© 2011 by Chih-Wei Hsu. All rights reserved.

SOLVING AUTOMATED PLANNING PROBLEMS WITH PARALLEL DECOMPOSITION

BY

CHIH-WEI HSU

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor Benjamin W. Wah, Chair and Director of Research
Professor Gerald DeJong
Professor Steven LaValle
Professor Martin Wong

Abstract

In this dissertation, we present a parallel decomposition method to address the complexity of solving automated planning problems. We have found many planning problems have good locality which means their actions can be clustered in such a way that nearby actions in the solution plan are usually also from the same cluster. We have also observed that the problem structure is regular and has lots of repetitions. The repetitions come from symmetric objects in the planning problem and a simplified instance with similar problem structure can be generated by reducing the number of symmetric objects.

We improve heuristic search in planning by utilizing locality and symmetry and applying parallel decomposition. Our parallel decomposition approach exploits these structural properties in a domain-independent way in three steps: action partitioning, constraint resolution, and subproblem solutions. In each step, we propose solutions to exploit localities and symmetries for minimizing solution time. Our key contribution lies in the design of simplification and generalization procedures to find good heuristics in action partitioning and constraint resolution.

In application of our method to solve propositional and temporal planning problems in three of the past International Planning Competitions, our results show that SGPlan_6, our proposed planner, can solve more instances than other top planners. We demonstrate SGPlan_6 performs well when action partitioning is useful in decreasing heuristic value. We also show SGPlan_6 can achieve better quality-time trade-off. By using the symmetry and locality, we are able to achieve good coverage using our domain-independent planner but still have good performance like domain-specific planners.

To My Family.

Acknowledgements

I would like to thank my adviser, Prof. Benjamin W. Wah, for his support and guidance during my graduate study at University of Illinois at Urbana-Champaign. This dissertation would not have been possible without his continuous encouragement and involvement.

I also would like to thank Professor Gerald DeJong, Professor Steven LaValle, and Professor Martin Wong for their precious time for serving on my dissertation committee. I also appreciate their fruitful comments and valuable suggestions.

This research was supported by National Science Foundation Grant IIS 03-12084 and by other financial support opportunities in the University of Illinois.

I would like to express my gratitude to many other people including my officemates and long-time friends in the University of Illinois including Batu, Bin, Chun-Cheng, Hong, Jing, Jing, Li-Lun, Lu-Chuan, Ming-Wei, Shan, Shui-Lung, Shuo-Heng, Soomin, Wei-Wen, and Yi-Ting for giving me enjoyable life and experience.

Finally, I want to thank my parents, my sister, and my wife for their unconditional love. This dissertation is dedicated to them.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem Formulation

Automated planning involves generating a sequence of actions to achieve a goal from an initial world description. The scheduled actions referred to as a plan are subject to constraints that can be propositional, numerical, or temporal. One can also define an optimization criterion such as the resource usage.

Automated planning is critical to robotics, space systems, manufacturing, and entertainment. Many real-world applications can be formulated as planning problems, including logistics control, transportation, model checking, game playing, and metabolic pathway. Also, solving planning problems would help the advance of areas like operations research and combinatorial optimization.

However, large planning problems are difficult to solve in general, even when finding a suboptimal plan. The computational complexity for propositional STRIPS [31] planning is PSPACE-complete [13]. In practice, existing planners employing a brute-force search will have difficulties in solving large planning instances whose search space grows exponentially with respect to problem size.

People have observed that domain-specific planners can perform better by utilizing problem structure. However, we cannot develop specific algorithm for each application as there are too many planning applications. Instead, we prototype each application in some formulation and develop one planner based on this.

Definition 1 *A planning problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ is a quadruple, where $\mathcal{V}$ is a set of state variables for a world description, $\mathcal{A}$ is the set of possible actions in $\mathcal{T}$, $\mathcal{I}$ is its initial state, and $\mathcal{G}$ is its problem goal.*

State variables can be logical, multi-valued, or numerical. Each action has some conditions on a subset of state variables in order to apply this action, as well as effects that change some state variables. The initial state is the original configuration of all the state variables, whereas the goal is the requirement to be reached on a subset of the state variables at the end of the plan.

Consider the Storage domain in IPC-5 [39] for unloading crates from containers and dropping them into depots by hoists. The space inside containers and depots is divided into areas, whereas each hoist can lift one crate at a time and move it from one area to its adjacent area and go in/out of a depot/container. Depots and containers are connected by loading areas, and movement in a loading area is unrestricted. Further, once an area is occupied by a crate or a hoist, it cannot be used by another activity until it is cleared. The optimization criterion can be the total time to deposit all crates to their desired locations.

Figure 1.1 illustrates a Storage planning problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ for the planning of three crates and three hoists. Here, $\mathcal{V}$ is the set of variables representing the state of crates, hoists, and areas. In $\mathcal{I}$, all hoists are in loading areas labeled "LA," and crates are stored in containers on the left. Actions $\mathcal{A}$ are specified for each hoist to move or to lift/drop crates in order to reach $\mathcal{G}$, where all crates are in depots on the right. Note that hoists can move almost independently as long as they do not go in the same area.

By executing a sequence of applicable actions, we can generate the state trajectory from $\mathcal{I}$. A valid plan is a sequence of applicable actions (state trajectory) to reach $\mathcal{G}$. Based on the trajectory, the plan metric can be the number of states visited. Since planning problems are solved by scheduling actions in the plan, the actions act like *variables*. Requirements (such as preconditions and mutual exclusions) to restrict the scheduling of actions are *constraints*. We define the constraints of a planning problem $\mathcal{T}$ as follows.

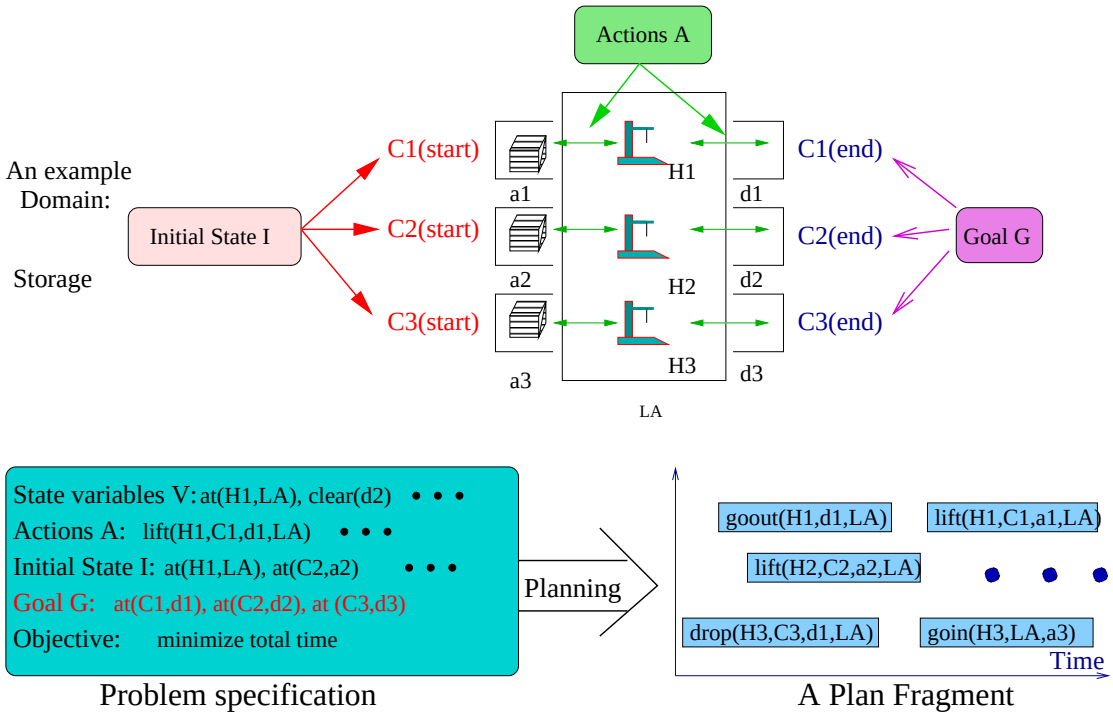


Figure 1.1: The topology of a Storage planning instance with 3 hoists and 3 crates. A sketch of the problem specification and a solution plan are shown in the lower panel.

Definition 2 *A planning problem entails the search of a sequence of actions that will achieve* $\mathcal{G}$*. The solution plan must satisfy the following* constraints*:*

1. *Its state trajectory starts from* $\mathcal{I}$ *and ends at* $\mathcal{G}$*.*

2. *Each action in the plan must be consistent with the state trajectory; that is, all (pre)-conditions must be supported and all effects must be fulfilled.*

3. *Simultaneous actions that cause* mutual exclusions *[6] cannot exist.*

In other words, an action in the solution plan is necessary because it supports either $\mathcal{G}$ or conditions of actions executed later, without introducing any mutual exclusions.

Based on Definition 2, we introduce an *action graph* to model the relations among actions. Constraints due to $\mathcal{I}$ and $\mathcal{G}$ are not included, as they are not related to actions but to state variables; that is, they are only used to constrain the starting and the termination points of a search trajectory.

Definition 3 *Given $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$, its* action graph *is a digraph (N, E), where nodes N are actions $\mathcal{A}$, and persistent (level independent) actions propagate the values of $\mathcal{V}$. There is an edge (n, n') in E iff n and n' are mutually exclusive, or the conditions of n' depend on n.*

Since the constraints of planning problems are too complicated to be handled at once, planning methods are derived from some properties or abstract representation of constraint definitions. In Section 1.2, we introduce properties of action graphs we exploited in this dissertation.

1.2 Locality and Symmetry

Figure 1.2 illustrates the action graphs of three Storage planning instances to plan for three crates. Here, a box represents an action, and a line between two actions denotes a constraint between them. When there are more than one hoists, the constraints can be partitioned by hoists, where between-cluster constraints prohibit the access of a hoist to the same area or the same crate, and within-cluster constraints ensure that each hoist can only move to a neighboring area and lift only one crate at a time.

There are two observations on this figure. Firstly, the locality of constraints is strong because the activities of each hoist rarely interact with each other, except when they access the same area or the same crate. Informally, *locality* in an action graph means that the actions involved in the problem structure, or specifically in the solution plan, consist of tightly coupled clusters. Hence, for a cluster of actions with strong locality, if action a from that cluster is used somewhere in the solution plan, it is likely nearby actions of a in the plan are also from the same cluster. Locality is derived from the problem structure, and a problem with good locality implies it is well structured, although the converse is not necessarily true.

Secondly, the problem structure is regular and has lots of repetitions from symmetric objects. In this domain where objects include hoists, crates, and areas, all hoists are symmetric

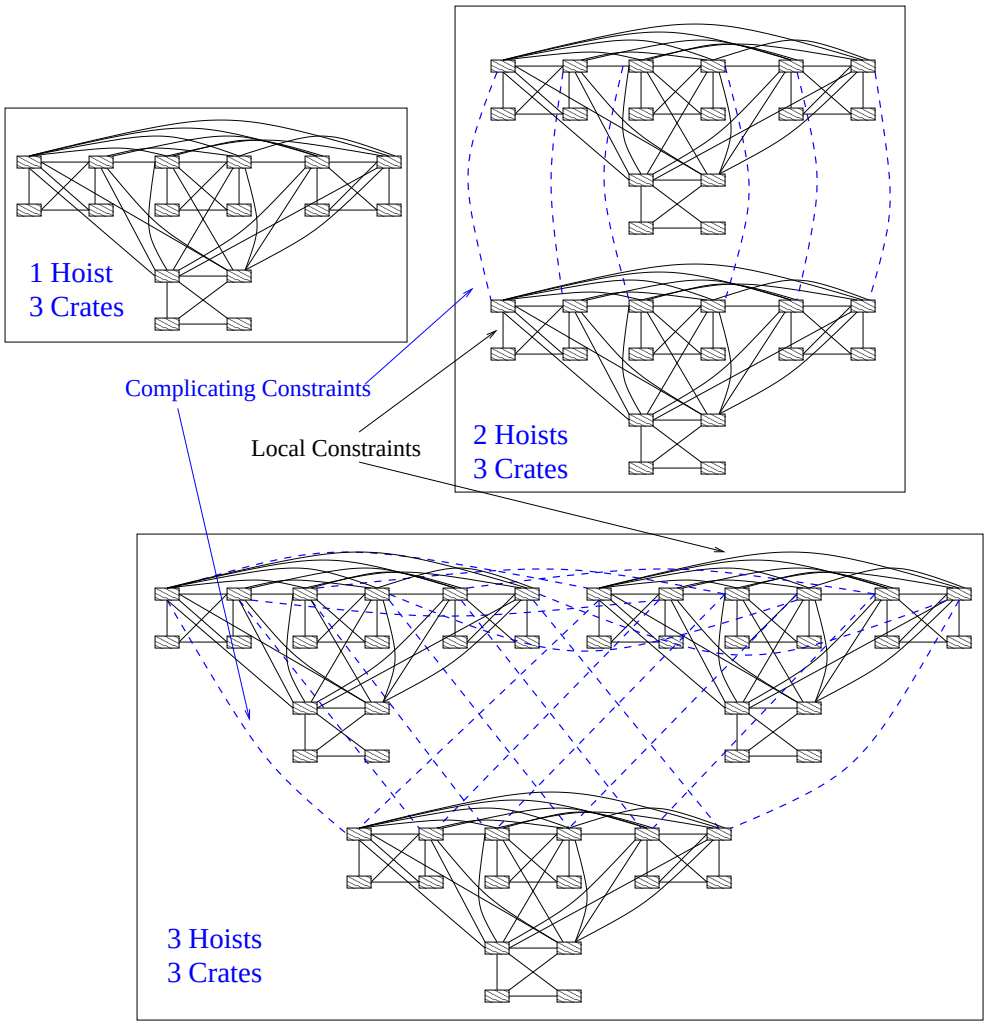


Figure 1.2: An action graph on the constraints in three Storage planning instances.

because any two can be interchanged without changing the problem semantics. Crates are also symmetric in the action graph, as we do not consider constraints due to the initial and goal states. That means any pair of crates can be interchanged in the action graph without changing the structure, although each crate may have its own source and destination. In contrast, areas are not symmetric because a loading area is functionally different from a depot area. Since each depot area can contain exactly one crate or hoist whereas a loading area has no capacity limit, interchanging them will result in a different topology and structure. In the following we specify functional and almost symmetries used in this paper.

Definition 4 (Functional symmetry [34]) *Two objects are functionally identical (symmetric) if substituting one for the other throughout* $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ *yields an identical problem.*

It has been observed that functional symmetries are uncommon in planning problems. They are also not useful for simplifying planning problems because their requirements are too strong. On the other hand, structures within a planning problem display *almost symmetries* [35] after eliminating chosen discrepancies in the initial state, actions, and goal between two objects.

Definition 5 (Almost symmetry [35]) *o and* o' *are almost symmetric in* $\mathcal{T}$ *if there exist an abstraction and the corresponding transformed problem* $\mathcal{T}'$ *in which o and* o' *are functionally symmetric.*

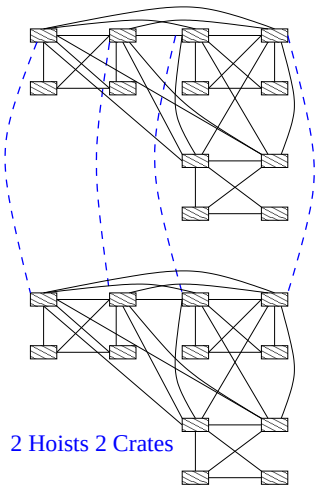


Figure 1.3: An action graph of a simplified Storage instance with 2 hoists and 2 crates (cf. the instance in Figure 1.2 with 2 hoists and 3 crates).

In the example in Figure 1.2, both hoists and crates form symmetry groups if we abstract the problem in such a way that all constraints not used in the action graph are eliminated. We can then exploit the resulting symmetry to generate a simplified problem with a similar action graph structure and other properties. Figure 1.3 illustrates the action graph of a simplified instance by choosing two hoists and two crates from the example in Figure 1.1.

It contains substructures that can be found in Figure 1.2 due to the existence of symmetric objects. In Section 2.3, we define and discuss the details of the abstraction used in our implementation.

1.3 Heuristic Search

Although there exist good localities and strong symmetries in planning problems, not every planner can exploit them to their advantage. In this dissertation, we concentrate on improving forward state-space heuristic search for planning with symmetry and locality as planners based on heuristic search have been proved successful in recent planning competitions [56]. In Section 3.1, we will also evaluate other alternatives and discuss the reasons of studying forward state-space heuristic search in this dissertation

The major component in forward state-space heuristic search is the heuristic function for estimating goal distance and guiding the search. The heuristic functions in state-space search are usually derived from relaxing or abstracting the constraints of planning problems. The performance of heuristic search depends on the quality of relaxation or abstraction for characterizing planning problems and the cost of computing heuristic function.

Definition 6 Heuristic function $h : S \rightarrow R^+$ *maps state* S *into a non-negative real number or* ∞. *In particular,* $h(S) = 0$ *if and only if* $S \vDash \mathcal{G}$. *In addition,* h *is completeness-preserving if* $h(S) = \infty \Rightarrow$ *there is no path from* S *to reach* $\mathcal{G}$.

In the rest of this dissertation, we assume all heuristic functions mentioned are completeness preserving. An important problem of heuristic function is that it may lose the necessary information for recognizing promising states during the relaxation or abstraction. Ideally, the heuristic function should identify the most promising state in the search frontier which leads to the decrease of heuristic function and the goal eventually. However, oftentimes we find no states in the search frontier can improve the heuristic value. It is also possible that

all states in the search frontier have identical heuristic value and heuristic searches works like uninformed search in this situation. Thus, number of explored states increases exponentially when it is difficult to decrease heuristic value. Here we define exit path where the heuristic function stops to provide guidance.

Definition 7 (Exit path and exit distance) *Given a solvable task $\mathcal{T}$ and a (completeness-preserving) heuristic function h, an exit path of a state S is a state trajectory from S to a state S' with $h(S') < h(S)$. In addition, the exit distance of S is the length of the shortest exit path.*

By identifying an exit path, we guide the search into a region where heuristic function continues to work. Note that Definition 7 is similar to the definition of exit or exit distance in local search topology studies [37, 54]. Because we assume $\mathcal{T}$ is solvable, for each search frontier, there must be a state with exit path as the path to goal must exist.

In order to improve heuristic search, we have to bring back necessary structural properties for finding exit paths. In particular, we show locality is beneficial to finding exit path meaning that oftentimes an exit path can be constructed by the actions from the same partition because each exit path is usually a small part of the entire solution plan. We observe that the heuristic function does suggest an action for decreasing heuristic value but search does not concentrate on the suggested direction because of uninformed heuristic values. In the following, we exemplify how exit paths can be constructed by complying localities with "ignore-delete-lists" heuristic which relaxes the planning problems by ignoring negative effects of actions.

Definition 8 *Given a STRIPS action a defined in a triple: $pre(a)$, add-effects $add(a)$, and delete-effects $del(a)$, $a^+ = (pre(a), add(a), \emptyset)$ is the "ignore-delete-lists" relaxation of a. Furthermore, the "ignore-delete-lists' heuristic for a planning problem $\mathcal{T}^+ = (\mathcal{V}, \mathcal{A}^+, \mathcal{S}, \mathcal{G})$ estimates $h(S)$ by using the length of a solution plan of the relaxed planning problem $\mathcal{T}^+ =$*

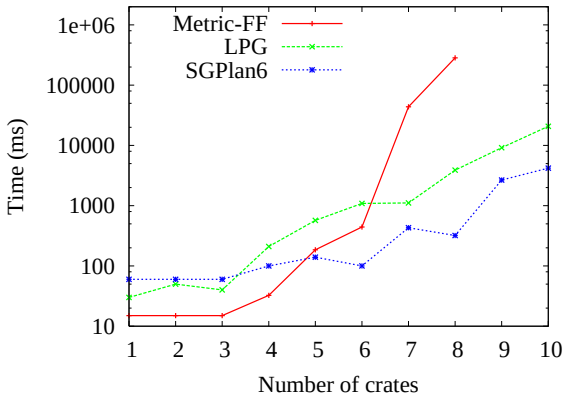


Figure 1.4: The exponential growth in complexity of three planners on the Storage domain. The x-axis shows the number of crates, and the y-axis, the solution time to find a feasible solution. All three planners were run on an AMD Athlon MP2800 PC with Linux AS4.

$(\mathcal{V}, \mathcal{A}^+, \mathcal{S}, \mathcal{G})$ *where* $\mathcal{A}^+$ *is constructed by apply "ignore-delete-lists" relaxation to each action in* $\mathcal{A}$*. Specifically,* $h^+(S)$ *estimates the goal distance by the length of optimal solution of* $\mathcal{T}^{+1}$.

We cannot always follow the (optimal) solution plan of $\mathcal{T}^+$ to decrease heuristic value as some delete-effects of earlier actions can block the execution of latter actions. However, one can find an exit path to decrease heuristic value by re-achieving the necessary delete-effects without invalidating other required facts. Actions in this exit path are tied by some constraints because there is some casual relationship and the goal is to achieve delete-effects of the first action of this exit path.

Figures 1.4 plots the solution times, with respect to different numbers of crates in the Storage domain, by FF [58] and LPG [41], two top planners in 2000/2002 International Planning Competitions (IPC). Without exploiting localities, the search complexity grows at a higher exponential rate, when compared to planners that utilizes localities, like our proposed SGPlan_6 in this paper.

[1]It is known that computing $h^+(S)$ is NP-complete but finding a feasible solution of $\mathcal{T}^+$ can be in polynomial time [58]. Although h^+ is not used in the implementation, it is used in the analysis.

1.4 Parallel Decomposition and Dissertation Outline

We have illustrated that exploiting localities can reduce the complexity of heuristic search. In this dissertation, we study the problem of utilizing localities and symmetry for improving heuristic search in planning. The goal is to achieve the performance of domain-specific planners by exploiting problem structure in a domain-independent way, we study how to use symmetry to generate simplified instances and to generalize search performance. Our use of symmetry is different from previous work where symmetry was used to guide the search directly [34, 35].

Decomposition is a general approach in problem solving that breaks a large problem into smaller subproblems and that solves each separately, either sequentially or in parallel. The decomposition of planning problems was first classified by Guestrin and Gordon [44] into serial and parallel decomposition. Freuder and Hubbe [38] named the same concepts for constraint satisfaction problems (CSPs) as disjunctive and conjunctive decomposition.

Serial decomposition partitions the state space of an optimization problem into subproblems in such a way that the combined state space is the union of the subproblem state spaces. Solving a serially decomposed problem involves finding one of the subproblems that solves the original problem. Many existing state-space search methods partition a problem by serial decomposition before solving the subproblems. Its disadvantage is that the complexity of each subproblem is very similar to that of the original problem.

Parallel decomposition [76], on the other hand, partitions the state space into subproblems in such a way that the combined state space is the cross product of the subproblem state spaces. Parallel decomposition leads to the partitioning of variables into (possibly overlapping) subsets. Those that are shared among the subsets are called *complicating variables*, and those in one subset only are *local variables*. As a result of the partitioning of the variables, the constraints are also partitioned: those involving variables in one subset are called *local constraints*, whereas those involving variables in more than one subset are *com-*

plicating constraints. Figure 1.2 illustrates local variables, local constraints and complicating constraints in the Storage problem.

Parallel decomposition is useful for solving problems with good localities. The original problem is solved by solving all the subproblems and by resolving the inconsistent complicating constraints and complicating variables. Its advantage is that the state space of each subproblem is exponentially smaller than that of the original problem; for instance, the number of actions and the length of a plan are much smaller if only one hoist is considered in Figure 1.2. As a result, parallel decomposition has the potential to significantly reduce the complexity of the original problem.

Based on the observation that action graphs of planning problems usually have good localities, we focus on studying parallel decomposition for solving large-scale planning problems in this paper. There are also some strategy parameters in our parallel decomposition framework. Based on symmetry identified, we use simplification and generalization to automatically adapt those parameters for each domain, rather than writing individual planner. Figure 1.5 shows the three inter-related steps when solving a problem partitioned by parallel decomposition.

The *partitioning* step partitions variables (and actions) into subsets and identifies the complicating variables and complicating constraints across the subsets. The resulting sub-

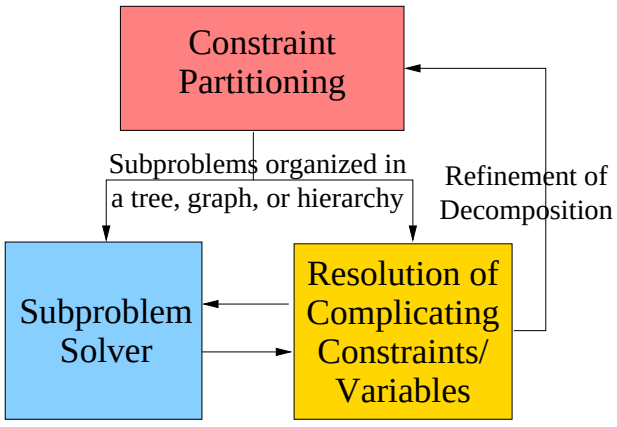


Figure 1.5: Solving a constrained planning problem by parallel decomposition.

problems may be organized in a two-level tree, a multi-level tree, or a graph. The main issue addressed in this paper is how to partition a planning problem in such a way that minimizes the time for solving the partitioned problem. We present in Chapter 2 our approach for discovering good partitioning attributes by solving simplified versions of the original problem and by generalizing these attributes to the original problem. Our approach is possible because the repetitive structure of an action graph allows it to be simplified into forms with similar partitioning behavior as the original problem. Our experimental results show that the best attribute found by solving simpler versions can be generalized to that of the original problem.

The *resolution* step resolves those inconsistent complicating variables and violated complicating constraints across subproblems. It may iterate a number of times and may require re-partitioning the original problem before a solution can be found. Based on an analysis of top-down/bottom-up and plan-space/state-space approaches, we present in Chapter 3 a top-down state-space framework that can be easily integrated with existing forward state-space planners and that can proactively guide constraint resolution with the aid of heuristic functions. By using heuristic functions to guide a best-first search, we show that good guidance heuristics can be generalized from those found by solving simpler versions of the original problem in a way similar to the generalization of partitioning attributes.

Lastly, the *subproblem solver* is used to solve the partitioned subproblems. It is very similar to that of solving the original problem, although some modifications may be needed for handling complicating variables and constraints. By exploiting localities in relaxed plan generations, the main issue addressed is the proper interleaving of the solver with the resolution step. By integrating a state-space search in our solver with a state-space resolution framework, we show in Chapter 3 that solution times can be substantially improved.

In Chapter 4, we evaluate our proposed method on benchmarks in three of the past International Planning Competitions. We show our planner can solve more instances than other top planners. By analyzing exit paths, we identify the condition for SGPlan_6 to work

well. Our planner can also achieve better time-quality tradeoffs than other top planners and solve more instances with more given time. We also show the results are not sensitive to the ordering of two generalization steps for finding partitioning attribute and heuristic function. Chapter 5 presents the conclusions and future work.

1.5 Contributions of This Research

The first contribution of this dissertation is the observation of locality and symmetry from planing problems and the utilization of them in improving forward state-space heuristic search. We demonstrate locality can facilitate finding exit paths in heuristic search. We also illustrate that planning problems can be simplified without changing characteristics by using symmetry.

The second contribution is the parallel decomposition framework for addressing the complexity of solving automated planning problems with locality and symmetry. The key methodology is to formulate and separate domain-dependent design options of parallel decomposition from domain-independent ones. Specifically, contributions in three inter-related steps of parallel decomposition are as follows:

In partitioning, we abstract the action partitioning problem into an attribute partitioning problem assuming that the best partitioning attribute can be identified later. This approach address the huge size of action graphs.

In resolution, we identify top-down state-space search is the most suitable resolution framework for parallel decomposition. We develop the search procedure for exploit locality identified in partitioning in order to reduce the number of nodes explored. In subproblem solver, we present an approach for reducing state evaluation cost by exploiting the locality again.

The third contribution is the design of simplification with symmetric objects and generalization procedures to find the best option for those domain-dependent components. We

apply this simplification and generalization framework to find the best partitioning attribute in action partitioning and the best heuristic function in resolution. It allows us to achieve good coverage using our domain-independent planner but still have good performance like domain-specific planners.

Chapter 2

Action Partitioning in Structured Planning Problems

In this chapter, we address two issues in partitioning action graphs of large-scale planning problems with good localities. First, given the large complexity of enumerating actions in large action graphs, we use a typing representation to classify planning objects into types (like "hoist") and group actions into several action schemata using types. In Section 2.2, we develop an approach based on *attribute partitioning*, where an attribute is a combination of types. Finding the best attribute is much easier than action partitioning because the number of attribute values is much smaller than the number of actions. Action partitioning is, therefore, transformed into *attribute identification*. Second, since it is impractical to find the best attribute by solving the original problem using different attributes, we present in Section 2.3 an approach that generates a simplified version of the original problem by sampling objects of the same type. We enumerate all possible attributes on the simplified problem in Section 2.4 and generalize the best attribute found to the original problem.

We evaluate in Section 2.5 our method on existing benchmarks and compare in Section 2.6 our approach with existing approaches. We show that our approach is more complete than SGPlan_4 [16] and is more suitable for search-based resolution methods than tree decomposition in factored planning methods.

2.1 A Comprehensive Example

To help understand the concepts and algorithms discussed, we present in this section a comprehensive example written in PDDL [33]. In PDDL, an *action schema* provides a

regular form for representing different actions associated with the same type of objects and with similar restrictions. Typing is useful for parameterizing actions and predicates, and the use of types as indexes leads to well structured planning problems with a large number of similar and related restrictions. Locality of actions is, however, implicit in such a representation and will have to be detected before it can be exploited.

