Faster Stackelberg Planning via Symbolic Search and Information Sharing

Álvaro Torralba,[1] Patrick Speicher,[2] Robert Künnemann,[2] Marcel Steinmetz,[3] Jörg Hoffmann [3]

[1] Aalborg University, Denmark
[2] CISPA Helmholtz Center for Information Security, Germany
[3] Saarland University, Saarland Informatics Campus, Germany
alto@cs.aau.dk, {patrick.speicher,robert.kuennemann}@cispa.saarland, {steinmetz,hoffmann}@cs.uni-saarland.de

Abstract

Stackelberg planning is a recent framework where a leader and a follower each choose a plan in the same planning task, the leader's objective being to maximize plan cost for the follower. This formulation naturally captures security-related (leader=defender, follower=attacker) as well as robustness-related (leader=adversarial event, follower=agent) scenarios. Solving Stackelberg planning tasks requires solving many related planning tasks at the follower level (in the worst case, one for every possible leader plan). Here we introduce new methods to tackle this source of complexity, through sharing information across follower tasks. Our evaluation shows that these methods can significantly reduce both the time needed to solve follower tasks and the number of follower tasks that need to be solved in the first place.

Introduction

Stackelberg planning (Speicher et al. 2018a) is a recent framework inspired by Stackelberg security games (Tambe 2011; Shieh et al. 2014). It models a single exchange of adversarial plan choice between two agents, *leader* and *follower*, acting in the same planning task. The leader's objective is to maximize the follower's plan cost. Solutions form a Pareto front containing pairs of leader/follower plans. This captures security-related scenarios like network security, where the leader must make it as hard as possible for an attacker to harm the system. It can also capture robustness-related scenarios, by assessing the impact of an adversarial event (constructed by the leader) on the cost of the follower's plan (the agent acting in the domain at hand). Stackelberg planning has been used in large-scale studies to evaluate the efficacy of new protocol proposals for the email infrastructure (Speicher et al. 2018b) and for the web (Tizio 2018) where performance and flexibility are key features.

Previous work by Speicher et al. (2018a) introduced a search algorithm in the space of leader actions, where each node corresponds to solving a follower task. Thus, one may need to solve exponentially many follower subtasks, each of which is a cost-optimal classical planning task.

We introduce Symbolic Leader Search (**SLS**), a new algorithm for solving Stackelberg planning tasks. **SLS** aims to effectively reuse as much information as possible between the different subtasks. We achieve this (1) by recognizing that most follower subtasks can be solved with bounded cost suboptimal planning techniques, which are often more efficient than optimal planning algorithms; and (2) by using a symbolic representation to share information across searches. Instead of considering the subtasks in isolation, **SLS** symbolically represents all possible follower subtasks that the leader can reach with a given cost. Not only does this avoid the explicit enumeration of all possible states in the leader state space, which can potentially provide an exponential advantage in terms of memory and time, but it also allows for sharing information, e.g., by reusing upper bound functions for cost-bounded planning algorithms.

Besides this algorithmic contribution, we also introduce an extension of Stackelberg planning, using soft goals and net-benefit planning at the follower level. They generalize the follower's goal and naturally occur both in the security-related and robustness-related scenarios. In the former, we can now model attackers that wish to inflict maximal damage (e.g., sum of users associated to compromised domains). In the latter, we can now model agents that maximize the achievable benefits subject to the damage inflicted by the adversarial event. Solving such more general Stackelberg planning tasks is, in principle, straightforward: we can use the well-known compilation from net-benefit to classical planning (Keyder and Geffner 2009).

Our empirical evaluation shows that **SLS** outperforms previous approaches for Stackelberg planning. The improvement is consistent, and is particularly pronounced in tasks with large leader action spaces and in net-benefit Stackelberg planning where follower subtasks are inherently harder to solve. We thus significantly extend the scope of Stackelberg planning tools.

Background

Classical Planning

A SAS$^+$ classical planning task is a tuple $(\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ where $\mathcal{V}$ is a set of variables of finite domain (Bäckström and Nebel 1995). A state is a value assignment to all variables in $\mathcal{V}$, and a partial state p is an assignment to a subset of variables $\mathcal{V}_p \subseteq \mathcal{V}$. Given a (partial) state s, and a set of variables $V \subseteq \mathcal{V}_s$, $s|_V$ denotes the projection of s onto V. Given two pairs of partial states s, t with disjoint variables, we define their union $s \cup t$ as a partial state over $\mathcal{V}_s \cup \mathcal{V}_t$ that agrees with both s and t. A partial state s satisfies another p (written $s \models p$) if $s[v] = p[v]$ for all $v \in \mathcal{V}_p \cup \mathcal{V}_s$. $\mathcal{I}$ is the initial state, and $\mathcal{G}$ is a partial state that represents the goal. $\mathcal{A}$ is a set of actions, where each action $a \in \mathcal{A}$ has a precondition, $pre(a)$, and an effect $\mathit{eff}(a)$, which are partial states. An action a is applicable in a state s iff $s \models pre(a)$. The resulting successor of applying a in s is $s[[a]] = s|_{\mathcal{V} \setminus \mathcal{V}_{\mathit{eff}(a)}} \cup \mathit{eff}(a)$.

A plan is a sequence of actions that go from $\mathcal{I}$ to any state s s.t. $s \models \mathcal{G}$. Moreover, each action has a cost $c(a) \in \mathbb{R}_0^+$, and the cost of a plan is the summed up cost of all its actions. A plan is optimal if it has a minimum cost.

Optimal planning is the problem of finding an optimal plan for any given planning task. Cost-bounded planning is the problem of, given a planning task and a cost bound B, returning a plan that has cost lower or equal to B or, if no such plan exists, returning “unsolvable”.

