Kernel Functions for Case-Based Planning

Ivan Serina
Free University of Bozen-Bolzano
Viale Ratisbona, 16
I-39042 Bressanone, Italy
ivan.serina@unibz.it

Technical Report

Abstract

Case-based planning can take advantage of former problem-solving experiences by storing in a plan library previously generated plans that can be reused to solve similar planning problems in the future. Although comparative worst-case complexity analyses of plan generation and reuse techniques reveal that it is not possible to achieve provable efficiency gain of reuse over generation, we show that the case-based planning approach can be an effective alternative to plan generation when *similar* reuse candidates can be chosen.

In this paper we describe an innovative case-based planning system, called OAKPLAN, which can efficiently retrieve planning cases from plan libraries containing more than ten thousand cases, choose heuristically a suitable candidate and adapt it to provide a good quality solution plan which is similar to the one retrieved from the case library.

Given a planning problem we encode it as a compact graph structure, that we call *Planning Encoding Graph*, which gives us a detailed description of the *topology* of the planning problem. By using this graph representation, we examine an approximate retrieval procedure based on *kernel functions* that effectively match planning instances, achieving extremely good performance in standard benchmark domains.

The experimental results point out the effect of the case base size and the importance of accurate matching functions for global system performance. Overall, we show that OAKPLAN is competitive with state-of-the-art plan generation systems in terms of number of problems solved, CPU time, plan difference values and plan quality when cases similar to the current planning problem are available in the plan library.

Finally when the adaptation phase is concluded and the solution plan is available to the agents, a new planning case corresponding to the current planning problem and its solution plan can be either inserted into the plan library or discarded.[1]

Keywords: Case-Based Planning, Domain-Independent Planning, Case-Based Reasoning, Heuristic Search for Planning, Kernel Functions.

[1]This technical report is an extended version of a paper published on the *Artificial Intelligence Journal*: I. Serina. Kernel Functions for Case-Based Planning. *Artificial Intelligence*, 174: 1369 – 1406, 2010.

Contents

1 Introduction **3**

2 Preliminaries **5**
2.1 Many-sorted logic 5
2.2 Planning formalism 6
2.3 Graphs 7
2.4 Kernel Functions for Labeled Graphs 9

3 Case-Based Planning **12**
3.1 Plan Retrieval 14
3.1.1 Object Matching 15
3.1.2 Screening Procedure 19
3.1.3 Kernel Functions for Object Matching 21
3.2 Plan Evaluation Phase 26
3.3 Plan Adaptation 28
3.3.1 Local Search Techniques for Plan Adaptation 29
3.4 Plan Revision 33
3.5 Case Base Update 33

4 Experimental Results **34**
4.1 Experimental Settings 34
4.2 Overall Results 37
4.2.1 Matching Functions 38
4.2.2 Object Names Renaming Analysis 41
4.2.3 Case base size analysis 42
4.3 Matching Functions Similarity Results 45
4.4 OAKPLAN vs. State of the Art Planners 49

5 Related Work on Case-Based Planning **53**

6 Summary and Future Work **63**

A Proofs **70**

B Variants **73**

C Additional results **78**

1 Introduction

Planning is a process which usually involves the use of a lot of resources. The efficiency of planning systems can be improved by avoiding repeating the planning effort whenever it is not strictly necessary. For example this can be done when the specification of the goals undergoes a variation during plan execution or execution time failures turn up: it is then advisable to change the existing plan rather than replanning from scratch. One might even think of basing the whole planning process on the modification of plans, a procedure also known as planning from *second principles* [61]. In fact this method does not generate a plan from scratch, but aims at exploiting the knowledge contained in plans that were generated before. The current problem instance Π is thus employed to search for a plan in a library that, maybe after a number of changes, might turn out useful to solve Π.

In *C*ase-*B*ased *P*lanning (CBP), previously generated plans are stored as cases in memory and can be reused to solve similar planning problems in the future. CBP can save considerable time over planning from scratch, thus offering a potential (heuristic) mechanism for handling intractable problems. Similarly to other *C*ase-*B*ased *R*easoning (CBR) systems, CBP is based on two assumptions on the nature of the world [50]. The first assumption is that the world is regular: similar problems have similar solutions; as a consequence, solutions for similar problems are a useful starting point for new problem-solving. The second assumption is that the types of problems an agent encounters tend to recur; hence future problems are likely to be similar to current problems.

Different case-based planners differ on how they store cases, how they adapt a solution to a new problem, whether they use one or more cases for building a new solution or not, etc [76]. From a theoretical point of view, in the worst case, adapting an existing plan to solve a new problem is not more efficient than a complete regeneration of the plan [61]. Moreover finding a good reuse candidate in a plan library may be already very expensive, because it leads to more computational costs than those that can be saved by reusing the candidate. In fact, the retrieval of a good plan from a library of plans represents a serious bottleneck for plan reuse in domain independent case-based planning systems. This happens because the problem of defining the best matching among the objects of two planning problems is NP-hard.

In this paper we present some data structures and new matching functions that efficiently address the problem of matching planning instances, which is NP-hard in the general case. These functions lead to a new case-based planner called OAKPLAN (acronym of *O*bject *A*ssignment *K*ernel case-based *plan*ner), which is competitive with state of the art plan generation systems when sufficiently similar reuse candidates can be chosen.

Following the formalisation proposed by Liberatore [52], a *planning case* is a pair $\langle \Pi_0, \pi_0 \rangle$, where Π_0 is a planning problem and π_0 is a plan for it, while a plan library is a set of cases $\{\langle \Pi_i, \pi_i \rangle | 1 \leq i \leq m\}$. Our approach is based on a compact graph representation which uses the initial and goal facts in order to define a detailed description of the *topology* of the planning problem examined. On the basis of this graph representation we use ideas from different research areas. In particular a lot of work has been done in molecular biology to analyse efficiently chemical databases which typically contain thousands of molecules encoded as graphs. Similarly to the RASCAL system [66], we use graph *degree sequences* [70] in order to filter out unpromising planning cases and reduce the set $\mathcal{C}^{ds} = \{\langle \Pi_i, \pi_i \rangle\}$ of cases that have to be examined accurately up to a suitable number.[2]

[2] The RASCAL system uses degree sequences and a rigorous maximum common edge subgraph (MCES) detection algorithm based on a maximum clique formulation to compute the exact degree and composition of similarity

Then in order to define a matching among the objects of the current planning problem and those of the selected planning cases we developed an approximate evaluation based on *Kernel functions* [71], since an exact matching computation is infeasible from a computational point of view also for a limited number of candidate cases. Our kernel functions are inspired by Fröhlich et al.'s work [26, 27, 25] on *kernel functions* for molecular structures, where a kernel function can be thought of as a special similarity measure that can be defined between arbitrarily structured objects, like vectors, strings, trees or graphs [46, 83].

We use kernel functions to define a matching among the objects of the planning problems considered and compute a similarity function that allows to choose a set of plans π_i; these plans are evaluated accurately through a simulated execution that determines the cost of removing the inconsistencies in the plans. The best plan is then adapted, if needed, in order to be applicable in the current initial state and solve the current goals; the adaptation phase is based on the LPG-adapt system [22] which showed excellent performance in many domains. Finally when the adaptation phase is finished and the solution plan is available to the agents, a new planning case corresponding to the current planning problem and its solution plan can be inserted into the library or can be cast off.

Following Nebel & Koehler's formalisation of matching functions [61], we examine the problem of defining a match between the objects of the current planning problem and those of the selected planning cases. Since an exact matching evaluation is infeasible from a computational point of view even for a limited number of candidate cases [61], we develop an *approximate* evaluation based on *kernel functions* [71] to define a match among the objects of the planning problems considered. Our kernel functions are inspired by Fröhlich et al.'s work [25, 26, 27] on kernel functions for molecular structures, where a kernel function can be thought of as a special similarity measure that can be defined among arbitrarily structured objects, like vectors, strings, trees or graphs [46, 83]. The computational attractiveness of kernel methods comes from the fact that they can be applied in high-dimensional feature spaces without suffering the high cost of explicitly computing the mapped data [71].

In contrast to other CBP approaches that define exact matching functions among the objects of Π_0 and those of the plan library whose computation requires exponential time [39, 45, 61], our kernel functions can compute in *polynomial time* an approximate matching function for each element of the set $\mathcal{C}^{ds}$; this matching function can choose a subset of the candidate plans efficiently for the successive *plan evaluation phase*. These plans are evaluated accurately through a simulated execution that determines the capacity of a plan π_i to solve the current planning problem. This phase is performed by executing π_i and evaluating the presence of inconsistencies corresponding to the unsupported preconditions of the actions of π_i; in the same way the presence of unsupported goals is identified. The best plan is then adapted, if necessary, in order to be applicable to the current initial state and solve the current goals. This phase is based on the LPG-adapt system [22] which has shown excellent performance in many domains. When the adaptation phase is concluded, a new planning case corresponding to the current planning problem and its solution plan can be inserted into the library or can be discarded.

OAKPLAN can efficiently retrieve planning cases from plan libraries with more than ten thousand elements, heuristically choose a suitable candidate, possibly the best one, and adapt it to provide a good quality solution plan similar to the one retrieved from the case base. We

---

in chemical databases. Our techniques use degree sequences and an approximate maximum common subgraph (MCS) detection algorithm based on kernel functions to analyse the cases of huge plan libraries and choose a good candidate for adaptation.

hope that this work will be able to renew interest in the case-based planning approach. Current research in planning has been devoted primarily to generative planning since no effective retrieval functions were available in the past. To the best of our knowledge this is the first case-based planner that performs an efficient domain independent objects matching evaluation. We examine it in comparison with state of the art plan generation systems showing that the case-based planning approach can be an effective alternative to plan generation when "sufficiently similar" reuse candidates can be chosen. This is a major improvement on previous approaches on CBP which can only handle small plan libraries (see section 5) and can hardly be compared with plan generation systems.

The paper is organised as follows. Section 2 introduces the essential notions required by the paper. In particular we expose the notion of union of graphs which is fundamental for the definition of the graphs used by our matching functions and we introduce the basic concepts of kernel functions. Section 3 presents the main phases of our case-based planner examining the different steps required by the Retrieval and Evaluation phases in detail. In section 4 a detailed analysis of the importance of an accurate matching function and of the case base size for the global system performance is provided. Then we examine the results produced by OAKPLAN in comparison with four state of the art plan generation systems. Finally, section 6 gives the conclusions and indicates future work.

2 Preliminaries

In the following we present some notations that will be used in the paper with an analysis of the computational complexity of the problems considered.

2.1 Many-sorted logic

Many-sorted logic can significantly increase the efficiency of a deductive inference system by eliminating useless branches of the search space of a domain [15, 16, 85]. In many-sorted logic, constants and variables are divided into subsets called *sorts*. A constant c can be associated with a sort t_c and denoted by

constant $c : t_c$

A variable x has a sort associated with it, denoted as

var $x : t$

Functions or predicates are also associated with appropriate sorts. Predicate symbols and function symbols are only defined on typed arguments. The syntax

$p(x_1 : t_1, ..., x_n : t_n)$

indicates that the predicate symbol p is defined only when its arguments are of sorts $t_1, ..., t_n$ respectively. The syntax

$f(t_1 \times ... \times t_n) \to t$

indicates that the function symbol f is defined only when the sort of a result term is t. The main difference between many-sorted logic and ordinary logic is that the universe of discourse in many-sorted logic is divided into sorts, or types. These sorts need not to be disjointed and, in general, form a partial order based on the set inclusion relation. The effect of the sort machinery is felt primarily in two places: *well-sortedness* and *unification* [15]. In particular well-sortedness requires that all formulae are well sorted, that is, the sort of every constituent term must match the sort of its argument position.

Well-sortedness: a key notion in many-sorted logic is that all formulae must be well sorted, that is, the sort of every constituent term must match the sort of its argument position. To keep the logic simple, here "match" means "being a subsort of". Therefore, in the inference process, the sort of every term (variable or non-variable) must be a subsort of its argument position. Formulae which are not well sorted are ignored by the inference machinery thus reducing the search space.

Unification: in many-sorted logic, the general unification algorithm in AI is modified so that every substitution has to obey certain sort restrictions. We write $t_1 \subset t_2$ when sort t_1 is a strict subsort of t_2. Note that $\subset$ is transitive. Sorts t_1 and t_2 are defined to be compatible if $\exists t_3 \neq \emptyset$ such that $t_3 \subset t_1$ and $t_3 \subset t_2$. It is defined that sorts of two variables being unified must be compatible and the sort of a non-variable must be a subsort of the variable that is substituted for. Consider the case where two variables x_1 and x_2 are being unified, and their sorts are t_1 and t_2 respectively. Sorts t_1 and t_2 must be compatible in order to unify x_1 and x_2; otherwise, the unification fails. If $t_1 \subset t_2$ then we can substitute x_1 for x_2 (denoted as x_1/x_2) and similarly if $t_2 \subset t_1$, we can substitute x_2 for x_1; otherwise, we must introduce a new variable x_3 of sort $t_3 = t_1 \cap t_2$ and make two substitutions: x_3/x_1 and x_3/x_2. If c is a constant or functional term, $c : t_1$ unifies with $x : t_2$ iff $t_1 \subseteq t_2$ and substitutes c for x. With the property of well sortedness, many-sorted logic requires either the sort of the former must be a subsort of the latter, or the two sorts must not be merely disjointed.

These restrictions act as a pruning mechanism of many-sorted logic.

2.2 Planning formalism

Similarly to Bylander's work [11], we define an instance of propositional planning as:

Definition 1 *A* **propositional STRIPS planning problem** *is a tuple* $\Pi = \langle \mathcal{P}_r, \mathcal{I}, \mathcal{G}, \mathcal{O}p \rangle$ *where:*

- $\mathcal{P}_r$ *is a finite set of ground atomic propositional formulae;*
- $\mathcal{I} \subseteq \mathcal{P}_r$ *is the initial state;*
- $\mathcal{G} \subseteq \mathcal{P}_r$ *is the goal specification;*
- $\mathcal{O}p$ *is a finite set of operators, where each operator* $o \in \mathcal{O}p$ *has the form* $o^p \Rightarrow o_+, o_-$ *such that*
  - $o^p \subseteq \mathcal{P}_r$ *are the propositional preconditions,*
  - $o_+ \subseteq \mathcal{P}_r$ *are the positive postconditions (add list),*
  - $o_- \subseteq \mathcal{P}_r$ *are the negative postconditions (delete list)*

  and $o_+ \cap o_- = \emptyset$*.*

We assume that the set of propositions $\mathcal{P}_r$ has a particular structure. Let $\mathbf{O}$ be a set of *typed* constants c_i, with the understanding that distinct constants denote distinct objects (corresponding to *individual entities* following the Conceptual Graphs notation [14]). Let $\mathbf{P}$ be a set of predicate symbols, then $\mathcal{P}_r(\mathbf{O}, \mathbf{P})$ is the set of all ground atomic formulae over this signature. Note that we use a *many-sorted logic* formalisation since it can significantly increase the efficiency of a deductive inference system by eliminating useless branches of the search space of a domain [15, 16, 85]. A fact is an assertion that some individual entities exist and that these entities are related by some relationships.

A plan π is a partially ordered sequence of actions $\pi = (a_1, \ldots, a_m, \mathcal{C})$, where a_i is an action (completely instantiated operator) of π and $\mathcal{C}$ defines the ordering relations between the actions of π. A *linearisation* of a partially ordered plan is a total order over the actions of the plan that is consistent with the existing partial order. In a totally ordered plan $\pi = (a'_1, \ldots, a'_m)$, a precondition f of a plan action a'_i is *supported* if *(i)* f is added by an earlier action a'_j and not deleted by an intervening action a'_k with $j \leq k < i$ or *(ii)* f is true in the initial state and $\nexists\ a'_k$ with $k < i$ s.t. $f \in del(a'_k)$. In a partially ordered plan, a precondition of an action is *possibly supported* if there exists a linearisation in which it is supported, while an action precondition is *necessarily supported* if it is supported in all linearisations. An action precondition is *necessarily unsupported* if it is not possibly supported. A *valid plan* is a plan in which all action preconditions are necessarily supported.

The complexity of STRIPS planning problems has been studied extensively in the literature. Bylander [11] has defined PLANSAT as the decision problem of determining whether an instance Π of propositional STRIPS planning has a solution or not. PLANMIN is defined as the problem of determining if there exists a solution of length k or less, i.e., it is the decision problem corresponding to the search problem of generating plans with minimal length. Based on this framework, he has analysed the computational complexity of a general propositional planning problem and a number of generalisations and restricted problems. In its most general form, both PLANSAT and PLANMIN are PSPACE-complete. Severe restrictions on the form of the operators are necessary to guarantee polynomial time or even NP-completeness [11].

It is important to remark that our approach is more related to the generative case-based planning approach than to the transformational approach, in that the plan library is only assumed to be used when it is useful in the process of searching for a new plan, and is not necessarily used to provide a starting point of the search process. If a planning case $\langle \Pi_0, \pi_0 \rangle$ is also known, the current planning problem Π cannot become more difficult, as we can simply disregard the case and find the plan using Π only. Essential to this trivial result is that, similarly to most modern plan adaptation and case-based planning approaches [3, 36, 79, 80], we do not enforce plan adaptation to be conservative, in the sense that we do not require to reuse as much of the starting plan π_0 to solve the new plan. The computational complexity of plan Reuse and Modification for STRIPS planning problems has been analysed in a number of papers [11, 12, 48, 52, 61]. While a problem cannot be made more difficult by the presence of a hint, which corresponds in our context to the solution of a planning case, it may become easier. Unfortunately the following theorem shows that this is not the case for plan adaptation.

Theorem 1 (see [52]) *Deciding whether there exists a plan for a STRIPS instance* Π*, given a case* $\langle \Pi_0, \pi_0 \rangle$*, is PSPACE-complete, even if* Π_0 *and* Π *only differ on one condition of the initial state.*

Moreover empirical analyses show that plan modification for similar planning instances is somewhat more efficient than plan generation in the average case [10, 22, 35, 36, 61, 77, 80].