Example 1 *In the domain definition of Storage-Propositional with a typed representation [21], there are ten types, including "hoist," "surface," and "crate." These types form a hierarchy. For example, "area" is a subtypes of "surface," which is a subtype of "object," the root of hierarchy. Types are used to parameterize predicates. For instance, predicate "on" with two parameters "?c" (of type "crate") and "?s" (of type "surface") is used to represent whether crate "?c" is on surface "?s."*

In this example, five action schemata are defined after the section on predicates, each with three parts: parameters, preconditions, and effects. Since each parameter of a defined type has multiple objects, it can be considered as an index when used in an action schema.

There exists a regular form for defining preconditions and effects of actions in an action schema. Instead of explicitly defining numerous actions with similar preconditions and effects, we can group them into an action schema and index them by objects. For example, action schema "move(?h ?from ?to)" represents a general class of actions for moving hoist "?h" from storearea "from" to storearea "to." To explicitly find a "move" action, one has to instantiate index values into the parameters of the action schema "move."

```
(define (domain Storage-Propositional)
(:requirements :typing)
(:types hoist surface place area - object
        container depot - place
        storearea transitarea - area
        area crate - surface)

(:predicates (clear ?s - storearea)
```

```
            (in ?x - (either storearea crate) ?p - place)
            (available ?h - hoist)
            (lifting ?h - hoist ?c - crate)
            (at ?h - hoist ?a - area)
            (on ?c - crate ?s - storearea)
            (connected ?a1 ?a2 - area)
            (compatible ?c1 ?c2 - crate))

(:action move
 :parameters (?h - hoist ?from ?to - storearea)
 :precondition (and (at ?h ?from) (clear ?to) (connected ?from ?to))
 :effect (and (not (at ?h ?from)) (at ?h ?to) (not (clear ?to)) (clear ?from)))

(:action lift
 :parameters (?h - hoist ?c - crate ?a1 - storearea ?a2 - area ?p - place)
 :precondition (and (connected ?a1 ?a2) (at ?h ?a2) (available ?h) (on ?c ?a1) (in ?a1 ?p))
 :effect (and (not (on ?c ?a1)) (clear ?a1) (not (available ?h))
   (lifting ?h ?c) (not (in ?c ?p))))

(:action drop
 :parameters (?h - hoist ?c - crate ?a1 - storearea ?a2 - area ?p - place)
 :precondition (and (connected ?a1 ?a2) (at ?h ?a2) (lifting ?h ?c) (clear ?a1) (in ?a1 ?p))
 :effect (and (not (lifting ?h ?c)) (available ?h) (not (clear ?a1)) (on ?c ?a1) (in ?c ?p)))

(:action go-out
 :parameters (?h - hoist ?from - storearea ?to - transitarea)
 :precondition (and (at ?h ?from) (connected ?from ?to))
 :effect (and (not (at ?h ?from)) (at ?h ?to) (clear ?from)))

(:action go-in
 :parameters (?h - hoist ?from - transitarea ?to - storearea)
 :precondition (and (at ?h ?from) (connected ?from ?to) (clear ?to))
 :effect (and (not (at ?h ?from)) (at ?h ?to) (not (clear ?to))))
)
```

A problem definition includes three parts: objects, initial state, and goal state; each can be varied to construct a new problem in the same domain. The following shows the problem definition of Storage-Propositional-P19*, with 27 "storearea" objects starting from* **depot0-1-1** *to* **container-2**, *9 "crate" objects starting from* **crate0** *to* **crate8**, *3 "hoist" objects, 3*

"container" objects, and 3 "depot" objects.

```
(define (problem storage-19)
(:domain Storage-Propositional)
(:objects
 depot0-1-1 depot0-1-2 depot0-1-3 depot0-2-1 ... container-2-9 - storearea
 hoist0 hoist1 hoist2 - hoist
 crate0 crate1 crate2 ... crate8 - crate
 container0 container1 container2 - container
 depot0 depot1 depot2 - depot
 loadarea transit0 - transitarea)

(:init
    (connected depot0-1-1 depot0-2-1)
    ...
    (available hoist2))

(:goal (and
    (in crate0 depot0)
    ...
    (in crate8 depot2)))
)
```

In conjunction with a problem definition, actions can be generated by enumerating all possible instantiations of parameters. *Instantiations* bind parameters with planning objects for each action schema. They are necessary because action schemata and typed representations are just convenient ways for representing many actions in a regular form and have nothing to do with a real planning process. For instance, given 9 objects of type "crate" and 27 objects of type "storearea" in Example 1, there can be up to $9 \times 27 \times 27 = 6,561$ actions associated with the action schema "move." The following example shows the action of moving **hoist0** from **depot0-1-1** to **depot0-1-2**.

Example 2 *The action of moving* **hoist0** *from* **depot0-1-1** *to* **depot0-1-2***:*

```
(:action move_hoist0_depot0-1-1_depot0-1-2
 :parameters ()
 :precondition (and (at hoist0 depot0-1-1) (clear depot0-1-2)
```

```
                (connected depot0-1-1 depot0-1-2))
:effect (and (not (at hoist0 depot0-1-1)) (at hoist0 depot0-1-2)
                (not (clear depot0-1-2)) (clear depot0-1-1)))
```

Given the highly regular structure expressed by a small set of action schemata for representing a large set of actions, we can abstract an action graph into a smaller graph called an attribute graph (discussed later). This is possible because actions involving the same object usually have closer relationship than those without shared objects. For example, all actions for moving **hoist0** in Figure 1.2 are tightly coupled and can be grouped together in partitioning.

Next, we need to to find a suitable method for abstracting an action graph in order to associate an action with several objects. For instance, action **move(hoist0, depot0-1-1, depot0-1-2)** can be associated with any of the three objects: **hoist0**, **depot0-1-1**, and **depot0-1-2**. Hence, one can group actions by hoists, meaning that actions corresponding to the same hoist are collapsed to one node, or use "area" objects like **depot0-1-1**, or even use a combination like **hoist0, depot0-1-1, depot0-1-2**.

Informally, *attribute partitioning* partitions the actions of an action graph by first choosing an attribute to reduce the graph to an attribute graph (discussed later) and by partitioning objects of the chosen attribute. Here, an *attribute* is a combination of types, where nodes in the attribute graph are all object instantiations for that combination of types. The resulting attribute graph is highly localized with respect to actions of the original action graph.

To find the best attribute for partitioning actions in such a way that the resulting partitioned planning problem can be solved in the shortest time, we propose to generalize the best attribute found when solving a simplified problem with the same domain definition and action schemata. Generalization is possible because objects of the same type generally induce repetitions in the action graph but do not change the nature of the problem. For example, the locality in Figure 1.2 is associated with hoists, although the action graph with

three hoists has one more cluster than that with two hoists. The functionality of all hoists is identical with repetitions of hoists of the same behavior.

Example 3 *This example illustrates the generalization of the attribute found by solving a simplified version of Storage-Propositional-P19 when the number of hoists is reduced from 3 to 2 and the number of crates from 10 to 6.*

```
(define (problem storage-19-simp)
(:domain Storage-Propositional)
(:objects
 depot0-1-1 ... depot0-1-3 depot0-2-1 ... container-2-0 - storearea
 hoist1 hoist2 - hoist
 crate4 crate5 ... crate8 - crate
 container0 container1 container2 - container
 depot0 depot1 depot2 - depot
 loadarea transit0 - transitarea)
(:init
        (connected depot0-1-1 depot0-2-1)
        ...
        (available hoist2))
(:goal (and
        (in crate4 depot1)
        ...
        (in crate8 depot2)))
)
```

Since the simplified instance of is easier to solve, we can find the best partitioning attribute by solving it using all possible combinations of attributes. It took 3.5 sec to solve the simplified instance using "(crate, hoist)" as the attribute, 5.6 sec using "crate," and 7 sec using "hoist." The total time of enumerating all 8 attributes starting from "hoist" is less than 56 seconds. The best attribute found by solving the simplified instance is "(crate, hoist)."

Using "(crate, hoist)" as the attribute, we can now solve the original Storage-Propositional-P19 in 3.5 sec. The solution time is worse when using any other attribute (such as 291.4 sec when using "crate" as the attribute). The solution time of the entire process is no more than 73.5 sec.

We have observed that many planning problems have highly regular and repeated structures, where actions can be represented by a smaller number of action schemata that are instantiated when planning is performed. (A non-typed representation is only useful when the planner cannot support a typed representation.) High regularity allows the generation of simplified instances with properties similar to those of the original problem.

Algorithm 1 summarizes the pseudo code of the action partitioning procedure presented in the rest of this section. It first calls Algorithm 3 in Section 2.3 to generate a series of simplified instances that mimic the structure of the original instance. Algorithm 4 in Section 2.4 then identifies the best attribute that consistently minimizes the solution time of the simplified instances. Finally, Algorithm 2 in Section 2.2 uses the best attribute found to partition the actions of the original instance. We discuss in the next section why action partitioning can be transformed into attribute partitioning.

2.2 Action Partitioning via Attribute Partitioning

In this section, we discuss attribute partitioning, the first step of exploiting the locality of actions in action graphs. Based on previous studies including problem reformulation [26], macro-operator generation [9], and inference of state invariants [36], we assume a typed representation in our problem specification. We define an attribute as a combination of types, and the values of an attribute as all possible object instantiations for that combination of types. Instead of partitioning an action graph directly, we first map its actions into a much smaller set of attribute values. We then define the projection operator for reducing all actions

Input: Problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$
Output: Actions $\mathcal{A}$ partitioned into K disjoint subsets: $\mathcal{A}_1$, ..., $\mathcal{A}_K$
1 $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$:= generate_simplified_instances($\mathcal{T}$) (Algorithm 3);
2 $\mathbb{O}$:= identify_best_attribute($\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$) (Algorithm 4);
3 $\mathcal{A}_1, \ldots, \mathcal{A}_K$:= partition_by_attribute($\mathcal{A}$, $\mathbb{O}$) (Algorithm 2);
Algorithm 1: *partition_actions*: finding action partitions in $\mathcal{A}$

involving the same attribute value into a node in an attribute graph. Finally, we carry out attribute partitioning on the much reduced attribute graph.

Definition 9 *An attribute $\mathbb{O}$ is a combination of types from the problem definition, where type t is associated with a set of objects O_t. The* set of attribute values *$D(\mathbb{O}) = \underset{t \in \mathbb{O}}{\times} O_t$ is the Cartesian product of O_t over all t. In particular, if $\mathbb{O}$ is an attribute of a single type, then $D(\mathbb{O}) = O_t$ for that type.*

Example 4 *For Storage-Propositional-P19 in Example 1, type t ="hoist" is an attribute, and O_t=*{**hoist0**, **hoist1**, **hoist2**} *is its set of objects. An attribute of two types $\mathbb{O}$ =("hoist", "crate") has the following set of attribute values $D(\mathbb{O})$ = {(***hoist0**, **crate0**), (**hoist0**, **crate1**), ..., (**hoist2**, **crate8**)}.*

Definition 10 *A projection $f_{\mathbb{O}} : \mathcal{A} \rightarrow D(\mathbb{O})$ is a function that maps actions $\mathcal{A}$ to $D(\mathbb{O})$. Here, $f_{\mathbb{O}}(a_1) = f_{\mathbb{O}}(a_2)$ iff a_1 and a_2 have the same object instantiations over $\mathbb{O}$. If an action is not instantiated from values in $D(\mathbb{O})$, then all values of $D(\mathbb{O})$ are equally possible in its projection.*

Example 5 *Given attribute $\mathbb{O}$ =("hoist", 'crate"), $f_{\mathbb{O}}$ maps actions* **lift(hoist0, crate0, depot0-1-1, depot0-1-2)** *and* **drop(hoist0, crate0, depot2-1-1, loadarea)** *to the same attribute value (***hoist0**, **crate0**).*

Definition 11 *(Attribute partitioning) Given problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$ with attribute $\mathbb{O}$ and projection $f_{\mathbb{O}}$, and the partitioning of attribute values $D(\mathbb{O})$ into K disjoint subsets: $D(\mathbb{O}) = D_1(\mathbb{O}) \bigcup \ldots \bigcup D_K(\mathbb{O})$, $\mathcal{A}$ can be partitioned into K partitions $\mathcal{A}_1$, ..., $\mathcal{A}_K$ in such a way that $\mathcal{A}_i = \{a | f_{\mathbb{O}}(a) \in D_i(\mathbb{O})\}$.*

Example 6 *Let "hoist" be the partitioning attribute of Storage-Propositional-P19. Assuming* **{hoist0, hoist1, hoist2}** *is partitioned into 3 disjoint subsets* {**hoist0**}, {**hoist1**}, *and* {**hoist2**}, *then actions of this problem can be partitioned into 3 disjoint subsets in such a way two actions are in the same subset if and only if they involve the same hoist.*

Algorithm 2 presents the procedure for partitioning an action graph $\mathcal{A}$ with attribute $\mathbb{O}$ into K disjoint subsets $\mathcal{A}_1, \ldots, \mathcal{A}_K$. It first sets K as the cardinality $D(\mathbb{O})$ and partitions $D(\mathbb{O})$ into K disjoint subsets (Line 1). Action a is then put into partition i based on a's object instantiation and $f_{\mathbb{O}}$, the projection $f_{\mathbb{O}}$ of attribute $\mathbb{O}$ (Lines 4 and 5).

Input: actions $\mathcal{A}$ and partitioning attribute $\mathbb{O}$
Output: K disjoint subsets: $\mathcal{A}_1, \ldots, \mathcal{A}_K$
1 $K := |D(\mathbb{O})|$; partition $D(\mathbb{O})$ into K disjoint subsets: $D_1(\mathbb{O}), \ldots, D_K(\mathbb{O})$;
2 $\forall i$, $A_i := \emptyset$;
3 **foreach** a *in* $\mathcal{A}$ **do**
4 | Find i such that $f_{\mathbb{O}} \in D_i(\mathbb{O})$;
5 | $A_i := A_i \cup a$;
6 **end**

Algorithm 2: *partition_by_attribute*: partitioning $\mathcal{A}$ by attribute $\mathbb{O}$

The identification of the best attribute used in Algorithm 2 entails finding one that that minimizes the solution time of the original planning problem $\mathcal{T}$.

Definition 12 *Given $\mathcal{T}$, attribute identification entails the search of attribute $\mathbb{O}$ for partitioning $D(\mathbb{O})$ that minimizes the total time of solving $\mathcal{T}$.*

In the following, we introduce the heuristic for selecting granularity (number of partitions). Note that an upper bound on the number of partitions is dictated by the partitioning attribute selected, since the number of values of an attribute corresponds to an upper bound on the number of partitions.

Table 2.1 summarizes the solution times of solving Storage-Propositional-P19 with respect to different granularities and six partitioning attributes. Because attributes "crate"

Table 2.1: Solution times of Storage-Propositional-P19 with respect to various number of partitions (on an AMD Athlon MP2800 PC running Linux AS4). Shaded boxes indicate those configurations that can be solved in less than 300 sec.

ATTRIBUTE	(number of partitions, time in seconds)						
AREA	(2, >300)	(4, >300)	(5, >300)	(6, >300)	(10, >300)	(15, >300)	(29, >300)
PLACE	(2, >300)	(3, >300)	-	-	-	-	-
CRATE	(2, >300)	(4, >300)	(9, 291.4)	-	-	-	-
HOIST	(2, >300)	(3, 76.7)	-	-	-	-	-
STOREAREA	(2, >300)	(4, >300)	(5, >300)	(7, >300)	(9, >300)	(14, >300)	(27, >300)
(CRATE,HOIST)	(2, >300)	(4, >300)	(5, >300)	(7, >300)	(9, >300)	(14, 37.5)	(27, 2.5)

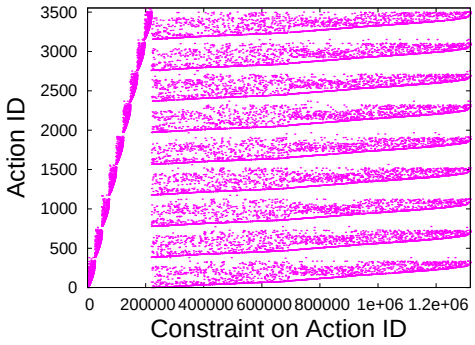
and "hoist" can solve the problem in less than 300 seconds, we also evaluate attribute ("crate", "hoist"). The results show that using the maximum granularity works well for attributes that lead to fast solution times, such as "crate", "hoist", and "(crate,hoist)." For attributes that cannot solve the problem in 300 seconds, it is difficult to know whether the maximum granularity is the best. In our results, the resolution algorithm (to be introduced in Chapter 3) uses the "ignore-delete-lists" heuristic function, and the subproblem solver is based on the best-first search in Metric-FF.

To understand why attribute identification followed by attribute partitioning is effective for action partitioning, we note that action partitioning cannot be directly solved by existing graph partitioners due to the large number of actions and their (pre-)conditions and effects. The number of actions can be more than 10,000, and the number of constraints can be quadratic to the number of actions. For instance, Storage-Propositional-P30 has 25,750 actions and 53,466,515 constraints. To reduce the complexity, we use the projection operator to reduce those actions involving the same attribute value into a node in an attribute graph and to partition the much smaller attribute graph.

Definition 13 *Given attribute $\mathbb{O}$, the attribute graph of problem $\mathcal{T}$ is a weighted graph with nodes $D(\mathbb{O})$. The weight of edge (o, o') is the number of edges (a, a') in the action graph of $\mathcal{T}$ where $o = f_{\mathbb{O}}(a)$ and $o' = f_{\mathbb{O}}(a')$.*

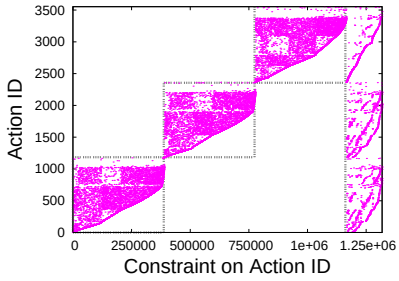
When the actions of a problem can be abstracted by types and action schemata, the corresponding action graph can be abstracted by a type-based attribute. For instance, Figure 1.2 can be abstracted by a three-node attribute graph, where each node represents a unique hoist. By using attribute partitioning, actions involving the same attribute value can be grouped together. In this case, the relation between two actions is represented by the relation between the corresponding attribute value. Figure 1.2 shows that actions associated with the same hoist are tightly coupled, and the relation between actions on different hoists is modeled by the relation between those two hoists.

	crt0	crt1	crt2	crt3	crt4	crt5	crt6	crt7	crt8
crt0	24207	14715	14806	14180	14855	13890	14029	14508	14380
crt1	15935	23803	15185	14588	15359	14454	14386	14704	14867
crt2	15931	15171	23921	14551	15348	14270	14416	14803	14874
crt3	16616	15780	15927	23942	15976	14963	15100	15524	15458
crt4	16150	15227	15332	14745	24142	14530	14603	15093	14990
crt5	16797	15833	16127	15460	16081	23857	15320	15724	15719
crt6	16909	16139	16188	15554	16261	15232	24085	15768	15833
crt7	16183	15564	15580	14903	15514	14603	14785	23848	15259
crt8	16596	15686	15790	15238	15890	14894	15006	15332	24129



a) Attribute and constraint graph based on type "crate"

	hoist0	hoist1	hoist2
hoist0	387111	26083	27075
hoist1	25996	386910	27062
hoist2	25322	25134	387228



b) Attribute and constraint graph based on type "hoist"

Figure 2.1: Partitioning the action graph of Storage-Propositional-P19 by two attributes. In each graph, the Y-axis enumerates the actions and the X-axis, the constraints between actions. There is a dot at (x, y) if constraint x involves action y.

For Storage-Propositional-P19, its action graph (with 3,552 nodes and 1,317,921 edges) can be transformed into a much smaller attribute graph by selecting an attribute and its projection operator. Figure 2.1 plots two attribute graphs in an adjacency-matrix representation using attributes "crate" and "hoist." For simplicity, we assume that each partition of actions corresponds to exactly one object, namely, a crate or a hoist. The abstraction based on "hoist" is much better, as weights on the diagonal dominate the off-diagonal entries. This means that actions associated with the same hoist have a much stronger relationship than those associated with different hoists. The right panel further illustrates the corresponding constraint graphs.

Although attribute partitioning is useful for partitioning regular action graphs, it may not be useful for problems with low regularity. For example, a problem may have constraints

Input: Original planning problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$;
Output: A series of simplified instances $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}$;
1 Initialize array R;
2 $\mathcal{T}_0$:= do_abstraction($\mathcal{T}$);
3 $\{O_1, O_2, \ldots, O_J\}$:= identify_symmetry_groups($\mathcal{T}_0$);
4 $i := 0$;
5 **foreach** r *in* R **do**
6 | $i := i + 1$;
7 | $\{O'_1, O'_2, \ldots, O'_J\} := g_r(\{O_1, O_2, \ldots, O_J\})$;
8 | $\mathcal{T}'$:= eliminate_problem_definitions($\mathcal{T}$, $O'_1 \cup O'_2 \cup \ldots \cup O'_J$);
9 | $\mathcal{T}_i$:= enhance_reachability($\mathcal{T}'$);
10 **end**

Algorithm 3: *generate_simplified_instances*: finding simplified instances

involving actions sampled from several distributions with small variances on the action indexes. Although it has good locality, its random structure cannot be exploited by identifying attributes.

2.3 Generating Simplified Instances

In this section, we present a method for generating simplified instances of a planning problem. By keeping the action schemata unchanged, Algorithm 3 separates the planning objects into groups of symmetric objects. It then samples each independently (in a stratified fashion) in order to produce simplified planning instances $\mathcal{T}_1, \ldots, \mathcal{T}$ of the original problem.

The procedure first initializes all sampling ratios in array R (Line 1) whose elements are used to control the size of the simplified instances. We use $R = [0.4, 0.6, 0.8, 1.0]$ in our implementation, where a larger value means a larger instance, and the last simplified instance ($R[4] = 1.0$) is identical to $\mathcal{T}$

Line 2 forms $\mathcal{T}_0$ by abstracting $\mathcal{T}$. Based on functional and almost symmetries defined in Section 1.2, it finds an abstraction that can exploit almost symmetries. Our approach is to retain those permanent relations between objects defined by static facts (whose truth value cannot be changed by any action) and to eliminate dynamic relations that would be

changed during planning.

Definition 14 *An almost symmetry is called* static *if the corresponding abstraction only keeps the static properties, including action schemata and static facts, in the problem definition. In other words, it ignores the initial values of state variables and goals.*

Example 7 *In Example 1, the location of each crate is not static because it can be changed during planning. Besides dynamic properties that include the initial locations and destinations, all crates are identical in terms of static facts. Thus, every pair of crates are statically symmetric but not functionally symmetric. In contrast, storeareas like* **depot0-1-1** *and* **depot0-1-2** *are neither statically nor functionally symmetric. Each has a unique role in the topology defined by static facts like (connected depot0-1-1 depot0-1-2).*

Static symmetry is important because they represent permanent relations between objects. For example, Macro-FF [9] generates statically connected components for macro-operators using static facts. Sampling statically symmetric facts utilize the redundancy in statically symmetric objects and learns the structure of the original problem by inheriting static facts. This abstraction is consistent with the use of action graphs discussed in Section 1.2. Because we do not directly use symmetry in searching the goal, we can use a loose version of symmetry to abstract problem structure.

Symmetry when further abstracted, such as treating objects of the same type to be symmetric, ignores structural information carried by static facts. Without these facts, additional static facts may be needed to repair the original structure. For instance, abstracting static facts in Example 7 may ignore connectivity information in the Storage domain and treat all storeareas to be symmetric. Sampling in this case may lead to a degenerate problem with disconnected storeareas. Further, repairs through additionally generated static facts may not lead to a simplified instance with a similar structure.

Line 3 examines each group of functionally symmetric objects in $\mathcal{T}_0$ in order to partition all objects into J symmetry groups $O_1, \ldots, O_J$. It starts by identifying all pairs of symmetric

objects. As the action schema is defined using types, symmetric objects must be of the same type. It then merges objects to a symmetry group if they are functionally symmetric, leading to multiple symmetry groups. However, objects of the same type are not necessarily placed in the same group. For instance, in Example 1, **container0** and **depot0** are different concepts, although they are both of the same type "place."

The above step is not exhaustive and can only identify a subset of the symmetries, since there may exist two object pairs (o_1, o_2) and (o'_1, o'_2) such that replacing (o_1, o_2) by (o'_1, o'_2) throughout yields an identical problem. Identifying full symmetries in planning problems is prohibitively expensive because it is equivalent to solving *graph automorphisms*, namely, finding a permutation of objects such that the transformed problem is isomorphic to the original one.

After finding statically symmetric objects, the loop between Lines 5 and 10 generates a simplified instance $\mathcal{T}_i$ corresponding to $R[i]$. Line 7 uses the *sampling operator* g_r to sample each group of functionally symmetric objects.

Definition 15 *Sampling operator g_r with sampling ratio $r \in (0, 1]$ is applied to objects O decomposed into sets of statically symmetric objects $O_1, O_2, \ldots, O_J$ It results in $O' = O'_1, O'_2, \ldots, O'_J$, where each object in O'_i is selected from O_i without replacement, and $|O'_i| = min(1, |O_i| \times r)$.*

Example 8 *By setting $r = 0.6$, we can generate Example 3 from Example 1. All crates are statically symmetric and so do hoists. We only sample objects of types "hoist" (from three to two) and "crate" (from nine to five). Areas are not statically symmetric because of their spatial relationships defined by static facts with predicate "connected" and "in." Places are also not statically symmetric because they may have different number of areas.*

After reducing all objects O to O' by g_r, Line 8 generates the simplified version $\mathcal{T}'$ of the original problem $\mathcal{T}$ by eliminating all state variables, actions, initial facts, and goals that

Table 2.2: Evaluation of partitioning attributes with respect to different simplification approaches. Each entry shows the solution time in seconds, and a shaded item for a given r requires the least solution time among the possible attributes sampled.

Attribute	$r = 0.4$	$r = 0.6$	$r = 0.8$	$r = 1.0$	$r = 0.4$	$r = 0.6$	$r = 0.8$	$r = 1.0$
Sampling on	Symmetry group from type "hoist"				Symmetry group from type "crate"			
AREA	-	>300	>300	>300	>300	>300	>300	>300
PLACE	-	>300	>300	>300	>300	>300	>300	>300
CRATE	-	59.31	59.31	291.4	157.7	>300	>300	291.4
HOIST	-	65.5	65.5	76.7	0.9	8.7	33.8	76.7
STOREAREA	-	>300	>300	>300	>300	>300	>300	>300
(CRATE,HOIST)	-	>300	>300	2.5	0.7	>300	231.5	2.5
Sampling on	Both from "hoist" and "crate"							
AREA	-	>300	>300	>300				
PLACE	-	>300	>300	>300				
CRATE	-	1.6	1.3	291.4				
HOIST	-	11.6	34.6	76.7				
STOREAREA	-	91.8	>300	>300				
(CRATE,HOIST)	-	0.4	1.0	2.5				
(CRATE,STOREAREA)	-	>300	>300	>300				
(HOIST,STOREAREA)	-	>300	>300	>300				

involve objects not in O', while keeping all state variables, actions, initial facts, and goals defined entirely on O'.

Although we have assumed in Definition 15 that all symmetry groups are used in sampling, it is possible to sample using a subset of the symmetry groups. For instance, there are two symmetry groups in Example 1, one formed by objects of type "crate" and the other by objects of type "hoist." Hence, there are three possibilities, using each of two groups and using both groups.

Table 2.2 enumerates the three alternatives in sampling symmetry groups on Storage-Propositional-P19 and shows that using all groups is the best. Each column shows the various solution times in seconds when enumerating under a given r. We skip r (= 0.4) that will reduce the number of values of one attribute to one or less, since it will lead to a degenerate attribute that cannot be evaluated. We evaluate an attribute with a combination of types only if all its elements are attributes that can solve the problem in 300 sec or less.

The result shows that sampling both symmetry groups is the best, as it provides consistent results on the best attribute with respect to solution time. Likewise, for a given

r, using both symmetry groups leads to less solution time. We also observe that the best attribute is "crate" when sampling objects in the symmetry group corresponding to type "hoist," whereas sampling objects in the group corresponding to type "crate" will not have consistent solution times across r. In short, sampling on hoists alone will worsen the locality associated with hoists, whereas sampling on crates alone will worsen the locality associated with crates. Sampling on both symmetry groups minimizes the chance of under-represented symmetry groups.

In general, we cannot guarantee that an instance is feasible after simplification, where infeasibility means that some combinations of state variables in O' are unreachable but those in $\mathcal{T}$ are feasible. For example, in Storage planning, one can set a goal such that each valid plan must use two distinct hoists. This means that a simplified instance with only one hoist is infeasible. To address this issue, in generating a simplified instance, we need to maintain the set of reachable facts entirely defined on O'. For example, if a hoist can freely move in every storearea and load every crate in $\mathcal{T}$, then this property should be preserved in $\mathcal{T}'$. For each state variable entirely defined on O', if we find an unreachable fact after simplification when generating the relaxed plan, we add this fact directly to the set of initial facts $\mathcal{I}'$. The step is illustrated as follows.

Example 9 *In the Storage domain in Section 1.2, (clear ?a) indicates that area "?a" is unoccupied. If a hoist occupying area A*1 *has been eliminated in the simplified instance, then there is no way to know that A*1 *is not occupied in the simplified instance. Our simplification procedure can remember the clearance of that area by including the corresponding fact in the initial state, making it possible for all movements to be blocked by this nonexistent hoist.*

Line 9 employs the "ignore-delete-lists" heuristic to detect those unreachable facts and to add a set of unreachable facts without over-constraining $\mathcal{T}'$. Specifically, if the difficulty of $\mathcal{T}'$ is measured by the length of a plan solving it, we maintain the set of reachable facts entirely defined on O' without making the plan of $\mathcal{T}'$ too short. The idea is to favor adding

shallow facts in the relaxed plan of $\mathcal{T}$, since they are more distant from satisfying $\mathcal{G}'$, and adding them will not significantly shorten the plan to $\mathcal{G}'$. Our approach is to do level-order traversing in the relaxed planning graph, starting from the level of the initial facts. Given i, if we determine that fact f is unreachable in $\mathcal{T}'$ but is reachable at level i in the original relaxed planning graph, we add f to $\mathcal{I}'$. We then re-evaluate the relaxed planning graph of $\mathcal{T}'$ and check all facts at level $i+1$.