Symbolic search is a well-known approach for exhaustive state space exploration in model-checking (McMillan 1993) and cost-optimal planning (Edelkamp and Kissmann 2009; Torralba et al. 2017). In symbolic search, Binary Decision Diagrams (BDDs) (Bryant 1986) are used to compactly represent sets of states as functions that map variable assignments to true or false, depending on whether such an assignment belongs to that set or not. We use the term V-BDD to denote a set of partial assignments over a specific subset of variables $V \subseteq \mathcal{V}$. BDDs offer a compact representation that often has an exponential gain in memory efficiency over listing all states in the set. Moreover, standard BDD operations can be used to operate on sets of states, e.g., the union of two sets of states corresponds to the disjunction of their BDDs ($S \vee S'$), and their intersection corresponds to the respective conjunction ($S \wedge S'$). The runtime of these operations depends on the BDD size, but not on the number of states in the set, translating the gain in memory to a gain in time.

Stackelberg Planning

A Stackelberg planning task is a tuple ($\mathcal{V}$, $\mathcal{A}^L$, $\mathcal{A}^F$, $\mathcal{I}$, $\mathcal{G}$). The set of actions is split into the leader actions $\mathcal{A}^L$ and the follower actions $\mathcal{A}^F$. The set of variables $\mathcal{V}$ can thus be categorized into three subsets. Leader ($\mathcal{V}^L$) and follower ($\mathcal{V}^F$) variables are those that appear in the effect of a leader/follower action. The subtask variables ($\mathcal{V}^T \subseteq \mathcal{V}^L$) are those leader variables that appear in the goal or in the precondition of a follower action. Note that these subsets may overlap. We assume w.l.o.g. that $\mathcal{V}_\mathcal{G} \subseteq \mathcal{V}^F$.

Each sequence of leader actions π^L that is applicable on $\mathcal{I}$ corresponds to an assignment to $\mathcal{V}^T$, which is $\mathcal{I}[[\pi^L]]|_{\mathcal{V}^T}$. Given an assignment X to $\mathcal{V}^T$, one can construct a planning task $\Pi_X = (\mathcal{V}^F, A_X, \mathcal{I}_X, \mathcal{G})$ that corresponds exactly to the follower subtask. All such tasks are defined over the same set of variables, $\mathcal{V}^F$, and contain the subset of follower actions $A_X \subseteq \mathcal{A}^F$ such that preconditions over variables in $\mathcal{V}^T \setminus \mathcal{V}^F$ are satisfied by X. The initial state $\mathcal{I}_X$ is $X|_{\mathcal{V}^F} \cup \mathcal{I}|_{\mathcal{V}^F \setminus \mathcal{V}^T}$.

A pair (π^L, π^F) is a Stackelberg plan if π^L is a sequence of leader actions applicable on $\mathcal{I}$, and π^F is an optimal plan for the follower subtask $\Pi_{\mathcal{I}[[\pi^L]]}$. A Stackelberg plan (π_1^L, π_1^F) is dominated by another (π_2^L, π_2^F) if $c(\pi_1^L) \geq c(\pi_2^L) \wedge c(\pi_1^F) \leq c(\pi_2^F)$. The domination is strict if one of the inequalities is strict. The Stackelberg planning problem is computing a Pareto front, i.e., a set of non strictly-dominated Stackelberg plans which dominate all other Stackelberg plans.

Let Π^S be a Stackelberg planning task, and PF a Pareto front of Π^S. We define the set of cost entries of PF as $c(PF) = \{(c(\pi^L), c(\pi^F)) \mid (\pi^L, \pi^F) \in PF\}$. Note that, even though there are many possible Pareto fronts for a planning task, they all have the same set of cost entries. Moreover, it is sufficient to include a single plan (π^L, π^F) in the Pareto front for each cost entry. Therefore, each task has a unique Pareto front size defined as $|PF(\Pi^S)| = |c(|PF(\Pi^S)|)|$.

Stackelberg tasks are solved by performing a search in the space of leader actions, and solving the corresponding follower subtask at every node. Previous work by Speicher et al. (2018a) proposed to use iterative-deepening search (IDS) on the leader state space. This allows for a simple but important optimization, caching the plan of the parent and testing whether the same plan is a solution for any successor. This avoids many unnecessary calls to the follower subsolver, greatly speeding the search.

Another important enhancement is *upper-bound pruning*. Given a global upper bound on the follower cost, we can prune any leader state that has leader cost greater than any other state with a follower cost equal to the upper bound. This upper bound can be determined whenever $\mathcal{V}^L \cap \mathcal{V}^F = \emptyset$.[1] In that case, all follower subtasks have the same initial state and, therefore, a follower task, Π^+, can be defined where all actions that can be disabled by the leader have been removed. As removing actions can only increase a task's solution cost, the follower cost of Π^+ is an upper bound on the follower cost of any other follower subtask.

Symbolic Leader Search

As in previous work by Speicher et al. (2018a), our main algorithm, symbolic leader search, **SLS** performs an exploration on the space of actions for the leader, using classical planning algorithms to solve the follower subtasks that result from leader plans. Thus, we can divide the overall algorithm into two parts: the search on the leader space and the follower subtasks. To solve the follower subtasks, previous work considered an optimal planner using explicit A^* search with the LM-cut heuristic (Helmert and Domshlak 2009). We also consider symbolic bidirectional blind search (Torralba et al. 2017), which, as we will discuss in the following subsections, has a synergy with **SLS** due to employing backward search and a symbolic representation.

SLS focuses on how to reuse information among subtasks. Outside of the subsolver, **SLS** (1) exploits the follower cost of previous follower subtasks that have been optimally solved as a bound for subsequent calls to the follower solver;

[1] This is a natural property whenever the leader has higher-level actuators than the follower: e.g. changing a road network versus moving in it; changing a network configuration versus trying to break into it. In these cases, the effect of leader actions (e.g., block a road, update a server) cannot be undone by the follower.