2.3 Graphs

Graphs provide a rich means for modelling structured objects and they are widely used in real-life applications to represent molecules, images, or networks. On a very basic level, a graph can be defined by a set of entities and a set of connections between these entities. Due to their universal definition, graphs have found widespread application for many years as tools for the representation of complex relations. Furthermore, it is often useful to compare objects represented as graphs to determine the degree of similarity between the objects. More formally:

Definition 2 *A* **labeled graph** *G is a 3-tuple $G = (V, E, \lambda)$, in which*

- *V is the set of vertices,*
- *$E \subseteq V \times V$ is the set of directed edges or arcs,*
- *$\lambda : V \cup E \to \wp_s(L_\lambda)$ is a function assigning labels to vertices and edges;*

where L_λ is a finite set of symbolic labels and $\wp_s(L_\lambda)$ represents the set of all the *multisets* on L_λ. Note that our label function considers *multisets* of symbolic labels, with the corresponding operations of union, intersection and join [7], since in our context they are more suitable than standard sets of symbolic labels in order to compare vertices or edges accurately as described later. The above definition corresponds to the case of directed graphs; undirected graphs are obtained if we require for each edge $[v_1, v_2] \in E$ the existence of an edge $[v_2, v_1] \in E$ with the same label. $|G| = |V| + |E|$ denotes the *size* of the graph G, while an empty graph such that $|G| = 0$ will be denoted by $\varnothing$. An arc $e = [v, u] \in E$ is considered to be directed from v to u; v is called the *source* node and u is called the *target* node of the arc; u is said to be a *direct successor* of v, v is said to be a *direct predecessor* of u, while v is said to be *adjacent* to the vertex u and vice versa.

Here we present the notion of graph union which is essential for the definition of the graphs used by our matching functions:

Definition 3 *The* **union** *of two graphs $G_1 = (V_1, E_1, \lambda_1)$ and $G_2 = (V_2, E_2, \lambda_2)$, denoted by $G_1 \cup G_2$, is the graph $G = (V, E, \lambda)$ defined by*

- *$V = V_1 \cup V_2$,*
- *$E = E_1 \cup E_2$,*
- $\lambda(x) = \begin{cases} \lambda_1(x) & \texttt{if } x \in (V_1 \backslash V_2) \vee x \in (E_1 \backslash E_2) \\ \lambda_2(x) & \texttt{if } x \in (V_2 \backslash V_1) \vee x \in (E_2 \backslash E_1) \\ \lambda_1(x) \uplus \lambda_2(x) & \texttt{otherwise} \end{cases}$

where $\uplus$ indicates the join, sometimes called *sum*, of two multisets [7], while $\lambda(\cdot)$ associates a multiset of symbolic labels to a vertex or to an edge.

In many applications it is necessary to compare objects represented as graphs and determine the similarity among these objects. This is often accomplished by using graph matching, or isomorphism techniques. Graph isomorphism can be formulated as the problem of identifying a one-to-one correspondence between the vertices of two graphs such that an edge only exists between two vertices in one graph if an edge exists between the two corresponding vertices in the other graph. Graph matching can be formulated as the problem involving the maximum common subgraph (MCS) between the collection of graphs being considered. This is often referred to as the maximum common substructure problem and denotes the largest substructure common to the collection of graphs under consideration. More precisely:

Definition 4 *Two labeled graphs $G = (V, E, \lambda)$ and $G' = (V', E', \lambda')$ are* **isomorphic** *if there exists a bijective function $f : V \to V'$ such that*

- *$\forall v \in V, \lambda(v) = \lambda'\big(f(v)\big)$,*
- *$\forall [v_1, v_2] \in E, \lambda\big([v_1, v_2]\big) = \lambda'\big([f(v_1), f(v_2)]\big)$,*

- $[u, v] \in E$ *if and only if* $\left[f(u), f(v)\right] \in E'$.

We shall say that f is an isomorphism function.

Definition 5 *An* **Induced Subgraph** *of a graph $G = (V, E, \lambda)$ is a graph $S = (V', E', \lambda')$ such that*

- $V' \subseteq V$ *and* $\forall v \in V'$, $\lambda'(v) \subseteq \lambda(v)$,
- $E' \subseteq E$ *and* $\forall e \in E'$, $\lambda'(e) \subseteq \lambda(e)$
- $\forall v, u \in V'$, $[v, u] \in E'$ *if and only if* $[v, u] \in E$

A graph G is a *Common Induced Subgraph* (*CIS*) of graphs G_1 and G_2 if G is isomorphic to induced subgraphs of G_1 and G_2. A common induced subgraph $G = (V, E, \lambda)$ of G_1 and G_2 is called *Maximum Common Induced Subgraph* (*MCIS*) if there exists no other common induced subgraph of G_1 and G_2 with $\sum_{v \in V} |\lambda(v)|$ greater than G. Similarly, a common induced subgraph $G = (V, E, \lambda)$ of G_1 and G_2 is called *Maximum Common Edge Subgraph* (*MCES*), if there exists no other common induced subgraph of G_1 and G_2 with $\sum_{e \in E} |\lambda(e)|$ greater than G. Note that, since we are considering multiset labeled graphs, we require a stronger condition than standard MCIS and MCES for labeled graph, in fact we want to maximise the total cardinality of the multiset labels of vertices/edges involved instead of the simple number of vertices/edges.

As it is well known, subgraph isomorphism and MCS between two or among more graphs are NP-complete problems [28], while it is still an open question if also graph isomorphism is an NP-complete problem. As a consequence, worst-case time requirements of matching algorithms increase exponentially with the size of the input graphs, restricting the applicability of many graph based techniques to very small graphs.

2.4 Kernel Functions for Labeled Graphs

In recent years, a large number of graph matching methods based on different matching paradigms have been proposed, ranging from the spectral decomposition of graph matrices to the training of artificial neural networks and from continuous optimisation algorithms to optimal tree search procedures.

The basic limitation of graph matching is due to the lack of any mathematical structure in the space of graphs. Kernel machines, a new class of algorithms for pattern analysis and classification, offer an elegant solution to this problem [71]. The basic idea of kernel machines is to address a pattern recognition problem in a related vector space instead of the original pattern space. That is, rather than defining mathematical operations in the space of graphs, all graphs are mapped into a vector space where these operations are readily available. Obviously, the difficulty is to find a mapping that preserves the structural similarity of graphs, at least to a certain extent. In other words, if two graphs are structurally similar, the two vectors representing these graphs should be similar as well, since the objective is to obtain a vector space embedding that preserves the characteristics of the original space of graphs.

A key result from the theory underlying kernel machines states that an explicit mapping from the pattern space into a vector space is not required. Instead, from the definition of a kernel function it follows that there exists such a vector space embedding and that the kernel function can be used to extract the information from vectors that is relevant for recognition. As a matter of fact, the family of kernel machines consists of all the algorithms that can be

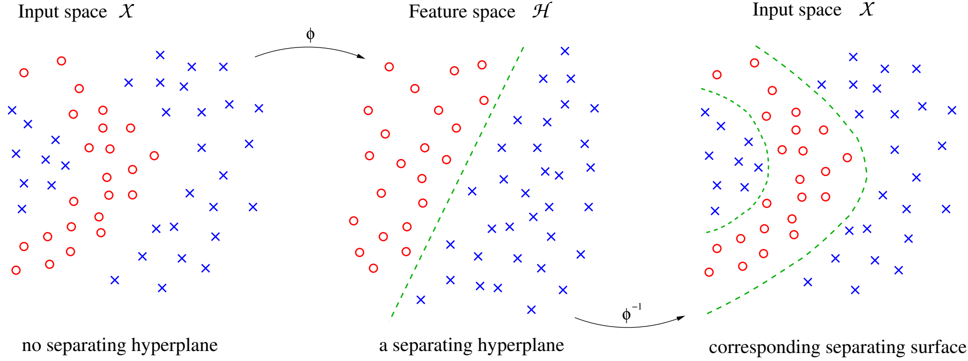


Figure 1: The kernel approach for classification. Left: non-linearly separable input provided by dots and crosses. Middle: perfect or approximate linear-separability can be achieved in feature space via the mapping ϕ. Right: linear decision surface in feature space defines a complex decision surface in input space.

formulated in terms of such a kernel function, including standard methods for pattern analysis and classification such as principal component analysis and nearest-neighbour classification. Hence, from the definition of a graph similarity measure, we obtain an implicit embedding of the entire space of graphs into a vector space.

A *kernel function* can be thought of as a special similarity measure with well defined mathematical properties [71]. Considering the graph formalism, it is possible to define a kernel function which measures the degree of similarity between two graphs. Each structure could be represented by means of its similarity to all the other structures in the graph space. Moreover a kernel function implicitly defines a *dot product* in some space [71]; i.e., by defining a kernel function between two graphs we implicitly define a vector representation of them without the need to explicitly know about it.

From a technical point of view a kernel function is a special similarity measure $k : \mathcal{X} \times \mathcal{X} \rightarrow \mathbb{R}$ between patterns lying in some arbitrary domain $\mathcal{X}$, which represents a dot product, denoted by $\langle \cdot, \cdot \rangle$, in some Hilbert space $\mathcal{H}$ [71]; thus, for two arbitrary patterns $x, x' \in \mathcal{X}$ it holds that $k(x, x') = \langle \phi(x), \phi(x') \rangle$, where $\phi : \mathcal{X} \rightarrow \mathcal{H}$ is an arbitrary mapping of patterns from the domain $\mathcal{X}$ into the feature space $\mathcal{H}$. In principle the patterns in domain $\mathcal{X}$ do not necessarily have to be vectors; they could be strings, graphs, trees, text documents or other objects. The vector representation of these objects is then given by the map ϕ. Instead of performing the expensive transformation step explicitly, the kernel can be calculated directly, thus performing the feature transformation only implicitly: this is known as *kernel trick*. This means that any set, whether a linear space or not, that admits a positive definite kernel can be embedded into a linear space.

More specifically, kernel methods manage non-linear complex tasks making use of linear methods in a new space. For instance, take into consideration a classification problem with a training set $\mathcal{S} = \{(u_1, y_1), \ldots, (u_n, y_n)\}$, $(u_i, y_i) \in \mathcal{X} \times Y$, for $i = 1, \ldots, n$, where $\mathcal{X}$ is an inner-product space $\left(\text{i.e. } \mathbb{R}^d\right)$ and $Y = \{-1, +1\}$. In this case, the learning phase corresponds to building a function $f \in Y^{\mathcal{X}}$ from the training set $\mathcal{S}$ by associating a class $y \in Y$ to a pattern $u \in \mathcal{X}$ so that the generalisation error of f is as low as possible.

A functional form for f consists in the hyperplane $f(u) = sign(\langle w, u \rangle + b)$, where $sign(\cdot)$ refers to the function returning the sign of its argument. The decision function f produces a prediction that depends on which side of the hyperplane $\langle w, u \rangle + b = 0$ the input pattern u

lies. The individuation of the *best* hyperplane corresponds to a convex quadratic optimisation problem in which the solution vector w is a linear combination of the training vectors:

$w = \sum_{i=1}^{n} \alpha_i y_i u_i$, for some $\alpha_i \in I\!R^+$, $i = 1, \ldots, n$.

In this way the linear classifier f may be rewritten as

$$f(u) = sign\left(\sum_{i=1}^{n} \alpha_i y_i \langle u_i, u \rangle + b\right)$$

As regards complex classification problems, the set of all possible linear decision surfaces might not be rich enough in order to provide a good classification, independently from the values of the parameters $w \in \mathcal{X}$ and $b \in I\!R$ (see Figure 1). The aim of the kernel trick is that of overcoming this limitation by adopting a linear approach to transformed data $\phi(u_1), \ldots, \phi(u_n)$ rather than raw data. Here ϕ indicates an embedding function from the input space $\mathcal{X}$ to a feature space $\mathcal{H}$, provided with a dot product. This transformation enables us to give an alternative kernel representation of the data which is equivalent to a mapping into a high-dimensional space where the two classes of data are more readily separable. The mapping is achieved through a replacement of the inner product:

$$\langle u_i, u \rangle \rightarrow \langle \phi(u_i), \phi(u) \rangle$$

and the separating function can be rewritten as:

$$f(u) = sign\left(\sum_{i=1}^{n} \alpha_i y_i \langle \phi(u_i), \phi(u) \rangle + b\right) \quad (1)$$

The main idea behind the kernel approach consists in replacing the dot product in the feature space using a kernel $k(u, v) = \langle \phi(v), \phi(u) \rangle$; the functional form of the mapping $\phi(\cdot)$ does not actually need to be known since it is implicitly defined by the choice of the kernel. A positive definite kernel [29] is:

Definition 6 *Let $\mathcal{X}$ be a set. A symmetric function $k : \mathcal{X} \times \mathcal{X} \rightarrow I\!R$ is a positive definite* **kernel function** *on $\mathcal{X}$ iff* $\forall n \in I\!N$, $\forall x_1, \ldots, x_n \in \mathcal{X}$, *and* $\forall c_1, \ldots, c_n \in I\!R$

$$\sum_{i,j \in \{1,\ldots,n\}} c_i c_j k(x_i, x_j) \geq 0$$

where $I\!N$ is the set of positive integers. For a given set $S_u = \{u_1, \ldots, u_n\}$, the matrix $\mathbf{K} = (k(u_i, u_j))_{i,j}$ is known as Gram matrix of k with respect to S_u. Positive definite kernels are also called Mercer kernels.

Theorem 2 (Mercer's property [58]) *For any positive definite kernel function $k \in I\!R^{\mathcal{X} \times \mathcal{X}}$, there exists a mapping $\phi \in \mathcal{H}^{\mathcal{X}}$ into the feature space $\mathcal{H}$ equipped with the inner product $\langle \cdot, \cdot \rangle_{\mathcal{H}}$, such that:*

$$\forall u, v \in \mathcal{X}, \qquad k(u, v) = \langle \phi(u), \phi(v) \rangle_{\mathcal{H}}$$

The kernel approach replaces all inner products in Equation 1 and all related expressions to compute the real coefficients α_i and b, by means of a Mercer kernel k. For any input pattern u, the relating decision function f is given by:

$$f(u) = sign\left(\sum_{i=1}^{n} \alpha_i y_i k(u_i, u) + b\right) \quad (2)$$

This approach transforms the input patterns $u_1, \ldots, u_n$ into the corresponding vectors $\phi(u_1), \ldots, \phi(u_n) \in \mathcal{H}$ through the mapping $\phi \in \mathcal{H}^{\mathcal{X}}$ (cf. Mercer's property, Theorem 2), and uses hyperplanes in the feature space $\mathcal{H}$ for the purpose of classification (see Figure 1). The dot product $\langle u, v \rangle = \sum_{i=1}^{d} u_i v_i$ of $I\!R^d$ is actually a Mercer kernel, while other commonly used Mercer kernels, like polynomial and Gaussian kernels, generally correspond to nonlinear mappings ϕ into high-dimensional feature spaces $\mathcal{H}$. On the other hand the Gram matrix implicitly defines the geometry of the embedding space and permits the use of linear techniques in the feature space so as to derive complex decision surfaces in the input space $\mathcal{X}$.

While it is not always easy to prove positive definiteness for a given kernel, positive definite kernels are characterised by interesting closure properties. More precisely, they are closed under sum, direct sum, multiplication by a scalar, tensor product, zero extension, pointwise limits, and exponentiation [71]. Well-known examples of kernel functions are:

- Radial Basis Functions $k_{RBF}(x, x') = exp\left(\frac{-\|x-x'\|^2}{2\sigma^2}\right)$;
- Homogenous polynomial kernels $k_{poly}(x, x') = \langle x, x' \rangle^d \ (d \in I\!N)$;
- Sigmoidal kernels $k_{Sig}(x, x') = tanh\left(\kappa(x \cdot x') + \theta\right)$
- Inv. multiquadratic kernels $k_{inv}(x, x') = \frac{1}{\sqrt{\|x-x'\|^2 + c^2}}$

A remarkable contribution to graph kernels is the work on convolution kernels, that provides a general framework to deal with complex objects consisting of simpler parts [41]. Convolution kernels derive the similarity of complex objects from the similarity of their parts. Given two kernels k_1 and k_2 over the same set of objects, new kernels may be built by using operations such as convex linear combinations and convolutions. The convolution of k_1 and k_2 is a new kernel k with the form

$$k_1 \star k_2(u, v) = \sum_{\{u1,u2\}=u;\{v1,v2\}=v} k_1(u_1, v_1) k_2(u_2, v_2)$$

where $u = \{u_1, u_2\}$ refers to a partition of u into two substructures u_1 and u_2 [41, 71]. The kind of substructures depends on the domain of course and could be, for instance, subgraphs or subsets or substrings in the case of kernels defined over graphs, sets or strings, respectively. Different kernels can be obtained by considering different classes of subgraphs (e.g. directed/undirected, labeled/unlabeled, paths/trees/cycles, deterministic/random walks) and various ways of listing and counting them [46, 62, 63]. The consideration of space and time complexity so as to compute convolution/spectral kernels is important, owing to the combinatorial explosion linked to variable-size substructures.

In the following section we present our *Optimal Assignment Kernel* as a symmetric and positive definite similarity measure for directed graph structures and it will be used in order to define the correspondence between the vertices of two directed graphs. For an introduction to kernel functions related concepts and notation, the reader is referred to Scholkopf and Smola's book [71].

3 Case-Based Planning

A *case-based planning system* solves planning problems by making use of stored plans that were used to solve analogous problems. CBP is a type of case-based reasoning, which involves the use of stored experiences (*cases*); moreover there is strong evidence that people frequently employ this kind of analogical reasoning [30, 69, 84]. When a CBP system solves a new

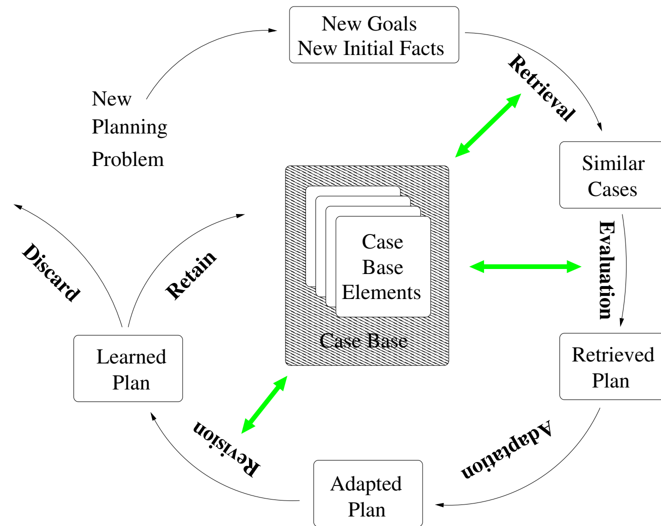


Figure 2: The case-based planning cycle.

planning problem, the new plan is added to its case base for potential reuse in the future. Thus we can say that the system learns from experience.