2.4 Generalization of Partitioning Attributes

We present in this section the generalization of the best partitioning attribute found by solving simplified instances. We define generalizability as follows.

Definition 16 *(Generalization) Given problem $\mathcal{T}$ and its simplified instance $\mathcal{T}'$, a hypothesis h can be generalized from $\mathcal{T}'$ to $\mathcal{T}$ if $h(\mathcal{T}) = h(\mathcal{T}')$.*

Here hypothesis h is the partitioning attribute chosen for a problem, and the image of h is the set of all possible attributes. In generalizing a partitioning attribute of $\mathcal{T}'$ to that of $\mathcal{T}$, the image of h is consistent in $\mathcal{T}'$ and $\mathcal{T}$, both having the same set of attributes; that is, we can infer $h(\mathcal{T})$ from $h(\mathcal{T}')$. Informally, h maps a finite set of attributes to an attribute that minimizes the solution time of the planning problem.

Algorithm 4 presents the procedure for identifying the best partitioning attribute. Using a series of simplified instances $\mathcal{T}_1, \ldots, \mathcal{T}$ generated by Algorithm 3, it outputs the best attribute for Algorithm 2.

Line 1 extracts the possible attributes from the problem definition. Note that attributes are from types in the action schemata when using attributes to do action partitioning. In Example 1, types "surface," "object," "container," and "depot" are not useful for grouping actions because they are not present in any action schemata. Further, to make projections meaningful, we set a threshold such that a majority of the actions must be instantiated from

Input: A series of simplified instances: $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$;
Output: Best partitioning attribute o for $\mathcal{T}$;
1 $\{\mathbb{O}_1, \ldots, \mathbb{O}_J\}$:= extract_possible_attribute($\mathcal{T}$);
2 $i := 0$;
3 **repeat**
4 | $i := i + 1$;
5 | o_i := enumerate_attribute($\mathcal{T}_i$, $\{\mathbb{O}_1, \ldots, \mathbb{O}_J\}$);
6 **until** $\mathcal{T}_i = \mathcal{T}$ *or consistent_attribute*(o_1, o_2, …, o_i) ;

Algorithm 4: *identify_best_attribute*: finding best partitioning attribute

the values of each possible attribute. In Example 1, type "transitarea" and combinations based on it are not reasonable attributes, as only a small set of actions are relevant to their values. To cover more than half of the actions, attributes are selected from five types ("hoist," "crate," "place," "area," "storearea") and their combinations.

Line 5 evaluates the J attributes using the simplified instance $\mathcal{T}_i$. To find o_i, we record the best attribute found so far and terminate its evaluation if it is impossible to be the best. In our resolution algorithm, we use the "ignore-delete-lists" guidance heuristic in Section 3.4. We defer to Chapter 3 to discuss the other details of the resolution algorithm and the subproblem solver.

In finding the best attribute efficiently, ordering their evaluations is important, In our approach, we order the attributes by their action-locality values and evaluate them in descending order. When partitioned by attribute j,

$$\mathrm{AL}_j \;=\; 1 - \left(\frac{|CC_j|}{|C_j|} + \sum_i \frac{|C_{i,j}|}{|C_j|} \times \frac{|\mathcal{A}_{i,j}|}{|\mathcal{A}_j|} \right), \tag{2.1}$$

where $|C_{i,j}|$ is the number of local constraints in the i^{th} action partition $\mathcal{A}_{i,j}$, $|C_j| = \sum_i |C_{i,j}|$ is total number of local constraints, $|\mathcal{A}_j| = \sum_i |\mathcal{A}_{i,j}|$. $|CC_j|$ is the total number of complicating constraints, and $|C|$ is the number of all constraints. Here, $\mathrm{AL}_j \in [0, 1)$; $\mathrm{AL}_j = 0$ represents a non-partitioned problem.

Intuitively, AL measures the fraction of the area not occupied by the block-angular part in the right panels of Figure 2.1. Consider the case when actions are partitioned into 9 subsets

by "crate" (Figure 2.1a). Because most of the constraints are complicating constraints, AL_{crate} for this partitioning is very small. On the other hand, when actions in Figure 2.1b are partitioned into three subsets by "hoist," $\text{AL}_{\text{hoist}} = 0.588$. This is computed from the following statistics: $|C_{1,\text{hoist}}| = 387111$, $|\mathcal{A}_{1,\text{hoist}}| = 1185$, $|C_{2,\text{hoist}}| = 386910$, $|\mathcal{A}_{2,\text{hoist}}| = 1177$, $|C_{3,\text{hoist}}| = 387228$, $|\mathcal{A}_{3,\text{hoist}}| = 1190$, and $|CC_{\text{hoist}}| = 156672$. The action locality is good, as the upper bound on AL for any 3-way partitioning is 2/3. Note that the exact calculation of AL_j can be expensive when C_j is large. In our implementation, we sample C_j without generating the action graph.

Given that the simplified instances are easy to solve, one can find the best attribute by enumerating all possible attributes with respect to their solution times. Enumerations are possible because the number of types in most action schemata is small, usually less than ten.

Line 6 ensures that the best attribute identified can be generalized to larger instances. If we find attribute o that is consistently the best for the simplified instances evaluated, we return o. Otherwise, we try a larger simplified instance. In the worst case, we evaluate the original problem $\mathcal{T}$ using $r = 1$.

Our approach can be viewed as a simple realization of explanation-based learning [20]. Based on training observations and an expert-supplied domain theory (that needs not always be robust), explanation-based learning constructs some new examples by the domain theory that can likely be generalized from the original observations. In our approach, we use the symmetry-based domain theory to generate the simplified instances, where the observations or labels of these instances are likely to be the same as those of the original instance. When there is an inconsistency in the simplified instances, we can resolve it by either a statistical learner or in our case a simple rule in Line 6 of Algorithm 4. The simplified instances serve a purpose similar to that of *virtual examples* [12] or *phantom examples* [72], since all are used to generalize the observations into new examples by prior domain knowledge. Note that our simplified instances may be biased because they are all generated from the original instance. Hence, the attributes found may not be generalizable to new instances or new domains.

2.5 Evaluation of Action Partitioning

In this section, we evaluate our action partitioning approach.

We first illustrate the entire process by fully evaluating the IPC-5 TPP-Propositional domain [39]. Figure 2.2 demonstrates the generalizability of the best attribute found. In computing solution times, the resolution algorithm uses the "ignore-delete-lists" guidance heuristic introduced in Chapter 3, and the subproblem solver uses the best-first search with the "ignore-delete-lists" heuristic. For each instance, we generate four simplified instances using $r = [0.4, 0.6, 0.8, 1.0]$. For each r, we evaluate the eight possible partitioning attributes for this domain: "level," "place," "depot," "market," "truck," "goods," ("truck," "goods"), and "none," the last denoting the case without partitioning. (To avoid congestion, we show the results of five attributes.)

The first 15 instances in the domain are not interesting because they all can be solved in 1 sec or less using most of the attributes. For the remaining 15 instances, we first examine the best attribute found for the original problem ($r = 1.0$), which is the target of generalization. Attribute "truck" is the best, except for TPP-Propositional-P25 where "none" is the best and TPP-Propositional-P29 where ("truck", "goods") is the best. TPP-Propositional-P30 cannot be solved using any attribute.

Next, we check whether "truck" is in general the best attribute when $r = 0.4$. We observe that "truck" is the best for a majority of the instances. There are some where ('truck", 'goods") is the best, although their number decreases as r becomes larger. When $r = 0.8$, ('truck", 'goods") is still the best for TPP-Propositional-P22 and TPP-Propositional-P23. Despite this is not the best when $r = 1.0$, the original instances can still be solved using this attribute. For TPP-Propositional-P25, ('truck", 'goods") is the best attribute for $r = 0.4$ and 0.6. However, "none" becomes the best when $r = 0.8$ or larger. In solving TPP-Propositional-P29, ('truck", 'goods") is consistently the best for every r.

Next we empirically evaluate the best attributes across simplified instances of every IPC-4, IPC-5, and IPC-6 propositional domain. In each domain, we select at least two

Table 2.3: The generalization results on all IPC-4, IPC-5, and IPC-6 propositional domains with respect to the best partitioning attribute. Each entry is of the form (best attribute, solution time corresponding to the best attribute in sec). Attribute "NONE" means a best-first search without parallel decomposition. Highlighted rows represent those with difficulties in generalization across r.

Instance	$r = 0.4$	$r = 0.6$	$r = 0.8$	$r = 1.0$
		IPC-4		
Airport-P48	(AIRPLANE,2.2)	(AIRPLANE,3.0)	(AIRPLANE,5.5)	(AIRPLANE,7.3)
Airport-P49	(AIRPLANE,2.6)	(AIRPLANE,4.1)	(AIRPLANE,10.9)	(AIRPLANE,15.5)
Pipesworld-P40	(PRODUCT,1.3)	(PRODUCT,0.6)	Unsolved	(PRODUCT,3.1)
Pipesworld-P41	(AREA,0.1)	(AREA,0.1)	(AREA,0.1)	(AREA,0.3)
PromelaOpt-P04	(NONE,0.5)	Unsolved	Unsolved	Unsolved
PromelaOptDP-P13	(NONE,0.5)	(NONE,1.6)	(NONE,6.2)	(NONE,17.1)
PromelaPhi-P14	(NONE,0.1)	(NONE,1.5)	(NONE,319.0)	Unsolved
PromelaPhiDP-P48	(NONE,0.6)	(NONE,2.3)	(NONE,13.5)	(NONE,48.5)
PSRsmall-P49	(NONE,0.4)	(NONE,0.4)	(NONE,0.4)	(NONE,0.4)
PSRsmall-P50	(NONE,0.1)	(NONE,0.1)	(NONE,0.1)	(NONE,0.1)
PSRmiddle-P49	(NONE,18.7)	(NONE,13.1)	(NONE,3.1)	(NONE,3.8)
PSRmiddle-P50	(NONE,3.2)	(NONE,4.1)	(NONE,3.3)	(NONE,4.0)
PSRlarge-P26	(NONE,50.4)	(NONE,82.5)	(NONE,96.3)	(NONE,157.0)
PSRlarge-P27	(NONE,48.0)	(NONE,62.5)	(NONE,133.1)	(NONE,211.0)
Satellite-P27	(DIRECTION,42.4)	(SATELLITE,105.0)	(SATELLITE,151.1)	(DIRECTION,118.6)
Satellite-P28	(SATELLITE,35.1)	(SATELLITE,61.1)	(SATELLITE,188.0)	(SATELLITE,113.2)
		IPC-5		
Openstacks-P29	(PRODUCT,32.4)	(PRODUCT,33.6)	(PRODUCT,45.0)	(PRODUCT,35.0)
Openstacks-P30	(PRODUCT,7.2)	(PRODUCT,7.2)	(PRODUCT,7.1)	(PRODUCT,7.2)
Pathways-P29	(LEVEL,1.5)	(LEVEL,1.5)	(LEVEL,1.5)	(LEVEL,1.5)
Pathways-P30	(LEVEL,1.7)	(LEVEL,1.6)	(LEVEL,1.7)	(LEVEL,1.7)
Pipesworld-P31	(AREA,19.0)	(AREA,7.2)	(AREA,7.8)	(AREA,5.1)
Pipesworld-P41	(AREA,0.1)	(AREA,0.1)	(AREA,0.1)	(AREA,0.4)
Rovers-P30	(WAYPOINT,1.6)	(WAYPOINT,25.1)	(WAYPOINT,25.8)	(WAYPOINT,200.3)
Rovers-P32	(WAYPOINT,2.9)	(WAYPOINT,12.9)	(WAYPOINT,202.6)	(WAYPOINT,37.5)
Storage-P20	(CRATE,40.1)	(PLACE,25.3)	(CRATE,17.3)	(CRATE,36.3)
Storage-P22	(CRATE,6.1)	(PLACE,56.7)	((PLACE,CRATE),0.6)	(STOREAREA,29.8)
TPP-P28	(TRUCK,21.4)	(TRUCK,24.3)	(TRUCK,43.6)	(TRUCK,56.0)
TPP-P29	((TRUCK,GOODS),3.6)	((TRUCK,GOODS),13.1)	((TRUCK,GOODS),61.2)	((TRUCK,GOODS),79.3)
Trucks-P09	(LOCATION,0.1)	(LOCATION,0.1)	(LOCATION,0.2)	(LOCATION,48.6)
Trucks-P14	(LOCATION,0.2)	(LOCATION,0.3)	(LOCATION,0.3)	(LOCATION,281.0)
		IPC-6		
Elevator-P29	(PASSENGER,1.8)	(COUNT,4.1)	(COUNT,12.8)	(PASSENGER,23.2)
Elevator-P30	(PASSENGER,2.0)	(PASSENGER,3.7)	(COUNT,7.9)	(COUNT,14.3)
Openstacks-P29	(ORDER,2.7)	(ORDER,2.0)	(ORDER,2.8)	(ORDER,2.1)
Openstacks-P30	(ORDER,1.5)	(ORDER,1.5)	(ORDER,1.8)	(ORDER,1.7)
Parcprinter-P29	(SHEET_T,0.1)	(SHEET_T,0.1)	(SHEET_T,0.1)	(SHEET_T,0.1)
Parcprinter-P30	(SHEET_T,0.2)	(SHEET_T,0.1)	(SHEET_T,0.1)	(SHEET_T,0.1)
Pegsol-P29	(NONE,0.1)	(LOCATION,0.1)	(NONE,0.1)	(LOCATION,0.2)
Pegsol-P30	(NONE,0.1)	(NONE,0.1)	(NONE,11.8)	(NONE,15.0)
Scanalyzer-P29	(NONE,0.1)	(NONE,0.1)	(CAR,224.1)	(NONE,197.2)
Scanalyzer-P30	(CAR,0.1)	(CAR,0.1)	(CAR,106.3)	(CAR,88.9)
Sokoban-P29	(LOCATION,0.1)	(STONE,0.7)	(NONE,15.3)	(NONE,460.4)
Sokoban-P30	((STONE,DIR.),0.3)	((STONE,DIR.),0.3)	((STONE,LOC.),3.0)	((STONE,DIR.),122.9)
Transport-P27	(VEHICLE,1.6)	(PACKAGE,14.6)	(PACKAGE,94.7)	(LOCATION,247.3)
Transport-P28	(LOCATION,33.6)	(VEHICLE,121.6)	(VEHICLE,14.7)	(VEHICLE,236.6)
Woodworking-P29	(PART,0.6)	(PART,1.1)	(PART,2.5)	(PART,2.7)
Woodworking-P30	(PART,1.2)	(PART,1.8)	(PART,2.7)	(PART,5.5)

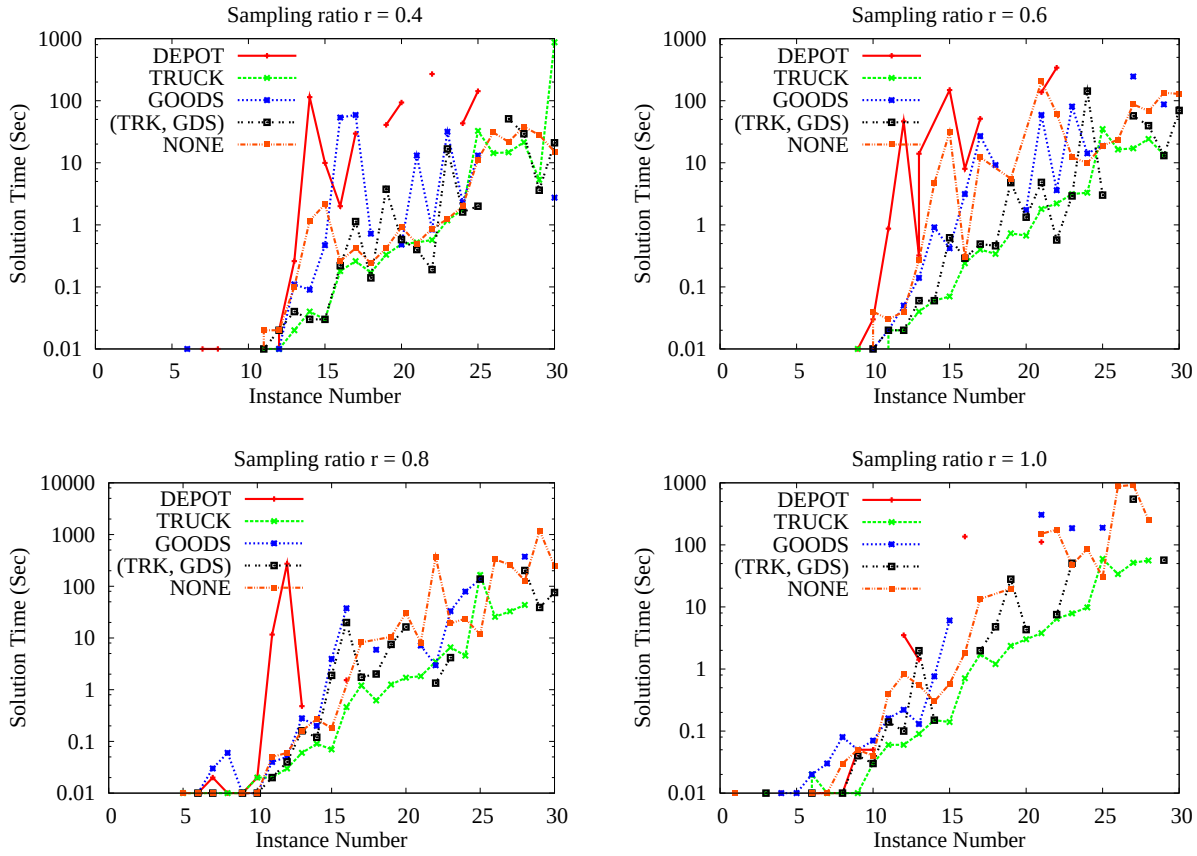


Figure 2.2: Generalization of the best partitioning attribute on the TPP-Propositional domain. Four sampling ratios and four different attributes are shown. The alternative "NONE" indicates a best-first search with parallel decomposition.

instances and compute its best attribute across four values of r. The evaluations are based on a resolution algorithm that uses the "ignore-delete-lists" heuristic function introduced in Chapter 3 and a subproblem solver that uses the best-first search with 'ignore-delete-lists" heuristic function. Table 2.3 shows that the best attributes for a majority of the instances can be generalized from those found for $r = 0.4$. For some instances, including Satellite-P27, Elevator-P30, Sokoban-P29, Sokoban-P30, and Transport-P28, although the best attribute for $r = 0.4$ is not consistent with the true best attribute, using a larger r will lead to the correct attribute. In Elevator-P30, using a suboptimal attribute "passenger" still allows the problem to be solved in 21.6 sec. Similarly, using "location" can solve Sokoban-P29 in 461.9 sec, which is very close to the best (460.4 sec). For Sokoban-P30, every attribute can solve

this instance in 145 sec. For Transport-P28, while "location" is the best when $r = 0.4$, "vehicle" is the second best with a time of 37.3 sec.

For results that do not show good generalizability, we analyze them by dividing them into two classes. In the first, there are two attributes that achieve similar timing results, and our approach cannot distinguish one from another. Examples include Pegsol-P29 (attributes "location" and "none") and Elevator-P29 (attributes "passenger" and "count"). In the second class, the hypothesis of merely generalizing on attributes is insufficient. For example, in Storage-P22 and Transport-P27, the best attribute is inconsistent across simplification. To resolve this issue, we further develop generalization in the resolution component in Chapter 3 in order to efficiently solve those instances, regardless of the attribute selected. There are also several unsolved instances, such as those in the Promela and Trucks domains, that need further study.

Yet there are cases in which the solution times of simplified versions are much smaller than that of the original instance. When the original instance is already simple, simplification will not be beneficial. There are also domains in which simplification cannot be applied. For example, the Openstacks and Pathways domains have no symmetry object groups and cannot be simplified. Fortunately, these domains are easy to solve, and identifying the best attribute is, therefore, not difficult.

2.6 Comparison with Other Constraint Partitioning Approaches

Some existing decomposition-based planning methods do not require constraint partitioning because they assume an implicit partitioning step carried out by users. Earlier planning work, like localized planning [71] or HTN planning [29], assumes a decomposition schema defined in the representation. Planning with MA-STRIPS [11] constructs a partitioning of a problem by defining the capability of each agent. Similarly, planning with logic-based

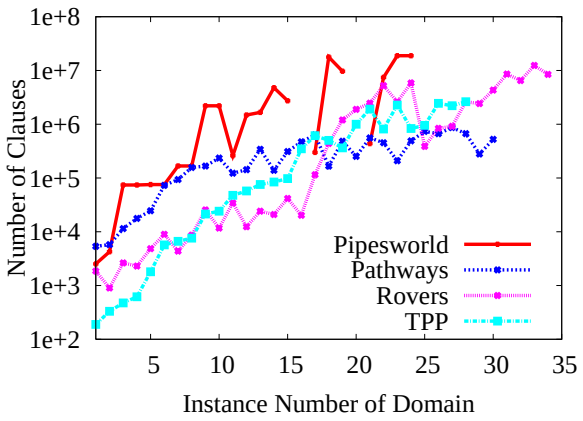


Figure 2.3: Minimum number of clauses needed by SatPlan2006 for representing all feasible plans for some IPC-5 instances.

Benders decomposition [61] assumes partitioning done by users.

When partitioning is not explicitly specified in the problem definition or done by users, constraint partitioning is generally carried out on an abstract representation that exposes the locality of some loosely coupled clusters of actions. This abstract representation can be built on either actions or state variables. The following describes some existing approaches and compares them with our attribute partitioning method.

2.6.1 Partitioning Based on Full Constraint Formulations

Formulations with all their constraints explicitly generated can be too large to be partitioned. Examples of such formulations include GraphPlan [6], SatPlan [65], and other CSP-planning methods. Figure 2.3 shows the exponential number of clauses after translating some IPC-5 propositional benchmarks [21] into conjunctive normal form (CNF) used in SatPlan2006 [66]. Each point is only an under-estimate of the number of clauses in the problem, where clauses are expanded until all goals are reached (in a relaxed fashion) and may not be feasible.

In general, constraints can be classified into level-independent and level-dependent. Level-independent constraints are those that always exist during the entire search, regardless of the level of search exploration. They include persistent mutexes [32], and action (pre-)conditions

and effects. In contrast, level-dependent constraints do not exist at every level of the search. Examples include the initial state and the goal constraints that are boundary conditions of the state trajectory. Non-persistent mutexes (inferred from persistent mutexes and the initial state) are also level-dependent, since they decay as the planning graph [6] expands.

In practice, the number of level-independent constraints dominates the number of level-dependent ones, and the number of initial states and goal constraints is negligible when compared to other types of constraints. Similarly, the number of non-persistent mutexes is negligible when compared to the number of persistent mutexes [32].

The constraints in our action-graph representation are level-independent and correspond to those constraints in one planning step. We do not have to expand more than one step because all level-independent constraints are duplicated in each step, and the locality structures across steps are almost identical. To reduce the complexity of partitioning (in which the number of level-independent constraints can be square of the number of actions), we have used attribute partitioning to group the constraints together by their attribute values.

2.6.2 Constraint Partitioning in SGPlan$_4$ and SGPlan$_5$

Similar to our approach of exploiting the locality of constraints, SGPlan$_4$ [16] uses an action-based graph, where nodes are actions and constraints are mutual exclusions or dependencies between actions. However, SGPlan$_4$ uses subgoals as its only attribute for partitioning action graphs. It assumes that locality is associated with a conjunctive list of explicitly specified top-level goals (subgoals) in a problem definition like PDDL. Each partitioned subproblem, therefore, corresponds to the planning of one top-level goal.

Although subgoal partitioning works well in many domains, it is inadequate for gauging planning complexity in terms of complicating constraints, which ultimately dominates the total complexity. For instance, in Storage-Propositional-P19 in Example 1, subgoal partitioning will choose "crate" as the partitioning attribute, based on the subgoals of transporting all crates to their desired locations. Figure 2.1 demonstrates that partitioning actions by

"crate" leads to worse locality than partitioning actions by "hoist."

$SGPlan_5$ [63] also exploits constraint localities represented in action graphs. It improves $SGPlan_4$ by using *guidance variables* as its partitioning attribute, where a guidance variable is a state variable used in goal constraints. By relaxing localities associated with subgoals, $SGPlan_5$ can analyze localities in problems with language features in PDDL3 [40]. However, it inherits the same limitation in $SGPlan_4$ because it does not fully evaluate all the partitioning alternatives. For instance, it chooses 'crate" as the partitioning attribute in solving Storage-Propositional-P19.

$SGPlan_5$ also tries to improve the granularity of partitioning over $SGPlan_4$. By observing that the number of bottleneck resources is a good indicator on the number of subproblems, it identifies *bottleneck variables* that are a group of state variables in which changes of other state variables must depend on. It then counts the number of bottleneck variables in estimating the number of bottleneck resources. This strategy has been subsumed in our current approach, since bottleneck variables are tied to some attributes, and our method also counts the number of attribute values to set granularity.

2.6.3 Tree Decomposition in Factored Planning

Factored planning [1, 10] was developed for solving planning problems in a discrete-time discrete-state environment, such as STRIPS or SAS^+ [4]. Following the common framework of solving CSPs, it uses a *causal graph* [23], where nodes are state variables and edges represent dependencies among variables. Because its resolution is based on a complete inference and consistency check on state variables, causal graphs are more suitable for describing variable interactions and for doing plan enumerations.

Intuitively, the goal of partitioning in factored planning is to minimize the interactions between factors, where a factor is a subset of state variables as well as actions defined on that subset. The interactions between factors exist in the form of state variables shared across factors. As the complexity of solving a CSP is an exponential function of tree-width [19]

(minimum width of an optimal *tree decomposition*), constraint partitioning is equivalent to tree decomposition on the causal graph with respect to tree-width.

The partitioning step in our attribute partitioning uses an action-graph representation that is very different from the state-variable representation in factored planning. Given that our goal is to find a feasible (rather than all) solution as quickly as possible, our resolution method (described in Section 3) does not enumerate the values of all possible shared variables; hence, minimizing the tree-width would not minimize the complexity of resolution. An action graph is more suitable because our method exploits the locality of constraints among actions instead of state variables.

Our proposed action partitioning approach can also be used to convert STRIPS problems into an MA-STRIPS representation and to construct an *agent interaction digraph* [11]. The reason is that both use an action-based representation that emphasizes interactions between actions.

2.7 Summary

In this chapter, we have formulated the action partitioning step as an attribute selection problem, based on the objective of minimizing the total solution time, and use simplification and generalization to find the best partitioning attribute. With the typing representation, we are able to abstract the action graph by a much smaller attribute graph. We show in Section 2.2 the action partitioning can be reduced into the attribute partitioning.

By observing the repetitions in the planning problem, we propose a simplification procedure in Section 2.3 by identifying and sampling symmetric objects. Since the simplified instances have shared properties with the original instance, we can generalize the best partitioning attribute without trying it on the original instance. We have presented the procedure of finding the best partitioning attribute in Section 2.4.

Section 2.5 demonstrates the best partitioning attribute can be generalized from simpli-

fied instances. We also compare related partitioning algorithm with our proposed attribute partitioning approach in Section 2.6.

Chapter 3

Resolution with State-Space Search

In this chapter, we present our approach for the resolution in parallel decomposition and discuss relevant previous work. Resolution aims to efficiently resolve violated complicating constraints when the actions of a planning problem have been decomposed by action partitioning into subproblems. Section 3.1 presents the possible variants of a resolution framework. Sections 3.2 and 3.3 then discuss our proposed top-down state-space search and the algorithm for selecting an appropriate partition for evaluation based on a guidance heuristic. This is how we reduce the number of evaluated nodes with the locality structure identified. Section 3.4 presents ways to generalize the best guidance heuristic found by solving simplified instances of the original problem, and in Section 3.5, the design of our subproblem solver. We present the design of the decomposed heuristic function for reducing state evaluation cost and its integration in the resolution component. Lastly, Section 3.6 evaluates the generalizability of the best guidance heuristics.

3.1 Variants of Resolution Framework

When a planning problem is decomposed by parallel decomposition, the inconsistent assignments or violated complicating constraints among the subproblems can be resolved either top-down or bottom-up. A top-down approach starts by resolving inconsistencies among the subproblems (master problem) and by propagating guidance information to the subproblems, whereas a bottom-up approach starts by solving the subproblems and by merging the solution of each subproblem into a global solution. In each approach, resolution can operate

on either the state space or the plan space. Below we describe the four combinations of a resolution framework.

3.1.1 Top-Down Plan-Space Resolution

This approach checks inconsistencies in the plan space by actively finding inconsistencies in the current plan. It then resolves those inconsistencies by providing guidance to subproblems in order to modify the current plan.

In serial decomposition, the approach starts from an abstract partial plan and improves it by resolving plan inconsistencies. The search selects an inconsistency and then proposes a modification to plan for that inconsistency. Examples include partial-order planning methods like UCPOP [81], RePOP [79], and VHPOP [89]. LPG [41, 42] uses a stochastic local search on an action-graph representation to solve planning problems in the space of partial plans.

In parallel decomposition, inconsistencies are different because they are clustered into partitions. To resolve them, a top-down approach needs to iterate over those partitions, starting from the most relaxed model to the most restricted one. Examples include *abstract planning* methods [68, 3, 90] that automatically cluster inconsistencies into different abstraction hierarchies, before iteratively refining the plan until all actions are consistent.

There are three reasons why a top-down plan-space approach is not effective under serial decomposition. Firstly, besides the much larger space needed for storing all the partial plans, the approach suffers from the huge search space incurred, since the number of different partial plans is much larger than the number of different states. Further, the complexity of matching partial plans is higher than that of state-space planning approaches, which can recognize repeated states by storing all visited states in a closed list.