Algorithm 1: Symbolic Leader Search (**SLS**)

```
   Input: Stackelberg Task $\Pi^S = (\mathcal{V}, \mathcal{A}^L, \mathcal{A}^F, \mathcal{I}, \mathcal{G})$
   Output: Pareto front
1  $c^L \leftarrow 0, c^F \leftarrow -1$
2  $F^+ \leftarrow$ GlobalUpperBound($\Pi^S$)
3  $Solved \leftarrow \emptyset$
4  $ParetoFront \leftarrow \emptyset$
5  while $c^F < F^+ \wedge c^L < \infty$ do
6      $S^L \leftarrow$ UniformCostSearchLayer($c^L$)
7      $S^T \leftarrow S^L|_{\mathcal{V}^T} \wedge \neg Solved$
8      while $S^T \neq \emptyset$ do
9          $\Pi^F \leftarrow$ ChooseSubtask($S^T$)
10         $plan, lb, ub \leftarrow$ Planner($\Pi^F, c^F, lb, ub$)
11         if plan not found then
12             $c^F \leftarrow \infty$
13             $S^T \leftarrow \emptyset$
14         else
15             $c^F \leftarrow \max(c^F, cost(plan))$
16             $sol, ub \leftarrow$ RegressPlan($plan, ub$)
17             $S^T \leftarrow S^T \wedge \neg sol$
18             $Solved \leftarrow Solved \vee sol$
19     AddIfNotDominated($ParetoFront, c^L, c^F$)
20     $c^L \leftarrow nextL$;
21 return ParetoFront
```

and (2) keeps track of the set of follower subtasks that are solved by some previously found plan. Inside the subsolver, three sources of information are exploited: a lower bound c^F on the follower cost on other subtasks with the same or lower leader cost, and two functions lb and ub. These functions allow for reusing information across follower subtasks and are discussed in detail in the next subsection.

SLS (see Algorithm 1) enumerates the leader search space layer by layer, using symbolic uniform-cost search on the space of leader actions over the set of leader variables. After exhausting a layer, the uniform cost search returns a set of leader states S^L that is reachable with leader cost c^L. These states are represented as a $\mathcal{V}^L$-BDD, avoiding their explicit enumeration, so that the algorithm can scale in tasks with large leader state spaces. Exploring the leader space layer by layer allows us to keep track of several global variables, namely c^L, c^F, the current Pareto front, and $Solved$. c^L is the current leader cost. As all layers of leader cost up to c^L have already been fully explored, the Pareto front up to that point has already been computed and can be reported up front, providing the user with partial results before the entire search is finished. Moreover, we also keep track of the current follower bound c^F, which is the highest optimal solution cost for any solved subtask with leader cost c^L or less. All subtasks with an optimal follower cost lower or equal to c^F do not contribute to the final Pareto front, so optimal solutions are not needed for them. Finally, we keep track of a set of solved subtasks, $Solved$, which are known to have an optimal cost lower or equal than the current c^F bound.

In each iteration, the first step is to obtain the set of candidate subtasks that could possibly add a new entry in the Pareto front, S^T, represented as a $\mathcal{V}^T$-BDD. This is done with standard BDD operations (see line 7), projecting S^L onto those leader variables that are relevant for the follower ($\mathcal{V}^T$), and subtracting all previously solved subtasks.

While there are follower subtasks in S^T that remain to be solved, one of them is chosen and solved by the follower solver. If the follower task is unsolvable, then we have found the last entry of the Pareto front (with $F = \infty$) and the algorithm ends. Otherwise, we update the value of c^F. The function RegressPlan returns the set of follower subtasks for which the same plan can be applied, as detailed in the next section. Different strategies could be applied, as long as it is guaranteed that the current subtask belongs to this set, so that no subtask is solved more than once. Once again, while there may be exponentially many such subtasks, we take advantage of a compact BDD representation to keep this computation tractable. All those subtasks are removed from S^T with a single BDD operation (line 17), possibly avoiding the enumeration of exponentially many leader states.

One important detail when using BDDs is the variable ordering, since the size of the BDD (and therefore the memory and running time of the algorithm) can be heavily affected by exponential factors under different variable orderings (Bryant 1986; Kissmann and Hoffmann 2014). We split the variables into three partitions ordered from top to bottom: (i) leader-only variables $\mathcal{V}^L \setminus \mathcal{V}^F$, (ii) common variables $\mathcal{V}^L \cup \mathcal{V}^F$, and (iii) follower-only variables $\mathcal{V}^F \setminus \mathcal{V}^L$. The intuition is that this simplifies the operations that project out these sets of variables (e.g., in line 7 of Algorithm 1). The order inside each partition is decided independently by a standard algorithm used to optimize BDD variable orderings in planning (Kissmann and Edelkamp 2011), only considering leader variables for partition (i), and only considering follower actions for partitions (ii) and (iii). This is important when the follower solver employs symbolic bidirectional search, as taking into account leader variables when optimizing the order of follower variables may be harmful for the performance of the follower subsolver.

Cost-Bounded Follower Solvers

A core idea of our approach is that it is not necessary to compute the optimal solution for every follower subtask to obtain the optimal Pareto front, as done previously. Instead, it suffices to compute the optimal solution for those subtasks that belong to the optimal Pareto front (i.e., at most one for each leader cost). For all other follower subtasks, it is enough to find any solution of lower or equal cost than any entry in the Pareto front with lower or equal leader cost. For this, one can use specialized bounded cost search algorithms (Thayer et al. 2012; Haslum 2013; Stern et al. 2014; Dobson and Haslum 2013) using the current c^F-bound. In practice, this is often much more efficient, even when the bound is the optimal follower cost (and therefore an optimal solution must be found anyway), since we can avoid proving optimality.

More precisely, our follower subtask solvers receive as input a classical planning task Π, and a cost bound B. The corresponding solution is a plan of cost B or less if one exists. Otherwise, it must return an optimal plan if one exists or "unsolvable" otherwise. Note that this is somewhere between cost-bounded and optimal planning. A straightforward approach is to use a cost-bounded planner and run an optimal planner in case it fails to find a solution.