In general the following steps are executed when a new planning problem must be solved by a CBP system:

1. *Plan Retrieval* to retrieve cases from memory that are analogous to the current (*target*) problem (see section 3.1 for a description of our approach).

2. *Plan Evaluation* to evaluate the new plans by execution, simulated execution, or analysis and choose one of them (see section 3.2).

3. *Plan Adaptation* to repair any faults found in the new plan (see section 3.3).

4. *Plan Revision* to test the solution new plan π for success and repair it if a failure occurs during execution (see section 3.4).

5. *Plan Storage* to eventually store π as a new case in the case base (see section 3.5).

In order to realise the benefits of remembering and reusing past plans, a CBP system needs efficient methods for retrieving analogous cases and for adapting retrieved plans together with a case base of sufficient size and coverage to yield useful analogues. The ability of the system to search in the library for a plan suitable to adaptation[3] depends both on the efficiency/accuracy of the implemented retrieval algorithm and on the data structures used to represent the elements of the case base.

A planning case of the case base corresponds to a planning problem Π (defined by an initial state I, a goal state G and a set of operators O) a solution π of Π and additional data structures derived by Π and stored in the case base so as to avoid their recomputation. The case base competence should increase progressively during the use of the case base system itself, every time solution plans enhancing the competence of the case base are produced.

[3]A plan suitable to adaptation has an adaptation cost that is lower with respect to the other candidates of the case base and with respect to plan generation.

The possibility of solving a large number of problems depends both on the size and on the competence of the library with respect to the agent activity. Furthermore this competence could be increased during the agent activity, in fact the solution plans of new problems could be added to the library.

Similarly to the Aamodt & Plaza's classic model of the problem solving cycle in CBR [1], Figure 2 shows the main steps of our case-based planning cycle and the interactions of the different steps with the case base. In the following we illustrate the main steps of our case-based planning approach, examining the different implementation choices adopted.

3.1 Plan Retrieval

Although the plan adaptation phase is the central component of a CBP system, the retrieval phase critically affects the system performance too. As a matter of fact the retrieval time is a component of the *total adaptation time* and the *quality* of the retrieved plan is fundamental for the performance of the successive adaptation phase. With OAKPLAN a number of functions for the management of the plan library and matching functions for the selection of the candidate plan for adaptation have been implemented.

The retrieval phase has to consider all the elements of the plan library in order to choose a good one that will allow the system to solve the new problem easily. Hence it is necessary to design a similarity metric and reduce the number of cases that must be evaluated accurately so as to improve the efficiency of the retrieval phase. Anyway the efficiency of a plan adaptation system is undoubtedly linked to the *distance* between the problem to solve and the plan to adapt. In order to find a plan which is *useful* for adaptation we have to reach the following objectives:

- The retrieval phase must identify the candidates for adaptation. The retrieval time should be as small as possible as it will be added to the adaptation time and so particular attention has been given to the creation of efficient data structures for this phase.
- The selected cases should actually contain the plans that are easier to adapt; since we assume that the world is regular, i.e. that similar problems have similar solutions, we look for the cases that are the most similar to the problem to solve (with respect to all the other candidates of the case base). In this sense, it is important to define a metric able to give an accurate measure of the similarity between the planning problem to solve and the cases of the plan library.

To the end of applying the reuse technique, it is necessary to provide a plan library from which "sufficiently similar" reuse candidates can be chosen. In this case, "sufficiently similar" means that reuse candidates have a large number of initial and goal facts in common with the new instance. However, one may also want to consider the reuse candidates that are similar to the new instance after the objects of the selected candidates have been systematically renamed. As a matter of fact, every plan reuse system should contain a matching component that tries to find a mapping between the objects of the reuse candidate and the objects of the new instance such that the number of common goal facts is maximised and the additional planning effort to achieve the initial state of the plan library is minimised. Following Nebel & Koehler's formalisation [61], we will have a closer look at this matching problem.

3.1.1 Object Matching

As previously said we use *a many-sorted logic* in order to reduce the search space for the matching process; moreover we assume that the operators are ordinary STRIPS operators using variables, i.e. we require that if there exists an operator o_k mentioning the typed constants $\{c_1 : t_1, ..., c_n : t_n\} \subseteq \mathbf{O}$, then there also exists an operator o_l over the arbitrary set of typed constants $\{d_1 : t_1, ..., d_n : t_n\} \subseteq \mathbf{O}$ such that o_l becomes identical to o_k if the d_i's are replaced by c_i's. If there are two instances

$$\Pi' = \langle \mathcal{P}_r(\mathbf{O}', \mathbf{P}'), \mathcal{I}', \mathcal{G}', \mathcal{O}p' \rangle$$

$$\Pi = \langle \mathcal{P}_r(\mathbf{O}, \mathbf{P}), \mathcal{I}, \mathcal{G}, \mathcal{O}p \rangle$$

such that (without loss of generality)

$$\mathbf{O}' \subseteq \mathbf{O}$$

$$\mathbf{P}' = \mathbf{P}$$

$$\mathcal{O}p' \subseteq \mathcal{O}p$$

then a *mapping*, or *matching function*, from Π' to Π is a function

$$\mu : \mathbf{O}' \to \mathbf{O}$$

The mapping is extended to ground atomic formulae and sets of such formulae in the canonical way, i.e.,

$$\mu(p(c_1 : t_1, ..., c_n : t_n)) = p(\mu(c_1) : t_1, ..., \mu(c_n) : t_n)$$

$$\mu(\{p_1(..), ..., p_m(..)\}) = \{\mu(p_1(..)), ..., \mu(p_m(..))\}$$

If there exists a bijective matching function μ from Π' to Π such that $\mu(G') = G$ and $\mu(I') = I$, then it is obvious that a solution plan π' for Π' can be directly reused for solving Π since Π' and Π are identical within a renaming of constant symbols, i.e., $\mu(\pi')$ solves Π. Even if μ does not match all goal and initial-state facts, $\mu(\pi')$ can still be used as a starting point for the adaptation process that can solve Π.

In order to measure the similarity between two objects, it is intuitive and usual to compare the features which are common to both objects [54]. The Jaccard similarity coefficient used in information retrieval is particularly interesting. Here we examine an extended version that considers two pairs of disjoint sets:

$$complete_simil_\mu(\Pi', \Pi) = \frac{|\mu(\mathcal{G}') \cap \mathcal{G}| + |\mu(\mathcal{I}') \cap \mathcal{I}|}{|\mu(\mathcal{G}') \cup \mathcal{G}| + |\mu(\mathcal{I}') \cup \mathcal{I}|} \tag{3}$$

In the following we present a variant of the previous function so as to overcome the problems related to the presence of irrelevant facts in the initial state description of the current planning problem Π and additional goals that are present in Π'. In fact while the irrelevant facts can be filtered out from the initial state description of the case-based planning problem Π' using the corresponding solution plan π', this is not possible for the initial state description of the current planning problem Π. Similarly, we do not want to consider possible "irrelevant" additional goals of G'; this could happen when Π' solves a more difficult planning problem with respect to Π. We define the following similarity function so as to address these issues:

$$simil_\mu(\Pi', \Pi) = \frac{|\mu(\mathcal{G}') \cap \mathcal{G}| + |\mu(\mathcal{I}') \cap \mathcal{I}|}{|\mathcal{G}| + |\mu(\mathcal{I}')|}. \tag{4}$$

Using $simil_\mu$ we obtain a value equal to 1 when there exists a mapping μ s.t. $\forall f \in I', \ \mu(f) \in I$ (to guarantee the applicability of π') *and* $\forall g \in G, \ \exists g' \in G'$ s.t. $g = \mu(g')$ (to guarantee the achievement of the goals of the current planning problem). Note that these similarity functions are not metric functions, although we could define a distance function in terms of the similarity as $Dist_\mu(\Pi', \Pi) = 1 - simil_\mu(\Pi', \Pi)$. and it easy to show that this distance function is indeed a metric.

Finally we define the following optimisation problem, which we call `obj_match`:
Instance: Two planning instances, Π' and Π, and a real number $k \in [0, 1]$.
Question: Does a mapping μ from Π' to Π such that $simil_\mu(\Pi', \Pi) = k$ exist and there is no mapping μ' from Π' to Π with $simil_{\mu'}(\Pi', \Pi) > k$?

It should be noted that this matching problem has to be solved for each potentially relevant candidate in the plan library to select the corresponding best reuse candidate. Of course, one may use structuring and indexing techniques to avoid considering all plans in the library. Nevertheless, it seems unavoidable solving this problem a considerable number of times before an appropriate reuse candidate is identified. For this reason, the efficiency of the matching component is crucial for the overall system performance. Unfortunately, similarly to Nebel & Koehler's analysis [61], it is quite easy to show that this matching problem is an NP-hard problem.

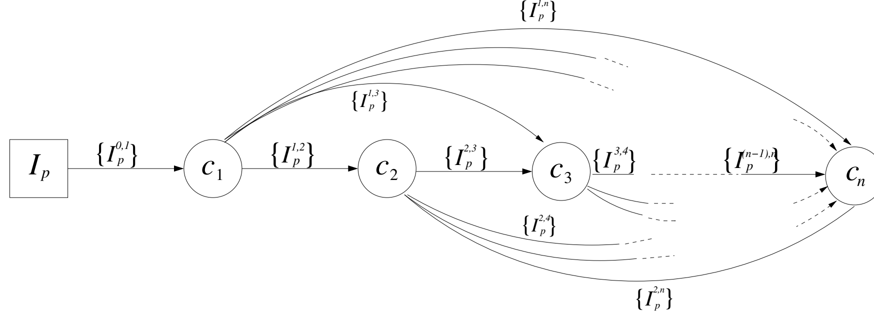
Theorem 3 `obj_match` *is NP-hard.*

The proof of this theorem and of the following ones can be found in Appendix A. This NP-hardness result implies that matching may be indeed a bottleneck for plan reuse systems. As a matter of fact, it seems to be the case that planning instances with complex goal or initial-state descriptions may not benefit from plan-reuse techniques because matching and retrieval are too expensive. In fact existing similarity metrics address the problem heuristically, considering approximations of it [60, 81]. However, this theorem is interesting because it captures the limit case for such approximations.

Planning Encoding Graph. We define a particular labeled graph data structure called *Planning Encoding Graph* which encodes the initial and goal facts of a single planning problem Π to perform an efficient matching between the objects of a planning case and the objects of the current planning problem. The vertices of this graph belong to a set $\mathbf{V}_\Pi$ whose elements are the representation of the objects $\mathbf{O}$ of the current planning problem Π and of the predicate symbols $\mathbf{P}$ of Π:

$$\mathbf{V}_\Pi = \mathbf{O} \cup \bigcup_{p \in \mathbf{P}} I_p \cup \bigcup_{q \in \mathbf{P}} G_q$$

i.e. for each predicate we define two additional nodes, one associated to the corresponding initial fact predicate called I_p and the other associated to the corresponding goal fact predicate called G_q. The labels of this graph are derived from the predicates of our facts and the sorts of our many-sorted logic. The representation of an entity (an object using planning terminology) of the application domain is traditionally called a *concept* in the conceptual graph community [14]. Following this notation a *Planning Encoding Graph* is composed of three kinds of nodes: *concept* nodes representing entities (objects) that occur in the application domain, *initial fact relation* nodes representing relationships that hold between the objects of the initial facts and *goal fact relation* nodes representing relationships that hold between the objects of the goal facts.



$$\lambda(I_p) = \{(I_p, 1)\} = \{I_p\},\ \ \lambda(c_1) = \{(t_1, 1)\} = \{t_1\}, ..., \lambda(c_n) = \{(t_n, 1)\} = \{t_n\}$$

Figure 3: Initial Fact Encoding Graph $\mathcal{E}^I(\mathbf{p})$ of the propositional initial fact $\mathbf{p} = p(c_1 : t_1, ..., c_n : t_n)$

Initial fact: (on A B)

$$\lambda(I_{on}) = \{I_{on}\},\ \ \lambda(A) = \{Obj\},\ \ \lambda(B) = \{Obj\}$$

Figure 4: Initial Fact Encoding Graph $\mathcal{E}^I(on\ A\ B)$ of the propositional initial fact $(on\ A\ B)$.

The *Planning Encoding Graph* of a planning problem $\Pi(I, G)$ is built using the corresponding initial and goal facts. In particular for each propositional initial fact $\mathbf{p} = p(c_1 : t_1, ..., c_n : t_n) \in I$ we define a data structure called *Initial Fact Encoding Graph* which corresponds to a graph that represents $\mathbf{p}$. More precisely:

Definition 7 *Given a propositional typed initial fact* $\mathbf{p} = p(c_1 : t_1, ..., c_n : t_n) \in I$ *of* Π*, the* **Initial Fact Encoding Graph** $\mathcal{E}^I(\mathbf{p}) = (V_\mathbf{p}, E_\mathbf{p}, \lambda_\mathbf{p})$ *of fact* $\mathbf{p}$ *is a directed labeled graph where*

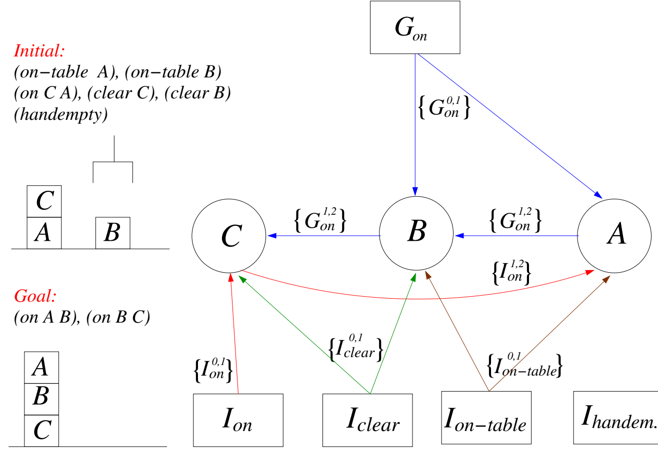
- $V_\mathbf{p} = \{I_p, c_1, ..., c_n\} \subseteq \mathbf{V}_\Pi$*;*
- $E_\mathbf{p} = \{[I_p, c_1], [c_1, c_2], [c_1, c_3], ..., [c_1, c_n], [c_2, c_3], [c_2, c_4], ..., [c_{n-1}, c_n]\} =$

$$= [I_p, c_1] \cup \bigcup_{i=1,...,n;\ j=i+1,...,n} [c_i, c_j]$$

- $\lambda_\mathbf{p}(I_p) = \{I_p\}, \lambda_\mathbf{p}(c_i) = \{t_i\}$ `with` $i = 1, ..., n$*;*
- $\lambda_\mathbf{p}([I_p, c_1]) = \{I_p^{0,1}\}$*;* $\quad \forall [c_i, c_j] \in E_\mathbf{p},\ \lambda_\mathbf{p}([c_i, c_j]) = \{I_p^{i,j}\}$*;*

i.e. the first node of the graph $\mathcal{E}^I(\mathbf{p})$, see Figure 3, is the initial fact relation node I_p labeled with the multiset $\lambda_\mathbf{p}(I_p) = \{(I_p, 1)\} = \{I_p\}$,[4] it is connected to a direct edge to the second node of the graph, the concept node c_1, which is labeled by sort t_1 (i.e. $\lambda_\mathbf{p}(c_1) = \{(t_1, 1)\} = \{t_1\}$); the node c_1 is connected with the third node of the graph c_2 which is labeled by sort t_2 (i.e. $\lambda_\mathbf{p}(c_2) = \{(t_2, 1)\} = \{t_2\}$) and with all the remaining concept nodes, the third node of

[4] In the following we indicate the multiset $\{(x, 1)\}$ as $\{x\}$ for sake of simplicity.



$\lambda(A) = \{(Obj, 3)\}$, $\lambda(B) = \{(Obj, 4)\}$ and $\lambda(C) = \{(Obj, 3)\}$

Figure 5: Planning Encoding Graph for the Sussman Anomaly planning problem in the BlocksWorld domain.

the graph c_2 is connected with c_3, c_4, ...,c_n and so on. The first edge of the graph $[I_p, c_1]$ is labeled by the multiset $\{I_p^{0,1}, 1\} = \{I_p^{0,1}\}$, similarly a generic edge $[c_i, c_j] \in E_\mathbf{p}$ is labeled by the multiset $\{I_p^{i,j}\}$.

For example, in Figure 4 we can see the Initial Fact Encoding Graph of the fact "$\mathbf{p} = (on\ A\ B)$" of the BlocksWorld domain. The first node is named as "I_{on}" and its label is the multiset $\lambda_\mathbf{p}(I_{on}) = \{(I_{on}, 1)\} = \{I_{on}\}$, the second node represents the object "A" with label $\lambda_\mathbf{p}(A) = \{(Obj, 1)\} = \{Obj\}$ and finally the third node represents the object "B" and its label is $\lambda_\mathbf{p}(B) = \{Obj\}$; the label of the $[I_{on}, A]$ arc is the multiset $\{(I_{on}^{0,1}, 1)\} = \{I_{on}^{0,1}\}$ and the label of the $[A, B]$ arc is the multiset $\{(I_{on}^{1,2}, 1)\} = \{I_{on}^{1,2}\}$.

Similarly to Definition 7 we define the **Goal Fact Encoding Graph** $\mathcal{E}^G(\mathbf{q})$ of the fact $\mathbf{q} = q(c'_1 : t'_1, ..., c'_m : t'_m) \in G$ using $\{G_q\}$ for the labeling procedure.

