
                     (on-table ?bm) (not (= ?bm ?bt)))
  :effect (and (not (clear ?bt)) (not (on-table ?bm))
               (on ?bm ?bt))))
```

Appendix 2: The Blocks World in PROLOG

```
       /* Precondition axioms */
poss(move-b-to-b(X,Y,Z),S):- clear(X,S),clear(Z,S), on(X,Y,S),not X=Z.
poss(move-b-to-t(X,Y),S) :- clear(X,S), on(X,Y,S).
poss(move-t-to-b(X,Z),S) :- ontable(X,S), clear(X,S), clear(Z,S).

        /* Succesor state axioms */
on(X,Y, [move-b-to-b(X,Z,Y) | S]).
on(X,Y, [move-t-to-b(X,Y) | S]).
on(X,Y, [A | S]) :- on(X,Y,S), not A=move-b-to-b(X,Y,Z),
                            not A=move-b-to-t(X,Y).

ontable(X, [move-b-to-t(X,Y) | S]).
ontable(X, [A | S]) :- ontable(X,S), not A=move-t-to-b(X,Y).

clear(X, [move-b-to-b(Y,X,Z) | S]).
clear(X, [move-b-to-t(Y,X) | S]).
clear(X, [A | S]) :- clear(X,S), not A=move-b-to-b(Y,Z,X)),
                            not A=move-t-to-b(Y,X).
```

References

[1] Vitaliy Batusov and Mikhail Soutchanski. A logical semantics for PDDL+. In *29th International Conference on Automated Planning and Scheduling, ICAPS 2019*, pages 40–48. AAAI Press, 2019.

[2] Riccardo De Benedictis, Nicola Gatti, Marco Maratea, Aniello Murano, Enrico Scala, Luciano Serafini, Ivan Serina, Elisa Tosello, Alessandro Umbrico, and Mauro Vallati. Preface to the Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (IPS-RCRA-SPIRIT 2023). In *Proceedings of the Italian Workshop on Planning and Scheduling, RCRA Workshop on Experimental evaluation of algorithms for solving problems with combinatorial explosion, and SPIRIT Workshop on Strategies, Prediction, Interaction, and Reasoning in Italy (IPS-RCRA-SPIRIT 2023) co-located with 22th International Conference of the Italian Association for Artificial Intelligence (AI* IA 2023)*, 2023.

[3] Daniel Bryce and Subbarao Kambhampati. Planning Graph Based Reachability Heuristics. *AI Mag.*, 28(1):47–83, 2007.

[4] Stephen A. Cook and Yongmei Liu. A complete axiomatization for blocks world. *J. Log. Comput.*, 13(4):581–594, 2003.

[5] Herbert Enderton. *A Mathematical Introduction to Logic*. Harcourt Press, 2nd edit., 2001.

[6] Hector Geffner and Blai Bonet. *A Concise Introduction to Models and Methods for Automated Planning*. Morgan & Claypool Publ., 2013.

[7] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning: Theory and Practice*. Morgan Kaufmann, 2004.

[8] C. Cordell Green. Application of theorem proving to problem solving. In *Proceedings of the 1st International Joint Conference on Artificial Intelligence (IJCAI), Washington, DC, USA, May 7-9, 1969*, pages 219–240, 1969.

[9] Claude Cordell Green. *"The Application of Theorem Proving to Question-Answering Systems"*. PhD thesis, Stanford Univ., available at https://www.kestrel.edu/home/people/green/publications/green-thesis.pdf https://en.wikipedia.org/wiki/Cordell_Green, 1969.

[10] Patrik Haslum, Nir Lipovetzky, Daniele Magazzeni, and Christian Muise. *An Introduction to the Planning Domain Definition Language*. Synthesis Lectures on Artificial Intelligence and Machine Learning. Morgan & Claypool Publishers, 2019.

[11] Malte Helmert. The Fast Downward Planning System. *J. Artif. Intell. Res.*, 26:191–246, 2006.

[12] Helmert et. al. Fast Downward at Github. https://github.com/aibasel/downward, 2022. Accessed: 2022-11-17.

[13] Jörg Hoffmann and Bernhard Nebel. The FF Planning System: Fast Plan Generation Through Heuristic Search. *J. Artif. Intell. Res.*, 14:253–302, 2001.

[14] Gerhard Lakemeyer and Hector J. Levesque. Evaluation-based reasoning with disjunctive information in first-order knowledge bases. In *Proc of the 8th KR-2002*, pages 73–81, 2002.

[15] H.J. Levesque, F. Pirri, and R. Reiter. Foundations for the situation calculus. *Linköping Electronic Articles in Computer and Information Science. Available at: http://www.ep.liu.se/ea/cis/1998/018/*, vol. 3, N 18, 1998.

[16] Fangzhen Lin. Situation calculus. In *Handbook of Knowledge Representation*, volume 3 of *Foundations of Artificial Intelligence*, pages 649–669. Elsevier, 2008.