To transform any search-based follower solver into a cost-bounded version, we consider two functions that map states of the follower task to numerical values: lb and ub:

- lb is an admissible estimate, i.e., it provides a lower bound on the goal distance from every state.
- ub is an upper bound on solution cost such that for all non-goal states s, $ub(s) \geq \min_{a \in \mathcal{A}^F} c(a) + ub(s[[a]])$. This property ensures that a plan can be reconstructed in polynomial time in the size of the task and the plan length if there are no 0-cost actions.[2]

We assume that lb and ub are represented symbolically, as a function from upper bounds to $\mathcal{V}^F$-BDDs. That is, for each possible return value, we keep a BDD that represents the set of follower states with this value. This is a common way to represent heuristics in symbolic search planning (Kissmann and Edelkamp 2011). Figure 1 summarizes how these functions are obtained, as detailed in the next sections, and used.

We modify all search algorithms to use ub as follows. During the search, any time a state s is generated, we check whether $g(s) + ub(s) \leq F$, where $g(s)$ is the cost of the path from $\mathcal{I}$ to s. If so, we stop the search and return a plan of cost $g(s) + ub(s)$ passing through s. As ub is represented symbolically, this is also possible in symbolic search.

To aggressively look for a solution below the cost bound B, we use a greedy best-first search with the FF heuristic (Hoffmann and Nebel 2001) and a time limit of 1s, pruning any node whose g-value plus the lower bound lb is greater than B. Due to the time limit and the inadmissible heuristic, this is an incomplete configuration, so if it finishes without finding a solution within the bound, we run one of the optimal solvers on the same subtask.

Even though using lb is straightforward for A^*-based solvers by taking the minimum among the heuristic and the value of lb, we do not use it here for practical reasons: extracting lb requires using the symbolic bidirectional search solver and none of our configurations use it with LM-cut. We do not use lb to bolster the symbolic search solver, as it performs blind search without using heuristics, and experimental analysis show that using heuristics is not always helpful in such setting (Speck, Geißer, and Mattmüller 2020).

Transferring Bounds Across Subtasks

The similarity between follower subtasks can be directly exploited by sharing lower and upper bounds on solution cost among them. As they share a common set variables and a common goal, we can define dominance across follower subtasks in the following way:

Definition 1. *Let Π_1 and Π_2 be two planning tasks with a common set of variables. We say that Π_1 dominates Π_2 if $h_1^*(s) \leq h_2^*(s)$ for all states s.*

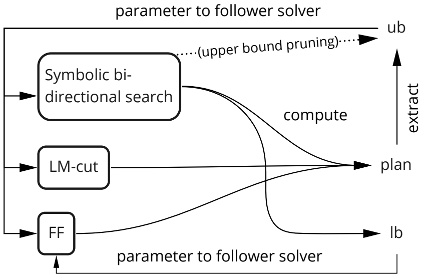
parameter to follower solver
ub
(upper bound pruning)
Symbolic bi-directional search
compute
extract
LM-cut
plan
FF
lb
parameter to follower solver

Figure 1: Computation and use of lower and upper bounds.

A sufficient criterion to identify that a task dominates another is to compare the set of actions.

Proposition 1. *Let $\Pi_1 = \langle \mathcal{V}^F, A_1, I_1, G \rangle$ and $\Pi_2 = \langle \mathcal{V}^F, A_2, I_2, G \rangle$ be two tasks with a common set of variables and goal, s.t. $A_2 \subseteq A_1$. Then, Π_1 dominates Π_2.*

Whenever dominance is established, we can use it to transfer bounds.

Proposition 2. *Let Π_1 and Π_2 be two tasks such that Π_1 dominates Π_2. Then any lower bound function for Π_1 is a lower bound for Π_2, and any upper bound function for Π_2 is an upper bound for Π_1.*

It is well known that symbolic backward search provides the exact goal distance on all states around the goal. This has been used in the past in order to compute admissible heuristics (Torralba, López, and Borrajo 2018). Therefore, whenever symbolic bidirectional search is used as a solver, we use the perimeter created by the backward search as a lower and upper bound that can be transferred to other tasks.

This is mainly useful to obtain lower bound functions for the FF solver, as follower subtasks are explored by increasing leader cost and, typically, leader actions tend to disable actions for the follower and not to enable them (since the objective of the leader is to increase follower cost). If that is the case, the initial task that is optimally solved at the beginning of the algorithm dominates all other tasks and therefore any symbolic backward search performed on it can be used as a lower bound function on all subsequent subtasks.

Moreover, we can also transfer any backward search on Π^+ as an upper bound function to all other subtasks. To boost this effect, whenever the upper-bound pruning optimization is enabled, we force symbolic bidirectional search to choose the backward direction for up to 30 seconds.

Plan Reuse Strategies

Inspired by a technique presented by Kolobov, Mausam, and Weld (2012) for probabilistic planning, we use the well-known notion of regression to *generalize* from individual follower plans. Generalization in our case pertains to two parts: (1) allowing for early termination of subsequent calls to the follower subsolver, and (2) removing entire follower subtasks from consideration.

Algorithm 2 depicts the function `RegressPlan`. It is called whenever a follower subtask has been solved in the leader search and a follower plan $\langle a_1, \dots, a_n \rangle$ have been

[2] In tasks with 0-cost actions, ub also needs to account for the number of 0-cost actions but we omit such details for simplicity.