Given a planning problem Π with initial and goal states I and G, the **Planning Encoding Graph** of Π, that we indicate as $\mathcal{E}_\Pi$, is a directed labeled graph derived by the encoding graphs of the initial and goal facts:

$$\mathcal{E}_{\Pi(I,G)} = \bigcup_{\mathbf{p}\in I} \mathcal{E}^I(\mathbf{p}) \cup \bigcup_{\mathbf{q}\in G} \mathcal{E}^G(\mathbf{q}) \tag{5}$$

i.e. the Planning Encoding Graph of $\Pi(I, G)$ is a graph obtained by merging the Initial and Goal Fact Encoding Graphs. For simplicity in the following we visualise it as a three-level graph. The first level is derived from the predicate symbols of the initial facts, the second level encodes the objects of the initial and goal states and the third level shows the goal fact nodes derived from the predicate symbols of the goal facts.[5]

Figure 5 illustrates the Planning Encoding Graph for the Sussman anomaly planning problem in the BlocksWorld domain. The nodes of the first and third levels are the initial and goal fact relation nodes: the vertices I_{on}, I_{clear} and $I_{on-table}$ are derived by the predicates of the

[5]Following the conceptual graph notation, the first and third level nodes correspond to initial and goal fact relation nodes, while the nodes of the second level correspond to concept nodes representing the objects of the initial and goal states.

initial facts, while G_{on} by the predicates of the goal facts. The nodes of the second level are concept nodes which represent the objects of the current planning problem A, B and C, where the label "Obj" corresponds to their type. The initial fact "*(on C A)*" determines two arcs, one connecting I_{on} to the vertex C and the second connecting C to A; the labels of these arcs are derived from the predicate symbol "on" determining the multisets $\{I_{on}^{0,1}\}$ and $\{I_{on}^{1,2}\}$ respectively. In the same way the other arcs are defined. Moreover since there is no overlapping among the edges of the Initial and Goal Fact Encoding Graphs, the multiplicity of the edge label multisets is equal to 1; on the contrary the label multisets of the vertices associated to the objects are:

$\lambda(A) = \{(Obj, 3)\}$, $\lambda(B) = \{(Obj, 4)\}$ and $\lambda(C) = \{(Obj, 3)\}$.

Moreover it could be useful to point out that if an object c appears more than once in an initial (goal) fact $p(c_1....c_n)$ of a planning problem Π, then the corresponding Initial (Goal) Fact Encoding Graph is built as usual (instantiating n nodes, one each c_i), while during the construction of the Planning Encoding Graph obtained by merging the Initial and Goal Encoding Graphs of Π, the nodes that correspond to the same object are merged into a single vertex node.

This graph representation can give us a detailed description of the "topology" of a planning problem without requiring any a priori assumptions on the relevance of certain problem descriptors for the whole graph. Furthermore it allows us to use Graph Theory based techniques in order to define effective matching functions. In fact a matching function from Π' to Π can be derived by solving the *Maximum Common Subgraph* problem on the corresponding Planning Encoding Graphs. A number of exact and approximate algorithms have been proposed in the literature so as to solve this graph problem efficiently. With respect to *normal conceptual graphs* [14] used for Graph-based Knowledge Representation, we use a richer label representation based on multisets. A single relation node is used to represent each predicate of the initial and goal facts which reduces the total number of nodes in the graphs considerably. This is extremely important from a computational point of view since, as we will see in the following sections, the matching process must be repeated several times and it directly influences the total retrieval time.

In the following we examine a procedure based on graph *degree sequences* that is useful to derive an upper bound on the size of the *MCES* of two graphs in an efficient way. Then we present an algorithm based on *Kernel Functions* that allows to compute an approximate matching of two graphs in polynomial time.

3.1.2 Screening Procedure

As explained previously, the retrieval phase could be very expensive from a computational point of view; so we have developed a screening procedure that can be used in conjunction with an object matching algorithm.

Similarly to the RASCAL system [66], we use *degree sequences* [70] to calculate an upper bound on the size of a Maximum Common Edge Subgraph (MCES) between a pair of graphs. Note that degree sequences of a graph have already been used by other authors to establish upper bounds on graph invariants [40, 55] and for indexing graph databases [64].

First, the set of vertices in each graph is partitioned into l partitions by label type, and then sorted in a non-increasing total order by degree.[6] Let L_1^i and L_2^i denote the sorted degree sequences of a partition i in the planning encoding graphs G_1 and G_2, respectively. An upper

[6]The degree or valence of a vertex v of a graph G is the number of edges which touch v.

bound on the number of vertices $Vertices(G_1, G_2)$ and edges $Edges(G_1, G_2)$ of the MCES graph can be computed as follows:

$$Vertices(G_1, G_2) = \sum_{i=1}^{l} min\left(|L_1^i|, |L_2^i|\right)$$

$$Edges(G_1, G_2) = \left\lfloor \sum_{i=1}^{l} \sum_{j=1}^{min(|L_1^i|,|L_2^i|)} \frac{min\left(|E(v_1^{i,j})|, |E(v_2^{i,j})|\right)}{2} \right\rfloor$$

where $v_1^{i,j}$ indicates the j-th element (vertex) of the L_1^i sorted degree sequence and $E(v_1^{i,j})$ indicates the set of arcs connected to the vertex $v_1^{i,j}$. An upper bound on the similarity between G_1 and G_2 can be expressed using Johnson's similarity coefficient [43]:

$$\begin{aligned} simil^{ds}(G_1, G_2) &= \frac{(Vertices(G_1, G_2) + Edges(G_1, G_2))^2}{(|V(G_1)| + |E(G_1)|) \cdot (|V(G_2)| + |E(G_2)|)} = \\ &= \frac{(Vertices(G_1, G_2) + Edges(G_1, G_2))^2}{|G_1| \cdot |G_2|} \end{aligned} \tag{6}$$

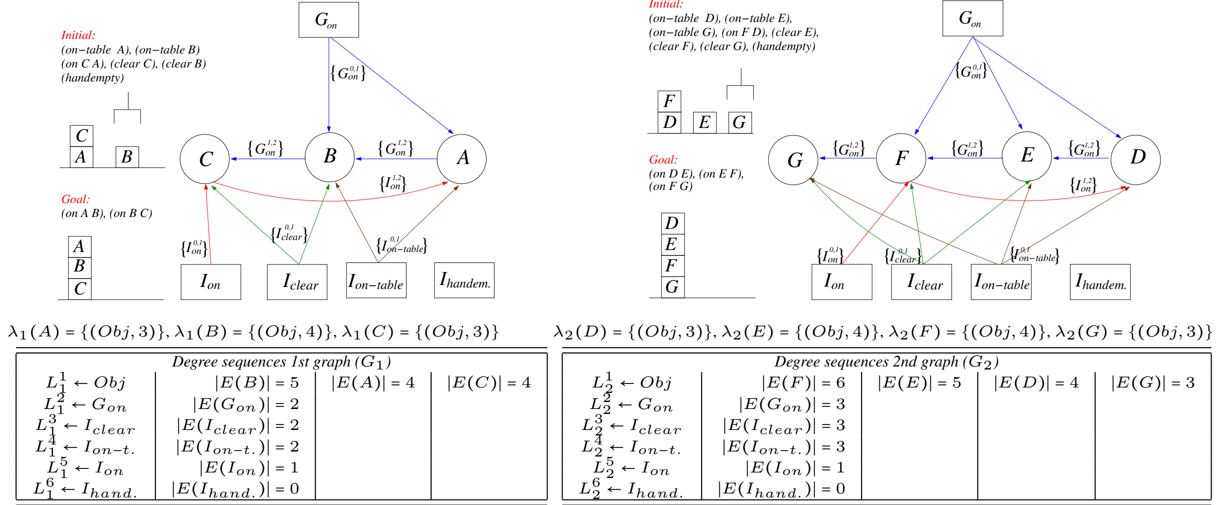
Since $Vertices(G_1, G_2)$ and $Edges(G_1, G_2)$ determine an upper bound on the number of nodes and arcs of the MCES of G_1 and G_2, the $simil^{ds}(G_1, G_2)$ similarity measure ranges from 0 to 1 and it is easy to show that it obeys the following inequality,

$$simil^{ds}(G_1, G_2) \geq \frac{(|V(G_{12})| + |E(G_{12})|)^2}{|G_1| \cdot |G_2|} = \frac{|G_{12}|^2}{|G_1| \cdot |G_2|}$$

where G_{12} is the MCES between graphs G_1 and G_2. Clearly, $simil^{ds}(G_1, G_2)$ can be used as an upper bound for the size of the MCES between G_1 and G_2 and this procedure provides a rapid screening mechanism which takes advantage of local connectivity and vertex labels to help eliminate unnecessary and costly MCES comparisons. For screening purposes, it is only necessary to specify a minimum acceptable value for the MCES based graph similarity measure. If the value determined by $simil^{ds}(G_1, G_2)$ is less than the minimum acceptable similarity, then the object matching comparison can be avoided. This procedure can be performed by using the quick sort algorithm in $O(n \cdot logn)$ time, where $n = max_i\left(|L_1^i|, |L_2^i|\right)$.

Figure 6 shows the degree sequences of two Planning Encoding Graphs and the corresponding similarity value $simil^{ds}$. To compute the degree sequences, the vertices of the graphs are first separated into partitions according to their label type. Considering the G_1 graph, the L_1^1 degree sequence is of type "Obj" and it has three elements: the vertices "$v_1^{1,1} = B$", "$v_1^{1,2} = A$" and "$v_1^{1,3} = C$" with degree 5, 4 and 4 respectively. Similarly the second degree sequence L_1^2 of G_1 is of type "G_{on}" which has only one element: the vertex "$v_1^{2,1} = G_{on}$" with degree 2. Moreover, the entry "$v_2^{1,3} = D$" of graph G_2 belongs to the sorted degree sequence L_2^1 of type "Obj", it is in third position in the degree sequence and its degree is equal to 4; while the entry "$v_2^{3,1} = I_{clear}$" of graph G_2 belongs to the degree sequence L_2^3 which is of type "I_{clear}", it is in first position in the degree sequence L_2^3 and it has degree 3. $Vertices(G_1, G_2)$ is calculated by summing the size of the set L^i with the fewest non-null elements in graphs G_1 and G_2. Since L_1^1 has 3 elements and L_2^1 has 4 elements, the first term of $Vertices(G_1, G_2)$ is equal to $min\left(|L_1^1|, |L_2^1|\right) = 3$. Considering the other partitions $L^2, ..., L^6$ we obtain $Vertices(G_1, G_2) = 8$.

Besides $Edges(G_1, G_2)$ is determined by summing the $min\left(|E(v_1^{i,j})|, |E(v_2^{i,j})|\right)$ values of each partition i, adding the resulting values together and then integer dividing the result



$\lambda_1(A) = \{(Obj,3)\}, \lambda_1(B) = \{(Obj,4)\}, \lambda_1(C) = \{(Obj,3)\}$

$\lambda_2(D) = \{(Obj,3)\}, \lambda_2(E) = \{(Obj,4)\}, \lambda_2(F) = \{(Obj,4)\}, \lambda_2(G) = \{(Obj,3)\}$

Degree sequences 1st graph (G_1)			
$L_1^1 \leftarrow Obj$	$\lvert E(B)\rvert = 5$	$\lvert E(A)\rvert = 4$	$\lvert E(C)\rvert = 4$
$L_1^2 \leftarrow G_{on}$	$\lvert E(G_{on})\rvert = 2$		
$L_1^3 \leftarrow I_{clear}$	$\lvert E(I_{clear})\rvert = 2$		
$L_1^4 \leftarrow I_{on-t.}$	$\lvert E(I_{on-t.})\rvert = 2$		
$L_1^5 \leftarrow I_{on}$	$\lvert E(I_{on})\rvert = 1$		
$L_1^6 \leftarrow I_{hand.}$	$\lvert E(I_{hand.})\rvert = 0$		

Degree sequences 2nd graph (G_2)				
$L_2^1 \leftarrow Obj$	$\lvert E(F)\rvert = 6$	$\lvert E(E)\rvert = 5$	$\lvert E(D)\rvert = 4$	$\lvert E(G)\rvert = 3$
$L_2^2 \leftarrow G_{on}$	$\lvert E(G_{on})\rvert = 3$			
$L_2^3 \leftarrow I_{clear}$	$\lvert E(I_{clear})\rvert = 3$			
$L_2^4 \leftarrow I_{on-t.}$	$\lvert E(I_{on-t.})\rvert = 3$			
$L_2^5 \leftarrow I_{on}$	$\lvert E(I_{on})\rvert = 1$			
$L_2^6 \leftarrow I_{hand.}$	$\lvert E(I_{hand.})\rvert = 0$			

$$Vertices(G_1,G_2) = 3{+}1{+}1{+}1{+}1{+}1 = 8;\;\; Edges(G_1,G_2) = \left\lfloor \frac{(5+4+4)}{2} + \frac{2}{2} + \frac{2}{2} + \frac{2}{2} + \frac{1}{2} \right\rfloor = 10;\;\; simil^{ds}(G_1,G_2) = \frac{(8+10)^2}{(8+10)*(9+15)} = 0.75$$

Figure 6: Planning Encoding Graphs, degree sequences and the corresponding similarity value for two planning problems in the BlocksWorld domain.

by two. For example, the first term of $Edges(G_1, G_2)$ is obtained by dividing by two the following value:

$$min\left(|E(v_1^{1,1})|, |E(v_2^{1,1})|\right) + min\left(|E(v_1^{1,2})|, |E(v_2^{1,2})|\right) + min\left(|E(v_1^{1,3})|, |E(v_2^{1,3})|\right)$$

$$= min\,(5,6) + min\,(4,5) + min\,(4,5) = 5 + 4 + 4.$$

If we consider the elements of the other partitions we obtain $Edges(G_1, G_2) = 10$; since $|V(G_1)| = 8$, $|E(G_1)| = 10$, $|V(G_2)| = 9$ and $|V(G_2)| = 15$, the degree similarity value of G_1 and G_2 $(simil^{ds}(G_1, G_2))$ is equal to 0.75.

3.1.3 Kernel Functions for Object Matching

As previously exposed `obj_match` is an NP-hard problem and its exact resolution is infeasible from a computational point of view also for a limited number of candidates in the case base. In the following we present an approximate evaluation based on kernel functions. Our kernel functions are inspired by Fröhlich et al.'s work [26, 27, 25] on kernel functions for molecular structures. Their goal is to define a kernel function which measures the degree of similarity between two chemical structures which are encoded as undirected labeled graphs. Our goal is to define a matching function among the objects of two planning problems encoded as directed graphs.

The intuition of these kernel functions is that the similarity between two graphs depends mainly on the matching of the pairs of vertices and the corresponding neighbourhoods; i.e., two graphs are more similar if the structural elements from both graphs fit together better and if these structural elements are connected in a more similar way in both graphs. Thereby, the graph properties of every single vertex and edge in both structures have to be considered. On a vertex level this leads to the idea of looking for those vertices in the two graphs that have the best match with regard to structural properties. In this way it is possible to consider not only direct neighbours, but also neighbours that are farther away, up to some maximal

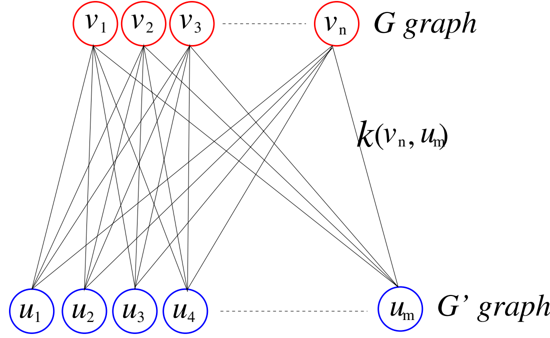


Figure 7: Possible assignments of vertices from G to those of Graph G'. The kernel function k measures the similarity of a pair of vertices (v, u). The goal is to find the optimal assignment of all vertices from G to those of G', which maximises the overall similarity score, i.e., the sum of edge weights in the bipartite graph, where each edge can be used at most once.

topological distance. We now want to associate each vertex in one graph to exactly one vertex in another graph such that the overall similarity is maximised. This problem can be modeled as a *maximum weight bipartite matching problem* where our kernel function determines the similarity between pairs of vertices. From this algorithm we know for each vertex in one graph to which vertex in the other graph it is assigned to. This guarantees us an easy way of interpreting and understanding our kernel function since a matching between an object of the planning problem Π and an object of Π' directly derives from a matching between a vertex of the planning encoding graph G_Π and a vertex of $G_{\Pi'}$.

Let us assume we have two graphs G and G', which have vertices $v_1, ..., v_n$ and $u_1, ..., u_m$ respectively. Let us further assume we have a kernel function k, which compares a pair of vertices v_i, u_j from both graphs, including information on their neighbourhoods. We now want to assign each vertex of the smaller of both graphs to exactly one vertex of the bigger one such that the overall similarity score, i.e., the sum of kernel values between individual vertices, is maximised. Mathematically this can be formulated as follows: let ζ denote a permutation of an n-subset of natural numbers $1, ..., m$, or a permutation of an m-subset of natural numbers $1, ..., n$, respectively. Then we are looking for the quantity

$$\mathcal{K}(G, G') = \begin{cases} max_\zeta \sum_{h=1}^{m} k(v_{\zeta(h)}, u_h) & \texttt{if } n \geq m \\ max_\zeta \sum_{h=1}^{n} k(v_h, u_{\zeta(h)}) & \texttt{otherwise} \end{cases} \tag{7}$$

$\mathcal{K}$ is a valid kernel function, as shown by [25], and hence a similarity measure for graphs. Implicitly it computes a dot product between two vector representations of graphs in some Hilbert space. Figure 7 illustrates this idea: between any pair of vertices from the upper and the lower structure there is some similarity, which can be thought as the edge weights of a bipartite graph. We now have to find a combination of edges such that the sum of edge weights is maximised. Thereby each edge can be used at most once. That means in the end exactly $min(n, m)$ out of $n \cdot m$ edges are used up.

We now have to define the kernel function k. For this purpose let us suppose we have two kernel functions k_v and k_e which compare the vertex and edge labels $\lambda(\cdot)$, respectively. In the following $e_j(v)$ denotes the j-th edge of the vertex v, while $n_j(v)$ denotes the node adjacent to the vertex v associated to the j-th edge $e_j(v)$. In the same way $e_j^{i/o}(v)$ denotes the j-th incoming/outgoing edge, while $n_j^{i/o}(v)$ denotes the direct predecessor/successor of the vertex v associated to the j-th incoming/outgoing edge $e_j^{i/o}(v)$. $\mathcal{N}(v_j)$ denotes the set of

vertices adjacent to the vertex v_j, while $E(v_j)$ denotes the set of incoming and outgoing edges of vertex v_j. Similarly $\mathcal{N}^{i/o}(v_j)$ denotes the set of direct predecessor/successor vertices of the vertex v_j.