Secondly, partial plans and their intermediate states (except the final state) are not useful for checking plan feasibility and optimality. Most current goal tests and plan costs, including violations on soft goals, are defined in state space, where goals are logical or

numerical conditions on state variables. Trajectory constraints can be similarly tested on state variables after adding some artificial variables and axioms.

Lastly, partial-order planning cannot completely describe its intermediate states during a search. Given a partial plan, one cannot predict the applicability of an action in a partial plan and, therefore, cannot compute the corresponding state transitions. Moreover, since flaws and threats in a partial plan do not always correlate to goals in state-based tests, it is difficult to identify good search directions [74]. Another limitation is that many existing heuristics for guidance and pruning are defined in state space, which are not useful when a large search space is exposed in plan-based approaches.

Additional complications also arise in parallel decomposition. In this case, subproblems cannot be solved by existing state-space forward-search methods because one cannot compute the complete intermediate states using a partial plan. Subproblems have to be solved by plan-space solvers that have those disadvantages mentioned above. Without good guidance on plan space, there may be unnecessary backtracking across subproblems. For example, abstract planning methods can produce exponentially longer plans and take exponentially longer time in some cases [4]. For these reasons, a top-down plan-space approach is not effective in parallel decomposition.

3.1.2 Bottom-Up State-Space Resolution

This approach checks consistencies on the basis of state space. A bottom-up approach implies that it pieces the solution plans of some smaller subproblems together in order to form the solution of the original problem.

Serial decomposition extends an existing plan into a solution by augmenting a valid plan-prefix with a well defined state. It works well when the prefix is necessary for the original problem. An example is incremental planning [69, 62] that extends a plan with some necessary conditions to the real goal.

In parallel decomposition, the approach first clusters inconsistencies and constraints over

state-space by grouping state variables and constraints into subproblems. After solving each subproblem by finding some solution plans that satisfy its local constraints, it then merges multiple solution plans into a valid plan for the overall problem. An example is factored planning [1, 10] that generates subproblem solutions by complete enumeration.

A major issue of the bottom-up state-space approach in parallel decomposition comes from plan merging in state space. Merging plans from different subproblems is nontrivial, as a state trajectory is not explicitly defined and may require enumerating possible shared variables. If subproblem plans are inconsistent, the subproblem solver will need to generate all possible states in order to accommodate different scenarios, possibly with feedbacks from consistency checks. This works well when the number of shared variables is small and the size of enumeration is acceptable. The approach is illustrated in factored planning that generates all possible plans up to a given length for each subproblem when resolving inconsistencies. The complexity is manageable for problems having small tree-widths with a small number of shared variables.

Another issue lies in the lack of a well-defined goal for problems partitioned by parallel decomposition. When subproblems are solved independently, existing search techniques developed for each subproblem cannot converge the multiple solutions found into a single coherent solution. In contrast, a global goal can be defined when looking for an optimal solution or all solutions of a planning problem. For instance, factored planning avoids the enumeration of infinitely long plans by limiting the search to a bounded length through iterative deepening. It then employs a tree search to resolve conflicts among all state variables one by one. The strategy is effective because it will always make progress until all state variables are eventually resolved.

3.1.3 Bottom-Up Plan-Space Resolution

The approach starts by solving one or more subproblems, checks their consistencies on the basis of plan space, and modifies the subproblem plans in order to adapt them to the target

problem. Unlike state-space methods, the search trajectory is composed of partial plans.

In serial decomposition, the approach modifies an existing plan for a slightly different problem in order to achieve the goal of the target problem. It works well when an existing plan is not very different from the desired solution. Examples of the approach include plan-reuse systems (e.g. SPA [45] and PRIAR [64]) that show good efficiency in several domains.

In parallel decomposition, the approach solves the subproblems and builds some global solutions from the subplans. It resolves the complicating constraints in a bottom-up manner by first solving the subproblems, merging the subplans if the complicating constraints are satisfied, and modifying the subplans if not. Examples of this approach include SGPlan_4 [16] and TSGP [17].

From a bottom-up perspective, the approach is ineffective in parallel decomposition. Similar to bottom-up state-space methods, the approach lacks global guidance when subproblems are independently solved. Without good guidance, it is difficult to determine the amount of search in each subproblem to generate subplans that can be merged. Such decisions are often based on measuring the distance to a feasible solution by hard-to-tune heuristic functions.

For example, SGPlan_4 [16] implements the approach by a penalty-based method. After generating an initial state for each subproblem, it solves the subproblems to find some subplans. If the subplans cannot be merged into a global plan that satisfies all complicating constraints, it uses penalties to penalizes those actions that introduce the violations, using the number of violated complicating constraints as a heuristic measure of the degree of violations. It then slowly increases the penalties until all the complicating constraints are eventually resolved. The approach is incomplete because penalties alone cannot help a search escape from a trapped infeasible space.

From an implementation perspective, a resolution strategy using a plan-space approach is more complex and has higher search costs, as discussed in Section 3.1.1. When applying a state-space solver to solve problems partitioned by parallel decomposition, it needs to

enumerate possible initial states because a subplan with flaws does not readily lead to new initial states for the solver. It is also more costly to search and backtrack under parallel decomposition, since the process does not follow a well-defined tree structure but rather a loosely connected two-level hierarchy of local and complicating constraints.

The difficulties are further illustrated in SGPlan$_4$. Due to the large branching factor in its resolution algorithm, it is impractical to record all possible choices in its search to help prune future enumerations. As a result, a search may get stuck in the same composed plan evaluated before. Further, the forward state-space search only finds total-ordered plans, and previous subplans cannot be reused in the current iteration if they do not have the same plan prefix.

3.1.4 Top-down State-Space Approach

This approach starts either from scratch or from a very abstract form of the solution represented by a state trajectory. By checking inconsistencies on constraints on state variables, it actively finds and resolves those inconsistencies by proposing one or more consistent variable assignments with less inconsistencies with respect to some quality measure.

In serial decomposition, the approach works like a state-space search. In forward state-space search, an intermediate solution is represented by a state trajectory from an initial state to the state evaluated. A heuristic function is used to measure inconsistencies between each successor state and the goal until the goal state is reached. Examples include HSP [8], FF [58], Optop [75], Yahsp [85], TP4 [46], Fast Downward [48], YochanPS [5], DTG-PLAN [14], LAMA [82], and abstraction heuristics [24, 47, 51]. For numerical and temporal planning, their heuristic functions are usually generalized from those developed for propositional planning. Example planners include Metric-FF [52], TLPlan [2], CPT [86], Sapa [22], MIPS [25], MIPS-XXL [28], and TFD [30].

In backward state-space search, usually mutual exclusions are detected to prune unreachable states. Planning graph is a data structure to efficiently represent the search space of

backward search. Examples are GraphPlan [6], IPP [70], SGP [87] AltAlt [80], STAN [73], and FDP [43]. Although the branching factor of backward search is smaller, the total search complexity is usually higher than that of forward search. It may evaluate more states to reach the solution because of worse heuristic guidance and undetectable unreachable states [7]. Also, we show the branching factor of forward search can be reduced by action partitioning in this work and eliminate the use of backward search.

In top-down parallel decomposition, after decomposing the original problem into subproblems, the evaluation of the subproblems can be guided by a master problem that refines its solution from the most relaxed model. Based on some guidance heuristics, the master problem may identify one or more subproblems to evaluate. A subproblem solver then finds a plan for each subproblem and propagates its feedback to the master problem. The success of the approach lies in the design of good guidance heuristics that can avoid misleading the search to infeasible traps.

One way of providing guidance to the master problem is to derive a Benders cut from previous infeasible solutions. In logic-based Benders decomposition (LBD) [61], subproblems are coupled by a master problem through complicating variables. In each iteration, the master problem is solved by a mixed-integer linear programming solver without constraints on non-complicating variables. Based on some values of the complicating variables, subproblems are then solved completely and optimally to derive a Benders cut for pruning the search space of complicating variables. The new constraint is then added to the master problem, and the process is repeated until all the complicating constraints are resolved.

In general, it is difficult to adapt LBD to automated planning in a top-down state-space framework because generating efficient cuts can be domain dependent. For instance, efficient methods for generating new cuts need to be manually developed [43, 15] for minimizing the number of parallel time steps in SatPlan-like planners. Without domain information, a subproblem planner must resort to a complete and optimal search in order to derive Benders cuts for each possible planning scenario. Oftentimes, the original problem can be solved

more efficiently without relying on these cuts. For example, the baseline planner in the sequential optimization track of IPC-6 [50] was able to solve many instances by simply using a uniform-cost search.

In short, to resolve constraints among subproblems partitioned by parallel decomposition, a bottom-up approach has difficulty in providing good guidance to subproblems, whereas a plan-space approach suffers from a much larger search space than a state-space approach and requires the aid of difficult-to-design guidance heuristics. In contrast, a top-down state-space approach can be integrated easily with existing forward state-space planners. In the next section, we illustrate our proposed top-down state-space approach that integrates pruning and backtracking in resolution.

3.2 Examples Illustrating Proposed Top-Down State-Space Resolution

Consider a planning problem with strong locality. Without considering locality in generating successors of a state, there may be superfluous searches when all actions in a branch are considered, especially when the branching factor is large. In contrast, by recognizing that neighboring actions are likely to be in the same action partition, we can significantly reduce the branching factor by limiting only actions in the same partition in a search. However, indiscriminately applying the approach may cause the search to get stuck in an action partition. The following example illustrates this point.

Example 10 *Consider a feasible plan of Storage-Propositional-P19 with 41 actions. Each line represents an action of the form "*time*: (*action*) [1]" where* time *specifies the schedule of the action and* action *is defined by the name of the action schema followed by its parameters.*

```
0.001: (GO-OUT HOIST2 DEPOT0-2-2 LOADAREA) [1]
1.002: (LIFT HOIST2 CRATE8 CONTAINER-2-0 LOADAREA CONTAINER2) [1]
2.003: (GO-IN HOIST2 LOADAREA DEPOT2-2-1) [1]
3.004: (DROP HOIST2 CRATE8 DEPOT2-1-1 DEPOT2-2-1 DEPOT2) [1]
```

```
4.005: (GO-OUT HOIST2 DEPOT2-2-1 LOADAREA) [1]
5.006: (LIFT HOIST2 CRATE5 CONTAINER-1-1 LOADAREA CONTAINER1) [1]
6.007: (GO-IN HOIST2 LOADAREA DEPOT1-2-2) [1]
7.008: (MOVE HOIST2 DEPOT1-2-2 DEPOT1-2-1) [1]
8.009: (DROP HOIST2 CRATE5 DEPOT1-2-2 DEPOT1-2-1 DEPOT1) [1]
9.010: (MOVE HOIST2 DEPOT1-2-1 DEPOT1-1-1) [1]
10.011: (MOVE HOIST2 DEPOT1-1-1 DEPOT1-1-2) [1]
11.012: (MOVE HOIST1 DEPOT0-1-2 DEPOT0-2-2) [1]
12.013: (LIFT HOIST2 CRATE5 DEPOT1-2-2 DEPOT1-1-2 DEPOT1) [1]
13.014: (DROP HOIST2 CRATE5 DEPOT1-1-3 DEPOT1-1-2 DEPOT1) [1]
14.015: (GO-OUT HOIST1 DEPOT0-2-2 LOADAREA) [1]
15.016: (LIFT HOIST1 CRATE3 CONTAINER-0-3 LOADAREA CONTAINER0) [1]
16.017: (DROP HOIST1 CRATE3 DEPOT1-2-2 LOADAREA DEPOT1) [1]
17.018: (GO-IN HOIST1 LOADAREA DEPOT2-2-1) [1]
18.019: (LIFT HOIST2 CRATE3 DEPOT1-2-2 DEPOT1-1-2 DEPOT1) [1]
19.020: (DROP HOIST2 CRATE3 DEPOT1-1-1 DEPOT1-1-2 DEPOT1) [1]
20.021: (GO-OUT HOIST1 DEPOT2-2-1 LOADAREA) [1]
21.022: (LIFT HOIST1 CRATE4 CONTAINER-1-0 LOADAREA CONTAINER1) [1]
22.023: (DROP HOIST1 CRATE4 DEPOT1-2-2 LOADAREA DEPOT1) [1]
23.024: (LIFT HOIST1 CRATE0 CONTAINER-0-0 LOADAREA CONTAINER0) [1]
24.025: (GO-IN HOIST1 LOADAREA DEPOT0-2-2) [1]
25.026: (DROP HOIST1 CRATE0 DEPOT0-1-2 DEPOT0-2-2 DEPOT0) [1]
26.027: (GO-OUT HOIST1 DEPOT0-2-2 LOADAREA) [1]
27.028: (LIFT HOIST1 CRATE1 CONTAINER-0-1 LOADAREA CONTAINER0) [1]
28.029: (GO-IN HOIST1 LOADAREA DEPOT0-2-2) [1]
29.030: (DROP HOIST1 CRATE1 DEPOT0-2-3 DEPOT0-2-2 DEPOT0) [1]
30.031: (GO-OUT HOIST1 DEPOT0-2-2 LOADAREA) [1]
31.032: (LIFT HOIST1 CRATE2 CONTAINER-0-2 LOADAREA CONTAINER0) [1]
32.033: (DROP HOIST1 CRATE2 DEPOT0-2-2 LOADAREA DEPOT0) [1]
33.034: (LIFT HOIST1 CRATE7 CONTAINER-1-3 LOADAREA CONTAINER1) [1]
34.035: (GO-IN HOIST1 LOADAREA DEPOT2-2-1) [1]
35.036: (MOVE HOIST1 DEPOT2-2-1 DEPOT2-2-2) [1]
36.037: (DROP HOIST1 CRATE7 DEPOT2-2-3 DEPOT2-2-2 DEPOT2) [1]
37.038: (MOVE HOIST1 DEPOT2-2-2 DEPOT2-2-1) [1]
38.039: (GO-OUT HOIST1 DEPOT2-2-1 LOADAREA) [1]
39.040: (LIFT HOIST1 CRATE6 CONTAINER-1-2 LOADAREA CONTAINER1) [1]
40.041: (DROP HOIST1 CRATE6 DEPOT2-2-1 LOADAREA DEPOT2) [1]
```

The solution plan demonstrates strong action locality on hoists: all actions before 10.011 involve HOIST2, whereas those after 20.021 involve HOIST1, and no action involves HOIST0.

With action partitioning by hoists and if the use of HOIST2 can be correctly predicted in the first 11 actions, we can focus on the action partition involving HOIST2 alone. In contract, without action partitioning, the branching factor is defined by the three hoists. For instance, at time 3.004, we can use HOIST0 or HOIST2 to move other crates to achieve the final goal, in addition to using HOIST2 for CRATE8.

Note that actions between 11.012 and 19.020 involve HOIST1 and HOIST2. Hence, for a search that focuses on one hoist at a time while fixing the states of the other hoists, it may get stuck in an infeasible state. For example, we cannot solely use HOIST2 to move crates to DEPOT0 after time 12.013, since we have already blocked the entrance of DEPOT0 (DEPOT0-2-2) by HOIST1.

One way to address the issue in the above example is to interleave a top-down search across multiple action partitions in order to avoid getting stuck in one partition. If we can determine when to switch to a new partition, then planning can be highly localized with reduced complexity. The approach is most suitable for a top-down search that keeps an abstract view of the global problem and does not focus on one particular action partition until it is necessary.

By combining top-down with existing forward state-space methods, we propose a new top-down state-space resolution algorithm for solving planning problems partitioned by parallel decomposition. Our approach uses heuristic values computed from relaxed plans to guide the search towards the final goal, similar to common heuristics-guided state-space searches. While such searches may have high complexity due to large branching factors, we employs a two-level strategy to reduce the branching factor. In each iteration, we only focus on states expanded in the same action partition. Using the heuristic values computed from the subproblems, we first identify the action partition to be searched. Since the locality structure boosts the accuracy of action-partition selection, there is a good chance that the search will stay in the same action partition in the next iteration. At some time, the search will switch to another partition with a better heuristic value. By using localities to limit the search to

one action partition at a time, while providing the flexibility to switch to another partition, our approach can approximate the performance of a best-first heuristic search with a lower branching factor.

Example 11 *Assuming heuristic $h_{\mathcal{G}}$ based on FF's* ignore-delete-lists *[58] is used to guide the search in solving Storage-Propositional-P19, we illustrate in this example the selection of the action partition in our two-level heuristic search. We assume that (***crate***,* **hoist***) has been identified by Algorithm 1 as the partitioning attribute for decomposing the action graph into subproblems.*

Consider the solution plan in Example 10. Given that the preceding action is `1.002: (LIFT HOIST2 CRATE8 CONTAINER-2-0 LOADAREA CONTAINER2) [1]`*, the next action is from the same action partition corresponding to (***crate8***,* **hoist2***) as suggested by $h_{\mathcal{G}}$. After executing the next two actions at* `2.003` *and* `3.004` *from the same partition, CRATE8 has been placed in DEPOT2. At this point, $h_{\mathcal{G}}$ suggests switching to the action partition corresponding to (***crate8***,* **hoist5***), and the next two actions at* `4.005` *and* `5.006` *are evaluated. By repeating the process, the CPU time needed to reach the goal is 2.5 sec.*

Without identifying a suitable action partition in each iteration, the search may wander around and make slow progress towards the final goal. For instance, if we always use a partition different from that suggested by $h_{\mathcal{G}}$, the CPU time needed to reach the goal state can exceed 15 minutes.

In addition to $h_{\mathcal{G}}$, we have studied heuristics $h_{\mathcal{G}_T}$ based on a transformed goal $\mathcal{G}_T$ and h_{LM} based on landmarks [60]. $h_{\mathcal{G}_T}$ first finds $\mathcal{G}'$ that is supposedly more informative than the original goal. It then calls $h_{\mathcal{G}}$ in the following search. h_{LM} counts the number of visited landmarks. Each time, we select the most promising state $\mathcal{I}'$ in terms of its heuristic value as the initial state of a subproblem, with a goal of achieving an unvisited landmark in that subproblem.

In general, the best guidance heuristic is unknown without solving the original problem. To this end, our approach is to evaluate $h_{\mathcal{G}}$, $h_{\mathcal{G}_T}$, and h_{LM} on smaller instances of the original problem and to generalize the best heuristic found. We illustrate the approach as follows and leave the details to the next section.

Example 12 *Assuming (***crate**, **hoist***) to be the partitioning attribute for Storage-Propositional-P19. We like to determine the best guidance heuristic in such a way that minimizes the time for solving the original problem. By applying a simplification and generalization approach similar to that of finding the best partitioning attribute, we derive simplified instances of the original problem using Algorithm 3, determine the best guidance heuristic for these instances, and generalize the best guidance heuristic to the original problem. Based on $h_{\mathcal{G}}$, $h_{\mathcal{G}_T}$, and h_{LM}, the following table shows that $h_{\mathcal{G}}$ found by solving the instance with $r = 0.4$ can be generalized to the original problem.*

Guidance	*Sampling Ratio r*			
Heuristic	0.4	0.6	0.8	1.0
$h_{\mathcal{G}}$	*0.2 sec.*	*0.4 sec.*	*1.0 sec.*	*2.5 sec.*
$h_{\mathcal{G}_T}$	*0.3 sec.*	*0.5 sec.*	*13.3 sec.*	*5.3 sec.*
h_{LM}	*0.4 sec.*	*1.7 sec.*	*3.7 sec.*	*4.5 sec.*

3.3 Constraint Resolution in Top-Down State-Space Parallel Decomposition

Motivated by the examples in the last section, we present in this section our proposed top-down state-space constraint-resolution method. We defer to the next section to present our strategy for finding the best guidance heuristic through simplification and generalization.

Algorithm 5 summarizes our proposed top-down state-space search for resolving constraints among subproblems. It is a heuristic search that selects the most promising state $\mathcal{I}'$ with the best heuristic value and that expands the neighboring states of $\mathcal{I}'$. By selecting

successors of each state only from an action partition, our approach limits the branching factor in each step.

Input: Problem $\mathcal{T} = (\mathcal{V}, \mathcal{A}, \mathcal{I}, \mathcal{G})$
Output: Solution plan
1 $\{\mathcal{A}_1, \ldots, \mathcal{A}_K\}$:= partition_actions($\mathcal{T}$) (Algorithm 1);
2 $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$:= generate_simplified_instances($\mathcal{T}$) (Algorithm 3);
3 h := identify_best_heuristic($\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$) (Algorithm 6);
4 *open* := $[\mathcal{I}]$; *closed* := [];
5 **while** *open is not empty* **do**
6 | $\mathcal{I}'$:= remove_next_state(h, *open*);
7 | **if** $\mathcal{I}' \in \mathcal{G}$ **then**
8 | | output_solution(); return;
9 | **end**
10 | **if** $\mathcal{I}' \notin$ *closed* **then**
11 | | $\mathcal{T}'$:= formulate_subproblem($\mathcal{T}$, h, $\mathcal{I}'$, $\mathcal{G}'$);
12 | | {*open*, *closed*} := solve_subproblem($\mathcal{T}'$, *open*, *closed*) (Alg. 7);
13 | **end**
14 **end**
Algorithm 5: *SGPlan6*: top-down state-space search with parallel decomposition

Line 1 partitions an action graph by Algorithm 1 into K disjoint subsets. Lines 2 and 3 then apply Algorithm 3 to generate simplified instances and Algorithm 6 to identify the best guidance heuristic h defined in Definition 6.

Line 4 initializes the open list to be the initial state and the closed list to be empty. Here, the open list is a priority queue of states that have not been fully explored, and a closed list contains those expanded states.

Lines 5-14 is a loop that iteratively expands the states by solving a series of subproblems until it reaches $\mathcal{G}$. Here, a subproblem is a group of states explored by the heuristic search in each iteration of the loop.

Definition 17 *Subproblem $\mathcal{T}$' = {$\mathcal{V}$, $\mathcal{A}$, $\mathcal{I}$', $\mathcal{G}$', k} is a quintuple, where $\mathcal{V}$ and $\mathcal{A}$ are inherited from original problem $\mathcal{T}$, $\mathcal{I}'$ is the initial state of the subproblem, $\mathcal{G}$' is the subproblem goal, and k is the selected action partition.*

The states explored include $\mathcal{I}'$ with the minimum heuristic value chosen from the open list

(Line 6), as well as all its neighboring states with respect to h and the search strategy. It then tests whether $\mathcal{I}'$ satisfies $\mathcal{G}$. If it does, we return the plan found (Line 8). If not and $\mathcal{I}'$ has not been expanded, we formulate $\mathcal{T}'$ with goal $\mathcal{G}'$ whose components are specific to h and $\mathcal{T}$ (Line 11).

Lastly, the subproblem solver (Algorithm 7) handles the state exploration of $\mathcal{G}'$, and updates the open and closed lists (Line 12). The solver computes the heuristic value with respect to h for each newly added node in the open list. Also, it guarantees that no goal state has been placed in the closed list.

In general, heuristic values are obtained by relaxing a subset of the conditions of a planning problem. For example, in common forward state-space searches, the heuristic value of a partial plan is derived by satisfying the constraints while relaxing the problem goal because only an applicable action that does not induce mutual exclusions is to be scheduled. In contrast, partial-order-planning methods start from an abstract plan that satisfies both the initial state and the goal. Therefore, the search is conducted in plan space in order to satisfy those precondition constraints and mutual exclusions. In this sense, their heuristic values are measured by the number of flaws in the partial plan.

In planning problems partitioned by parallel decomposition, their heuristic values can be computed in a similar fashion by relaxing their constraints. However, they cannot be computed by relaxing mutual exclusions because a parallel forward search will switch indiscriminately among action partitions when mutual exclusions among parallel subplans are relaxed.

In our approach, we exploit the fact that the set of goal constraints is usually smaller and less complex than the set of mutual exclusions. To this end, we design a forward heuristics-guided search towards the satisfaction of the goal while maintaining the feasibility of precondition constraints and mutual exclusions. The result is a state-space search, since inconsistencies due to parallel decomposition can only happen on the state trajectory.

Another decision the subproblem solver needs to make is to determine the active partition

in which $\mathcal{T}$ is to be evaluated. Of course, actions in the active partition must be directly related to the satisfaction of $\mathcal{G}$. Fortunately, there are usually multiple partitions available, and the task is to merely eliminate those unsuitable ones. Our solution is to order the relaxed plan $\{a_1, a_2, \ldots\}$ from $\mathcal{I}'$ to $\mathcal{G}$ by the *ignore-delete-lists* heuristic and to select $\mathcal{A}_k$ if $a_1 \in \mathcal{A}_k$, regardless of the actual heuristic h used in the subproblem solver. From the aspect of finding exit paths with respect to the *ignore-delete-lists* heuristic, we observe that we can decrease heuristic value with a_1 if we re-achieve the delete effects of a_1. Thus, the exit path must have some action mutual exclusive with a_1 and like in the same action partition as a_1.

With respect to those unselected action partitions, instead of pruning their actions as in Benders cuts, we prioritize them in such a way that they will not be used unless no action in the selected partition is applicable. The ability to prioritize the partitions and to switch to another reduces the risk of getting stuck in one with worse heuristic values. In general, the search will likely continue in the same partition because the locality structure boosts the accuracy of action selection. The details of this step are presented in Section 3.5.

In short, we have formulated constraint resolution in parallel decomposition as a top-down state-space search. By selecting a proper action partition, we can reduce the branching factor of the search. In the next section, we study the trade-offs in determining the best guidance heuristic function.

3.4 Learning Generalized Heuristic Functions

In this section, we develop heuristics for guiding top-down state-space searches in parallel decomposition. Our approach can be considered in the context of learning and planning. In the common learning paradigm in automated planning, a hypothesis space is defined, followed by the identification of generalizable hypotheses from given solution plans or search traces. The system then uses the learned hypotheses to benefit planning for other similar instances.

Existing learning methods assume some properties that can be generalized from training instances or traces of test problems. They usually assume some training instances and their solutions, where such instances are smaller problems in the same benchmark suite. They can be classified into three types.

1. Control knowledge can be learned for a forward state-space search. This knowledge can be in the form of heuristic functions and qualitative reactive policies, including measure of progress and decision-list rules [88].

2. Generalized plans [84] is the most aggressive way of learning to reuse existing plans. This assumes an algorithm-like plan that can be used for solving a class of problems. After learning a generalized plan, solving different problems just involves varying the number of objects or other instance-dependent features. Earlier work like L2ACT [67] also generates a linear sequence of actions selected by the learned reactive policies.

3. Macro-actions, each with more than one actions, can be learned to dramatically reduce the depth of a search tree. By learning macro-actions from a training set with the aid of a domain-component abstraction, Macro-FF [9] applies them to solve new instances in the same domain. Marvin [18] learns plateau-escaping macro-actions for FF from previous searches of similar plateaus. However, an exhaustive search is generally needed for plateaus with unseen structures. A more general setting is to learn macros using genetic algorithms without domain properties [78].

While previous methods focus on learning strategies for solving planning subproblems, our approach here addresses learning in the context of parallel decomposition, in particular, the learning and generalization of guidance heuristics in the top-down state-space search. In the generalization of guidance heuristics (Definition 16), the hypothesis is to output heuristic h that minimizes the time for solving the original problem $\mathcal{T}$. Here, h can be generalized from $\mathcal{T}'$ to $\mathcal{T}$ when the best heuristic of $\mathcal{T}'$ and $\mathcal{T}$ are the same; that is, we can infer $h(\mathcal{T})$ via $h(\mathcal{T}')$ by using the run-time performance of h on $\mathcal{T}'$.

Our approach consists of two parts. First, we propose in Section 3.4.1 three heuristics in order to cover different trade-offs between the number of states evaluated and the evaluation cost. We use a fixed set of heuristics because it is difficult to design general heuristics at run time in an automated fashion. Second, we present in Section 3.4.2 a generalization method that exploits the similarity in structures between the original problem and its simplified instances. The generalization paradigm is similar to that for designing the best partitioning attribute in Section 2.4. Example 12 has illustrated that the performance of heuristics can be generalized from smaller instances of the original problem.

3.4.1 Heuristics for Guiding Top-Down State-Space Search

In this section, we present three heuristics, each targeting a specific trade-off between the number of states evaluated and the state evaluation cost. Roughly speaking, $h_{\mathcal{G}_T}$ aims to largely reduce the number of states evaluated at the cost of slower state evaluation than $h_{\mathcal{G}}$. h_{LM} aims to largely reduce the state evaluation cost at the cost of more states evaluated than $h_{\mathcal{G}}$.

Guidance heuristic $h_{\mathcal{G}}$ based on "Ignore-delete-lists." This employs the *ignore-delete-lists* heuristic to compute the length of a relaxed plan to the original goal $\mathcal{G}$. Despite action partitioning, it considers the entire planning problem when computed. In evaluating a subproblem using $h_{\mathcal{G}}$, the solver expands all unseen neighboring states of $\mathcal{I}'$ from the open list, compute their heuristic values, inserts the resulting states into the open list, and places $\mathcal{I}'$ in the closed list.

Guidance heuristic $h_{\mathcal{G}_T}$ based on a transformed goal $\mathcal{G}_T$. This is motivated by the observation that the goal specified in the *ignore-delete-lists* heuristic is not sufficiently informative, leading to a large number of state evaluated though at relatively low evaluation costs.