Algorithm 2: `RegressPlan`

```
Input: Stackelberg Task ($\mathcal{V}$, $\mathcal{A}^L$, $\mathcal{A}^F$, $\mathcal{I}$, $\mathcal{G}$)
Input: Follower plan: $\langle a_1, \ldots, a_n \rangle$
Input: Previous upper bound function: $ub$
Output: Set of solved states and updated $ub$ function
1 $g \leftarrow 0$ ;
2 $C_{n+1} \leftarrow \mathcal{G}$ ;
3 $ub[g] \leftarrow ub[g] \lor BDD(C_{n+1})$ ;
4 for $i \in [n...1]$ do
5     $C_i \leftarrow regress_{\mathcal{V}^F \cup \mathcal{V}^T}(C_{i+1}, a_i)$ ;
6     $g \leftarrow g + c(a)$ ;
7     $ub[g] \leftarrow ub[g] \lor BDD(C_i)$ ;
8 return $BDD(C_1|_{\mathcal{V}^T}), ub$ ;
```

found. (If the follower subtask was unsolvable, the leader search terminates anyway.) `RegressPlan` then identifies a condition C_i at every step $1 \leq i \leq n$ along the found plan such that for every state s, $s \models C_i$ implies that $\langle a_i, \ldots, a_n \rangle$ is a follower plan for s. The conditions are computed by traversing the plan back to front, and computing in each step the regression of the respective action with the previously computed condition. In the regression, we must take into account all relevant variables $\mathcal{V}^F \cup \mathcal{V}^T$ to correctly compute these conditions. $\mathcal{V}^F$ alone identifies states within all follower tasks, while $\mathcal{V}^T$ provides the context, i.e., it constrains the follower subtasks to those in which the actions in the plan have not been disabled by the leader.

For the purpose of (1), the extracted conditions feed directly into the follower-cost upper bound ub. For every state that satisfies C_i, we have now found a plan with cost $c(\langle a_i, \ldots, a_n \rangle)$. The ub function is updated with appropriate BDD operations (lines 3 and 7). When a new follower subtask $s^T \in S^T$ is chosen to be solved, ub is transformed into an upper bound function for the corresponding follower subtask by computing $(ub[g] \land BDD(s^T|_{\mathcal{V}^T \setminus \mathcal{V}^F}))|_{\mathcal{V}^F}$ for each follower cost g. This restricts the upper bounds to those computed plans whose conditions projected onto the leader preconditions are still satisfied. In a final step, the resulting BDDs are projected onto $\mathcal{V}^F$ to obtain the $\mathcal{V}^F$-BDD that represents states of the follower subtask.

Regarding (2), consider the condition C_1 computed for the entire plan $\langle a_1, \ldots, a_n \rangle$. Since all variables of $\mathcal{V}^F \setminus \mathcal{V}^T$ in C_1 must necessarily be assigned to their respective values in $\mathcal{I}$, C_1 projected onto $\mathcal{V}^T$ hence identifies all subtasks with follower plan $\langle a_1, \ldots, a_n \rangle$. At this point, we know, however, that any new entry in the Pareto front must have an optimal follower plan cost higher than the cost of this plan. Hence, all subtasks identified by $C_1|_{\mathcal{V}^T}$ can safely be pruned. Note that $C_1|_{\mathcal{V}^T}$ always represents the subtask that was solved by the last call to the follower sub-solver. This property is required (and sufficient by itself) to guarantee the termination of the overall leader search.

Net-Benefit Stackelberg Planning

Previous work on Stackelberg planning assumed that all follower goals must be achieved. However, for many applications, this is too restrictive. Consider, for example, the robustness of a road network in a logistics-like scenario. The follower's goal is to deliver packages from their respective pick-up location to different target locations. As soon as the leader blocks any pick-up or delivery location, the follower task becomes unsolvable. Consequently, all configurations where a package cannot be delivered have the same utility, regardless of how many packages still can. This limits the scope of the robustness analysis. It is more interesting to identify *how many* packages can still be delivered given the damage inflicted by the leader.

Similar in the pentesting scenario. Here, the follower tries to compromise a set of assets, i.e., computers holding sensitive information. Prior work by Speicher et al. (2018a) considered the goal of accessing *all* assets. In that case, once one of the assets has been protected, the follower cost is infinite, even if other assets can be compromised. Vice versa, if the attacker's goal is to access *any* assets, then a defense measure that protects a single asset is regarded equally good as one that protects many more. In practice, assets hold different kinds of information that are valuated during risk assessment (Tsiakis and Stephanides 2005). Therefore, it is desirable to quantify the leaked information by associating a utility to each piece of information. This is used in various DAG-based modelling approaches (Kordy, Piètre-Cambacédès, and Schweitzer 2014), which are typically used on the local network level, or when quantifying the potential impact of compromised infrastructure in the internet, e.g., name servers (Ramasubramanian and Sirer 2005) or content delivery networks (Simeonovski et al. 2017).

Net-benefit Stackelberg planning allows the specification of soft goals, each being a partial state goal with an associated utility. The goal of the follower is to find a plan that maximizes utility minus cost. This is a natural way to model problems where the follower has more than a single goal.

Net-benefit Stackelberg planning tasks can be handled by the same algorithms as Stackelberg planning. We simply compile the utilities of soft goals into action cost, as done in classical planning (Keyder and Geffner 2009). Namely, for each soft goal we introduce an auxiliary action that achieves this fact[3] with a cost equal to the utility. These auxiliary actions remove a flag that is a precondition for all ordinary actions, so that once an auxiliary action achieves a soft goal, no ordinary actions are applicable.

Evaluation

We implemented our new **SLS** algorithm on top of the Stackelberg framework by Speicher et al. (2018a), built upon the Fast Downward planning system (Helmert 2006). We ran our experiments with the Downward Lab toolkit (Seipp et al. 2017) on an Intel Xeon CPU E5-2650 v3, 2.30 GHz. For each task, we set a 30 minute time limit and a 4 GB memory limit, ignoring the translate and preprocessing phase which is equal for all configurations. Our source code, benchmarks, and results are publicly available (Torralba et al. 2021).

[3] We assume that soft goals are single facts. Otherwise an auxiliary fact representing the soft goal needs to be introduced as well.