Given a pairs of vertices v and u, we use the kernel function $k_v(v,u) = \gamma_0(v,u) \cdot \frac{|\lambda(v) \cap \lambda(u)|}{|\lambda(v) \cup \lambda(u)|}$, where $\gamma_0(v,u)$ is equal to 1.1 if u and v correspond to the same object (it is verified considering the names of the objects represented by vertices u and v), otherwise it is equal to 1.0 . The γ_0 coefficient has been introduced in our kernel functions in order to allow a greater stability in the activity assignment which is useful especially when human agents are handled by the planner. For example, in a logistic domain, we would like the drivers to be assigned the same set of activities as much as possible. While for pairs of edges we use $k_e(e_k(v), e_j(u)) = \frac{|\lambda(e_k(v)) \cap \lambda(e_j(u))|}{|\lambda(e_k(v)) \cup \lambda(e_j(u))|}$ if $e_k(v)$ and $e_j(u)$ are both incoming or outgoing edges of the vertices v and u, otherwise $k_e(e_k(v), e_j(u))$ is equal to 0. Formally, this corresponds to the multiplication by a so-called δ-kernel.

We define the *base kernel* between two vertices v and u, including their direct neighbourhoods as

$$k_{base}(v,u) = k_v(v,u) + \frac{1}{|\mathcal{N}^i(v)| \cdot |\mathcal{N}^i(u)|} \sum_{h,h'} k_v\left(n_h^i(v), n_{h'}^i(u)\right) \cdot k_e\left(e_h^i(v), e_{h'}^i(u)\right) +$$

$$+ \frac{1}{|\mathcal{N}^o(v)| \cdot |\mathcal{N}^o(u)|} \sum_{h,h'} k_v\left(n_h^o(v), n_{h'}^o(u)\right) \cdot k_e\left(e_h^o(v), e_{h'}^o(u)\right) \tag{8}$$

This means that the similarity between two vertices consists of two parts: first the similarity between the labels of the vertices and second the similarity of the neighbourhood structure. It follows that the similarity of each pair of neighbours $n_h^{i/o}(v), n_{h'}^{i/o}(u)$ is weighed by the similarity of the edges leading to them. The normalisation factors before the sums are introduced to ensure that vertices with a higher number of arcs do not automatically achieve a higher similarity. Hence we divide the sums by the number of the addends in them. It is also interesting to point out that the previous definition is just a classical convolution kernel as introduced by Haussler [41].

In the following we define a more accurate kernel R_1, which compares all the direct neighbours of the vertices (v,u) as the optimal assignment kernel between all the neighbours of v and u and the edges leading to them so that we can to improve the similarity values that can be obtained simply using k_{base}; more precisely:

$$R_1(v,u) = \begin{cases} \frac{1}{|E(v)|} max_\zeta \sum_{i=1}^{|E(v)|} k_v\left(n_{\zeta(i)}(v), n_i(u)\right) \cdot k_e\left(e_{\zeta(i)}(v), e_i(u)\right) & \texttt{if } |E(v)| \geq |E(u)| \\ \frac{1}{|E(u)|} max_\zeta \sum_{i=1}^{|E(u)|} k_v\left(n_i(v), n_{\zeta(i)}(u)\right) \cdot k_e\left(e_i(v), e_{\zeta(i)}(u)\right) & \texttt{otherwise} \end{cases} \tag{9}$$

Similarly to the graph kernel $\mathcal{K}$ of equation (7), the intuition behind this kernel function is that the similarity between two nodes depends not only on the nodes structure but also on the matching of the corresponding neighbourhoods; i.e., two nodes are more similar if their neighbourhood elements are connected in a more similar way in both nodes.

Theorem 4 *R_1 is a kernel function.*

Of course it would be beneficial not to consider the match of direct neighbours only, but also that of indirect neighbours and vertices having a larger topological distance. For this purpose we can evaluate R_1 not at (v,u) only, but also at all pairs of neighbours, indirect

neighbours and so on, up to some topological distance L. The weighed average of all these values corresponds to the weighed mean match of all indirect neighbours and vertices of a larger topological distance. Adding them to $k_v(v,u)$ leads to the following definition of the neighbourhood kernel $k_{\mathcal{N}}$:

$$k_{\mathcal{N}}(v,u) = k_v(v,u) + \gamma(1)R_1(v,u) + \sum_{l=2}^{L} \gamma(l)R_l(v,u) \tag{10}$$

where $\gamma(l)$ denotes a decay parameter which reduces the influence of neighbours that are at topological distance l;[7] similarly, R_l denotes the average of all R_1 evaluated for neighbours at distance l and it is computed from R_{l-1} via the recursive relation:

$$R_l(v,u) = \frac{1}{|\mathcal{N}^i(v)| \cdot |\mathcal{N}^i(u)|} \sum_{h,h'} R_{l-1}\left(n_h^i(v), n_{h'}^i(u)\right) \cdot k_e\left(e_h^i(v), e_{h'}^i(u)\right) +$$

$$+ \frac{1}{|\mathcal{N}^o(v)| \cdot |\mathcal{N}^o(u)|} \sum_{h,h'} R_{l-1}\left(n_h^o(v), n_{h'}^o(u)\right) \cdot k_e\left(e_h^o(v), e_{h'}^o(u)\right) \tag{11}$$

i.e., we can compute $k_{\mathcal{N}}(v,u)$ by iteratively revisiting all direct neighbours of v and u. The first addend in equation (10) takes into account the nodes (v,u), while the second addend takes into account the direct neighbours of (v,u) computing the $R_1(v,u)$ kernel function, then the next addend (i.e. $\gamma(2) \cdot R_2(v,u)$) computes the average of the match of all neighbours which have topological distance 2 by evaluating R_1 for all direct neighbours of (v,u). The fourth addend (i.e. $\gamma(3) \cdot R_3(v,u)$) does the same for all neighbours with topological distance 3. Finally, the last addend considers all neighbours which have topological distance L by evaluating R_1 for all neighbours at topological distance $L-1$. Furthermore we can easily show that:

Theorem 5 *Let $\gamma(l) = p^l$ with $p \in (0, 0.5)$. If there exists a $C \in \mathbb{R}^+$ such that $k_v(v_1, v_2) \le C$ for all v_1, v_2 and $k_e(e_1, e_2) \le 1$ for all e_1, e_2 then (eq. 10) converges for $L \to \infty$.*

To briefly summarise, our approach works as follows: we first compute the similarity of all vertex and edge features using the kernels k_v and k_e. Having these results we can compute the match of direct neighbours R_1 for each pair of vertices from both graphs by means of equation (9). From R_1 we can compute $R_2, ..., R_L$ by iteratively revisiting all direct neighbours of each pair of vertices and computing the recursive update formula (11). Having k_v and $R_1, ..., R_L$ directly gives us $k_{\mathcal{N}}$, the final similarity score for each pair of vertices, which includes structural information as well as neighbourhood properties. With $k_{\mathcal{N}}$ we can finally compute the optimal assignment kernel between two graphs G and G' using Equation (7). Moreover (7) can be calculated efficiently by using the *Hungarian method* [49] in $O(n^3)$, where n is the maximum number of vertices of both graphs.

The kernel functions k_{base} and $k_{\mathcal{N}}$ can be used in equation (7) to define the optimal assignment kernels $\mathcal{K}_{base}$ and $\mathcal{K}_{\mathcal{N}}$ respectively. Our optimal assignment kernel functions also define a permutation ζ that allows to easily determine the matching function μ associating each object in the smaller planning problem to exactly one object in the other planning problem.

The worst case scenario is determined when we consider two complete graphs with the same number of nodes n. In this case, equation (8) has computational complexity $O(n^2)$ and

[7] The $\gamma(\cdot)$ function used in our experimental evaluation is defined in section 4.1.

Algorithm RELAXEDPLAN
Input: a set of goal facts (G), the set of facts that are true in the current state (*INIT*),
a possibly empty relaxed plan (A)
Output: a relaxed plan $ACTS$ estimating a minimal set of actions required to achieve G
1. $G = G - INIT$; $ACTS = A$
2. $F = \bigcup_{a \in ACTS} Add(a)$
3. `while` $G - F \neq \emptyset$
4. g = "a fact in $G - F$"
5. $bestact = Bestaction(g)$
6. $Rplan$ = RelaxedPlan$(Pre(bestact), INIT, ACTS)$
7. $ACTS = Aset(Rplan) \cup \{bestact\}$
8. $F = \bigcup_{a \in ACTS} Add(a)$
9. `return` $ACTS$

Figure 8: Algorithm for computing a relaxed plan estimating a minimal set of actions required to achieve a set of facts G from the state *INIT*. $Bestaction(g)$ is the action that is heuristically chosen to support g as described in [31].

Algorithm EVALUATEPLAN
Input: a planning problem $\Pi = (I, G)$, an input plan π and an adaptation cost limit $Climit$
Output: a relaxed plan to adapt π in order to resolve Π
1. *CState*= I; $Rplan = \emptyset$
2. `forall` $a \in \pi_i$ `do`
3. `if` $\exists f \in Pre(a)$ s.t $f \notin CState$ `then`
4. $Rplan$ =RELAXEDPLAN$(Pre(a), CState, Rplan)$
5. `if` $|Rplan| > Climit$ `then`
6. `return` $Rplan$
7. $CState = (CState / Del(a)) \cup Add(a)$
8. `if` $\exists g \in G$ s.t $g \notin CState$ `then`
$Rplan$ =RELAXEDPLAN$(G, CState, Rplan)$
9. `return` $Rplan$

Figure 9: Algorithm to evaluate the ability of π to solve the planning problem Π

since it has to be examined n^2 time in equation (7) we obtain a computational complexity of $n^2 \cdot O(n^2) + O(n^3) = O(n^4)$ for the $\mathcal{K}_{base}$ kernel function. Moreover, function R_1 has a computational complexity of $O(n^3)$ since we use the Hungarian method for its computation; similarly to k_{base} the R_l function has computational complexity $O(n^2)$ and, limiting the $k_{\mathcal{N}}$ evaluation to a topological distance L which is a polynomial of n, we obtain a computational complexity for $k_{\mathcal{N}}$ equal to $1 + O(n^3) + O(n) \cdot O(n^2) = O(n^3)$. The $k_{\mathcal{N}}$ kernel function has to be examined n^2 times in equation (7), determining a computational complexity for $\mathcal{K}_{\mathcal{N}}$ equal to $n^2 \cdot O(n^3) + O(n^3) = O(n^5)$.

As it will be described in the next section, in OAKPLAN both $\mathcal{K}_{base}$ and $\mathcal{K}_{\mathcal{N}}$ have been used; $\mathcal{K}_{base}$, which has a lower computational complexity, has been used in order to prune unpromising case base candidates. It allows to define a first matching function μ_{base} and the corresponding similarity function $simil_{\mu_{base}}$, as described in the following section. On the other hand $\mathcal{K}_{\mathcal{N}}$ has been used to define a final matching function μ and the corresponding similarity function $simil_{\mu}$.

Algorithm RETRIEVEPLAN
Input: a planning problem Π, a case base $C = \langle \Pi_i, \pi_i \rangle$
Output: candidate plan for the adaptation phase

```
1.1.  π_R = EVALUATE_PLAN(Π, EMPTY_PLAN, ∞)
1.2.  Define the set of initial relevant facts of Π using π_R: I_{π_R} = I ∩ ⋃_{a∈π_R} pre(a)
1.3.  Compute the Planning Encoding Graphs E_Π and E_{Π_R} of Π(I,G) and Π_R(I_{π_R},G)
          respectively, and the degree sequences L^j_{Π_R}
1.4.  forall Π_i ∈ C do
1.5.      simil_i = simil^{ds}(E_{Π_i}, E_{Π_R})
1.6.      push((Π_i, simil_i)), queue)
1.7.      best_ds_simil = max(best_ds_simil, simil_i)
2.1.  forall (Π_i, simil_i) ∈ queue  s.t.  best_ds_simil − simil_i ≤ limit do*
2.2.      Load the Planning Encoding Graph E_{Π_i} and compute the matching function μ_base
          using K_base(E_{Π_i}, E_Π)
2.3.      push((Π_i, μ_base), queue_1)
2.4.      best_μ_base_simil = max(best_μ_base_simil, simil_{μ_base}(Π_i, Π))
3.1.  forall (Π_i, μ_base) ∈ queue_1  s.t.  best_μ_base_simil − simil_{μ_base}(Π_i, Π) ≤ limit do
3.2.      Compute the matching function μ_N using K_N(E_{Π_i}, E_Π)
3.3.      if  simil_{μ_N}(Π_i, Π) ≥ simil_{μ_base}(Π_i, Π)  then  μ_i = μ_N
          else μ_i = μ_base
3.4.      push((Π_i, μ_i), queue_2)
3.5.      best_simil = max(best_simil, simil_{μ_i}(Π_i, Π))
4.1.  best_cost = α_G · |π_R|;   best_plan =EMPTY_PLAN
4.2.  forall (Π_i, μ_i) ∈ queue_2  s.t.  best_simil − simil_{μ_i}(Π_i, Π) ≤ limit do
4.3.      Retrieve π_i from C
4.4.      cost_i = |EVALUATEPLAN(Π, μ_i(π_i), best_cost · simil_{μ_i}(Π_i, Π))|
4.5.      if best_cost · simil_{μ_i}(Π_i, Π) > cost_i  then
4.6.          best_cost = cost_i/simil_{μ_i}(Π_i, Π)
4.7.          best_plan = μ_i(π_i)
5.1.  return best_plan
```

* We limited this evaluation to the best 700 cases of *queue*.

Figure 10: Algorithm to find a suitable plan for the adaptation phase from a set of candidate cases or the empty plan (in case the "generative" approach is considered more suitable).

3.2 Plan Evaluation Phase

The purpose of plan evaluation is that of defining the capacity of a plan π to resolve a particular planning problem. It is performed by simulating the execution of π and identifying the unsupported preconditions of its actions; in the same way the presence of unsupported goals is identified. The plan evaluation function could be easily defined as the number of inconsistencies in the current planning problem. Unfortunately this kind of evaluation considers a uniform cost in order to resolve the different inconsistencies and this assumption is generally too restrictive. Then our system considers a more accurate inconsistency evaluation criterion so as to improve the plan evaluation metric. The inconsistencies related to unsupported facts are evaluated by computing a relaxed plan starting from the corresponding state and using the RELAXEDPLAN algorithm in LPG [31]. The number of actions in the relaxed plan determines the difficulty to make the selected inconsistencies supported; the number of actions in the final relaxed plan determines the accuracy of the input plan π to solve the corresponding planning problem.

Figure 8 describes the main steps of the RELAXEDPLAN function.[8] It constructs a relaxed plan through a backward process where $Bestaction(g)$ is the action a' chosen to achieve a (sub)goal g, and such that: (i) g is an effect of a'; (ii) all preconditions of a' are reachable from the current state *INIT*; (iii) the reachability of the preconditions of a' requires a minimum number of actions, evaluated as the maximum of the heuristically estimated minimum number of actions required to support each precondition p of a' from *INIT*; (iv) a' subverts a minimum number of supported precondition nodes in $\mathcal{A}$ (i.e., the size of the set *Threats*(a') is minimal).

Figure 9 describes the main steps of the EVALUATEPLAN function. For all actions of π (if any), it checks if at least one precondition is not supported. In this case it uses the RELAXEDPLAN algorithm (step 4) so as to identify the additional actions required to satisfy the unsupported preconditions. If $Rplan$ contains a number of actions greater than $Climit$ we can stop the evaluation, otherwise we update the current state *CState* (step 7). Finally we examine the goal facts G (step 8) to identify the additional actions required to satisfy them, if necessary.

Figure 10 describes the main steps of the retrieval phase. We initially compute a relaxed plan π_R for Π (step 1.1) using the EVALUATEPLAN function on the empty plan which is needed so as to define the *generation* cost of the current planning problem Π (step 4.1)[9] and an *estimate* of the initial state relevant facts (step 1.2). In fact we use the relaxed plan π_R so as to filter out the irrelevant facts from the initial state description.[10] This could be easily done by considering all the preconditions of the actions of π_R:

$$I_{\pi_R} = I \cap \bigcup_{a \in \pi_R} pre(a).$$

Then in step 1.3 the Planning Encoding Graph of the current planning problem Π and the degree sequences that will be used in the screening procedure are precomputed. Note that the degree sequences are computed considering the Planning Encoding Graph $\mathcal{E}_{\Pi_R}$ of the planning problem $\Pi_R(I_{\pi_R}, G)$ which uses I_{π_R} instead of I as initial state. This could be extremely useful in practical applications when automated tools are used to define the initial state description without distinguishing among relevant and irrelevant initial facts.

Steps 1.4 – 1.7 examine all the planning cases of the case base so as to reduce the set of candidate plans to a suitable number. It is important to point out that in this phase it is not necessary to retrieve the complete planning encoding graphs of the case base candidates $G_{\Pi'}$ but only their sorted degree sequences $L^i_{\Pi'}$ which are precomputed and stored in the case base. On the contrary the planning encoding graph and the degree sequences of the input planning problem are only computed in the initial preprocessing phase (step 1.3).

All the cases with a similarity value sufficiently close[11] to the best degree sequences similarity value ($best_ds_simil$) are examined further on (steps 2.1–2.4) using the $\mathcal{K}_{base}$ kernel function. Then all the cases selected at steps $2.x$ with a similarity value sufficiently close to the best $simil_{\mu_{base}}$ similarity value ($best_\mu_{base}_simil$) (step 3.1) are accurately evaluated using the $\mathcal{K}_{\mathcal{N}}$ kernel function, while the corresponding $\mu_{\mathcal{N}}$ function is defined at step 3.2. In steps 3.3–3.5 we select the best matching function found for Π_i and the best similarity value found until now.

[8] RELAXEDPLAN is described in detail in [31]. It also computes an estimation of the earliest time when all facts in G can be achieved, which is not described in this paper for sake of simplicity.