To reduce the number of states evaluated, we design a new heuristic $h_{\mathcal{G}_T}$ that aims to achieve a more informed goal $\mathcal{G}_T$. $\mathcal{G}_T$ implies or includes $\mathcal{G}$ but has more top-level goals,

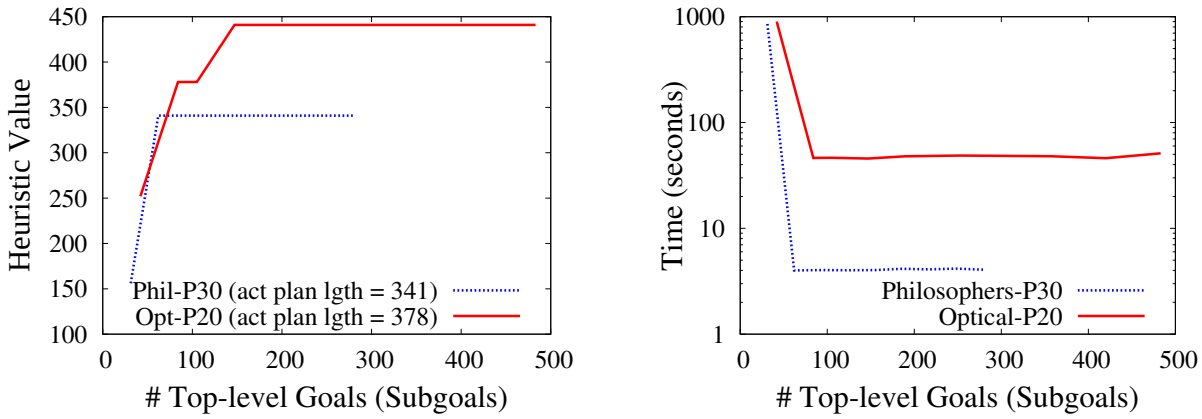


Figure 3.1: Relationship between heuristic values/solution times and complexity of goal constraints in terms of the number of top-level goals.

whose goal constraints are in the form that are consistent with and lead to $\mathcal{G}$. Because $\mathcal{G}_T$ does not need to be achieved before reaching $\mathcal{G}$, we only have to check $\mathcal{G}$ in our goal test when evaluating heuristic values.

Example 13 *To illustrate that relatively small efforts to increase top-level goals in $\mathcal{G}_T$ will lead to large reductions in solution times, consider two IPC-4 instances: Optical-P20 and Philosophers-P30. We enumerate various numbers of top-level goals (derived from a real solution) in the problem specification and record their solution times and heuristic values in Figure 3.1 (when evaluated using serial decomposition). We observe that $\mathcal{G}$ is not sufficiently informative, and the corresponding heuristic value is underestimated. In Optical-P20 with 42 top-level goals, the heuristic value (length of relaxed plan) with respect to these goals is 252, whereas the length of a real plan is 378. Likewise, in Philosophers-P30 with 31 top-level goals, the heuristic value with respect to these goals is 155, whereas the length of a real plan is 341. We also observe in Figure 3.1 that both the heuristic values and the total time converge quickly as the number of top-level goals is increased.*

New top-level goals can be added in $\mathcal{G}_T$ when $\mathcal{G}$ is composed of *accompanying state variables* whose values can be inferred from the values of some governing state variables. For each accompanying variable in $\mathcal{G}$, we add a top-level goal based on its governing state

variable without introducing any inconsistency.

Example 14 *In* storage-19-simp*,* on(crate1, depot0-1-1) *implies* in(crate1, depot0)*, given that* in(depot0-1-1 depot0) *is true (where* depot0-1-1 *is a storage area in* depot0*). For a top-level goal* in(crate1, depot0)*, we can add* on(crate1, depot0-1-1) *to enrich the goal information. The identification of accompanying state variables is possible because actions* add on(crate1, depot0-*i*-*j*) *for each storage area* depot0-*i*-*j* *in* depot0 *must also* add in(crate1, depot0)*, whereas* delete in(crate1, depot0) *must also* delete on(crate1, depot0-*i*-*j*) *for some i-j pair.*

On the other hand, if there are no accompanying state variables, then we carry out a backward search from $\mathcal{G}$ to collect conditions of actions that directly assert the goal. For each top-level goal in $\mathcal{G}$, we find an action that establishes it. We then add all conditions on that action without inducing inconsistencies with $\mathcal{G}$. Because this technique adds more top-level goals than the first one and increases the risk of inconsistencies, it should only be used when there are no accompanying state variables.

In general, the technique is incomplete and suboptimal. It may create an infeasible $\mathcal{G}_T$, as some inconsistencies cannot be detected without solving the problem. Consequently, it should only be applied on the simplified instances, which may provide hints on whether it is applicable to the original problem.

Guidance heuristic h_{LM} based on landmarks. This computes the number of landmarks that have not been accepted or are accepted but are required again [82], where landmarks [60] are facts that must be true at some time in every solution plan. (We assume that the partial orders among landmarks have been detected using existing techniques [60, 16, 82].)

In our implementation, when selecting $\mathcal{I}'$ from the open list, we require the subplan from $\mathcal{I}'$ to a desirable landmark L to have a heuristic value different from that of $\mathcal{I}'$. Since the action partition selected is the one with the first action $\mathcal{I}'$ of the relaxed plan to $\mathcal{G}$, we set L, the first desirable landmark to be achieved in the relaxed plan, to depend on $\mathcal{I}'$. We then

use the *ignore-delete-lists* heuristic to compute the length of the relaxed plan to L and call the subproblem solver to search for plans to L. If L cannot be achieved using the selected action partition, then we switch to the next action partition.

Unlike $h_{\mathcal{G}}$ and $h_{\mathcal{G}_T}$, the subproblem solver using h_{LM} can now place multiple states in the closed list. This happens when L cannot be immediately achieved from $\mathcal{I}'$, and the search takes multiple actions to reach L. Further, since each landmark L is necessary in order to reach $\mathcal{G}$, it is guaranteed that no goal state is overlooked in the solver. Regarding the update of the open list, we have to recompute h_{LM} for each state returned by the subproblem solver. This overhead is small as h_{LM} is easily computed by counting the landmarks.

When compared to $h_{\mathcal{G}}$ and $h_{\mathcal{G}_T}$, h_{LM} exploits different trade-offs between evaluation cost and number of states evaluated. While LAMA [82] combines h_{LM} and $h_{\mathcal{G}}$ in order to have a more informed single heuristic, we achieve different trade-offs by using them separately. h_{LM} is more abstract because its computation measures changes to at least one of the landmarks, which involves variables only related to landmarks. In contrast, the computation of $h_{\mathcal{G}}$ and $h_{\mathcal{G}_T}$ involves all state variables when evaluating all unseen successors of a state. The evaluation cost of h_{LM} is also smaller and the relaxed plan is shorter because landmarks are usually reached before $\mathcal{G}$. However, h_{LM} can be less accurate, which may lead to a longer search depth or more states evaluated.

3.4.2 Generalization of the Best Guidance Heuristic

Algorithm 6 identifies the best guidance heuristic using a series of simplified instances, $\mathcal{T}_1$, ..., $\mathcal{T}$, generated by Algorithm 3. It is very similar to Algorithm 1 for identifying the best partitioning attribute.

The loop between Lines 4 and 3 enumerates each of the three heuristics in Section 3.4.1 on the sequence of simplified instances. In each iteration, it records the best heuristic found so far. It leaves the loop if has found a heuristic that is consistently the best for the sequence of simplified instances evaluated; otherwise, it tries a larger instance until it reaches the last

Input: A series of simplified instances: $\{\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}\}$
Output: Best guidance heuristic h_i for generalization to $\mathcal{T}$

```
1 i := 0;
2 repeat
3 |   i := i + 1;
4 |   h_i := enumerate_heuristic(T_i);
5 until T_i = T or consistent_heuristic(h_1, h_2, ..., h_i) ;
```

Algorithm 6: *identify_best_heuristic*: selecting the best guidance heuristic

instance (the original problem $\mathcal{T}$). To reduce the time of evaluating simplified instance $\mathcal{T}_i$, we try the heuristics in ascending order of solution times on $\mathcal{T}_{i-1}$.

3.5 The Subproblem Solver in Parallel Decomposition

In this section, we present the design of a solver for generating a subplan for a partitioned subproblem $\mathcal{T}'$ (Definition 17). Our solver is implemented as a top-down state-space search to allow better integration with the top-down state-space resolution.

The main overhead of the solver lies the evaluation of the guidance heuristic, which entails the construction a relaxed graph and the scheduling of its actions. The relaxed graph has strong locality, given the strong locality in the original action graph. By utilizing this locality, it leads to significant reduction in the overhead in evaluating the guidance heuristic. Similar to Example 10, the following illustrates the relation between locality and relaxed planning.

Example 15 *Consider an IPC-4 airport ground-traffic-control task [57]. Its goal is to construct the travel routes of all the airplanes in order to reach their destination gates, while avoiding collisions and other unsafe situations.*

Assume that airplane A2 *is initially at* g5*, that* A3 *is at* g13*, and that the only route for* A2 *to travel to* g12 *is through the following sequence of gates:*

$$\mathsf{g5(A2)} \Longrightarrow \mathsf{g4} \Longrightarrow \mathsf{g3} \Longrightarrow \mathsf{g2} \Longrightarrow \mathsf{g13(A3)} \Longrightarrow \mathsf{g12}.$$

To compute a relaxed plan to subgoal (at A2 g12) *using FF, the solver needs to move* A2 *from* g5 *to* g12*, as well as move* A3 *away from* g13*, since* A3 *blocks the route of* A2 *to* g12*. It follows that most of the actions in the relaxed plan are related to* A2*. In other words, necessary facts like* (at A2 g4) *and* (at A2 g3) *can only be achieved using actions related to* A2*.*

In the above example, the generation of the first few layers of the relaxed plan involves a small number of actions from the action partition containing the initial state. In general, although one cannot guarantee that all actions in the relaxed plan are from this partition, one way to reduce the complexity is to initially ignore those actions not in this partition and to construct the final relaxed plan using the entire set of actions only when needed.

Existing planners generally do not exploit localities in generating relaxed plans. Although one can eliminate irrelevant actions that will never be part of any solution, few actions will be proved irrelevant to goal satisfaction [59]. For instance, since A3 in Example 15 blocks the route of A2, those actions corresponding to A3 are relevant to subgoal (at A2 g12) and cannot be eliminated. For this reason, SGPlan_4 does not exploit localities in subproblems. In LBD, the reduction of complexity is addressed by its partitioning component that generates small subproblems solved efficiently by a constraint programming solver. In factored planning, low complexity is achieved by minimizing the interactions across subproblems and by using an enumeration-based solver.

Algorithm 7 presents the state-space search implemented in our subproblem solver. It is solution preserving, as it does not eliminate any actions. The states in the *open* list are arranged by their heuristic values computed using one of three guidance heuristics $h_{\mathcal{G}}$, $h_{\mathcal{G}_T}$ and h_{LM} described in Section 3.4.1. As discussed earlier, it starts its search in the same partition k as that of $\mathcal{I}'$ but does not limit the search to it, since actions in the relaxed plan may not always be from this partition. Similar to the approach in Fast Downward [48] and LAMA [82] for prioritizing preferred operators, it maintains two open lists: *op* for actions in the k^{th} partition and *open*$'$ for all other actions. The union of these two lists is the search

Input: $\mathcal{T}' = (\mathcal{V}, \mathcal{A}, \mathcal{I}', \mathcal{G}', k)$, open list *open*, and closed list *closed*
Output: Modified open list *open* and closed list *closed*

```
1  open' := []; op := [I];
2  closed' := closed;
3  while open' is not empty or op is not empty do
4  |  if op is not empty then
5  |  |  s := remove_next_state(op);
6  |  else
7  |  |  s := remove_next_state(open');
8  |  end
9  |  if open' ⊨ G' then
10 |  |  open := update_open_list(open, op, open');
11 |  |  closed := closed';
12 |  |  return open, closed;
13 |  end
14 |  if s ∉ closed' then
15 |  |  closed' := closed' ∪ s;
16 |  |  foreach a ∈ applicable_actions(s) do
17 |  |  |  if a ∈ A_k then
18 |  |  |  |  op := op∪ successor(s, a);
19 |  |  |  else
20 |  |  |  |  open' := open'∪ successor(s, a);
21 |  |  |  end
22 |  |  end
23 |  end
24 end
```

Algorithm 7: *solve_subproblem*: subproblem solver with state-space search

frontier that initially contains $\mathcal{I}'$ (Line 1). However, *open*$'$ is not used until *op* is empty (Lines 4-7). The closed list *closed*$'$ will be inherited from *closed* in the resolution component (Line 2).

Next, it checks whether $\mathcal{G}'$ has been reached. If so, it adds all the nodes in *op* and *open*$'$ to *open* (Line 10). In case that h_{LM} is used, the heuristic value of each state in Algorithm 5 needs to be recomputed because the order of the search is different with respect to ordering the nodes by $h_{\mathcal{G}}$. It then moves all newly added states of *closed*$'$ to *closed* (Line 11) and returns to Algorithm 5.

If $\mathcal{G}'$ has not been reached and the current state s is not in *closed*$'$, then it places s in

closed′ (Line 15) and generates all successors of s by enumerating all the applicable actions a of s. It places a in *op* if a is in the k^{th} partition (Line 18); otherwise, it places a in *open*′ to demote its priority (Line 20). In generating a successor state of s and its heuristic value by the *successor* function, we have developed two techniques for reducing the evaluation cost.

First, we compute the heuristic value of s by exploiting action locality. In building the relaxed plan, we initially use actions in the active partition, and apply actions in other partitions only when the problem cannot be solved.

We illustrate this technique on the computation of heuristic values using the *ignore-delete-lists* function $h_{\mathcal{G}}$. In building the relaxed plan, actions in the active partition and their add effects are incrementally inserted into the planning graph when their preconditions get supported, whereas actions not in the active partition are postponed until no actions and effects can be inserted. If the goal is not achieved using actions in the active partition, we first apply those postponed actions and generate the relaxed planning graph as before. The extraction of the relaxed plan is done by a recursive backward search of the relaxed planning graph [58]. The approach is solution-preserving because actions can only be delayed but not eliminated. However, it may generate a longer relaxed plan when we restrict actions to only the active partition, as compared to a relaxed plan with the same effect but not limited to the active partition. Without loss of generality, we can assume that all actions are related to $\mathcal{G}'$. (If not, we can eliminate them using the second technique.) If the set of reachable facts using actions in the active partition are very similar to the set of all reachable facts, then this technique will lead to great saving in generating a relaxed plan. On the other hand, the overhead (both the number of actions used and the number of facts achieved) is negligible if the set of reachable facts using actions in the active partition is much smaller than the set all reachable facts. In both cases, this technique will have no negative impact on the cost of generating a relaxed plan.

Second, we ignore those actions irrelevant to $\mathcal{G}'$ using *backward relevance analysis* implemented in SGPlan$_4$ when computing heuristic values. Whereas SGPlan$_4$ and other work

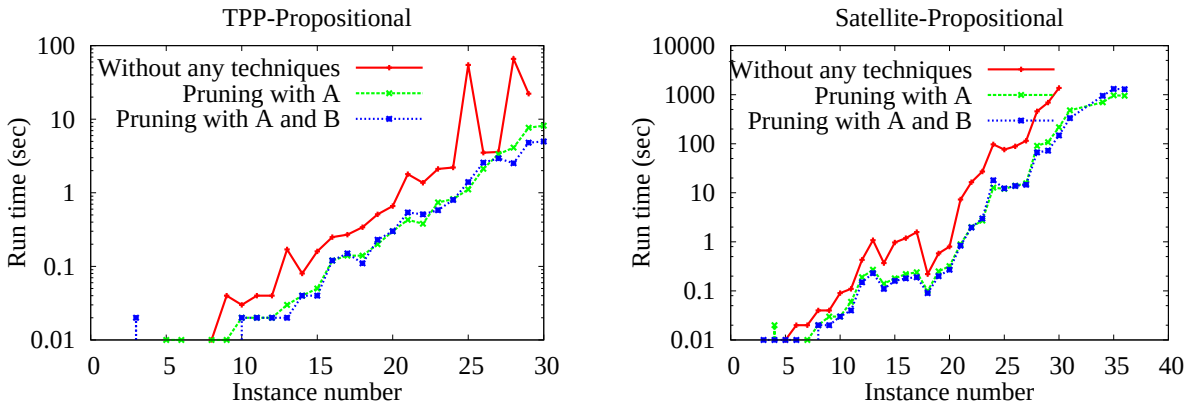


Figure 3.2: Sensitivity analysis of techniques for computing heuristic values. A: action-set partitioning; B: backward relevance analysis.

(such as RIFO [77]) use backward relevance analysis to reduce the search space, our current approach does not prune the search space, as actions irrelevant to $\mathcal{G}'$ may still be related to $\mathcal{G}$. Instead, we postpone the execution of those irrelevant actions in generating a relaxed planning graph, just like what we have done for actions not in the active partition.

Figure 3.2 demonstrates the effectiveness of the two techniques on the Satellite-Strips [57] and the TPP-Propositional [39] domains. Using h_{LM} and assuming **truck** as the partitioning attribute for TPP-Propositional and **satellite** for Satellite-Strips, the results show substantial saving in solution times when generating a relaxed plan after action partitioning. Without building the relaxed planning graph in a decomposed manner, some instances cannot be solved within 30 minutes of CPU time. On the other hand, pruning with backward relevance analysis has performance gain when solving some instances, such as the three largest instances in TPP-Propositional (P28, P29, and P30), although it is not effective for the Satellite-Strips domain.

In short, our solution-preserving technique uses the locality of actions to reduce the cost of generating relaxed plans. By using two open lists, it avoids getting stuck in dead-ends or incurring significantly deeper searches.

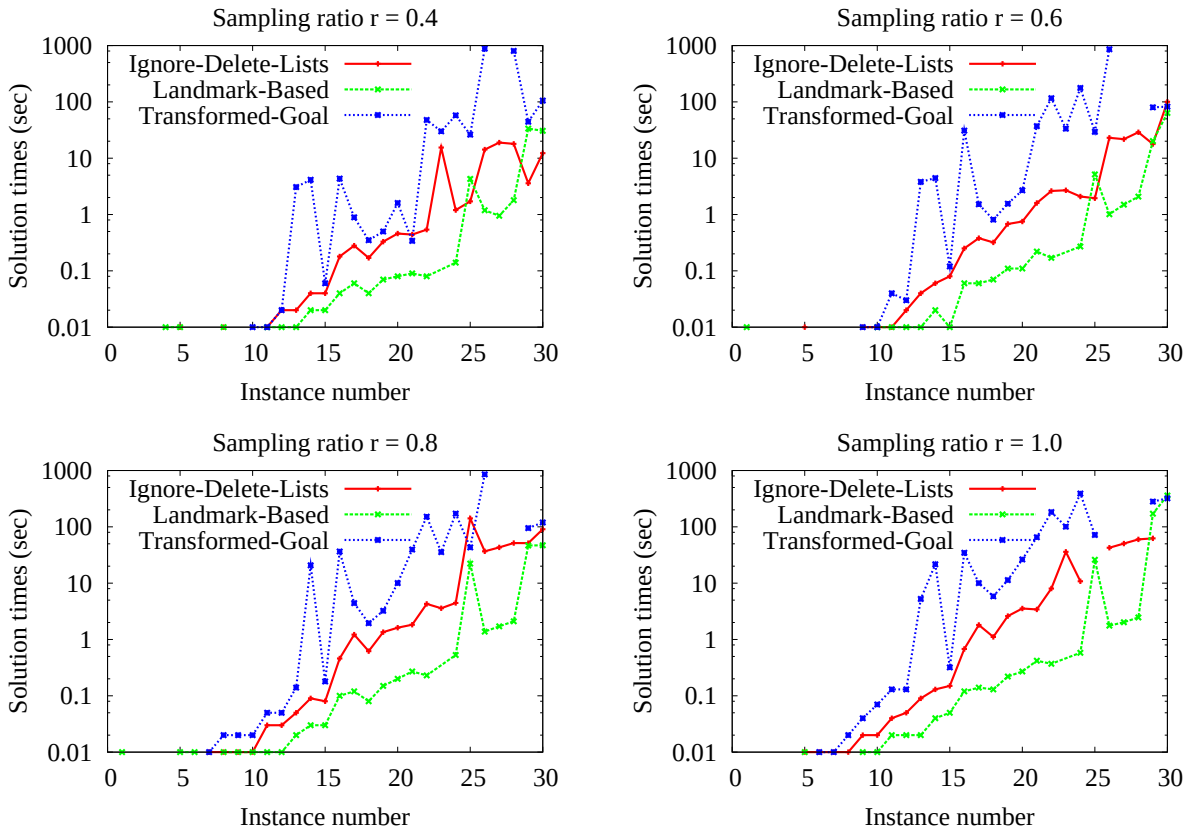


Figure 3.3: Generalization of the three guidance heuristics: $h_{\mathcal{G}}$, $h_{\mathcal{G}_T}$, and h_{LM} in the TPP-Propositional domain. For each instance, the best partitioning attribute is identified from Figure 2.2.

3.6 Evaluation of Top-Down State-Space Constraint Resolution

In this section, we evaluate our search algorithm on the TPP-Propositional domain, using the best partitioning attribute found in Figure 2.2. Figure 3.3 plots the results using the three guidance heuristics on four sampling ratios r.

The first 10 instances of this domain are not interesting because each can be solved in 0.1 sec or less. For most of the remaining instances, the landmark-based heuristic h_{LM} is the best irrespective of r. However, for TPP-Propositional-P25, the *ignore-delete-lists* heuristic $h_{\mathcal{G}}$ is the best when $r = 0.4$ or 0.6, but h_{LM} is the best when r is 0.8 or 1.0. For TPP-Propositional-P29, $h_{\mathcal{G}}$ is the best when $r = 0.4$, 0.6, or 1.0, but h_{LM} performs better when

$r = 0.8$. However, the difference in performance between the two heuristics is insignificant when $r = 0.6$ or 0.8. For TPP-Propositional-P30, h_{LM} is the best when $r \geq 0.6$. In short, the generalizability of h_{LM} is strong in this domain.

Next we empirically find the best guidance heuristics across four r values of every IPC-4, IPC-5, and IPC-6 propositional domain. In each domain, we select at least two instances, and for each instance, we identify the best partitioning attribute using Algorithm 1.

Table 3.1 shows that for most instances the best guidance heuristics are the same across all values of r. Among them, $h_{\mathcal{G}}$ is consistently the best for a majority of the instances.

For some instances, the best guidance heuristic when $r = 1.0$ is not the same as that found for one of the smaller r. For example, for TPP-P30 and Pegsol-P29 (*resp.* Pipesworld-P40, TPP-P29, and Scanalyzer-P29), the best heuristic when $r = 0.4$ (*resp.* $r = 0.8$) is not the same as that when $r = 1.0$. In these cases, the actual best heuristic can be generalized from a majority of the heuristics found for $r < 1.0$.

For some instances, using a heuristic other than the best does not lead to a significant degradation in solution time. For example, Elevator-P30 is solvable within 30.5 sec using the suboptimal h_{LM}; Openstacks-P29 can be solved by h_{LM} in 2.3 sec, which is very close to the best value of 2.1 sec.

For some other instances, using a suboptimal heuristic may render the problem unsolvable. For example, Algorithm 5 identifies h_{LM} as the best heuristic for Airport-P49 using $r = 0.4$ and 0.6 and Algorithm 4 identifies "airplane" at the best partitioning attribute, but the instance is unsolvable when $r = 1.0$. In this case, the instance can only be solved by using $h_{\mathcal{G}}$.

In sum, our results show that our proposed simplification and generalization approach is effective for finding the best guidance heuristics. By using h_{LM} and $h_{\mathcal{G}_T}$, we are now able to solve many more instances in the Promela, Satellite, Rovers, and Storage domains.

Table 3.1: The generalization results on all IPC-4, IPC-5, and IPC-6 propositional domains with respect to the best guidance heuristics. The second column shows the best partitioning attribute found using Algorithm 4. For each sampling ratio r, we show the result in the form (best guidance heuristic corresponding to the best attribute, solution time in sec). "ID," "TG," and "LM" denote, respectively, the guidance heuristics $h_{\mathcal{G}}$, $h_{\mathcal{G}_T}$, and h_{LM}. The heavily shaded instances cannot be solved by the heuristics generalized from $r = 0.4$ and $r = 0.6$, whereas the lightly shaded instances have minor inconsistencies whose behavior is explained in the text.

Instance	Attribute	$r = 0.4$	$r = 0.6$	$r = 0.8$	$r = 1.0$
		IPC-4			
Airport-P48	AIRPLANE	(LM,2.6)	(LM,2.6)	(LM,4.0)	(LM,5.2)
Airport-P49	AIRPLANE	(LM,2.1)	(LM,3.0)	(ID,10.9)	(ID,15.5)
Pipesworld-P40	PRODUCT	(ID,1.3)	(ID,0.6)	Unsolved	(ID,3.1)
Pipesworld-P41	AREA	(ID,0.1)	(ID,0.1)	(ID,0.1)	(ID,0.3)
PromelaOpt-P48	NONE	(TG,11.2)	(TG,54.2)	(TG,201.4)	(TG,555.8)
PromelaOptDP-P13	NONE	(ID,0.5)	(ID,1.6)	(ID,6.2)	(ID,17.1)
PromelaPhi-P48	NONE	(TG,3.4)	(TG,17.4)	(TG,75.1)	(TG,250.9)
PromelaPhiDP-48	NONE	(ID,0.6)	(ID,2.3)	(ID,13.5)	(ID,48.5)
PSRsmall-P49	NONE	(ID,0.4)	(ID,0.4)	(ID,0.4)	(ID,0.4)
PSRsmall-P50	NONE	(ID,0.1)	(ID,0.1)	(ID,0.1)	(ID,0.1)
PSRmiddle-P49	NONE	(ID,18.7)	(ID,13.1)	(ID,3.1)	(ID,3.8)
PSRmiddle-P50	NONE	(ID,3.2)	(ID,4.1)	(ID,3.3)	(ID,4.0)
PSRlarge-P26	NONE	(ID,50.4)	(ID,82.5)	(ID,96.3)	(ID,157.0)
PSRlarge-P27	NONE	(ID,48.0)	(ID,62.5)	(ID,133.1)	(ID,211.0)
Satellite-P35	SATELLITE	(LM,1137.5)	(LM,594.0)	(LM,995.3)	(LM,1313.6)
Satellite-P36	SATELLITE	(LM,521.7)	(LM,525.9)	(LM,708.1)	(LM,1212.0)
		IPC-5			
Openstacks-P29	PRODUCT	(ID,32.4)	(ID,33.6)	(ID,45.0)	(ID,35.0)
Openstacks-P30	PRODUCT	(ID,7.2)	(ID,7.2)	(ID,7.1)	(ID,7.2)
Pathways-P29	LEVEL	(ID,1.5)	(ID,1.5)	(ID,1.5)	(ID,1.5)
Pathways-P30	LEVEL	(LM,0.5)	(LM,0.5)	(LM,0.5)	(LM,0.5)
Pipesworld-P31	AREA	(ID,19.0)	(ID,7.2)	(ID,7.8)	(ID,5.1)
Pipesworld-P41	AREA	(ID,0.1)	(ID,0.1)	(ID,0.1)	(ID,0.4)
Rovers-P39	WAYPOINT	(LM,6.2)	(LM,5.6)	(LM,12.0)	(LM,18.6)
Rovers-P40	WAYPOINT	(LM,21.2)	(LM,11.6)	(LM,14.1)	(LM,137.9)
Storage-P29	CRATE	(TG,0.1)	(TG,1.6)	(TG,1.3)	(TG,1.9)
Storage-P30	CRATE	(TG,0.3)	(TG,2.5)	(TG,1.7)	(TG,2.3)
TPP-P29	(TRUCK,GOODS)	(ID,3.6)	(ID,17.9)	(LM,46.5)	(ID,62.4)
TPP-P30	(TRUCK,GOODS)	(ID,12.3)	(LM,63.6)	(LM,47.1)	(LM,359.8)
Trucks-P09	LOCATION	(ID,0.1)	(ID,0.1)	(ID,0.2)	(ID,48.6)
Trucks-P14	LOCATION	(ID,0.2)	(ID,0.3)	(ID,0.3)	(ID,281.0)
		IPC-6			
Elevator-P29	COUNT	(LM,2.2)	(LM,3.6)	(LM,7.0)	(LM,9.3)
Elevator-P30	PASSENGER	(LM,1.5)	(LM,3.5)	(LM,11.7)	(ID,21.6)
Openstacks-P29	ORDER	(LM,1.9)	(LM,1.9)	(LM,2.0)	(ID,2.1)
Openstacks-P30	ORDER	(ID,1.5)	(ID,1.5)	(ID,1.8)	(ID,1.7)
Parcprinter-P29	SHEET_T	(ID,0.1)	(ID,0.1)	(ID,0.1)	(ID,0.1)
Parcprinter-P30	SHEET_T	(ID,0.2)	(ID,0.1)	(ID,0.1)	(ID,0.1)
Pegsol-P29	LOCATION	(LM,0.1)	(ID,0.1)	(ID,37.2)	(ID,0.2)
Pegsol-P30	NONE	(ID,0.1)	(ID,0.1)	(ID,11.8)	(ID,15.0)
Scanalyzer-P29	NONE	(ID,0.1)	(ID,0.1)	(LM,454.7)	(ID,197.2)
Scanalyzer-P30	CAR	(ID,0.1)	(ID,0.1)	(ID,106.3)	(ID,88.9)
Sokoban-P29	NONE	(ID,14.7)	(ID,11.7)	(ID,15.3)	(ID,460.4)
Sokoban-P30	(STONE,DIR.)	(ID,0.3)	(ID,0.3)	(ID,5.6)	(ID,122.9)
Transport-P27	PACKAGE	(LM,3.9)	(LM,7.7)	(ID,101.6)	Unsolved
Transport-P28	VEHICLE	(LM,0.6)	(LM,1.2)	(LM,2.5)	(LM,7.0)
Woodworking-P29	PART	(ID,0.6)	(ID,1.1)	(ID,2.5)	(ID,2.7)
Woodworking-P30	PART	(ID,1.2)	(ID,1.8)	(ID,2.7)	(ID,5.5)

3.7 Summary

In this chapter, we have presented our solution for the resolution step in parallel decomposition. In Section 3.1, we evaluate possible variants of a resolution framework and identify the top-down state-space approach is suitable for out applications.