		$\|PF(\Pi^S)\|$		LMcut					Symbolic Bidirectional				
				IDS	**SLS**				IDS	**SLS**			
		avg	*max*	Π^+	—	+*ub*	+Π^+	+ FF	Π^+	—	+*ub*	+Π^+	+ FF
OLD	Logistics (416)	1.85	3	**26**	**26**	**26**	**26**	**26**	**18**	**18**	**18**	**18**	**18**
	Mystery (218)	1.59	3	168	**172**	**172**	**172**	**172**	137	145	150	150	**151**
	Pentesting (213)	1.26	2	**201**	149	149	168	167	150	149	149	**168**	167
	Rovers (474)	1.86	3	**30**	**30**	**30**	**30**	**30**	50	70	**71**	**71**	**71**
	Sokoban (224)	1.92	2	**218**	**218**	**218**	**218**	**218**	191	**196**	**196**	**196**	**196**
	Tpp (132)	2.00	2	**4**	**4**	**4**	**4**	**4**	8	**9**	**9**	**9**	**9**
	Visitall (310)	2.32	7	34	31	34	**35**	34	32	36	38	**41**	**41**
	$\sum$ (1987)			**681**	630	633	653	651	586	623	631	**653**	**653**
NEW	Logistics (140)	2.59	6	65	72	**90**	89	88	27	30	30	**31**	**31**
	Nomystery (128)	2.60	7	82	**86**	84	84	**86**	73	106	**107**	**107**	**107**
	Pentesting (234)	1.71	4	**234**	**234**	**234**	232	**234**	179	230	**233**	232	**233**
	Rovers (112)	1.84	3	**44**	**44**	**44**	**44**	**44**	77	**95**	94	94	94
	Tpp (130)	2.15	7	**70**	68	69	**70**	**70**	98	114	114	**118**	**118**
	Transport (140)	4.24	17	40	62	**68**	65	66	75	100	**102**	100	99
	Visitall (134)	2.95	7	96	97	109	109	**110**	72	101	**104**	**104**	**104**
	$\sum$ (1018)			631	663	**698**	693	**698**	601	776	784	**786**	**786**
NET	Logistics (140)	3.67	16	56	62	78	78	**86**	19	26	28	**29**	**29**
	Nomystery (128)	3.29	13	61	62	**66**	**66**	65	68	86	87	**97**	95
	Pentesting (234)	3.51	13	180	182	189	214	**217**	151	179	181	**189**	188
	Rovers (112)	3.51	9	37	40	40	**43**	**43**	54	86	87	**92**	**92**
	Tpp (130)	2.35	13	**61**	54	56	55	55	96	101	103	**116**	**116**
	Transport (140)	5.45	34	38	47	52	**58**	**58**	66	76	82	89	**90**
	Visitall (134)	4.06	13	57	58	64	68	**69**	53	73	75	**83**	79
	$\sum$ (1018)			490	505	545	582	**593**	507	627	643	**695**	689

Table 1: Maximum and average size of the Pareto front $|PF(\Pi^S)|$ on solved instances and coverage using LMcut and symbolic bidirectional search subsolvers. We enable **SLS**'s features one by one: reusing upper bounds (ub), upper bound pruning (Π^+) and cost-bounded search with the FF heuristic (FF). We highlight the best configurations for each subsolver.

Benchmark Set

We evaluate our algorithms on three benchmark sets: OLD, NEW, and NET. OLD is the one used by Speicher et al. (2018a). We include it to have a direct comparison with the same instances. It contains instances of classical planning IPC domains, but extended with n leader actions that disable preconditions of some follower actions. For each instance, we chose the versions ($n = 2, 4, 8, \ldots, 512$) out of 19 versions Speicher et al. considered. Upper bound pruning is applicable on all instances. All of them (with the exception of the Pentesting domain, cf.) involve the transportation of some objects to some locations. But the number of locations is small compared to the number of objects or goals, and thus succinct leader plans are often sufficient to render the follower's goal unsolvable. The left-hand side of Table 1 shows the maximum and average size of the Pareto front for each instance. Most solved instances in the OLD set have very small Pareto front. Note that a Pareto front size of 1 means that the only Pareto-optimal choice for the leader is doing nothing. A size of 2 indicates that it has only one other option, which is typically to make some follower goal(s) unreachable with few actions.

NEW extends the previous benchmark set with instances better suited for the evaluation of Stackelberg planning algorithms. We base our new instance set on the same domains, but increase the number of locations relative to the number of objects and goals. To do this, we generated instances of those domains with a size chosen s.t. the baseline planner can solve the instance in ≈10-60 seconds. Then, we scaled the number of locations. This provides the follower with more paths to the goal, and the leader with more options to increase the follower's cost, leading to larger Pareto fronts. We also replaced Mystery by the more modern Nomystery domain, and Sokoban (which does not admit a high number of locations without making the task significantly easier) with Transport. Compared to other domains, Transport features a more realistic road map, where locations are assigned coordinates in a plane and the action cost to travel between locations is their straight-line distance. This leads to larger Pareto fronts than any other domain.

NET is a set of net-benefit instances, discussed below.

Evaluation of SLS

Table 1 shows overall coverage. Of course, the choice of the underlying classical planner depends on the domain, e.g., LM-cut works best on Logistics and Sokoban, whereas symbolic bidirectional search is superior on Rovers, Transport and TPP. Our new algorithm, **SLS**, clearly outperforms the baseline, IDS (Speicher et al. 2018a), with both solvers. The only exception is the Pentesting domain where the symbolic

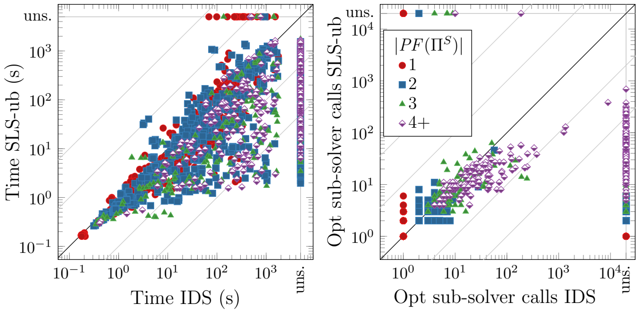

Figure 2: Total time and optimal follower searches for IDS and **SLS**-ub with symbolic bidirectional search.