[9] The α_G coefficient gives more or less importance to plan adaptation vs plan generation; if $\alpha_G > 1$ then it is more likely to perform plan adaptation than plan generation.

[10] In the relaxed planning graph analysis the negative effects of the domain operators are not considered and a solution plan π_R of a relaxed planning problem can be computed in polynomial time [42].

[11] In our experiments we used $limit = 0.1$.

We use the relaxed plan π_R in order to define an estimate of the *generation* cost of the current planning problem Π (step 4.1). The `best_cost` value allows to select a good candidate plan for adaptation (which could also be the empty plan). This value is also useful during the computation of the adaptation cost through EVALUATEPLAN, in fact if such a limit is exceeded then it is wasteful to use CPU time and memory to carry out the estimate and the current evaluation could be terminated. The computation of the adaptation cost of the empty plan allows to choose between an *adaptive* approach and a *generative* approach, if no plan gives an adaptation cost smaller than the empty plan.

For all the cases previously selected with a similarity value sufficiently close to $best_simil$ (step 4.2) the adaptation cost is determined (step 4.4). If a case of the case base determines an adaptation cost which is lower than `best_cost`$\cdot simil_{\mu_i}(\Pi_i, \Pi)$ then it is selected as the current best case and also the `best_cost` and the `best_plan` are updated (steps 4.5–4.7). Note that we store the encoded plan $\mu_i(\pi_i)$ in `best_plan` since this is the plan that can be used by the adaptation phase for solving the current planning problem Π. Moreover we use the $simil_{\mu_i}(\Pi_i, \Pi)$ value in steps 4.4 – 4.6 as an *indicator* of the effective ability of the selected plan to solve the current planning problem maintaining the original plan structure and at the same time obtaining low distance values.

3.3 Plan Adaptation

As previously exposed, the plan adaptation system is a fundamental component of a case-based planner. It consists in reusing and modifying previously generated plans to solve a new problem and overcome the limitation of planning from scratch. As a matter of fact, in planning from scratch if a planner receives exactly the same planning problem it will repeat the very same planning operations. In our context the input plan is provided by the plan retrieval phase previously described; but the applicability of a plan adaption system is more general. For example the need for adapting a precomputed plan can arise in a dynamic environment when the execution of a planned action fails, when the new information changing the description of the world prevents the applicability of some planned actions, or when the goal state is modified by adding new goals or removing existing ones [22, 31].

Different approaches have been considered in the literature for plan adaptation; strategies vary from attempting to reuse the structure of an existing plan by constructing bridges that link together the fragments of the plan that fail in the face of new initial conditions [37, 38, 39, 44], to more dynamic plan modification approaches that use a series of plan modification operators to attempt to repair a plan [53, 80]. From a theoretical point of view, in the worst case, plan adaptation is not more efficient than a complete regeneration of the plan [61] when a conservative adaptation strategy is adopted. However adapting an existing plan can be in practice more efficient than generating a new one from scratch, and, in addition, this worst case scenario does not always hold, as exposed in [3] for the Derivation Analogy adaptation approach. Plan adaptation can also be more convenient when the new plan has to be as “similar” as possible to the original one.

Our work uses the LPG-adapt system given its good performance in many planning domains but other plan adaptation systems could be used as well. LPG-adapt is a local-search-based planner that modifies plan candidates incrementally in a search for a flawless candidate. We describe the main components of the LPG-adapt system in the following section. It is important to point out that this paper relates to the description of a new efficient case-based planner and in particular to the definition of effective plan matching functions, no significant changes were made to the plan adaptation component (for a detailed description of it see [22]).

3.3.1 Local Search Techniques for Plan Adaptation

Here we present the search techniques used in LPG-adapt. We start with a description of the general local search scheme in the space of Action graphs (A-graph)[31]. Then we concentrate on the heuristics of LPG-adapt and on its methods for maintaining the A-graph representation during the search, for computing the solution plans and for deriving good quality plans incrementally.

An action graph (A-graph) $\mathcal{A}$ of a planning graph $\mathcal{G}$ is a subgraph of $\mathcal{G}$ such that, if a is an action node of $\mathcal{G}$ in $\mathcal{A}$, then also the fact nodes of $\mathcal{G}$ corresponding to the preconditions and positive effects of a are in $\mathcal{A}$, together with the edges connecting them to a [34]. The general scheme for searching for a solution graph (a final state of the search) consists of two main steps. The first step is an initialisation of the search in which we construct an initial A-graph. The second step is a local search process in the space of all A-graphs, starting from the initial A-graph. We can generate an initial A-graph in several ways; in our context the A-graph is obtained from an existing plan given as input to the adaptation process by the retrieval phase. Further details on the initialisation step can be found in earlier papers on planning through local search and action graphs [31, 34, 36]. Once we have computed an initial A-graph, each basic search step selects an inconsistency in the current A-graph. If this is an unsupported fact node, then in order to resolve (eliminate) it, we can either add an action node that supports it, or we can remove an action node that is connected to that fact node by a precondition edge. The strategy for selecting the next inconsistency to handle may have a significant impact on the overall performance and it has been extensively studied in the context of causal-link partial-order planning [33, 65]; the default strategy corresponds to selecting the lowest level inconsistency.

Given an action graph $\mathcal{A}$ and an inconsistency σ in $\mathcal{A}$, the *neighbourhood* $N(\sigma, \mathcal{A})$ of σ in $\mathcal{A}$ is the set of A-graphs obtained from $\mathcal{A}$ by applying a graph modification that resolves σ. The two basic modifications consist of an extension of the A-graph to include a new action node, or a reduction of the A-graph to remove an action node (and the relevant edges). At each step of the local search scheme, the elements of the neighbourhood are evaluated according to a function estimating their quality, and an element with the best quality is then chosen as the next possible A-graph (search state). The quality of an A-graph depends on a number of factors, such as the number of inconsistencies, the estimated number of search steps required to resolve them and the overall cost of the actions in the represented plan.[12]

The local search strategy used by LPG-adapt is Walkplan a local search algorithm with restarts which is similar to the heuristic used by Walksat [47, 72] to solve boolean satisfiability problems. In Walkplan, see Figure 11, the best element in the neighbourhood is the A-graph which has the *lowest decrease of quality* with respect to the current A-graph, i.e., it does not consider possible improvements. Like Walksat, our strategy uses a *noise parameter* p. Given an A-graph $\mathcal{A}$ and an inconsistency σ, if there is a modification for σ that does not decrease the quality of $\mathcal{A}$, then this modification is performed, and the resulting A-graph is chosen as the next A-graph; otherwise, with probability p one of the graphs in $N(\sigma, \mathcal{A})$ is chosen randomly, and with probability $1 - p$ the next A-graph is chosen according to the minimum value of the evaluation function. If a solution graph is not reached after a certain number of search steps, the current A-graph is reinitialised and the search is repeated up to a user-defined maximum number of times.

[12] For simple STRIPS domains the execution cost of the plan is measured in terms of the number of actions (i.e., each action has cost 1), while the plan makespan is ignored.

Algorithm WALKPLAN($\Pi(I,G)$,max_steps,$max_restarts$,p)

Input: a planning problem Π, the maximum number of search steps max_steps,
the maximum number of search restarts $max_restarts$, a noise factor p ($0 \leq p \leq 1$)
and an input plan π_0 for the adaptation process.

Output: a solution graph representing a plan solving Π or `fail`.

1. **for** i = 1 *to* $max_restarts$ **do**
2. $\mathcal{A}$ = "an initial A-graph derived from π_0"
3. **for** j = 1 **to** max_steps **do**
4. **if** $\mathcal{A}$ is a solution graph **then**
5. **return** $\mathcal{A}$
6. σ = "an inconsistency in $\mathcal{A}$"
7. $N(\sigma,\mathcal{A})$ = "neighbourhood of $\mathcal{A}$ for σ"
8. **if** $\exists\, \mathcal{A}' \in N(\sigma,\mathcal{A})$ such that the quality of $\mathcal{A}'$ is not worse than the quality of $\mathcal{A}$
9. **then** $\mathcal{A}$ = $\mathcal{A}'$ (if there is more than one $\mathcal{A}'$-graph, choose randomly one)
10. **else if** `random` < p **then**
11. $\mathcal{A}$ = "an element of $N(\sigma,\mathcal{A})$ randomly chosen"
12. **else** $\mathcal{A}$ = " best element in $N(\sigma,\mathcal{A})$"
13. **return** `fail`.

Figure 11: General scheme of Walkplan with restarts. `random` is a randomly chosen value between 0 and 1. The quality of an action graph in the neighbourhood is measured using an evaluation function estimating the cost of the graph modification used to generate it from the current action graph.

The behaviour of LPG-adapt is controlled by an evaluation function that is used to select between different candidates in the neighbourhood. The elements of the neighbourhood are evaluated according to an *action evaluation function* E. This function is used to estimate the cost of either adding ($E(a)^i$) or of removing ($E(a)^r$) an action node a to the current partial plan π undergoing adaptation [31]. In order to properly manage adaptation planning problems the function E has been extended to include an additional evaluation term that takes into account the actions that do not belong to the input plan π_0. The idea is that of penalising the insertion and removal of the actions that increase the distance of the current partial plan π under adaptation from the input plan π_0. The function E is fully described in [31], while the corresponding extensions for plan adaptation problems are described in [22].

Briefly, $\mathsf{Eval}(a)^i$ returns a relaxed plan, π_r, that contains a minimal set of actions for achieving the unsupported preconditions of a and the set of preconditions of other actions in the current partial plan that would become unsupported by adding a to it. Similarly, $\mathsf{Eval}(a)^r$ returns a minimal set of actions required to achieve the preconditions that would become unsupported if a was removed from π, The relaxed subplans used in $\mathsf{Eval}(a)^{i/r}$ are computed by the RELAXEDPLAN algorithm. In the adaptation context, the last term of the evaluation function E measures the distance between $\pi_0 - \pi$, where π is the current partial plan (before modification), and the relaxed plan π_r built by Eval^i or Eval^r. This is calculated to give an estimate of the expected distance between the finished plan we can expect to be constructed from the result of modifying π with action a. We use the relaxed plan π_r to represent the portion on the plan that is likely to be added into π in the subsequent planning process to arrive at a complete plan.

RelaxedPlan constructs the relaxed plan π_r through a backward process using the function $Bestaction$ in order to select the actions to insert in π_r [31]. In the adaptation context we ex-

tended $Bestaction$ introducing a penalisation coefficient $\Delta_{best}(a)$ that evaluates the insertion in π_r of an action a considering the elements of π_0, π and π_r itself:

$$\Delta_{best}(a) = \begin{cases} 1 & \texttt{if } a \notin (\pi_0 - \pi) - \pi_r \\ 0 & \texttt{otherwise} \end{cases} \tag{12}$$

i.e we penalise the evaluation of a if it is not an action of π_0 or it has already been selected for the insertion in π or π_r.

In general, E consists of three weighed terms, evaluating three aspects of the quality of the current plan that are affected by the addition or removal of a:[13]

$$E(a)^{i/r} = \frac{1}{max_S} \cdot Search_cost(a)^{i/r} + \frac{\mu_E}{max_E} \cdot Execution_cost(a)^{i/r} + \\ + \frac{\mu_\Delta}{max_\Delta} \cdot |\mathsf{Eval}^{i/r}(a) \cap (\pi_0 - \pi)| \tag{13}$$

The first two terms of the two forms of the function E are unchanged from the standard behaviour of LPG [31]. The first term of E estimates the minimum number of search steps required to repair the flaws introduced by the insertion/remotion of a in/from the current action graph; where the second estimates the total cost of the new actions defining the search cost previously described, while the cost of an action is defined by the plan metric function and is equal to 1 for pure STRIPS planning problems (see [31] for a detailed description). The third term in the expressions for E is the adaptation term, estimating how plan modification will affect the increase in distance from π_0. This term is only used for adaptation planning problems.

The coefficients of each of these terms are used to normalise them, and to weigh their relative importance. Thus μ_E and μ_Δ are non-negative coefficients that weigh the relative importance of the execution and "difference" costs, respectively. Their values can be set by the user, or they can be automatically derived from the expression defining the plan metrics in the formalisation of the problem. The factors $1/max_E$, $1/max_S$ and $1/max_\Delta$ are used to normalise the terms of E to a value that is (upper) bounded by 1. max_E and max_Δ are respectively the maximum value of the second and third term of E over all elements of the neighbourhood, multiplied by the number κ of inconsistencies in the current partial plan. The term max_S is defined as the maximum value of $Search_cost$ over all possible action insertions or removals that eliminate the inconsistency under consideration.[14] Without this normalisation the second term of E could be much higher than the other ones. This would guide the search towards good quality plans without paying sufficient attention to their validity and similarity with respect to π_0. On the contrary, we would like the search to give more importance to reducing the search cost, rather than to optimising the quality of the plan, especially when the current partial plan contains many inconsistencies.

The LPG-adapt planner has an anytime behaviour, in fact it can produce a succession of valid plans, where each plan is an improvement of the previous ones in terms of its "adaptation quality". This is a process that incrementally improves the *total* quality of the plans, which can be stopped at any time to give the best plan computed so far (π_{best}). Each time we start a new search, the input plan π_0 provided by the retrieval phase is used to initialise the data

[13]The LPG system considers an additional term related to the temporal execution cost that we have not examined for sake of simplicity.

[14]The role of κ is to decrease the importance of the second and third optimisation terms when the current plan contains many inconsistencies, and to increase them when the search approaches a valid plan.

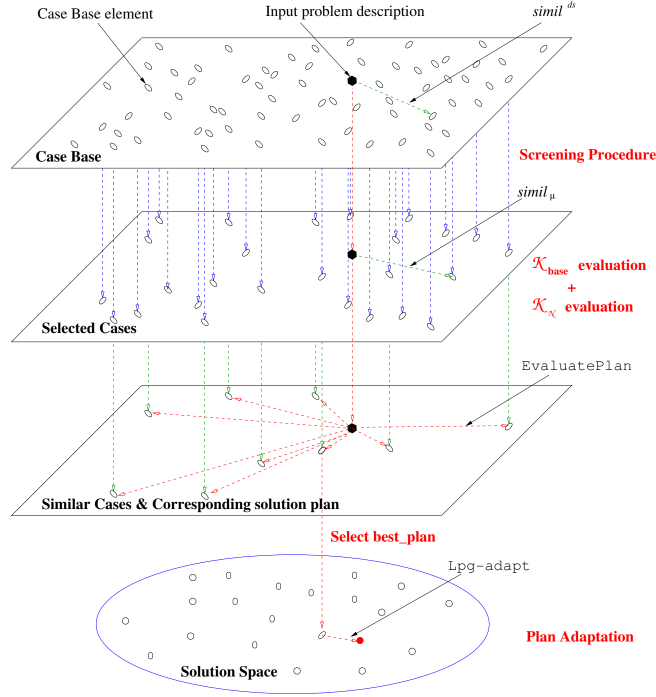


Figure 12: Retrieval and Adaptation phases.

structures. Furthermore, during search some random inconsistencies are forced in the plan currently undergoing adaptation when a valid plan that does not improve π_{best} is reached. This is a mechanism for leaving local optima. In our adaptation context the total quality of a plan is derived considering a weighed evaluation of the metric quality and the "distance" of π from π_0:

$$Q_{\pi_0}(\pi, \pi_{best}) = \alpha \frac{Quality(\pi)}{Quality(\pi_{best})} + (1 - \alpha) \frac{|\pi \cap \pi_0|}{|\pi_{best} \cap \pi_0|} \tag{14}$$

the rationale of this choice is that of trying to balance the quality of the plan produced and the distance from the input plan π_0.[15] The $Q_{\pi_0}(\cdot, \cdot)$ value is used to choose between the new valid plan π and the best plan π_{best} found until now.

Figure 12 visualises the main steps of the retrieval and adaptation phases. The screening procedure computes the $simil^{ds}$ distance function so as to filter out the dissimilar cases and reduce the number of planning cases that have to be examined accurately up to a suitable number. Then we compute the $\mathcal{K}_{base}$ and $\mathcal{K}_{\mathcal{N}}$ kernel functions and the corresponding μ_{base} and $\mu_{\mathcal{N}}$ matching functions to define the corresponding $simil_\mu$ similarity values and identify the candidate plans for the adaptation phase. These plans are evaluated using the EVALUATEPLAN procedure and if the best case has an adaptation cost inferior to the generation cost then it is used by LPG-adapt in order to find a suitable solution, otherwise a complete generation phase is performed providing the empty plan to LPG-adapt.

[15] In our tests we used $\alpha = 0.5$.

Algorithm INSERT_CASE($Case$, π, $\Pi(I, G)$)
Input: a case base $Case$, a solution plan π for planning problem Π with initial state I and goal state G.
Output: insert the planning case in $Case$ if not present.
1. Define the set of initial state relevant facts I_π of Π using the input plan π
2. Compute the Planning Encoding Graph $\mathcal{E}_\pi$ of $\Pi_\pi(I_\pi, G)$
3. `for` each case $(\Pi_i, \pi_i) \in Case$
4. Compute the matching function μ_i using $\mathcal{K}_\mathcal{N}(\mathcal{E}_{\Pi_i}, \mathcal{E}_\pi)$
5. `if` $complete_simil_{\mu_i}(\Pi_i, \Pi_\pi) = 1 \wedge |\pi_i| \leq |\pi|$ `then`
6. `return` FALSE;
7. `enfor`
8. Insert the planning problem $\Pi_\pi(I_\pi, G)$, its solution plan π, the Planning Encoding Graph $\mathcal{E}_\pi$ and the data structures for the screening procedure in $Case$
9. `return` TRUE;

Figure 13: High-level description of INSERT_CASE.

3.4 Plan Revision

Any kind of planning system that works in dynamic environments has to take into account failures that may arise during plan generation and execution. In this respect case-based planning is not an exception; this capability is called plan revision and it is divided in two subtasks: evaluation and repair. The evaluation step verifies the presence of failures that may occur during plan execution when the plan does not produce the expected result. When a failure is discovered, the system may react by looking for a repair or aborting the plan. In this first hypothesis the LPG-adapt system is invoked on the remaining part of the plan; in the latter hypothesis the system repeats the CBP cycle so as to search a new solution.