Section 3.2 and Section 3.3 illustrate and describe our proposed search algorithm given then identified partitioning attribute. Our strategy for reducing the branching factor is to focus our search within the selected partition, observing that the solution plan complies with some locality oftentimes.

Heuristic function is important in guiding the search to problem goal regardless of parallel decomposition. In Section 3.4, we introduce three heuristic functions used in this work and present the procedure for identifying the best one. We also survey relevant work on learning search strategy in planning.

We have presented the design of our subproblem solver in Section 3.5. Again, we exploit the locality for reducing state evaluation cost. Integrated with the resolution step, we demonstrate the generalizability of the best guidance heuristics in Section 3.6.

Chapter 4

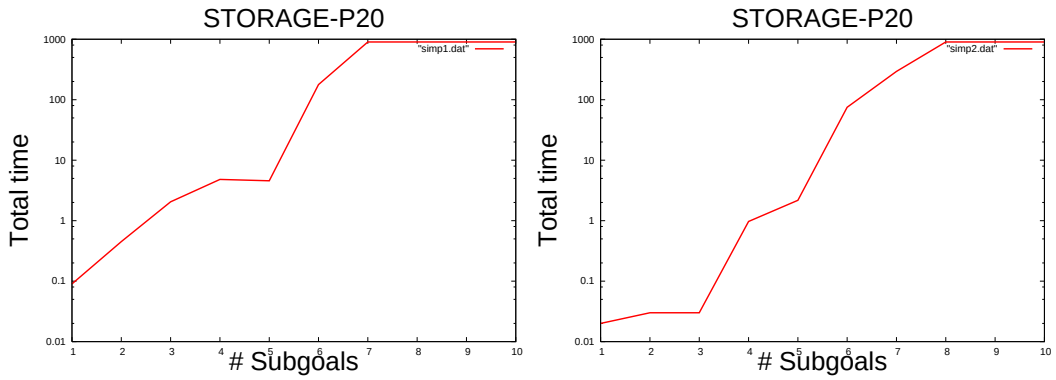
Application on PDDL2.2 Domains

In this chapter, we evaluate our planner SGPlan_6 for solving IPC-4, IPC-5, and IPC-6 benchmarks specified in PDDL2.2 and compare its performance with that of four other planners. In Section 4.1, we present the detailed implementation of SGPlan_6 for solving sequential and temporal domains defined in PDDL. In Section 4.2, we apply SGPlan_6 and other selected planners to all the benchmarks to see how many instances can be solved. We discuss the results by domains and study the run time performance of each planner. We also identify domains suitable for SGPlan_6 by analyzing exit paths. In Section 4.3, we focus on solution quality in evaluation and show SGPlan_6 can either solve more instances or improve solution quality with more time given. Section 4.4 presents the results of identifying heuristic function first and shows the results are not sensitive to the ordering of two generalization steps.

4.1 Implementation Details of SGPlan_6

The pre-processor of SGPlan_6 was designed based on the parser of Metric-FF [52] and SGPlan_4 [16] and can handle STRIPS, ADL, numerical, and temporal planning from early versions of PDDL, as well as derived predicates and timed initial literals in PDDL2.2 [27].

The handling of STRIPS problems with a typed representation has been discussed in Chapters 2 and 3. For the small number of untyped benchmarks (such as IPC4-PSR-Small) whose attributes cannot be identified and action partitioning cannot be done, SGPlan_6 employs the usual best-first search without parallel decomposition. For these benchmarks,



a) Subgoal removal b) Symmetric objects removal

Figure 4.1: Solving times of Storage-Propositional-P20 corresponding to different number of subgoals using two different simplification methods. In left figure, instances with more than five subgoals are too difficult to solve whereas instances with five or less subgoals are too relaxed and therefore useless. In right figure, the solving time increases smoothly as the size of simplified problem grows.

without attributes, we cannot generate simplified instances using Algorithm 3. To find the best guidance heuristics through simplification, we generate the simplified instances by removing subgoals as follows. Given a sampling ratio r, we generate a simplified instance by randomly sampling $r \cdot |\mathcal{G}|$ subgoals from the $|\mathcal{G}|$ subgoals in $\mathcal{T}$ and by keeping the other definitions unchanged. Removing some subgoals can lead to a strictly relaxed problem because the action set is not reduced. Hence, it is unnecessary to detect unreachable facts after simplification.

However, it is unclear the relaxed instance is structurally similar to and easier than the original one. Figure 4.1a) plots the times of solving Storage-Propositional-P20 with respect to different number of subgoals. It shows that the subgoal removal approach generates either much easier instances with different structural properties or instances with similar difficulty and properties. As a result, this subgoal sampling technique is used only when simplification using symmetric objects is not applicable.

In handling ADL features, we first compile the features away and transform the original problem into a STRIPS problem. This means that an ADL action with conditional effects or disjunctive preconditions is converted into several STRIPS actions. However, the instanti-

ated STRIPS actions are less efficient for computing symmetry when there are quantifiers in the definition. To this end, SGPlan$_6$ works on an ADL action schema directly in identifying statically symmetric objects but still works on STRIPS actions in the other components of action partitioning (Algorithm 1).

In incorporating derived predicates in the propositional part into a state-space search, we have to evaluate the values of those derived variables by the corresponding derived rule for each newly generated state (*successor()* in Algorithm 7). Since there may be ADL features in these rules, we treat the rules as actions and apply the same compilation technique. Similarly, the derived-predicate schema plays the same role as action schema in computing symmetric objects and in generating simplified instances. Although we cannot reduce the branching factor by partitioning derived predicates, the complexity of computing the guidance heuristic depends on the number of derived rules, as all derived rules need to be tested in expanding a relaxed graph. Since the state-evaluation cost may be reduced by partitioning the derived rules, we merge all derived rules into the action set before action partitioning.

In the same way as SGPlan$_4$ [16], SGPlan$_6$ processes a temporal planning problem in two stages: sequential planning and scheduling. In sequential planning, it converts each durative action in a temporal task into an atomic action by removing all time specifiers on conditions, effects, and duration information. The scheduling stage then annotates the sequential plan with suitable timestamps by applying the critical-path algorithm [25]. As a result, it can only produce temporal plans that have the corresponding sequential solutions. It is possible to make temporal planning more complete by either employing a plan-space search or by searching in an extended state space with timestamps, scheduled effects, and conditions [30, 22]. Because both approaches will result in higher complexity and since the focus of this research is not on temporal planning, we have decided to continue using the two-stage framework.

To handle timed initial literals in a temporal problem, we first find a sequential plan through back-tracking. When evaluating the guidance heuristic, the subproblem solver also

considers the timed initial literals and computes using the critical-path algorithm the earliest planning time to achieve each fact in the relaxed graph. It then prunes those branches that cannot comply with all timed initial literals. In addition, it generates the relaxed plan by following a critical path in order to have a consistent heuristic estimate.

We rely on the implementation of Metric-FF for dealing with continuous variables and metric values. The new addition is the handling of action cost introduced in IPC-6.

In addition to finding feasible solutions, we have implemented an anytime option for optimizing quality, including plan cost, makespan, and plan length. The option is implemented by a series of independent weighted A^* search with non-decreasing weights, each guided by heuristic values of the form $wg+h$, where g is the plan quality, h is the guidance value in the original best-first search, and w is the relative weight between f and g. Unlike Metric-FF, our h does not estimate the cost of the relaxed plan no matter how g is defined. Each search also prunes branches that will not improve the best known solution. When the quality metric involves makespan, we re-use the results of the critical-path algorithm to compute the lower bound of makespan.

The first search is a best-first search which is equivalent to $w = \infty$. The remaining values of w are $[8, 4, 2, 1, 1, \ldots]$. As discussed before, the guidance heuristic may not be able to discriminate different states with uniform heuristic value. Therefore, using smaller w at the beginning will result in a search more like a breadth-first one which is hopeless in solving large problems. Refer to [83] for more implementation considerations and an empirical study.

SGPlan_6 was implemented by a combination of C and Python. The search engine was written in C, using inputs that include the partitioning attribute, the selected guidance heuristic, the sampling ratio, and the weight of the anytime search. The anytime version as well as Algorithms 4 and 6 were implemented in Python that invoke the search engine for solving the original problem and possibly the simplified instances.

4.2 Experimental Results and Analysis

In this section, we compare SGPlan_6 to four planning systems: LAMA [82],[1] Temporal Fast Downward (TFD) [30],[2] SGPlan_4 [16],[3] and the baseline system that shares the same subproblem solver and state-space search with SGPlan_6 but does not have action partitioning and always uses the *ignore-delete-lists* guidance heuristic $h_{\mathcal{G}}$ in its best-first search. Note that LAMA can only do propositional planning, and TFD can only do temporal planning possibly with numerical variables. In our experiments, we test the baseline planner for sensitivity with respect to SGPlan_6.

All runs were carried out on an AMD Athlon MP2800 PC with RedHat Linux AS4 and 4-Gbyte main memory. In solving one instance by SGPlan_6, we set the memory and time limits to, respectively, 2 GB and 30 minutes, the latter including the times for solving the simplified instances as well as the original problem. We further limit the time for solving a simplified instance by SGPlan_6 to be, respectively, 1 min, 3 min, and 6 min when the sampling ratio r is 0.4, 0.6, and 0.8. When a simplified instance cannot be solved within the time limit, we give up and try another attribute/heuristic function.

For the following experimental results, we partition the benchmark domains into sequential domains and temporal domains and compare SGPlan_6 with available planners.

4.2.1 Sequential Domains

Table 4.1 summarizes the performance of the four systems on all the IPC-4, IPC-5, and IPC-6 propositional domains. It shows that SGPlan_6 can solve the maximum number of instances. In general, there is no domain that SGPlan_6 performs much worse, and the simplification and generalization framework can increase the robustness of planning.

[1] Winner of the IPC-6 sequential satisficing track: http://www.informatik.uni-freiburg.de/šrichter/software/lama.tar.gz

[2] Runner-up of the IPC-6 temporal satisficing track: http://tfd.informatik.uni-freiburg.de/downloads/version-0.2.1/tfd-src-0.2.1.tgz

[3] First Prize Winner of the IPC-4 Suboptimal Temporal Metric Track and Second Prize Winner of the IPC-4 Suboptimal Propositional Track, http://wah.cse.cuhk.edu.hk/wah/programs/SGPlan/SP4/sgplan42.zip

Table 4.1: Summary on the number of sequential planning instances solved by the four planners. The second column shows the number of instances in that domain. The solver solving the most instances in each domain is highlighted.

Domain	Total	$SGPlan_6$	LAMA	$SGPlan_4$	Baseline
IPC-4					
Airport-Nontemp	50	50	35	44	34
Pipesworld-NotankNontemp	50	37	44	50	33
Promela-OpticalTele	48	48	2	14	1
Promela-OpticalTeleDP	48	48	2	19	37
Promela-Philosophers	48	48	12	29	12
Promela-PhilosophersDP	48	48	29	48	48
PSR-Small	50	50	50	47	50
PSR-Middle	50	50	50	50	50
PSR-Large	50	31	30	11	29
Satellite-Strips	36	35	32	30	23
IPC-5					
Openstacks-Propositional	30	30	30	30	30
Pathways-Propositional	30	29	30	30	11
Pipesworld-Propositional	50	24	38	34	24
Rovers-Propositional	40	36	40	37	23
Storage-Propositional	30	30	19	14	18
TPP-Propositional	30	29	30	30	26
Trucks-Propositional	30	11	15	9	10
IPC-6					
Elevators-Sequential	30	30	26	30	30
Openstacks-Sequential	30	30	30	30	30
Parcprinter-Sequential	30	30	20	30	30
Pegsol-Sequential	30	30	29	16	30
Scanalyzer-Sequential	30	30	30	28	27
Sokoban-Sequential	30	28	19	1	29
Transport-Sequential	30	30	30	27	13
Woodworking-Sequential	30	30	28	30	11
Overall	958	872	700	718	659

In the following, we analyze the domains where $SGPlan_6$ performs well. By comparing the results with the baseline planner, we show good performance is due to either a good guidance heuristic or an action partitioning such that exit paths consist of actions from the same partition most of times. As all guidance heuristics used in Chapter 3 are based on *ignore-delete-lists* heuristic, we study the exit path with respect to *ignore-delete-lists* heuristic. Note that finding the exit distance for given state is as hard as finding the solution plan to $\mathcal{G}$ (PSPACE-complete for STRIPS domains [55]. Hence, there does not exist a simple algorithm which can automatically analyzes exit paths for all domains.

We assume that a state S has an optimal relaxed plan $\mathcal{P}$ starting from an action a. For analysis purpose, we use $\mathcal{P}$ associated with h^+ defined in Definition 8 instead of suboptimal $h_{\mathcal{G}}$ used in the implementation of SGPlan_6 and FF. We also define S' is the destination of an exit path of S.

For the Airport-Nontemporal domain, the exit path for most states can be constructed using actions associated with the same "airplane". An exit path involving two actions occurs only when one airplane blocks another airplane. We have found that this case rarely happens in the benchmark.

For the Promela-OpticalTele domains, it can be shown that the exit distance can be arbitrarily long for some states [54]. The exit path also involves actions from all partitions regardless of partitioning attribute selected. In this case, finding a more effective heuristic function is more important than attribute partitioning. SGPlan_6 performs much better than the other planners because the guidance heuristic based on transformed goals $\mathcal{G}_T$ is useful for this domain and it can be identified and generalized by solving simplified instances. Since it can be shown that optimal plans of this domain can be generated in polynomial time [49], there exists a heuristic function for solving this domain efficiently.

For the Promela-Philosophers domains, although it can be shown that the exit distance for any state is at most 32 [54], but the exit path still involves actions from all partitions regardless of partitioning attribute selected. Thus, although *ignore-delete-lists* heuristic performs better on these domains compared with Promela-OpticalTele domains, it is still inefficient as 32 is too large. Similar to Promela-OpticalTele domains, SGPlan_6 performs much better than the other planners because of the use of $\mathcal{G}_T$. Similar to Promela-OpticalTele, the problem is finding an efficient heuristic as it can also be shown that optimal plans of this domain can be generated in polynomial time [49].

For the Satellite-Strips domain, it can be shown that the exit distance is at most five for each state S [53]. In fact, the shortest exit path can be constructed by using actions associated with one "satellite" only by looking into the proof of bounded exit distance.

For the Pathway-Propositional domain, the problem of *ignore-delete-lists* heuristic function is that all molecules are always available once being generated. We never consume molecules in relaxed planning and $\mathcal{P}$ can underestimate the distance to goal. One can construct an arbitrarily long exit path by having a very long reaction chain and this exit path cannot be constructed from single action partition. The major reason that SGPlan_6 performs better than the baseline planner is the use of landmark-based heuristic h_{LM} which guides the search to one landmark at each time.

For the Storage-Propositional domain, one can also construct an arbitrarily long exit path when a is "move" action. As relaxed planning views all areas as clear, a can block the move of other crates in $\mathcal{P}$, whether those crates are loaded by the same hoist or not. Like Promela domains, $\mathcal{G}_T$ is useful for this case.

For the Rovers-Propositional domain, it can be shown that the exit distance of S is at most $3d + 2$ where d is the longest distance between two waypoints in this instance [53]. It means that *ignore-delete-lists* heuristic is inaccurate when d is large. Also, the exit path cannot be constructed from single action partition. The major reason that SGPlan_6 performs better than the baseline planner is the use of h_{LM} which restricts the search to one landmark first.

For the Transport-Sequential domain, we show that the exit path for each state can be constructed using actions associated with the same "vehicle". We enumerate all possible a by the corresponding action schema in the following.

If a is a "drive" action which changes vehicle location, the exit path for S must be associated with the same vehicle involved in a. It is clear that driving another vehicle has nothing to do with a. Note that the exit distance can be unbounded in this case just like Rovers-Propositional.

If a is a "pick-up" action, the delete effect of a only blocks the actions in $\mathcal{P}$ that want to use the same vehicle to pick up another package but do not have enough capacity. In order to get enough capacity, one has to drop the package picked up by a to the destination and

drives the "vehicle" to pick up another package. It is clear that these actions construct an exit path for S and involve the same "vehicle".

If a is a "drop" action, the exit distance is one because we only drop a package at its destination in $\mathcal{P}$ and restoring capacity has no negative impact on the remaining actions in $\mathcal{P}$.

For the Woodworking-Sequential domain, we show that the exit path for each state either can be constructed using actions associated with the same "part" or is trivial to construct.

If a is a "do-immersion-varnish", a "do-spray-varnish", a "do-glaze", a "do-grind", or a "do-plan" action, the exit path of S can be constructed by only using actions associated with the "part" involved in a. Once a "part" is available, the changes to that "part" has nothing to do with other "parts" by observing that the effects and conditions of those types of actions. Moreover, the exit distance of this case is bounded by a constant number as the optimal solution plan for each part is already bounded by a constant number.

If a is a "load-highspeed-saw" action, there must be an action a' in the optimal relaxed plan $\mathcal{P}$ to cut a board by the same saw. Otherwise, it is pointless to have a in the optimal relaxed plan $\mathcal{P}$ because a does not directly add a goal fact. If there is no other action besides a' in $\mathcal{P}$ to use the saw loaded by a, a itself constructs an exit path for S. Otherwise, (a, a') is an exit path for S to reduce heuristic value by 1 as we can remove (a, a') from $\mathcal{P}$ and add an "unload-highspeed-saw" action into $\mathcal{P}$ for the next saw use to construct a relaxed plan for S'.

Since the add effects of each "unload-highspeed-saw" action are already in the initial state, an "unload-highspeed-saw" action appears in $\mathcal{P}$ only when we just apply a "load-highspeed-saw" action. The discussion for "load-highspeed-saw" shows the exit distance of this case is only one.

For the other types of actions ("cut-board-small", "cut-board-medium", "cut-board-large", "do-saw-small", "do-saw-medium", and "do-saw-large"), the only delete effect is to invalidate that fact that this "part" is unused. The delete effect is to make sure we do

not cut/saw the same board twice. However, since we never cut/saw the same board twice in $\mathcal{P}$, the exit distance for S of this case is only one.

SGPlan$_6$ cannot solve most instances in PSR-large, Trucks-Propositional, and the two Pipesworld domains. In PSR-Large, there are many instantiated derived predicates (more than 100,000 in the largest task) when all the rules of the derived predicates are evaluated to set the derived variables in each step of its state-space search. In this case, action partitioning is not effective because the number of derived predicates is much larger than the number of actions, and the branching factor cannot be reduced by partitioning the derived predicates. In fact, finding an efficient heuristic is more critical because PSR in P [49]. As a result, the difference among SGPlan$_6$, LAMA, and the baseline system only comes from the guidance heuristic used.

For the two Pipesworld domains, there are two reasons why SGPlan$_6$ is not competitive. First, SGPlan$_6$ does not have a good guidance heuristic for both the global search and the subproblem solver for these domains: $h_{\mathcal{G}}$ is not informative because it ignores too much dependency information; $h_{\mathcal{G}_T}$ is ineffective because the transformed goal cannot be found; and h_{LM} may let the search can wander without good guidance. Exit paths can be very long and no partitioning attribute works well. In contrast, LAMA utilizes a more informative guidance heuristic that combines the *ignore-delete-lists* and the landmark-counting functions. It also detects more landmarks by constructing a domain transition graph. SGPlan$_4$ also uses a better guidance heuristic by integrating domain-specific constraints. By employing the incomplete but more greedy and memory-efficient enforced hill-climbing search [58], it solves more instances in the Notankage variant than LAMA but gets stuck in dead-ends more often in IPC5-Pipesworld.

Secondly, although there are symmetric oil derivatives when transporting derivatives to their destinations, reducing derivatives through symmetry does not simplify the resulting problem. The reason is that derivatives are also resources for moving other derivatives in the network. Reducing the number of derivatives below the resource limit effectively makes the

pipe disconnected and the simplified instance infeasible. On the other hand, if infeasibility caused by a disconnected pipe can be detected by *enhanced_reachability* in Algorithm 3, then the derivatives will virtually pass through that pipe, making the simplified instance trivial. In both cases, the simplified instances do not contribute to solving the original problem, even with a good guidance heuristic. In fact, the overhead of solving the simplified instances and the mis-identification of the best attribute/guidance heuristic make SGPlan_6 less competitive when compared to SGPlan_4 and LAMA.

SGPlan_6 also performs worse than LAMA in solving the Trucks-Propositional domain. The better performance of LAMA is attributed to the large overhead for solving the simplified instances in SGPlan_6 that usually identifies "NONE" as the partitioning attribute. Without partitioning, SGPlan_6's guidance heuristics are intrinsically sequential and are not suitable for minimizing completion times when encoding a planning instance with deadlines into a propositional setting. Moreover, the simplified instance constructed for delivering a smaller number of packages within a duration is not useful for solving the harder original problem that requires the delivery of a larger number of packages within the same duration. To better address the deadlines of tasks, the guidance heuristic in SGPlan_6 has to distinguish between durative and atomic actions in a relaxed plan. Further, tasks should be partitioned by their deadlines in order to focus on tasks with pending deadlines. Such an approach is beyond our action partitioning and landmark generation. In addition, the simplification approach needs to consider deadlines in goal definitions as well.

Figure 4.2 presents the search times of the four planners in solving IPC-4 sequential domains. Although solving the simplified instances took a significant fraction of search time in SGPlan_6, the identified partitioning attribute and heuristic function can help SGPlan_6 to achieve better scalability in Airport, Promela, and Satellite domains.

Figure 4.3 presents the search times of the four planners in solving IPC-5 sequential domains. LAMA is usually the most efficient planner on IPC-5 sequential domains. However, SGPlan_6 performs the best on the Storage domain as it is able to learn a suitable heuristic

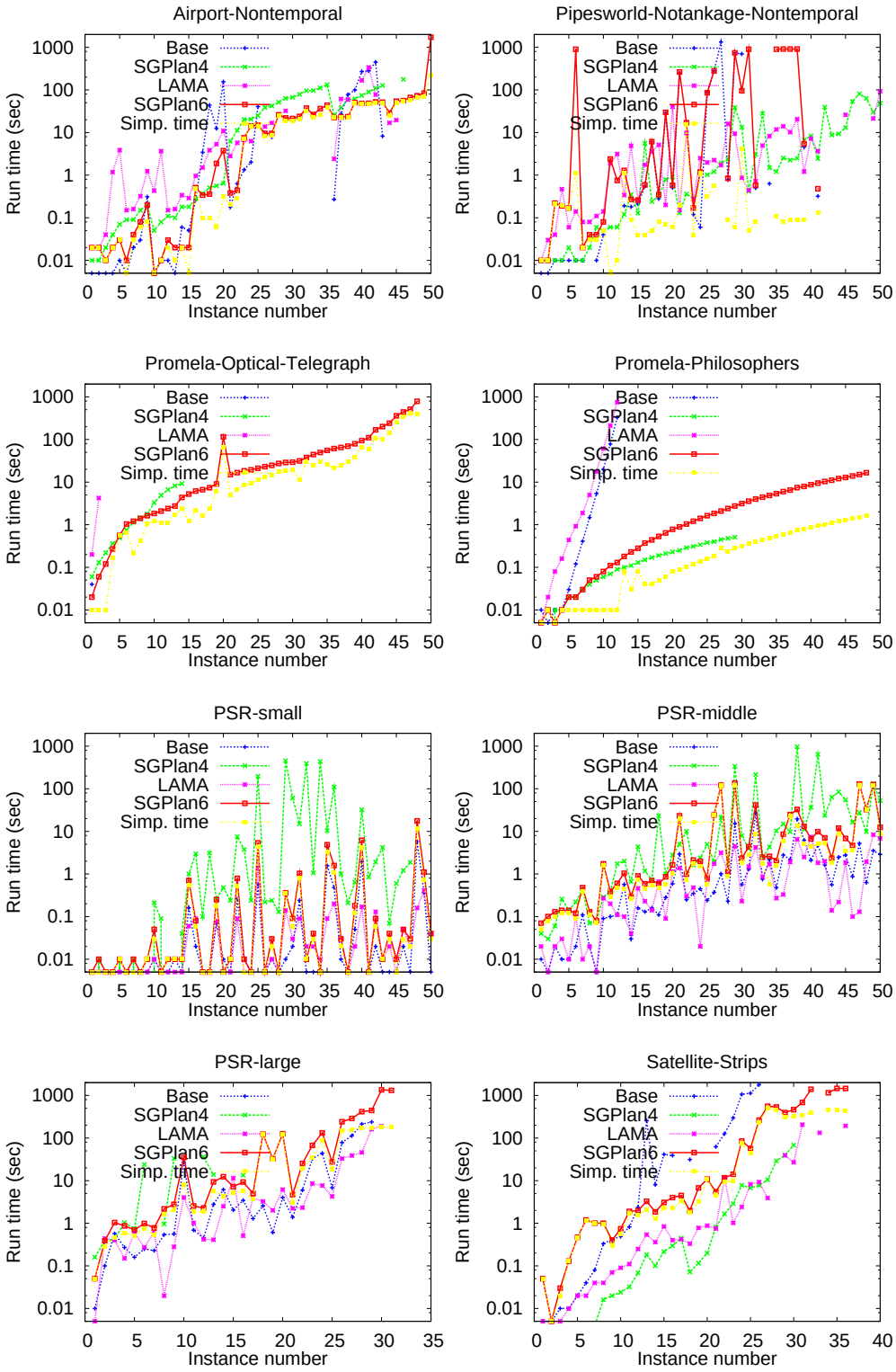


Figure 4.2: Comparison of search times for solving the sequential domains in IPC-4. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

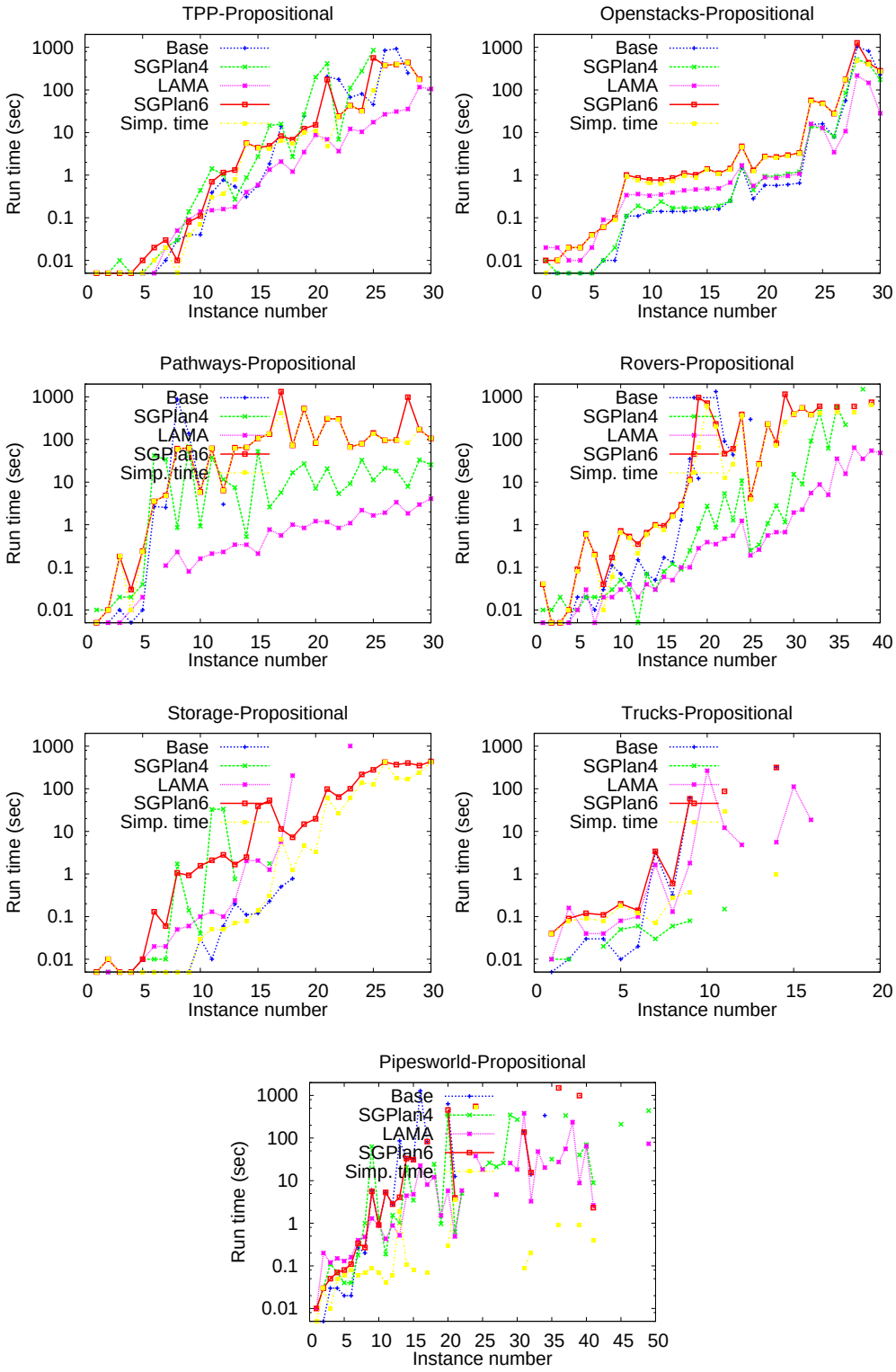


Figure 4.3: Comparison of search times for solving the sequential domains in IPC-5. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

function.

Figure 4.4 presents the search times of the four planners for solving those sequential domains in IPC-6. Some domains are easy for every planner, although no single planner outperforms the others in all the domains evaluated. The results also show that the times spent on solving the simplified instances in SGPlan_6 are significant in some domains, including the Elevators-Sequential, Openstacks-Sequential, Parcprinter-Sequential, and Woodworking-Sequential domains. In this case, simplification is ineffective for solving easy domains.