[17] Fangzhen Lin and Raymond Reiter. How to Progress a Database. *Artificial Intelligence*, 92:131–167, 1997.
[18] Nir Lipovetzky and Hector Geffner. Width-based algorithms for classical planning: New results. In *21st European Conference on AI, ECAI-2014*, pages 1059–1060, 2014.
[19] Nir Lipovetzky and Hector Geffner. Best-first width search: Exploration and exploitation in classical planning. In *31st AAAI-2017*, pages 3590–3596, 2017.
[20] Lipovetzky and Geffner. Best First Width Search Planner, Github repository. https://github.com/nirlipo/BFWS-public, 2022. Accessed: 2022-11-17.
[21] Yongmei Liu. *Tractable Reasoning in Incomplete First-Order Knowledge Bases*. PhD thesis, Department of Computer Science, University of Toronto, 2005.
[22] Yongmei Liu and Gerhard Lakemeyer. On First-Order Definability and Computability of Progression for Local-Effect Actions and Beyond. In *21st IJCAI-2009*, pages 860–866, 2009.
[23] Carlos Linares López, Sergio Jiménez Celorrio, and Angel García Olaya. The deterministic part of the seventh international planning competition. *Artif. Intell.*, 223:82–119, 2015.
[24] Arman Masoumi, Megan Antoniazzi, and Mikhail Soutchanski. Modeling Organic Chemistry and Planning Organic Synthesis. In *Global Conference on AI, GCAI-2015, Georgia*, volume 36 of *EPiC Series in Computing*, pages 176–195. EasyChair, 2015.
[25] John McCarthy. Situations, actions and causal laws. Technical Report Memo 2, Stanford University AI Laboratory, Stanford, CA, 1963. Reprinted in Marvin Minsky, editor, Semantic Information Processing, MIT Press, 1968.
[26] John McCarthy and Patrick Hayes. Some Philosophical Problems from the Standpoint of Artificial Intelligence. In B. Meltzer and D. Michie, editors, *Machine Intelligence*, volume 4, pages 463–502. Edinburgh Univ. Press, 1969.
[27] Edwin P. D. Pednault. ADL and the State-Transition Model of Action. *J. of Logic and Comput.*, 4(5):467–512, 1994.
[28] Fiora Pirri and Ray Reiter. Some contributions to the metatheory of the situation calculus. *Journal of the ACM (JACM)*, 46(3):325–361, 1999.
[29] Hadi Qovaizi. Efficient Lifted Planning with Regression-Based Heuristics, Master Thesis. Technical report, TMU, Toronto Metropolitan (formerly Ryerson) University, Department of Computer Science, Dec 2019.
[30] Raymond Reiter. An Approach to Deductive Question-Answering. BBN Technical Report 3649 (Accession Number : ADA046550), Bolt Beranek and Newman, Inc., 1977.
[31] Raymond Reiter. On Closed World Data Bases. In *Logic and Data Bases*, pages 55–76. Plenum, 1978.
[32] Raymond Reiter. Equality and Domain Closure in First-Order Databases. *J. ACM*, 27(2):235–249, 1980.
[33] Raymond Reiter. The Frame Problem in the Situation Calculus: A Simple Solution (sometimes) and a Completeness Result for Goal Regression. In V. Lifschitz, editor, *AI and Mathematical Theory of Computation: Papers in Honor of John McCarthy*, pages 359–380, San Diego, 1991. Academic Press.
[34] Raymond Reiter. Proving Properties of States in the Situation Calculus. *Artif. Intell.*,

64(2):337–351, 1993.

[35] Raymond Reiter. *Knowledge in Action. Logical Foundations for Specifying and Implementing Dynamical Systems*. MIT, http://cognet.mit.edu/book/knowledge-action, 2001.

[36] Silvia Richter and Matthias Westphal. The LAMA planner: Guiding cost-based anytime planning with landmarks. *J. Artif. Intell. Res.*, 39:127–177, 2010.

[37] Mikhail Soutchanski. Planning as Heuristic Controlled Reasoning in the Situation Calculus. PROLOG source code, TMU (formerly Ryerson), Dep. of Computer Science, https://www.cs.torontomu.ca/~mes/, Toronto, Canada, July 2017.

[38] Mikhail Soutchanski and Ryan Young. Planning as theorem proving with heuristics. *CoRR, https://doi.org/10.48550/arXiv.2303.13638*, 2023.

[39] R. Waldinger. Achieving Several Goals Simultaneously. In *Machine Intelligence*, volume 8, pages 94–136, Edinburgh, Scotland, 1977. Ellis Horwood.