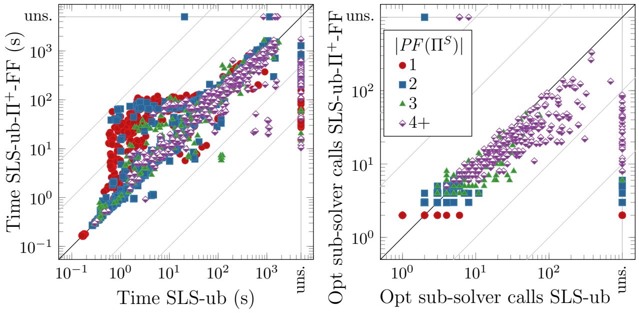

Figure 3: Total time and optimal follower searches of **SLS**-ub and **SLS**-ub-Π^+-FF with symbolic bidirectional search.

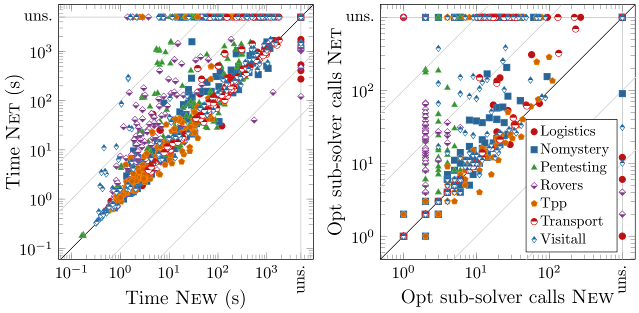

Figure 4: Comparison of NEW and NET instances when using **SLS**-ub with symbolic bidirectional search.

representation suffers due to the huge number of variables. The time spent at the leader search level is a low percentage of the total time, indicating that the main bottleneck is the time spent by the follower subsolvers. This means that the advantage of **SLS** mainly comes from two factors: (1) the number of calls to the follower subsolver, and (2) the speedup provided by sharing the bound functions.

Our ablation analysis, enabling an optimization at a time, provides insights on what the main reasons for the advantage of **SLS** are. The main difference between IDS and the basic version of **SLS** is that the latter explores the leader space with symbolic search layer by layer, reusing all previously found plans, instead of only the parent's plan. Hence the performance difference between both is mainly due to the number of calls to the subsolver. Moreover, using the ub function is almost always beneficial, speeding up each call to the follower subsolver by allowing an early termination.

Figure 2 shows a comparison of the baseline IDS, and **SLS**-ub in terms of runtime and number of subsolver calls. The plot displays a clear trend regarding the planner performance with respect to the Pareto front size. Whenever, the Pareto front size has only 1 or 2 entries, few follower subsearches are needed by both algorithms. In those tasks, the overall running time is only determined by how efficient the follower subsolver is and there is little room of improvement for reducing the number of calls to the subsolver or reusing information among different calls. Tasks with a Pareto front of 3 or more, however, require a much higher number of subsolver searches. By better reusing information among them, **SLS**-ub significantly reduces the number of searches, and the total time often by more than an order of magnitude.

Figure 3 shows in detail the effect of enabling upper-bound pruning (Π^+) and cost-bounded FF search. Both have up to $30s$ of pre-processing in the form of symbolic backward search to obtain upper (ub) and lower (lb) bounds. However, as the results of Table 1 indicate, this often pays off in terms of coverage and it is also slightly beneficial in runtime on instances where **SLS**-ub uses more than $100s$.

Evaluation of Net-benefit Planning

To analyze the gap in difficulty between standard and net-benefit Stackelberg planning, we introduce the NET instance set. The instances are the same as in NEW, but consider each goal fact to be an individual soft goal with a utility of 10000. This is significantly higher than the cost of any leader plan in these domains, so that the follower maximizes the number of achieved goals, while reducing its cost. Rather than modifying the PDDL encoding, we implemented the soft-goal compilation as a final step in Fast Downward's translator (Helmert 2009). Therefore, NET and NEW instances use the same variable representation so this constitutes a very direct evaluation of the impact on having soft goals.

Figure 4 compares **SLS**-ub on both instance sets, and other configurations have very similar results in this regard. Net-benefit instances are often much harder, though the impact depends on the domain. Higher impact can be observed in domains like Rovers or Pentesting, where the average Pareto front size increased the most (see Table 1). Many NEW instances of these domains have tiny Pareto fronts and require very few follower sub-searches. However, this is no longer true in their net-benefit counterparts. Coverage results show that **SLS** is particularly good on this set beating IDS even on the Pentesting domain, due to the higher proportion of hard tasks where many follower subtasks must be solved.

Conclusion

We introduced **SLS**, an algorithm for solving Stackelberg planning tasks. It exploits symbolic leader search and cost-bounded follower search to share information between subtasks. **SLS** thus consistently outperforms previous approaches, in particular when the leader action space is large or when we consider soft goals.

Acknowledgments

Álvaro Torralba was employed by Saarland University and the CISPA Helmholtz Center for Information Security during most of the development of this paper. This work was partially supported by DFG grant 389792660 as part of TRR 248 (see https://perspicuous-computing.science).

References

Bäckström, C.; and Nebel, B. 1995. Complexity Results for SAS$^+$ Planning. *Computational Intelligence* 11(4): 625–655.

Bryant, R. E. 1986. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers* 35(8): 677–691.

Dobson, S.; and Haslum, P. 2013. Heuristics for Bounded-Cost Search. In *Workshop on Heuristic Search and Domain Independent Planning (HSDIP'17)*.

Edelkamp, S.; and Kissmann, P. 2009. Optimal Symbolic Planning with Action Costs and Preferences. In Boutilier, C., ed., *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, 1690–1695. Pasadena, California, USA: Morgan Kaufmann.