3.5 Case Base Update

After finding the plan from the library and after repairing it with the LPG-adapt techniques the solution plan can be inserted into the library or be discarded. The case base maintenance is clearly important for the performance of the system and different strategies have been proposed in the literature [75, 79]. Furthermore our attention has been oriented towards the improvement of the competence of the case base; a solved planning problem is not added to the case base only if there is a case that solves the same planning problem with a solution of a better quality.[16] Such a check has been introduced to the end of keeping only the best solution plans for certain kinds of problems in the library as there can be different plans that can solve the same problems with different sets of actions.

Figure 13 describes the main steps of the function used to evaluate the insertion of a planning problem Π solved in the case base. First of all we compute the set of initial state relevant facts I_π using the input plan π; this set corresponds to a subset of the facts of I relevant for the execution of π. It can be easily computed, as described in section 3.2, using the preconditions of the actions in π:

$$I_\pi = I \cap \bigcup_{a \in \pi} pre(a).$$

[16]In our experiments we have considered only the number of actions for distinguishing between two plans that solve the same planning problem but other and more accurate metrics could be easily added, i.e. consider for example actions with not unary costs.

Note that I_π identifies all the facts required for the execution of the plan π and that this definition is consistent with the procedure used in the RETRIEVEPLAN algorithm for the relaxed plan π_R.[17] Then we compute the Planning Encoding Graph $\mathcal{E}_\pi$ of the new planning problem $\Pi_\pi(I_\pi, G)$ having I_π as initial state instead of I. At steps 3–6 the algorithm examines all the cases of the case base and if it finds a case that solves Π with a plan of a better quality with respect to π then it stops and exits. In order to do so we use the similarity function $complete_simil_{\mu_i}$, described in section 3.1.1, which compares all the initial and goal facts of two planning problems. Otherwise if there is no case that can solve Π_π with a plan of a better quality with respect to π then we insert the solved problem in the case base. As we can observe at step 8, a planning case is made up not only by Π_π and π, but also other additional data structures are precomputed and added to the case base so that their recomputation during the *Retrieval Phase* can be avoided.

An extension to the system could be that of developing a more thorough evaluation of the competence of the library and developing a case base Maintenance Policy as described in [79], which is left for future work.

4 Experimental Results

In this section, we present an experimental study aimed at testing the effectiveness of OAKPLAN in a number of standard benchmark domains. In the first subsection, we describe the experimental settings and then, in the second subsection, we present the system overall results. In particular we examine the behaviour of OAKPLAN when different matching functions are used and we experimentally analyse the impact of the size of the case base (number of planning cases) in the overall performance of the system. In the third subsection, we experimentally investigate the similarity values of our matching functions when the case base objects are progressively renamed. Finally, we compare our planner with four state-of-the-art planners.[18]

4.1 Experimental Settings

Here we present and discuss the general results for the experimental comparison, moreover we examine the importance of the matching functions and the size of the case base in the overall performance of the system. In Appendix C we provide the plots of all the experiments.

OAKPLAN is written in C++ and uses the SQLite3 library[19] for storing and retrieving the data structures of the case base and the VFLIB library [17] so as to create and elaborate our graph data structures.[20] The OAKPLAN code and the benchmark planning problems are available from the OAKPLAN website `http://pro.unibz.it/staff/iserina/OAKplan/`.

[17] We have used this simple definition instead of using the causal links in π in order to compute the set of relevant facts since it allows to obtain slightly better performance than the corresponding version based on causal links.

[18] We compared OAKPLAN with the following planners:

- METRIC-FF winner of the 2nd IPC;
- LPG winner of the 3rd IPC;
- DOWNWARD 1st Prize, Suboptimal Propositional Track 4th IPC;
- SGPLAN-IPC5 winner of the 5th IPC.

[19] SQLite is a software library that implements a self-contained, serverless, zero-configuration, transactional SQL database engine; further information can be found at http://www.sqlite.org/.

[20] Although the VFLIB library can solve subgraph-isomorphism problems, its use turns out to be computationally too expensive and the kernel functions described in section 3.1.3 are used instead.

We have conducted all the experimental tests using an AMD Sempron(tm) Processor 3400+ (with an effective 2000 MHz rating) with 1 Gbyte of RAM. Unless otherwise specified, the CPU-time limit for each run is 10 minutes for OAKPLAN and 30 minutes for all the other planners, after which termination is forced.[21] In the following tests the maximum topological distance L considered for the computation of the $k_{\mathcal{N}}$ kernel function in equation (10) is set to half of the number of nodes of the smaller of the two graphs examined $\left(L = \left\lfloor \frac{min(|V_1|,|V_2|)}{2} \right\rfloor\right)$; since this value is sufficiently small to avoid convergence problems the $\gamma(l)$ coefficient of equation (10) is set equal to $\gamma(l) = \left(1 - \frac{1}{L}\right)^l$.

Since our planner and LPG use a randomised search algorithm, the corresponding results are median values over five runs for each problem considered. Moreover, since OAKPLAN and LPG are incremental planners we evaluate their performance with respect to three different main criteria: CPU-time required to compute a valid plan, the plan *stability* [22] of the generated plans with respect to the corresponding solutions of the target plans and the quality of the best plan generated within the given CPU-time limit.

In these tests the solution plans of the planning cases are obtained by using the *domain dependent* planner TLPLAN [4] unless otherwise specified. TLPLAN is a planning system that utilises domain specific search control information to guide simple forward chaining search, and it is the winner of the Hand Coded track of the 3rd IPCs. Its use allows us to use a high quality input plan with comparatively low investment of initial computation time. Using a plan from a different planner also ensures that we are not artificially enhancing stability by relying on the way in which the planner explores its search space.

Often the best quality solution computed by OAKPLAN requires much more time than the first (unoptimised) solution. However, it should be noted that when a planning problem can be solved by a satisficing planner within a reasonable amount of CPU-time (e.g., 10 CPU-minutes), the quality of the computed solutions is practically very important, and for many applications it can be more meaningful than CPU-time. Moreover, Metric-LPG generates a sequence of valid plans with increasing quality, and often it produces good quality *intermediate* plans much more quickly than the best (last) plan. In Section 4.4, we give empirical evidence for this behaviour. In general, in Metric-LPG there is a tradeoff between plan quality and CPU-time that can be exploited by the user, depending on whether for the application domain under consideration planning speed is more important than plan quality, or vice versa.

Our tests are conducted on a series of variants of problems from different domains:

- BlocksWorld and Logistics Additionals (2nd International Planning Competition),
- DriverLog and ZenoTravel Strips (3rd IPC),
- Rovers-IPC5 and TPP Propositional (5th IPC).

These tests are generally performed by taking six problems from the benchmark test suite in each case and then methodically generating a series of variants for those problems for a total of 216 planning problems for each domain. In each case, we take three medium size problems and three largest size problem instances as base problems,[22] so as to make the question of case-

[21] We use 10 minutes for OAKPLAN since many experimental tests have been conducted on it and a CPU limit of 30 minutes has turned out to be computationally too expensive. However, as we will see in Table 1, the CPU limit of 10 minutes is enough for OAKPLAN to solve more than 95% of the problems attempted and additional CPU-time could only modify the number of solved problems, the plan quality and the difference values obtained slightly.

[22] Specifically the base problems considered are for

- BlocksWorld Additionals: probBLOCKS-40-0, probBLOCKS-60-0, probBLOCKS-80-0, probblocks-100-0, probblocks-120-1 and probblocks-140-1;

based planning versus replanning an interesting one.[23] The variant problems are generated by modifying the initial and goal facts of the original problem. These modifications are performed randomly, although the number of modifications is increased systematically: we consider from zero to five modifications of the goal set and from zero to five modifications of the initial state.[24]

Although we use only six base problems in each domain, we generate a large number of variants and we consider problems from several domains, so these results can be considered representative of the behaviour of the system for other similarly sized base problems. To confirm that the results are not an artifact of the particular problem instances chosen, we adopt a different problem generation strategy for creating problem instances in the Logistics domain. Thus we select problems randomly from the benchmark suites considering the "Additionals" planning problems created in the 2nd IPC for the Domain Dependent planners, distributed across the smallest and the largest problem instances, and generate variant problems for each case. We use the same scheme as above to determine the combination of modification values for the initial state and goals, but select the base problem to apply the modifications randomly. The list of base problems selected for each domain and the random modifications applied are described in Appendix B.

For each of the benchmark domains we build a case base library used by OAKPLAN. All the problems generated in the different IPCs belong to these libraries. Using the problem generators provided by the IPC organisers, a number of planning problems, with the same features as the IPC planning problems considered, are generated and added to the libraries, for a total of 10000 planning problems for each of the benchmark domains considered except TPP where we only use the original IPC planning problems since it is not possible to use TLPLAN to solve the planning problems of this domain; then we use SGPLAN-IPC5 to determine the solutions of the TPP planning cases.

In the following we report the summary results obtained by OAKPLAN considering (1) case base libraries whose cases use the same objects as the planning problems in the test set, so as to verify the system behaviour when the matching function of OAKPLAN could be simply obtained by using the identity function (we add the suffix "Ons", which stands for *O*riginal object *n*ame*s*, to the corresponding results as in OAKPLAN-Ons); (2) case base libraries where the object names of the planning cases are randomly modified with respect to the objects of the planning problems in the test set, to verify the system behaviour when completely new problems are provided to OAKPLAN (we add the suffix "Nns", which stands for *N*ew object *n*ame*s*, to the corresponding results as in OAKPLAN-Nns); (3) case base libraries which contain only the base problems used to generate the variants of the benchmark set (we add the suffix "small"

- Logistics Additionals Track2: randomly selected from logistics-16-0 to logistics-100-1;
- Driverlog: pfile14, pfile17, pfile20, pfile-HC03, pfile-HC06, pfile-HC09;
- Zenotravel: pfile14, pfile17, pfile20, pfile-HC14, pfile-HC17, pfile-HC20;
- Rovers-IPC5: pfile35, pfile36, pfile37, pfile38, pfile39, pfile40;
- TPP: pfile25, pfile26, pfile27, pfile28, pfile29, pfile30.

[23]For small problems, the difference among these strategies is not particularly interesting except for the situation in which the stability of the plan produced is fundamental.

[24]In the following experimental results when a planning problem is solved by OAKPLAN it is not inserted in the case base but simply discarded.

Results for OAKPLAN-NNS						
Domain	Solutions	Speed	Matching Time	Quality	Stability	Differences
BlocksWorld	187 (86%)	214244.8	121535.6	346.0	0.92	49.6
Logistics	213 (98%)	88928.4	69606.3	390.2	0.88	76.6
DriverLog	197 (91%)	112556.6	31107.4	230.2	0.91	23.6
ZenoTravel	211 (97%)	86722.1	34123.0	194.6	0.84	47.2
Rovers	214 (99%)	62421.3	53719.5	374.4	0.98	11.4
TPP	210 (97%)	26837.8	859.2	308.8	0.96	20.9
TOTAL	1232 (95%)	96162.0	50777.4	307.8	0.92	38.2

Table 1: Results of OAKPLAN-Nns in the different domains: number of solutions found, average CPU-time of the first solutions (in milliseconds) and corresponding average matching time, average best plan quality, average best plan stability and average best plan differences.

to the corresponding results as in OAKPLAN-small).[25] A more detailed comparison of the results produced in the different domains is available in Appendix C.

4.2 Overall Results

In this section we report the overall results of OAKPLAN considering the number of solutions found, the CPU-time, the plan quality and the plan stability [22] of the solutions produced by the adaptation process with respect to the plan obtained by the RETRIEVEPLAN function ($best_plan$). While the first three terms are standard evaluation parameters commonly adopted in planning, the plan stability deserves some additional considerations. The importance of plan stability has been examined by Fox et al. [22] in the context of plan adaptation, where the authors use the term *plan stability* to refer to a measure of the difference a process induces between an original (source) plan and a new (target) plan. In [22] the plan stability is measured considering the *distance*, expressed in terms of number of different actions, between the source plan π and the target plan π_0. In this paper we also consider an additional plan stability function (ϱ) derived by the formalisation presented by Srivastava et al. [77]:

$$\varrho(\pi, \pi_0) = 1 - \left(\frac{|\pi - \pi_0|}{|\pi| + |\pi_0|} + \frac{|\pi_0 - \pi|}{|\pi| + |\pi_0|} \right)$$

The second term represents the contribution of the actions in π to the plan stability, while the third term indicates the contribution of π_0 to ϱ. This function assumes a 0 value when the two plans are completely different and a value equal to 1 when π and π_0 have exactly the same actions. From a practical point of view we think that plan stability can be quite important in different real world applications. For example more stable plans offer a greater opportunity for graceful elision of activities and less stress on execution components. Preserving plan stability also reduces the cognitive load on human observers of a planned activity, by ensuring coherence and consistency of behaviours, even in the face of dynamic environments [22]. Finally, in a case-based approach, a high plan stability is a clear indicator that the system correctly selects plans that can be easily adapted.

In Table 1 we present the results of OAKPLAN-Nns in the different benchmark domains. Here we consider the average CPU-time and the Matching Time for the first solutions generated (in milliseconds). In the fifth column we present the average plan quality of the best solution generated in the different variants and finally the average plan stability and plan distance (in

[25] The number of cases of the plan libraries in the "small" tests is always lower than 15 except in the Logistics domain where we consider 170 base planning problems.

Results for OAKPLAN-NNS-KBASE *and percent errors of* OAKPLAN-NNS-KBASE *vs* OAKPLAN-NNS						
Domain	Solutions	Speed	Matching	Quality	Stability	Differences
BlocksWorld	68.0 (-64%)	244907 (173%)	122391 (198%)	330 (56%)	0.71 (-19%)	280 (407%)
Logistics	123 (-42%)	101027 (362%)	87582 (390%)	293 (14%)	0.59 (-31%)	328 (415%)
DriverLog	157 (-20%)	113622 (110%)	18982 (3.3%)	209 (19%)	0.60 (-33%)	98.9 (401%)
ZenoTravel	142 (-33%)	142705 (252%)	51823 (176%)	191 (31%)	0.25 (-69%)	161 (290%)
Rovers	216 (0.93%)	47931 (-25%)	38183 (-30%)	374 (0.08%)	0.97 (-0.7%)	29.6 (136%)
TPP	198 (-5.7%)	73923 (198%)	928 (9.4%)	332 (9.4%)	0.20 (-79%)	76.5 (267%)
TOTAL	904 (-27%)	101961 (122%)	41886 (69%)	293 (13%)	0.55 (-39%)	132 (338%)

Table 2: Results of OAKPLAN-Nns-$\mathcal{K}_{base}$ vs OAKPLAN-Nns

Results for OAKPLAN-NNS-KNODE *and percent errors of* OAKPLAN-NNS-KNODE *vs* OAKPLAN-NNS						
Domain	Solutions	Speed	Matching	Quality	Stability	Differences
BlocksWorld	65.0 (-65%)	275902 (244%)	147690 (317%)	317 (57%)	0.41 (-54%)	293 (462%)
Logistics	103 (-52%)	129886 (620%)	125599 (725%)	241 (7.8%)	0.31 (-63%)	282 (320%)
DriverLog	108 (-45%)	54862 (353%)	46725 (553%)	94.6 (4.1%)	0.07 (-91%)	137 (676%)
ZenoTravel	108 (-49%)	98908 (245%)	78695 (290%)	96.6 (-2.5%)	0.17 (-78%)	151 (267%)
Rovers	214 (0.0%)	36560 (-42%)	18985 (-65%)	346 (-7.5%)	0.15 (-85%)	345 (2940%)
TPP	4.00 (-98%)	502450 (10901%)	740 (24%)	520 (106%)	0.20 (-79%)	504 (2701%)
TOTAL	602 (-51%)	95935 (133%)	66690 (120%)	236 (3.3%)	0.19 (-78%)	258 (708%)

Table 3: Results of OAKPLAN-Nns-$\mathcal{K}_{node}$ vs OAKPLAN-Nns

terms of number of different actions) of the best solution produced with respect to the plan obtained by the RETRIEVEPLAN function ($best_plan$). OAKPLAN-Nns solves 95% of the problems attempted and the average difference with respect to the target plans is 38.2, i.e. considering all the 1232 planning problems solved by OAKPLAN there are on average 38 actions introduced or removed with respect to the target plans which corresponds to a stability of 92%. It requires 96 seconds to solve the different benchmark planning problems of which 51 seconds are required by the matching process. It is important to point out that more than 10000 cases belong to each plan library, which have to be considered by the matching process. We think that such a high number of cases is hardly required by real applications: in fact case base maintenance policies [75] could be used in real word applications in order to reduce the number of cases that have to be handled by a case-based planner significantly.

In the following part of this section we examine the relevance of the kernel functions used by OAKPLAN considering the $\mathcal{K}_{base}$ kernel function described in section 3.1.3 and a new kernel function, that we call $\mathcal{K}_{node}$, which simply uses the k_v kernel function in equation (7) and only compares the labels of the pairs of nodes considered. Finally we examine the influence of the case base size on the system performance and how OAKPLAN performs when a planning problem with the same objects names as the case base is considered.

4.2.1 Matching Functions

To verify the relevance of the matching functions used by our system, we compare OAKPLAN with simpler matching functions; in particular we examine the system behaviour considering the $\mathcal{K}_{base}$ and the $\mathcal{K}_{node}$ kernel functions.