4.2.2 Temporal Domains

Table 4.2 shows that SGPlan_6 can solve the maximum number of instances in the temporal domains. Action partitioning is effective for handling the large number of actions in Airport-Temp, Airport-Temp-Deadline, and Satellite-Time. Also, the guidance heuristic based on $h_{\mathcal{G}_T}$ is effective for solving Storage-Time, whose problem structure is very similar to that of Storage-Propositional. The exit path analysis for their sequential counterparts in Chapter 4.2.1 can also be applied here to see why SGPlan_6 works well here.

SGPlan_6 outperforms the baseline system in solving Pipesworld-NotankDead because it can solve some larger instances through action partitioning and the *ignore-delete-lists* guidance heuristic. When compared to SGPlan_4, SGPlan_6 and the baseline system are better in dealing with deadlines represented by timed initial literals. Both generate relaxed plans along critical paths that provide better guidance. In contrast, the relaxed plans in SGPlan_4 favor less actions that do not usually lead to smaller makespan.

SGPlan_6 performs well in solving Transport-NumericTempo where action partitioning is effective for reducing the branching factor. Using the guidance heuristic based on landmarks, the state evaluation cost is further improved.

SGPlan_6 performs better than the other planners for Rovers-MetricTime, Pathways-MetricTime, TPP-MetricTime, and Sokoban-Temporal, although it still cannot solve most of the instances in these domains. For those it can solve, it mainly exploits simplification

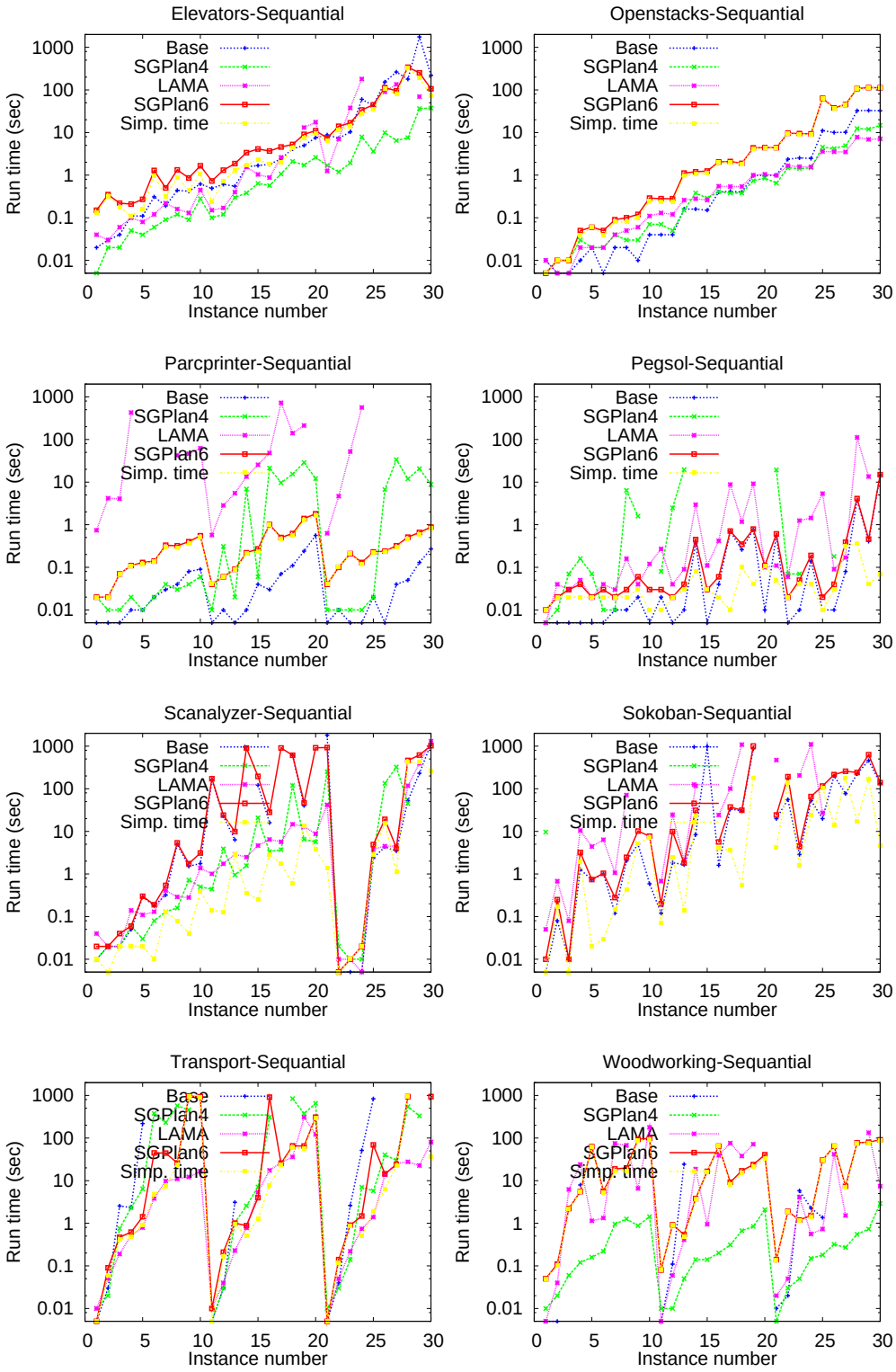


Figure 4.4: Comparison of search times for solving the sequential domains in IPC-6. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

Table 4.2: Summary on the number of (numeric-)temporal planning instances solved by the four planners. The second column shows the number of instances in each domain. The planner solving the most instances in each domain is highlighted. '−' means that the planner does not support the language features in the benchmark (TFD cannot handle timed initial literals, and the optimization metric is a combination of makespan and some other metric values).

Domain	Total	$SGPlan_6$	TFD	$SGPlan_4$	Baseline
IPC-4					
Airport-Temp	50	50	11	44	34
Airport-Temp-Deadline	50	50	-	43	34
Pipesworld-NotankTemp	50	37	20	50	33
Pipesworld-NotankDead	30	25	-	8	22
Satellite-Time	36	34	7	30	23
Satellite-TimeTW	36	7	-	24	6
Satellite-Complex	36	25	0	28	17
Satellite-ComplexTW	36	6	-	24	5
UMTS-Temporal	50	50	0	50	50
UMTS-TemporalTW	50	50	-	50	50
UMTS-FlawTemporal	50	50	0	50	50
UMTS-FlawTemporalTW	50	50	-	50	50
IPC-5					
Openstacks-Time	20	20	18	0	20
Openstacks-MetricTime	20	20	20	0	20
Pathways-MetricTime	30	5	-	0	2
Pipesworld-MetricTime	50	24	14	34	24
Rovers-MetricTime	40	16	13	0	12
Storage-Time	30	30	15	14	20
TPP-MetricTime	40	24	-	0	19
Trucks-Time	30	30	18	-	30
IPC-6					
Crewplannng-Temporal	30	29	29	0	29
Elevators-Temporal	30	29	17	0	29
Elevators-NumericTempo	30	30	23	30	30
Modeltrain-NumericTempo	30	15	1	0	6
Openstacks-Temporal	30	30	30	30	30
Openstacks-NumericTempo	30	30	30	30	30
Parcprinter-Temporal	30	30	13	0	27
Pegsol-Temporal	30	30	28	21	30
Sokoban-Temporal	30	18	12	9	17
Transport-NumericTempo	30	28	10	11	6
Woodworking-NumericTempo	30	29	28	27	27
Overall	1114	881	367	657	782

and generalization to find the best partitioning attributes and guidance heuristics.

SGPlan_6 is not effective for solving Satellite-Complex and Rovers-MetricTime. These are path-finding problems involving shortest paths specified by numerical constraints in the action preconditions or path costs in the quality metric. The *ignore-delete-lists* guidance heuristic in SGPlan_6 is not effective, as it underestimates the distance of paths in its relaxed planning graph when it overlooks the existence of multi-valued state variables (for instance, the location of a rover must be one of its waypoints) and produces arbitrarily long exit distance. The situation is worse than the sequential counterparts since exit paths usually involve multiple action partitions because of resource constraints. In fact, the path-finding or unbounded exit path problem can be solved by constructing a *domain transition graph* [48] of those multi-valued state variables that involves the connectivity of waypoints. A heuristic based on a domain-transition graph, like the *causal graph heuristic* in Fast Downward and TFD, would be more effective for these domains. Similarly, SGPlan_4 uses a special path optimization algorithm to minimize path lengths, which is more effective for handling time constraints represented by timed initial literals in Satellite-TimeTW and Satellite-ComplexTW.

SGPlan_6 is not effective when solving Pathways-MetricTime. In this domain, the branching factor grows as the search tree expands. Since there are less mutual exclusions on actions when compared to the other domains, it is harder to find actions that can be clustered by some attributes. Due to the large branching factor, a backward search or an incomplete local search would be more effective than our method based on a complete forward search.

For TPP-MetricTime, it is possible to solve this domain by the "truck" attribute and the landmark-based guidance heuristic. However, SGPlan_6 does not find this combination because it sequentially identifies the best attribute and then the best guidance heuristic based on the attribute found. It cannot find the best combination through enumeration due to the large overhead involved in solving simplified instances. To address this issue, one may consider the two identification problems together but prune some combinations by studying

the relation between the attribute and the guidance heuristic. For example, since landmarks in TPP-MetricTime are to store some goods, it will be redundant to pair attribute "goods" and the landmark-based heuristic.

$SGPlan_6$, however, does not perform well on the Pipesworld-NontankTemp and the Pipesworld-MetricTime domains when compared to $SGPlan_4$. For the same reasons explained earlier, the temporal and propositional versions of these domains have identical problem structures.

$SGPlan_6$ cannot solve many of the simplified instances in Modeltrain-NumericTemporal. The solver sometimes cannot solve a subproblem of arranging one train due to the large number of actions, and the locality observed does not support further partitioning. Similarly, the maze in Sokoban-Temporal is usually crowded with players and stones and a tight relationship among all planning objects, making it difficult to predict what player or stone will be involved when given the last action and a poor locality. Some instances also have enormous search-tree depth, rendering all our guidance heuristics ineffective. In the future, we plan to study these two domains further.

Figure 4.5 presents the search times of the four planners in solving IPC-4 temporal domains. For the Satellite and UMTS domains, we do not show the variants with time windows as TFD cannot solve them at all. The results are very similar to Table 4.2 in the sense that algorithm with better scalability solves more instances in general.

Figure 4.6 presents the search times of the four planners in solving IPC-5 temporal domains. $SGPlan_6$ has the best search time performance on all domains besides two Openstacks domains. Since the Openstacks domains are easy to solve, it is not worthy to solve simplified instances in $SGPlan_6$.

Figure 4.7 presents the search times of the four planners in solving IPC-6 temporal domains. For the Elevators and Openstacks domains, we only present their NumericTemporal variants, as their temporal variants are very similar. We do not show the Modeltrain domain, as only $SGPlan_6$ can solve more than five instances. Similar to Figure 4.4, there is no

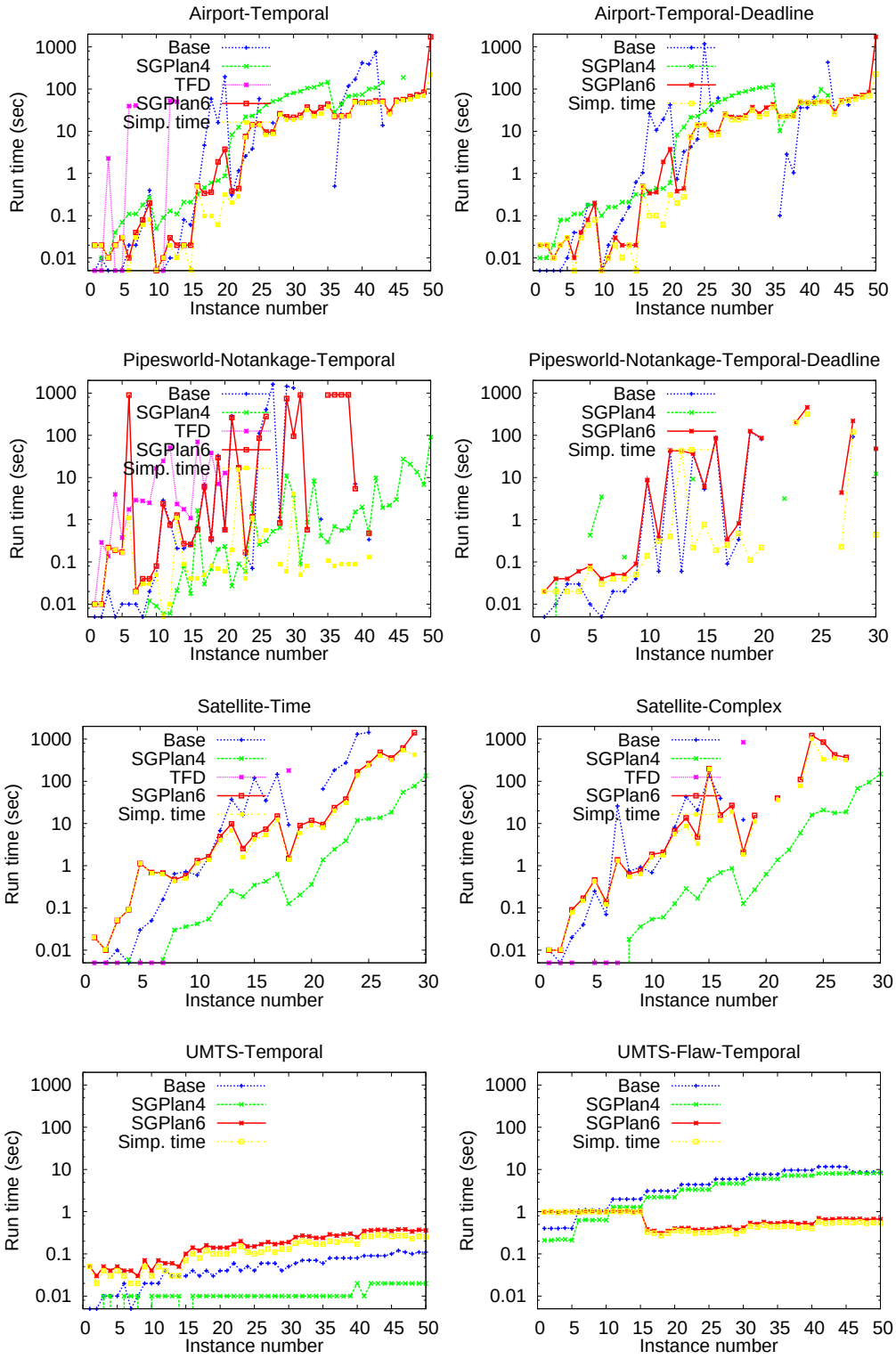


Figure 4.5: Comparison of search times for solving the temporal domains in IPC-4. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

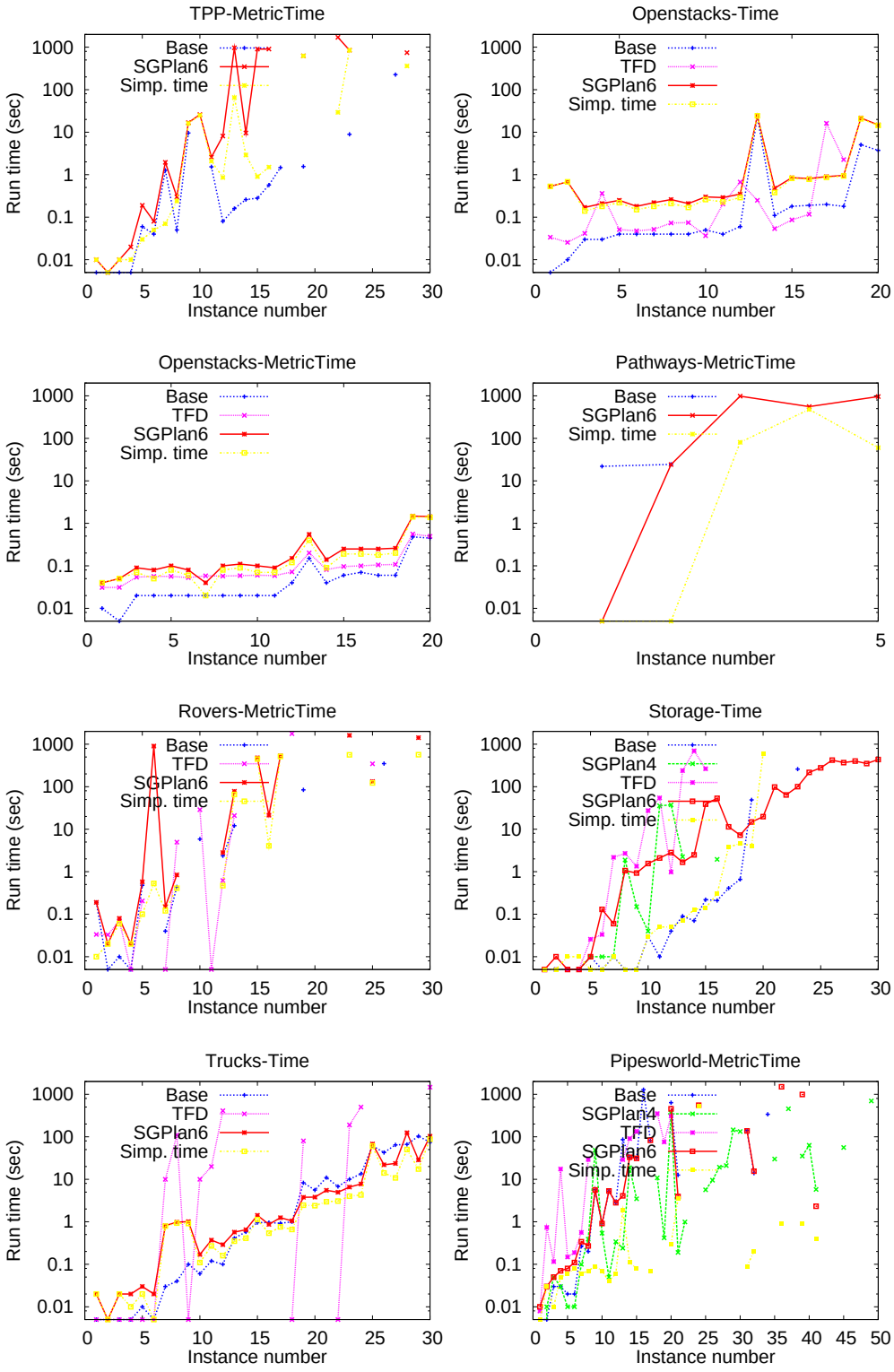


Figure 4.6: Comparison of search times for solving the temporal domains in IPC-5. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

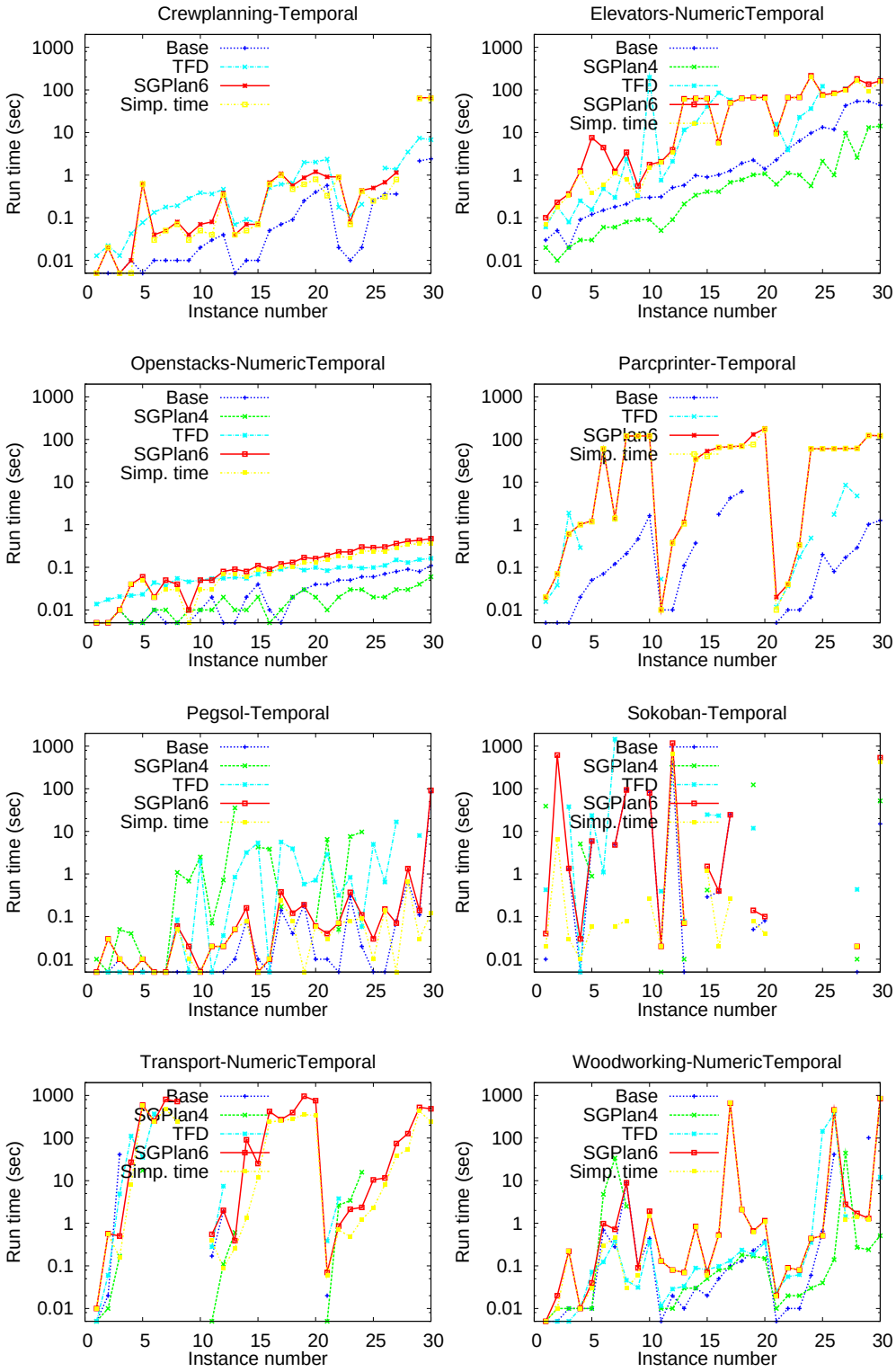


Figure 4.7: Comparison of search times for solving the temporal domains in IPC-6. The line "Simp. time" plots the search times for solving those simplified instances in SGPlan_6.

single planner that outperforms the others in all the domains evaluated. Also, solving the simplified instances took a significant fraction of search time in $SGPlan_6$.

4.3 Quality and Time Trade-off

In this section, we evaluate the solution quality of $SGPlan_6$ with the anytime search implementation. We also compare the quality and time trade-off of $SGPlan_6$ with other top planners.

Figure 4.8 plots the smallest plan metric value achieved within 30 minutes of CPU time for each IPC-6 sequential instance. Since $SGPlan_4$ does not have an anytime search option, the metric value of the single solution found is presented. Table 4.3 further presents a score for each domain, where the score of a solution is the ratio between the metric of this solution to the best metric of the four planners. Each planner can score up to 30 for each domain, since there are 30 instances in each domain. $SGPlan_6$ achieves better quality when solving the Woodworking-Sequential domain, as action partitioning can help the search quickly reach a good plan. However, it was not able to improve the metric of Transport-Sequential with the anytime search after committing to the partitioning attribute in its solution. For the other domains, there is no significant difference in plan quality between $SGPlan_6$ and the best planner.

Table 4.3: Overall scores on IPC-6 sequential domains using the IPC-6 evaluation criterion. The best entry in each domain is highlighted.

Domain	$SGPlan_6$	LAMA	$SGPlan_4$	Baseline
Elevators-Sequential	23.61	24.42	24.08	25.27
Openstacks-Sequential	27.16	28.97	13.37	28.01
Parcprinter-Sequential	29.21	19.79	25.29	29.55
Pegsol-Sequential	27.65	28.65	12.02	27.61
Scanalyzer-Sequential	28.14	28.49	24.03	24.45
Sokoban-Sequential	26.86	18.63	0.69	28.24
Transport-Sequential	25.11	28.85	17.65	10.78
Woodworking-Sequential	29.79	23.76	27.17	10.33
Overall	217.55	201.56	144.31	184.24

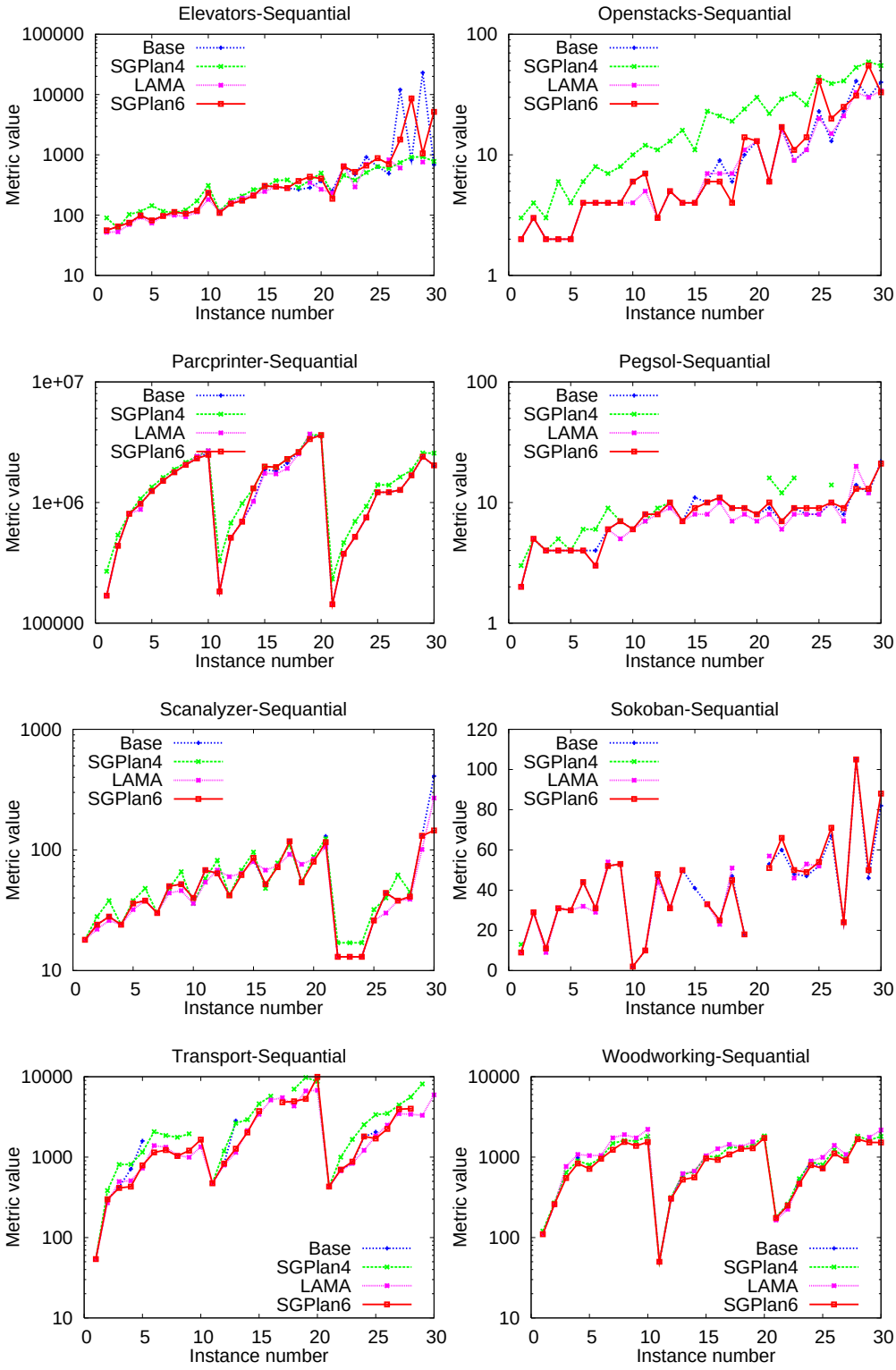


Figure 4.8: Comparison of the best plan metric values achieved within 30 minutes of CPU time when solving the IPC-6 sequential domains.