Haslum, P. 2013. Heuristics for Bounded-Cost Search. In Borrajo, D.; Fratini, S.; Kambhampati, S.; and Oddi, A., eds., *Proceedings of the 23rd International Conference on Automated Planning and Scheduling (ICAPS'13)*. Rome, Italy: AAAI Press.

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

Helmert, M. 2009. Concise Finite-Domain Representations for PDDL Planning Tasks. *Artificial Intelligence* 173: 503–535.

Helmert, M.; and Domshlak, C. 2009. Landmarks, Critical Paths and Abstractions: What's the Difference Anyway? In Gerevini, A.; Howe, A.; Cesta, A.; and Refanidis, I., eds., *Proceedings of the 19th International Conference on Automated Planning and Scheduling (ICAPS'09)*, 162–169. AAAI Press.

Hoffmann, J.; and Nebel, B. 2001. The FF Planning System: Fast Plan Generation Through Heuristic Search. *Journal of Artificial Intelligence Research* 14: 253–302.

Keyder, E.; and Geffner, H. 2009. Soft Goals Can Be Compiled Away. *Journal of Artificial Intelligence Research* 36: 547–556.

Kissmann, P.; and Edelkamp, S. 2011. Improving Cost-Optimal Domain-Independent Symbolic Planning. In Burgard, W.; and Roth, D., eds., *Proceedings of the 25th National Conference of the American Association for Artificial Intelligence (AAAI'11)*, 992–997. San Francisco, CA, USA: AAAI Press.

Kissmann, P.; and Hoffmann, J. 2014. BDD Ordering Heuristics for Classical Planning. *Journal of Artificial Intelligence Research* 51: 779–804.

Kolobov, A.; Mausam; and Weld, D. S. 2012. Discovering Hidden Structure in Factored MDPs. *Artificial Intelligence* 189: 19–47.

Kordy, B.; Piètre-Cambacédès, L.; and Schweitzer, P. 2014. DAG-based attack and defense modeling: Don't miss the forest for the attack trees. *Computer science review* 13: 1–38.

McMillan, K. L. 1993. *Symbolic Model Checking*. Kluwer Academic Publishers.

Ramasubramanian, V.; and Sirer, E. G. 2005. Perils of transitive trust in the domain name system. In *Proceedings of the 5th ACM SIGCOMM conference on Internet Measurement*, 35–35.

Seipp, J.; Pommerening, F.; Sievers, S.; and Helmert, M. 2017. Downward Lab. https://doi.org/10.5281/zenodo.790461.

Shieh, E. A.; Jiang, A. X.; Yadav, A.; Varakantham, P.; and Tambe, M. 2014. Unleashing Dec-MDPs in Security Games: Enabling Effective Defender Teamwork. In Schaub, T., ed., *Proceedings of the 21st European Conference on Artificial Intelligence (ECAI'14)*, 819–824. Prague, Czech Republic: IOS Press.

Simeonovski, M.; Pellegrino, G.; Rossow, C.; and Backes, M. 2017. Who controls the internet? analyzing global threats using property graph traversals. In *Proceedings of the 26th International Conference on World Wide Web*, 647–656.

Speck, D.; Geißer, F.; and Mattmüller, R. 2020. When Perfect Is Not Good Enough: On the Search Behaviour of Symbolic Heuristic Search. In *Proceedings of the 30th International Conference on Automated Planning and Scheduling (ICAPS'20)*, 263–271. AAAI Press.

Speicher, P.; Steinmetz, M.; Backes, M.; Hoffmann, J.; and Künnemann, R. 2018a. Stackelberg Planning: Towards Effective Leader-Follower State Space Search. In McIlraith, S.; and Weinberger, K., eds., *Proceedings of the 32nd AAAI Conference on Artificial Intelligence (AAAI'18)*, 6286–6293. AAAI Press.

Speicher, P.; Steinmetz, M.; Künnemann, R.; Simeonovski, M.; Pellegrino, G.; Hoffmann, J.; and Backes, M. 2018b. Formally Reasoning about the Cost and Efficacy of Securing the Email Infrastructure. In *Proceedings of the 2018 IEEE European Symposium on Security and Privacy (EuroS&P'18)*, 77–91.

Stern, R.; Felner, A.; van den Berg, J.; Puzis, R.; Shah, R.; and Goldberg, K. 2014. Potential-based bounded-cost search and Anytime Non-Parametric A*. *Artificial Intelligence* 214: 1–25. ISSN 0004-3702.

Tambe, M. 2011. *Security and Game Theory: Algorithms, Deployed Systems, Lessons Learned*. Cambridge University Press.

Thayer, J. T.; Stern, R.; Felner, A.; and Ruml, W. 2012. Faster Bounded-Cost Search Using Inadmissible Estimates. In Bonet, B.; McCluskey, L.; Silva, J. R.; and Williams, B., eds., *Proceedings of the 22nd International Conference* *on Automated Planning and Scheduling (ICAPS'12)*. AAAI Press.

Tizio, G. D. 2018. *Pareto-Optimal Defensive Strategies Against JavaScript Injections*. Master's thesis, University of Trento, Italy.

Torralba, Á.; Alcázar, V.; Kissmann, P.; and Edelkamp, S. 2017. Efficient symbolic search for cost-optimal planning. *Artificial Intelligence* 242: 52–79.

Torralba, Á.; López, C. L.; and Borrajo, D. 2018. Symbolic perimeter abstraction heuristics for cost-optimal planning. *Artificial Intelligence* 259: 1–31.

Torralba, Á.; Speicher, P.; Künneman; Steinmetz, M.; and Hoffmann, J. 2021. Code, Benchmarks, and Data of Faster Stackelberg Planning via Symbolic Search and Information Sharing. https://doi.org/10.5281/zenodo.4320574.

Tsiakis, T.; and Stephanides, G. 2005. The economic approach of information security. *Computers & security* 24(2): 105–108.