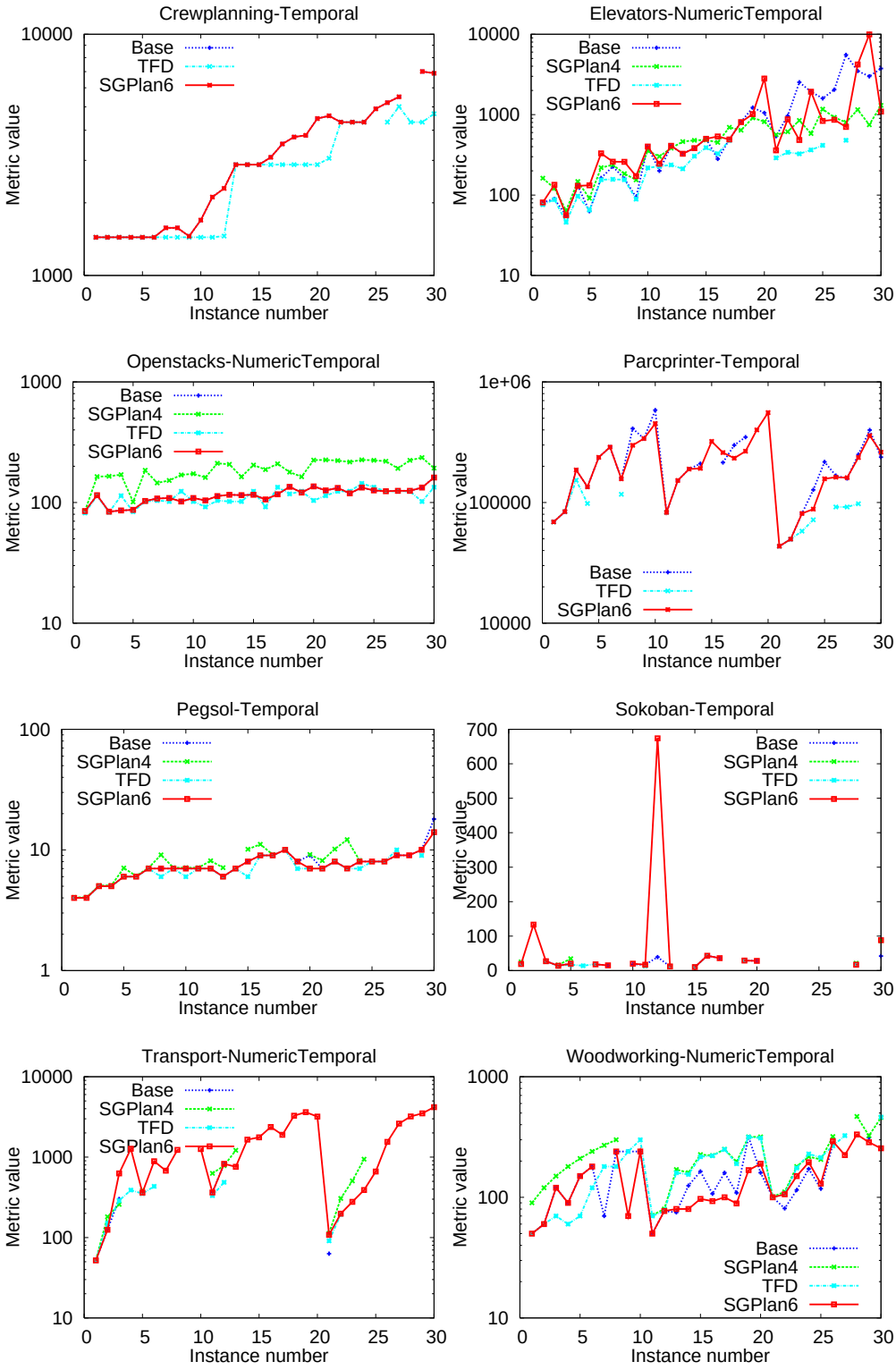


Figure 4.9: Comparison of the best makespans achieved within 30 minutes in solving the IPC-6 temporal domains.

Table 4.4: Overall results on IPC-6 temporal domains using the IPC-6 evaluation criterion. The planner with the best score in each domain is highlighted.

Domain	$SGPlan_6$	TFD	$SGPlan_4$	Baseline
Crewplannng-Temporal	25.61	29.00	0	25.61
Elevators-Temporal	23.58	16.38	0	19.49
Elevators-NumericTempo	19.05	22.71	20.94	19.53
Modeltrain-NumericTempo	14.52	0.96	0	5.91
Openstacks-Temporal	24.53	28.89	18.49	27.86
Openstacks-NumericTempo	28.23	29.19	17.93	28.23
Parcprinter-Temporal	27.11	13.00	0	22.67
Pegsol-Temporal	29.07	27.90	18.17	28.63
Sokoban-Temporal	16.04	11.86	7.25	16.60
Transport-NumericTempo	26.24	9.42	7.52	5.35
Woodworking-NumericTempo	26.15	19.92	15.59	22.38
Overall	260.14	209.23	105.90	222.26

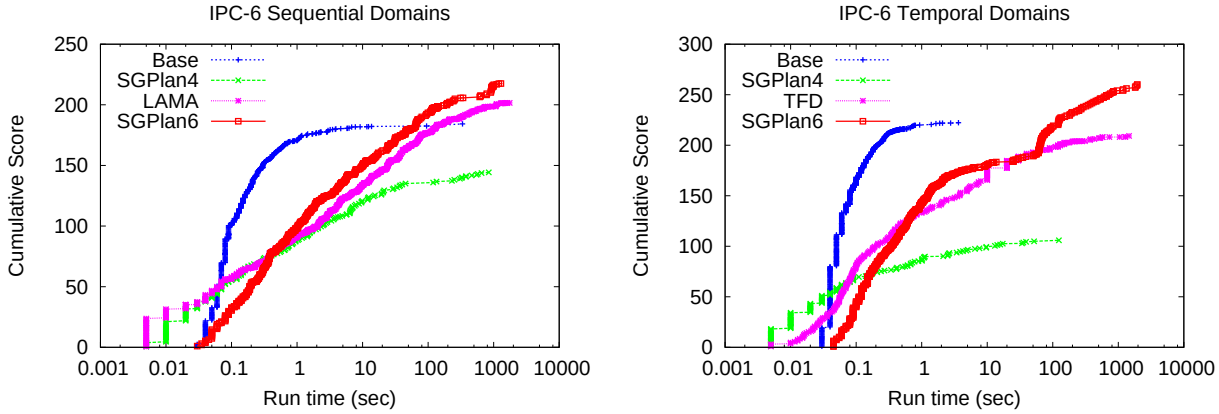


Figure 4.10: Cumulative IPC-6 evaluation score over time.

Figure 4.9 presents the best makespans achieved within 30 minutes of CPU time for all selected IPC-6 temporal instances. Table 4.4 further presents the scores for the IPC-6 temporal domains. $SGPlan_6$ can achieve better quality in solving Woodworking-NumericTempo because it can afford more enumeration in finding better plans for the simplified instances. On the other hand, TFD can achieve better quality on Crewplannng-Temporal and Elevators-NumericTempo because it searches in the temporal state space rather than converting an instance into a sequential one. For the other domains, no planner is the best in terms of both the number of solved instances and makespan.

So far we have been evaluating performance of each planner by setting maximal CPU

time to be 30 minutes. It is interesting to compare performance with different maximal run times, especially to see which algorithm will be better if we have more CPU resource.

Figure 4.10 enumerates maximal run time and plots cumulative IPC-6 evaluation score over time on all IPC-6 domains. It shows that SGPlan_6 becomes the best planner on IPC-6 sequential domains when the maximal run time is greater than one minute and becomes the best planner on temporal domains when the maximal run time is greater than two minutes. When the given run time is too small, the baseline system performs the best as SGPlan_6 has the overhead for solving simplified instances.

Further, the performance score of the baseline system and SGPlan_4 converges quickly. When their search strategies and heuristics are not powerful enough for some domains, their computational complexity grows exponentially and they will not benefit much from more run time. The scalability of SGPlan_6 is consistently better than LAMA and TFD after the maximal run time is greater than one second.

In fact, many problems due to misbehaved generalization become solvable when our computational resource is unlimited. One can either spend more time in solving larger simplified instances or even completely enumerate every combination of partitioning attribute and heuristic function. Therefore, the framework of SGPlan_6 is more like a long-term solution as we will have more computational power in the future.

4.4 Results of Identifying the Best Heuristic Function First

As we cannot afford to completely enumerate all possible combinations of partitioning attribute and heuristic function, our proposed approach is a greedy search which identifies the best partitioning attribute first and then identifies the best heuristic function given the best partitioning attribute. In this section, we evaluate a different ordering which starts from identifying the best heuristic function. The approach here is to apply Algorithm 6 assuming

the partitioning attribute is the attribute with the largest action-locality (AL) value. All other experiment setup including the scheduling of simplified instances is not changed.

Table 4.5 compares the results of two different orderings. In terms of number of instances solved, two orderings produce similar results and converge to the same combination on most solved instances. Identifying the best partitioning attribute first will solve more instances on sequential domains whereas identifying the best heuristic function first will solve more instances on temporal domains.

The reason why we have similar results for both orderings is because our initial guess for both partitioning attribute and heuristic function is the correct guess for many domains. Even if one generalization step is more crucial to run-time performance than the other step, one can still identify the best combination by taking more time in solving simplified instances. As a result, there is a discrepancy when the initial guess is wrong and the following simplification and generalization cannot recognize the correct answer.

For the TPP-MetricTime domain, using the landmark-based heuristic function is critical. With the initial guess of partitioning attribute, the landmark-based heuristic function outperforms the other heuristic functions more often in solving simplified instances. Hence, we can solve more instances for this domain with a different ordering.

On the other hand, using "waypoint" as the partitioning attribute is important for solving the Rovers-Propositional domain. However, the initial guess for partitioning attribute is "camera" since it has the highest AL value. This leads to a suboptimal heuristic function (landmark-based heuristic function) for larger instances. The simplification and generalization in the second stage cannot identify "waypoint" as the partitioning attribute with this suboptimal heuristic function as well.

Table 4.5: Summary on the number of planning instances solved by identifying partitioning attribute first and identifying heuristic function first. We highlight the entries where two orderings solve different number of instances for that domain.

Sequential Domains			Temporal Domains		
Domain	Attribute First	Heuristic First	Domain	Attribute First	Heuristic First
IPC-4					
Airport-Nontemp	50	50	Airport-Temp	50	50
Pipesworld-NotankNontemp	37	37	Airport-Temp-Deadline	50	50
Promela-OpticalTele	48	48	Pipesworld-NotankTemp	37	37
Promela-OpticalTeleDP	48	48	Pipesworld-NotankDead	25	25
Promela-Philosophers	48	48	Satellite-Time	34	34
Promela-PhilosophersDP	48	48	Satellite-TimeTW	7	7
PSR-Small	50	50	Satellite-Complex	25	25
PSR-Middle	50	50	Satellite-ComplexTW	6	6
PSR-Large	31	31	UMTS-Temporal	50	50
Satellite-Strips	35	35	UMTS-TemporalTW	50	50
			UMTS-FlawTemporal	50	50
			UMTS-FlawTemporalTW	50	50
IPC-5					
Openstacks-Propositional	30	30	Openstacks-Time	20	20
Pathways-Propositional	29	27	Openstacks-MetricTime	20	20
Pipesworld-Propositional	24	24	Pathways-MetricTime	5	5
Rovers-Propositional	36	30	Pipesworld-MetricTime	24	24
Storage-Propositional	30	30	Rovers-MetricTime	16	16
TPP-Propositional	29	28	Storage-Time	30	30
Trucks-Propositional	11	11	TPP-MetricTime	24	33
			Trucks-Time	30	30
IPC-6					
Elevators-Sequential	30	30	Crewplannng-Temporal	29	29
Openstacks-Sequential	30	30	Elevators-Temporal	29	30
Parcprinter-Sequential	30	30	Elevators-NumericTempo	30	28
Pegsol-Sequential	30	30	Modeltrain-NumericTempo	15	15
Scanalyzer-Sequential	30	28	Openstacks-Temporal	30	30
Sokoban-Sequential	28	28	Openstacks-NumericTempo	30	30
Transport-Sequential	30	30	Parcprinter-Temporal	30	30
Woodworking-Sequential	30	29	Pegsol-Temporal	30	30
			Sokoban-Temporal	18	21
			Transport-NumericTempo	28	27
			Woodworking-NumericTemp	29	30
Overall	872	860	Overall	881	892

4.5 Summary

In this chapter, we have presented the implementation details of $SGPlan_6$ its application to PDDL2.2 domains. We have shown that $SGPlan_6$ is able to solve more planning instances than other top planners. We also discuss the performance for each domain and explain where our approach works. In particular, we analyze exit paths of domains with respect to *ignore-delete-list* heuristic to measure the quality of heuristic and the impact of action partitioning. Many problems are not solved by $SGPlan_6$ due to a lack of powerful heuristic function. On the other hand, our generalization procedure increases the coverage of the solver at the cost of solving simplified instances.

$SGPlan_6$ outperforms other top planners when solution quality is taken into account. Although we limit our search space by parallel decomposition, the solution quality is not degraded with the use of anytime best-first search. The time-quality trade-off also shows the scalability of $SGPlan_6$ is better than other top planners as long as the benchmark set is not too trivial.

We have also demonstrated the experimental results are not very sensitive to the ordering of two generalization steps. The reason is that our initial guess of partitioning attribute and heuristic function works on a majority of instances. Our generalization procedure can also lead to the best selection when the initial guess is wrong.

Chapter 5

Conclusions and Future Work

In this chapter we summarize the conclusions of this dissertation and present some of the future directions to refine the proposed approach and to extend the same parallel decomposition work further for a wider range of planning problems.

5.1 Summary

In this dissertation, we have developed a parallel decomposition method to address the complexity of solving automated planning problems. We have identified that planners based on heuristic search are not efficient when there exist long exit paths in search topologies as the heuristic function stops to provide guidance there. By observing the locality that nearby actions in a solution plan or in an exit path are usually from the same partition, we employ action partitioning in parallel decomposition to reduce the branching factor in finding exit paths and state-evaluation cost.

For a given planning instance, we select the best partitioning attribute and the guidance heuristic by solving some simplified instances before applying them to the original instance. Unlike previous work that symmetry was used to eliminate redundant search efforts, we identify and utilize symmetry in order to generate simplified instances for learning performance of partitioning attribute and heuristic function. By utilizing locality and symmetry, we proposed a parallel decomposition framework to improve heuristic search in planning.

Our results show that our planner, SGPlan$_6$, can solve more instances than other top planners in IPC-4, IPC-5, and IPC-6 because of better scalability. The experimental results

also imply that the performance gain of $SGPlan_6$ is significant when solving a problem whose exit paths consist of actions from the same partition. On the other hand, studying a more efficient heuristic function would be more crucial when the exit paths are too long and unlikely stay in the same action partition.

5.2 Future Work

There are some problems along with this research direction. First, we restrict in this research that each planning problem can only have one action partitioning for all states. As a result, we can only use action graphs to abstract the overall structure and search for an attribute that works for most of states. In fact, Chapter 4.2.1 shows exit paths can be very different for different states. It is unnecessarily restricted to have single partitioning attribute covering all the exit paths. One can dynamically change the partitioning based on state or trajectory. We plan to extend our analysis for exit paths for this problem. One possibility is to utilize *casual graph* [23] or RIFO [77] for finding actions sufficient for specific exit path. The problem here is how to prune this sufficient action set into a necessary one for reducing branching factor.

Second, we need more understanding about where a heuristic function works. Although automatically proving that a heuristic function is inefficient for some domain is difficult, it might be possible to empirically probe the search topologies. One idea is to monitor the length of exit paths in solving simplified instances. If we find the search progress is consistently slower and the search gets into plateaus many times for some heuristic function, we should stop trying this guidance heuristic and focus on the others.

Third, we want to capture essential problem structures relevant to exit paths in simplification. The analysis in Chapter 4.2.1 shows many properties of exit paths can be generalized from one instance to another. We plan to refine the simplification and generalization approach to better exploit the generalizability of exit paths as exit paths are crucial to both

partitioning and heuristic search.

Last, we plan to apply the proposed parallel decomposition work to PDDL3 [40] domains. Although static symmetry and action graphs in our current approach can capture localized problem structures, they are not realistic for problems with trajectory constraints in PDDL3, since these problem may become trivial when trajectory constraints need not be satisfied. In addition, there are both soft trajectory constraints and goals in PDDL3 domains that are not considered in the optimization of action graphs and static symmetry. We will also need to study other heuristic functions and techniques for complexity reduction in those PDDL3 domains.

References

[1] E. Amir and B. Engelhardt. Factored planning. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 929–935, 2003.

[2] F. Bacchus. The AIPS'00 planning competition. *AI Magazine*, 22(3):47–56, 2001.

[3] F. Bacchus and Q. Yang. Downward refinement and the efficiency of hierarchical problem solving. *Artificial Intelligence*, 71(1):43–100, 1994.

[4] C. Bäckström and B. Nebel. Complexity results for SAS^+ planning. *Computational Intelligence*, 11:625–656, 1995.

[5] J. Benton, S. Kambhampati, and M. B. Do. YochanPS: PDDL3 simple preferences as partial satisfaction planning. In *Proc. Fifth Int'l Planning Competition*. Int'l Conf. on Automated Planning and Scheduling, June 2006.

[6] A. L. Blum and M. L. Furst. Fast planning through planning graph analysis. *Artificial Intelligence*, 90:281–300, 1997.

[7] B. Bonet and H. Geffner. Planning as heuristic search: New results. In *Proc. European Conf. on Planning*, pages 360–372, 1999.

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

[9] A. Botea, M. Müller, and J. Schaeffer. Fast planning with iterative macros. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 1828–1833, 2007.

[10] R. I. Brafman and C. Domshlak. Factored planning: How, when, and when not. In *Proc. AAAI Conf. on Artificial Intelligence*, 2006.

[11] R. I. Brafman and C. Domshlak. From one to many: Planning for loosely coupled multi-agent systems. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 28–35, 2008.

[12] C. J. C. Burges and B. Schölkopf. Improving the accuracy and speed of support vector machines. In *NIPS*, pages 375–381, 1996.

[13] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69(1-2):165–204, 1994.

[14] Y. Chen, R. Huang, and W. Zhang. Fast planning by search in domain transition graph. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 886–891, 2008.

[15] Y. Chen, R. Huang, X. Zhao, and W. Zhang. Long-distance mutual exclusion for planning. *Artificial Intelligence*, 173(2):365–391, 2009.

[16] Y. X. Chen, B. W. Wah, and C. W. Hsu. Temporal planning using subgoal partitioning and resolution in SGPlan. *J. of Artificial Intelligence Research*, 26:323–369, August 2006.

[17] A. Coles, M. Fox, D. Long, and A. Smith. Planning with respect to an existing schedule of events. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 81–88, 2007.

[18] A. Coles and K. A. Smith. Marvin: A heuristic search planner with online macro-action learning. *J. Artificial Intelligence Research*, 28:119–156, 2007.

[19] R. Dechter. *Constraint Processing*. Morgan Kaufman, 2003.

[20] G. DeJong. Toward robust real-world inference: A new perspective on explanation-based learning. In *ECML*, pages 102–113, 2006.

[21] Y. Dimopoulos, A. Gerevini, P. Haslum, and A. Saetti. The benchmark domains of the deterministic part of IPC-5. In *Proc. Fifth Int'l Planning Competition*. Int'l Conf. on Automated Planning and Scheduling, June 2006. `http://zeus.ing.unibs.it/ipc-5/`.

[22] M. B. Do and S. Kambhampati. Sapa: A multi-objective metric temporal planner. *J. Artificial Intelligence Research*, 20:155–194, 2003.

[23] C. Domshlak and R. I. Brafman. Structure and complexity in planning with unary operators. In *Proc. Int'l Conf. on AI Planning and Scheduling (AIPS)*, pages 34–43, 2002.

[24] S. Edelkamp. Planning with pattern databases. In *Proc. European Conf. on Planning*, pages 13–24, 2001.

[25] S. Edelkamp. Taming numbers and duration in the model checking integrated planning system. *J. of Artificial Intelligence Research*, 20:195–238, 2003.

[26] S. Edelkamp and M. Helmert. Exhibiting knowledge in planning problems to minimize state encoding length. In *Proc. European Conf. on Planning*, pages 135–147, 1999.

[27] S. Edelkamp and J. Hoffmann. PDDL2.2: The language for the classic part of the 4th International Planning Competition. Technical report, Tech. Rep. 195, Institut für Informatik, Freiburg, Germany, January 2004.

[28] S. Edelkamp, S. Jabbar, and M. Nazih. Large-scale optimal PDDL3 planning with MIPS-XXL. In *Proc. Fifth Int'l Planning Competition*. Int'l Conf. on Automated Planning and Scheduling, June 2006.

[29] K. Erol, J. A. Hendler, and D. S. Nau. HTN planning: Complexity and expressivity. In *Proc. National Conf. on Artificial Intelligence*, pages 1123–1128, 1994.

[30] P. Eyerich, R. Mattmüller, and G. Röger. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, 2009.

[31] R. E. Fikes and N. J. Nilsson. STRIPS: A new approach to the application of theorem proving to problem solving. *Artificial Intelligence*, 2(3 & 4):189–208, 1971.

[32] M. Fox and D. Long. Utilizing automatically inferred invariants in graph construction and search. In *Proc. Int'l Conf. on AI Planning and Scheduling (AIPS)*, pages 102–111, 2000.

[33] M. Fox and D. Long. PDDL+ Level 5: An extension to PDDL2.1 for modeling planning domains with continuous time-dependent effects. *Tech. Rep., Dept. of Computer Science, Univ. of Durham, Durhan, UK*, September 2001.

[34] M. Fox and D. Long. Extending the exploitation of symmetries in planning. In *Proc. Int'l Conf. on AI Planning and Scheduling (AIPS)*, pages 83–91, 2002.

[35] M. Fox, D. Long, and J. Porteous. Abstraction-based action ordering in planning. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 1220–1225, 2005.

[36] Maria Fox and Derek Long. The automatic inference of state invariants in TIM. *J. of Artificial Intelligence Research*, 9:367–421, 1998.

[37] J. Frank, P. Cheeseman, and J. Stutz. Where grsavity fails: Local search topology. *J. of Artificial Intelligence Research*, 7:249–281, 1997.

[38] E. C. Freuder and P. D. Hubbe. A disjunctive decomposition control schema for constraint satisfaction. In *Principles and Practice of Constraint Programming, 1st International Workshop, PPCP'93*, Newport, Rhode Island, 1993.

[39] A. Gerevini, P. Haslum, D. Long, A. Saetti, and Y. Dimopoulos. Deterministic planning in the fifth international planning competition: PDDL3 and experimental evaluation of the planners. *Artificial Intelligence*, 173(5-6):619–668, 2009.

[40] A. Gerevini and D. Long. Plan constraints and preferences for PDDL3. Technical report, R.T. 2005-08-07, Dept. of Electronics for Automation, U. of Brescia, Brescia, Italy, August 2005.

[41] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs in LPG. *J. Artificial Intelligence Research*, 20:239–290, 2003.

[42] A. Gerevini, A. Saetti, and I. Serina. An approach to temporal planning and scheduling in domains with predictable exogenous events. *J. of Artificial Intelligence Research*, 25:187–231, 2006.

[43] S. Grandcolas and C. Pain-Barre. Filtering, decomposition and search space reduction for optimal sequential planning. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 993–998, 2007.

[44] C. Guestrin and G. J. Gordon. Distributed planning in hierarchical factored MDPs. In *Proc. Conf. on Uncertainty in Artificial Intelligence*, pages 197–206, 2002.

[45] S. Hanks and D. S. Weld. A domain-independent algorithm for plan adaptation. *J. of Artificial Intelligence Research*, 2:319–360, 1995.

[46] P. Haslum. Improving heuristics through relaxed search - an analysis of TP4 and HSP*a in the 2004 planning competition. *J. of Artificial Intelligence Research*, 25:233–267, 2006.

[47] P. Haslum, A. Botea, M. Helmert, B. Bonet, and S. Koenig. Domain-independent construction of pattern database heuristics for cost-optimal planning. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 1007–1012, 2007.

[48] M. Helmert. The Fast Downward planning system. *J. of Artificial Intelligence Research*, 26:191–246, 2006.

[49] M. Helmert. New complexity results for classical planning benchmarks. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 52–62, 2006.

[50] M. Helmert, M. Do, and I. Refandis. Deterministic part, 6th International Planning Competition. `http://ipc.informatik.uni-freiburg.de/`, 2008.

[51] M. Helmert, P. Haslum, and J. Hoffmann. Flexible abstraction heuristics for optimal sequential planning. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, 2007.

[52] J. Hoffmann. The Metric-FF planning system: Translating "Ignoring Delete Lists" to numeric state variables. *J. of Artificial Intelligence Research*, 20:291–341, 2003.

[53] J. Hoffmann. Where "Ignoring Delete Lists" works: Local search topology in planning benchmarks. Technical report, Tech. Rep. 185, Albert-Ludwigs-Universität, Institut für Informatik, Freiburg, Germany, 2003.

[54] J. Hoffmann. Where "Ignoring Delete Lists" works: Local search topology in planning benchmarks. *J. of Artificial Intelligence Research*, 24:685–758, 2005.

[55] J. Hoffmann. Analyzing search topology without running any search: On the connection between causal graphs and h^+. *J. of Artificial Intelligence Research*, 41:155–229, 2011.

[56] J. Hoffmann. Everything you always wanted to know about planning - (but were afraid to ask). In *Proc. Annual German Conf. on Artificial Intelligence*, pages 1–13, 2011.

[57] J. Hoffmann, S. Edelkamp, S. Thiébaux, R. Englert, F. S. Liporace, and S. Trüg. Engineering benchmarks for planning: the domains used in the deterministic part of IPC-4. *J. of Artificial Intelligence Research*, 26:453–541, 2006.

[58] J. Hoffmann and B. Nebel. The FF planning system: Fast plan generation through heuristic search. *J. of Artificial Intelligence Research*, 14:253–302, 2001.

[59] J. Hoffmann and B. Nebel. RIFO revisited: Detecting relaxed irrelevance. In *European Conf. on Planning*, pages 325–336, 2001.

[60] J. Hoffmann, J. Porteous, and L. Sebastia. Ordered landmarks in planning. *J. of Artificial Intelligence Research*, 22:215–278, 2004.

[61] J. N. Hooker. Planning and scheduling by logic-based Benders decomposition. *Operations Research*, 55:588–602, 2007.

[62] C. W. Hsu, Y. X. Chen, and B. W. Wah. Subgoal ordering and granularity control for incremental planning. *Int'l J. of Artificial Intelligence Tools*, 16(4):707–723, August 2007.

[63] C. W. Hsu, B. W. Wah, R. Huang, and Y. X. Chen. Constraint partitioning for solving planning problems with trajectory constraints and goal preferences. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 1924–1929. AAAI, January 2007.

[64] S. Kambhampati and J. A. Hendler. A validation-structure-based theory of plan modification and reuse. *Artificial Intelligence*, 55(2):193–258, 1992.

[65] H. A. Kautz and B. Selman. Planning as satisfiability. In *Proc. European Conf. on Artificial Intelligence*, pages 359–363, 1992.

[66] H. A. Kautz, B. Selman, and J. Hoffmann. SatPlan: Planning as satisfiability. In *Proc. Fifth Int'l Planning Competition*. Proc. Int'l Conf. on Automated Planning and Scheduling, June 2006.

[67] R. Khardon. Learning action strategies for planning domains. *Artificial Intelligence*, 113(1-2):125–148, 1999.

[68] C. A. Knoblock. Automatically generating abstractions for planning. *Artificial Intelligence*, 68(2):243–302, 1994.

[69] J. Koehler and J. Hoffmann. On reasonable and forced goal ordering and their use in an agenda-driven planning algorithm. *J. of Artificial Intelligence Research*, 12:339–386, 2000.

[70] J. Koehler, B. Nebel, J. Hoffmann, and Y. Dimopoulos. Extending planning graphs to an ADL subset. In *Proc. European Conf. on Planning*, pages 273–285, 1997.

[71] A. L. Lansky. Localized planning with action-based constraints. *Artificial Intelligence*, 98(1-2):49–136, 1998.

[72] S. H. Lim, L.-L. Wang, and G. DeJong. Integrating prior domain knowledge into discriminative learning using automatic model construction and phantom examples. *Pattern Recognition*, 42(12):3231–3240, 2009.

[73] D. Long and M. Fox. Efficient implementation of the plan graph in STAN. *J. of Artificial Intelligence Research*, 10:87–115, 1999.

[74] D. V. McDermott. Using regression-match graphs to control search in planning. *Artificial Intelligence*, 109(1-2):111–159, 1999.

[75] D. V. McDermott. Estimated-regression planning for interactions with web services. In *Proc. Int'l Conf. on AI Planning and Scheduling (AIPS)*, pages 204–211, 2002.

[76] N. Meuleau, M. Hauskrecht, K. Kim, L. Peshkin, L. Pack Kaelbling, T. Dean, and C. Boutilier. Solving very large weakly coupled Markov decision processes. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 165–172, 1998.

[77] B. Nebel, Y. Dimopoulos, and J. Koehler. Ignoring irrelevant facts and operators in plan generation. In *Proc. European Conf. on Planning*, pages 338–350, 1997.

[78] M. A. H. Newton, J. Levine, M. Fox, and D. Long. Learning macro-actions for arbitrary planners and domains. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 256–263, 2007.

[79] X. Nguyen and S. Kambhampati. Reviving partial order planning. In *Proc. Int'l Joint Conf. on Artificial Intelligence*, pages 459–466, 2001.

[80] R. S. Nigenda, X. Nguyen, and S. Kambhampati. AltAlt: Combining the advantages of Graphplan and heuristic state search. Technical report, Arizona State University, 2000.

[81] J. Penberethy and D. Weld. UCPOP: A sound, complete, partial order planner for ADL. In *Proc. 3rd Int'l Conf. on Principles of Knowledge Representation and Reasoning*, pages 103–114. KR Inc., 1992.

[82] S. Richter, M. Helmert, and M. Westphal. Landmarks revisited. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 975–982, 2008.

[83] S. Richter, J. T. Thayer, and W. Ruml. The joy of forgetting: Faster anytime search via restarting. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 137–144, 2010.

[84] S. Srivastava, N. Immerman, and S. Zilberstein. Learning generalized plans using abstract counting. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 991–997, 2008.

[85] V. Vidal. A lookahead strategy for heuristic search planning. In *Proc. Int'l Conf. on Automated Planning and Scheduling*, pages 150–160, 2004.

[86] V. Vidal and H. Geffner. CPT: An optimal temporal POCL planner based on constraint programming. In *Proc. 4th Int'l Planning Competition*, pages 59–60. ICAPS, 2004.

[87] D. S. Weld, C. R. Anderson, and D. E. Smith. Extending Graphplan to handle uncertainty & sensing actions. In *Proc. AAAI Conf. on Artificial Intelligence*, pages 897–904, 1998.

[88] S. Yoon, A. Fern, and R. Givan. Learning control knowledge for forward search planning. *J. of Machine Learning Research*, 9(Apr):683–718, 2008.

[89] H. L. S. Younes and R. G. Simmons. VHPOP: Versatile heuristic partial order planner. *J. Artificial Intelligence Research*, 20:405–430, 2003.

[90] R. M. Young, M. E. Pollack, and J. D. Moore. Decomposition and causality in partial-order planning. In *Proc. Int'l Conf. on AI Planning and Scheduling (AIPS)*, pages 188–194, 1994.