In Table 2 we compare OAKPLAN-Nns with one of its reduced versions called OAKPLAN-Nns-$\mathcal{K}_{base}$ that avoids the computation of steps $3.x$ in the RETRIEVEPLAN procedure, i.e. the best matching function is obtained only by considering the $\mathcal{K}_{base}$ kernel function. In this

Results for OAKPLAN-NNS-ADAPT-KBASE *and percent errors of* OAKPLAN-NNS-ADAPT-KBASE *vs* OAKPLAN-NNS						
Domain	Solutions	Speed	Matching	Quality	Stability	Differences
BlocksWorld	65.0 (-65%)	172152 (117%)	35607 (0.20%)	338 (67%)	0.44 (-49%)	294 (447%)
Logistics	178 (-16%)	130661 (132%)	45745 (0.43%)	411 (21%)	0.30 (-66%)	488 (574%)
DriverLog	157 (-20%)	112493 (112%)	18328 (-0.1%)	209 (19%)	0.74 (-18%)	99.0 (406%)
ZenoTravel	160 (-24%)	156323 (198%)	21714 (-0.2%)	214 (31%)	0.44 (-47%)	194 (347%)
Rovers	214 (0.0%)	63247 (1.3%)	53737 (0.03%)	375 (0.08%)	0.96 (-2.1%)	26.8 (136%)
TPP	203 (-3.3%)	79418 (225%)	858 (0.35%)	340 (11%)	0.88 (-8.5%)	86.7 (325%)
TOTAL	977 (-21%)	109291 (112%)	29153 (0.11%)	319 (16%)	0.67 (-27%)	180 (434%)

Table 4: Results of OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ vs OAKPLAN-Nns

Results for OAKPLAN-NNS-ADAPT-KNODE *and percent errors of* OAKPLAN-NNS-ADAPT-KNODE *vs* OAKPLAN-NNS						
Domain	Solutions	Speed	Matching	Quality	Stability	Differences
BlocksWorld	69.0 (-63%)	181379 (111%)	36344 (0.08%)	348 (68%)	0.42 (-52%)	314 (455%)
Logistics	181 (-15%)	133987 (133%)	45543 (-1.0%)	410 (19%)	0.18 (-80%)	523 (617%)
DriverLog	108 (-45%)	15154 (25%)	7146 (-0.2%)	95.0 (4.7%)	0.06 (-93%)	144 (716%)
ZenoTravel	126 (-40%)	109391 (169%)	21098 (-1.0%)	139 (8.9%)	0.09 (-88%)	223 (419%)
Rovers	214 (0.0%)	71984 (15%)	53745 (0.04%)	351 (-6.2%)	0.19 (-81%)	351 (2986%)
TPP	4.00 (-98%)	437260 (9463%)	585 (-1.7%)	518 (109%)	0.19 (-81%)	507 (3460%)
TOTAL	702 (-43%)	98775 (92%)	36588 (-0.4%)	289 (11%)	0.17 (-81%)	338 (781%)

Table 5: Results of OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ vs OAKPLAN-Nns

Table and in the following ones, we report the *percent error* in brackets[26] with respect to OAKPLAN-Nns where, except for the column of the solutions found, we consider only the problems solved by both planners. By using this less accurate matching function the number of problems solved is 904 (down 27%), the average CPU-time required is 101 seconds (up 122% considering only the problems solved by both systems) and, most important of all, a plan difference of 132 actions (up 338%) and a plan stability of 0.55 (down 39%). Note that the CPU-time required by the matching process increases significantly (plus 69%) since a less accurate matching function determines a greater number of problems that have to be examined by steps 4.2–4.7 of the RETRIEVEPLAN procedure.

In Table 3 we compare OAKPLAN-Nns with a relaxed version of it called OAKPLAN-Nns-$\mathcal{K}_{node}$ which avoids the computation of steps $3.x$ of the RETRIEVEPLAN procedure and uses the $\mathcal{K}_{node}$ kernel function instead of the $\mathcal{K}_{base}$ function at step 2.2. In this test we want to examine the system behaviour when a very simple matching function is used. The number of problems solved is 602 (down 51%), the CPU-time required to solve the planning problems is 95.9 seconds (up 133% considering the problems solved by both systems), the plan difference is of 258 actions (up 708%) and plan stability is only 0.19 (down 78%). This clearly indicates the extraordinary importance of an accurate matching function for the global system performance, not only in order to obtain low distance values but also to solve a reasonable number of planning problems.

In Tables 4 and 5 we examine the situation where the best planning case is selected by the standard RETRIEVEPLAN procedure but the $best_plan$ at step 4.7 is not identified by using the best matching function found until now but by applying the corresponding $\mu_{base}(\pi_i)$ or $\mu_{node}(\pi_i)$[27] matching functions. The relating results are indicated respectively with OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$. In this way we provide the same planning

[26]Given two values a and b the percent error of a with respect to b is equal to $\frac{|a-b|}{|b|} \cdot 100\%$. Since our values are always positive we have not considered the absolute value in the previous formula and a negative percent error indicates that a is less than b.

[27]The μ_{base} matching function is computed using the $\mathcal{K}_{base}$ kernel function, similarly the μ_{node} matching function is computed using the $\mathcal{K}_{node}$ kernel function.

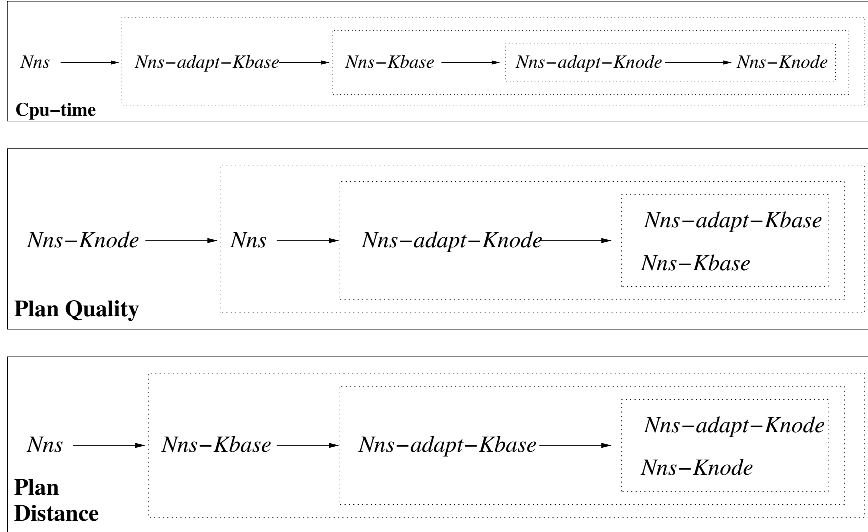


Figure 14: Partial order of the performance of OAKPLAN-Nns, OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ according to the Wilcoxon signed rank test for our benchmark problems.

case used by OAKPLAN-Nns to the adaptation process while the encoded solution plan (step 4.7 Figure 10) is obtained by using the $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions. The number of problems solved increases considerably since the correct planning case is provided to LPG-adapt, but the average CPU-time and the average plan differences remain significantly greater than the corresponding ones of OAKPLAN-Nns. In particular OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ determines an average plan distance of 180 actions (up 434%), while OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ determines an average plan distance of 338 actions (up 781%) and a plan stability of 0.17. The plan distance values are even greater than the ones obtained in the previous tests essentially because now the system is able to solve much more difficult planning problems.

We carried out a statistical analysis based on the Wilcoxon signed rank test [86] to understand the significance of the performance gaps in the planners compared during the experiments. The organisers of IPC-3 have also utilised this statistical test to study the performance of the planners in the competition [56]. The data necessary to effect the Wilcoxon test are obtained in the following way. The difference between the CPU-times of the two planners compared is computed and the samples of the test for the CPU-time analysis are defined. The absolute values of these differences are then ranked by increasing numbers, starting from the lowest value. (The lowest value is ranked 1, the next lowest value is ranked 2, and so on.) After that the ranks of the positive differences and the ranks of the negative differences are summed respectively. Should it happen that the performance of the planners compared are not very different, then the number of the positive differences is more or less equal to the number of the negative differences. Moreover the sum of the ranks in the set of the positive differences is approximately equal to the sum of the ranks in the other set. From an intuitive point of view, the test takes into consideration a weighted sum of the number of times one planner performs better than the other. The test makes use of the performance gap to give a rank to each performance difference, thus we say that the sum is weighted.

Figure 14 gives a graphical summary of the Wilcoxon results for the relative performance of OAKPLAN-Nns with OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-

Results for OAKPLAN-ONS *and percent errors of* OAKPLAN-ONS *vs* OAKPLAN-NNS						
Domain	Solutions	Speed	Matching	Quality	Stability	Differences
BlocksWorld	213 (14%)	66831 (-73%)	10648 (-92%)	335 (-7.3%)	0.98 (6.8%)	13.9 (-76%)
Logistics	211 (-0.9%)	33729 (-62%)	17804 (-74%)	371 (-4.3%)	0.94 (6.8%)	31.7 (-59%)
DriverLog	199 (1.0%)	112552 (-4.2%)	29052 (-9.1%)	233 (-0.1%)	0.91 (0.27%)	22.8 (-4.3%)
ZenoTravel	211 (0.0%)	39625 (-54%)	7989 (-77%)	178 (-8.8%)	0.91 (8.8%)	19.8 (-58%)
Rovers	214 (0.0%)	61688 (-1.2%)	52975 (-1.4%)	374 (-0.0%)	0.98 (-0.1%)	11.2 (-1.3%)
TPP	211 (0.47%)	24537 (-13%)	748 (-13%)	308 (-0.5%)	0.97 (0.31%)	17.3 (-18%)
TOTAL	1259 (2.2%)	55988 (-45%)	19846 (-61%)	301 (-3.2%)	0.95 (3.6%)	19.4 (-50%)

Table 6: Summary Table OAKPLAN-Ons vs OAKPLAN-Nns

$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ in terms of CPU-time, plan quality and difference values for our benchmark problems. [28] A solid arrow from a planner A to a planner B (or to a cluster of planners B) indicates that the performance of A is statistically different from the performance of B (every planner in B), and that A performs better than B (every planner in B) with a confidence level of 99.9%. A dashed arrow from A to B indicates that A is better than B with a confidence level of 99%. Here we can observe as expected that OAKPLAN-Nns is statistically better than the other OAKPLAN variants both in terms of CPU-time and plan distance values. Quite interesting we can note that OAKPLAN-Nns-$\mathcal{K}_{node}$ is statistically the most efficient planner in terms of quality of the plans generated followed by OAKPLAN-Nns. This can be explained by the fact that during the incremental adaptation process it has not been able to reduce significantly the plan distance values but only the quality of the plans produced.

4.2.2 Object Names Renaming Analysis

In Table 6 we compare OAKPLAN-Ons and OAKPLAN-Nns. The CPU-time required by OAKPLAN-Ons is lower than the CPU-time of OAKPLAN-Nns since the $\mathcal{K}_{base}$ kernel function in OAKPLAN-Nns produces lower similarity values than in OAKPLAN-Ons, as we can see more precisely in the following subsection. These lower values determine a greater number of cases that must be evaluated using $\mathcal{K}_{\mathcal{N}}$ while the number of solutions produced and the plan qualities are very close. On the contrary the difference values decrease considerably with respect to the values of Table 1 (38.2 vs. 19.4): it is important to point out that with OAKPLAN-Ons it has been possible to use the solution plans stored in the case base directly to compute the distance values since these test problems and the planning cases have the same domain objects. In particular while in DriverLog, Rovers and TPP the plans produced by OAKPLAN-Nns and OAKPLAN-Ons are very similar, in BlocksWorld and Logistics the plans produced by OAKPLAN-Nns are clearly worse than the corresponding ones produced by OAKPLAN-Ons with respect to the difference values. In the BlocksWorld domain the main difficulties are related to the very simple typed encoding which sometimes does not allow our kernel functions to easily identify the best object matching function. In this domain, the initial and goal state descriptions are very homogeneous since all objects are of the same type "Obj" and this leads to many different matching possibilities. As regards the Logistic domain, the main drawbacks are related to the fact that sometimes some trucks are assigned to different cities with respect to the original ones, unfortunately in this domain the trucks can be used only if they are positioned in specific cities and incorrect truck assignments could determine a high number of not applicable actions.

Considering real word applications we think that the effective performance of OAKPLAN should be placed between the results obtained by OAKPLAN-Nns and OAKPLAN-Ons. Al-

[28] Detailed results are reported in Appendix C at page 79.

Domain	Solutions	Speed	Matching	Quality	Stability	Differences
Results for OAKPLAN-SMALL-NNS *and percent errors of* OAKPLAN-SMALL-NNS *vs* OAKPLAN-NNS						
BlocksWorld	200 (6.9%)	118823 (-56%)	1596 (-99%)	359 (-0.3%)	0.91 (0.12%)	54.5 (-2.2%)
Logistics	214 (0.46%)	30131 (-67%)	11436 (-84%)	392 (0.03%)	0.88 (-0.1%)	76.9 (0.37%)
DriverLog	205 (4.1%)	97810 (-28%)	4724 (-86%)	242 (0.05%)	0.92 (1.7%)	24.3 (-4.2%)
ZenoTravel	216 (2.4%)	56226 (-39%)	3146 (-92%)	196 (-0.4%)	0.84 (0.55%)	46.3 (-3.3%)
Rovers	216 (0.93%)	27221 (-56%)	22527 (-58%)	374 (0.0%)	0.98 (0.0%)	11.3 (0.0%)
TPP	210 (0.0%)	26604 (-0.9%)	858 (-0.1%)	309 (0.0%)	0.96 (0.0%)	20.9 (-0.1%)
TOTAL	1261 (2.3%)	58584 (-47%)	7502 (-85%)	312 (-0.1%)	0.92 (0.37%)	39.0 (-1.4%)

Table 7: Summary Table OAKPLAN-small-Nns vs OAKPLAN-Nns

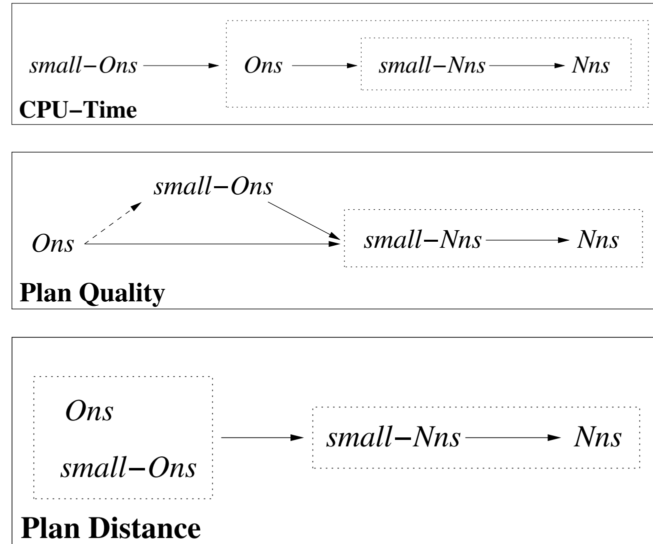


Figure 15: Partial order of the performance of OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons according to the Wilcoxon signed rank test for our benchmark problems.

though it is not realistic that all the current planning problem objects have to belong to the case base, it is quite common that the *domain topology* does not change significantly during the system evolution. For example, considering a department robotics domain where one or more robots have to move some packages from different locations, it is reasonable that the department locations and the corresponding connections do not change significantly as time goes by.

4.2.3 Case base size analysis

In Table 7 we compare OAKPLAN-small-Nns to OAKPLAN-Nns. We can observe that the "small version" is clearly faster than the complete version since the number of cases is considerably lower; OAKPLAN-small-Nns is able to solve 29 more problems than OAKPLAN-Nns (up 2.3%) with an average CPU-time of 58.5 seconds (down 47% with respect to OAKPLAN-Nns) and if we consider the matching CPU-time, it requires 7.5 seconds (down 85% with respect to OAKPLAN-Nns). It follows that the number of problems solved, the plan quality and the difference values are very close.

Figure 15 gives a graphical summary of the Wilcoxon results for the relative performance of OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons in terms

of CPU-time, plan quality and difference values for our benchmark problems.[29] Here we can observe that the use of case bases with the same objects names as the test set ("Ons" variants) reduces the CPU-time required to find a solution, the plan quality and the plan stability of the plans produced. As expected the "small" variants are faster than the corresponding variants with huge case bases. Moreover OAKPLAN-small-Nns is statistically better than OAKPLAN-Nns in terms of plan distance values since it can devote much more CPU-time to the incremental adaptation process.

Hence we can observe that OAKPLAN is significantly faster and solves more problems when it runs on small rather than large case bases, with only minimal impact on solution quality, stability and differences. This clearly indicate the importance in CBP of developing highly scalable retrieval mechanisms to analyse efficiently the case base, in fact all CBP systems have at least a retrieval component, and the success of a given system depends critically on the efficient retrieval of the right case at the right time. In this paper we consider a relatively simple screening procedure that filters out efficiently irrelevant cases; moreover our procedure could be combined with other retrieval techniques based on a model of case competence [75, 79] so as to improve the global system efficiency, which is left as future work.

Here we examine the CPU-time (in seconds) required by the different phases of OAKPLAN-Nns vs. the number of elements in the corresponding case base considering some specific benchmark planning problems. In particular in Figures 16-17 we show the cumulative CPU-time required by the different phases of OAKPLAN-Nns; the CPU-times can be simply derived by considering the distance of the corresponding line from the previous one. So we can obtain the CPU times required:

1. by the *preprocessing* phase so as to instantiate the data structures used by OAKPLAN, compute the mutex relations, connect to the case base and load the objects and predicated indexes;
2. by the *screening* procedure to retrieve the degree sequences from the case base and compute the $simil^{ds}$ values (steps 1.4 – 1.7 of the RETRIEVEPLAN procedure);
3. by the planning encoding graph retrieval procedure and the computation of the $\mathcal{K}_{base}$ kernel function on the cases selected in the previous phase (steps 2.1 – 2.4 of the RETRIEVEPLAN procedure);
4. by the computation of the $\mathcal{K}_{\mathcal{N}}$ kernel function and the corresponding *matching* function on the cases selected in the previous phase (steps 3.1 – 3.5 of the RETRIEVEPLAN procedure);
5. by the *evaluation* of the selected plans so as to define the corresponding adaptation cost (steps 4.2 – 4.7 of the RETRIEVEPLAN procedure);
6. by the LPG-adapt system to find a first solution; the *adaptation time* can be obtained by the difference between the *Total time* required to find a first solution and the total *Evaluation* time.

Here we can observe that the screening procedure is extremely fast and the CPU-time required by the preprocessing and evaluation phases is always limited. Quite interesting in the BlocksWorld variant the CPU-time required by the $\mathcal{K}_{\mathcal{N}}$ computation is particularly relevant since the $\mathcal{K}_{base}$ kernel function is not precise enough to filter out a significant number of cases. In fact, in this domain, a correct matching of the objects of two different planning problems is particularly difficult since all the objects are of the same type called "Obj" as exposed previously. We can also observe in the BlocksWorld variant that the CPU-time required by the

[29]Detailed results are reported in Appendix C at page 88.

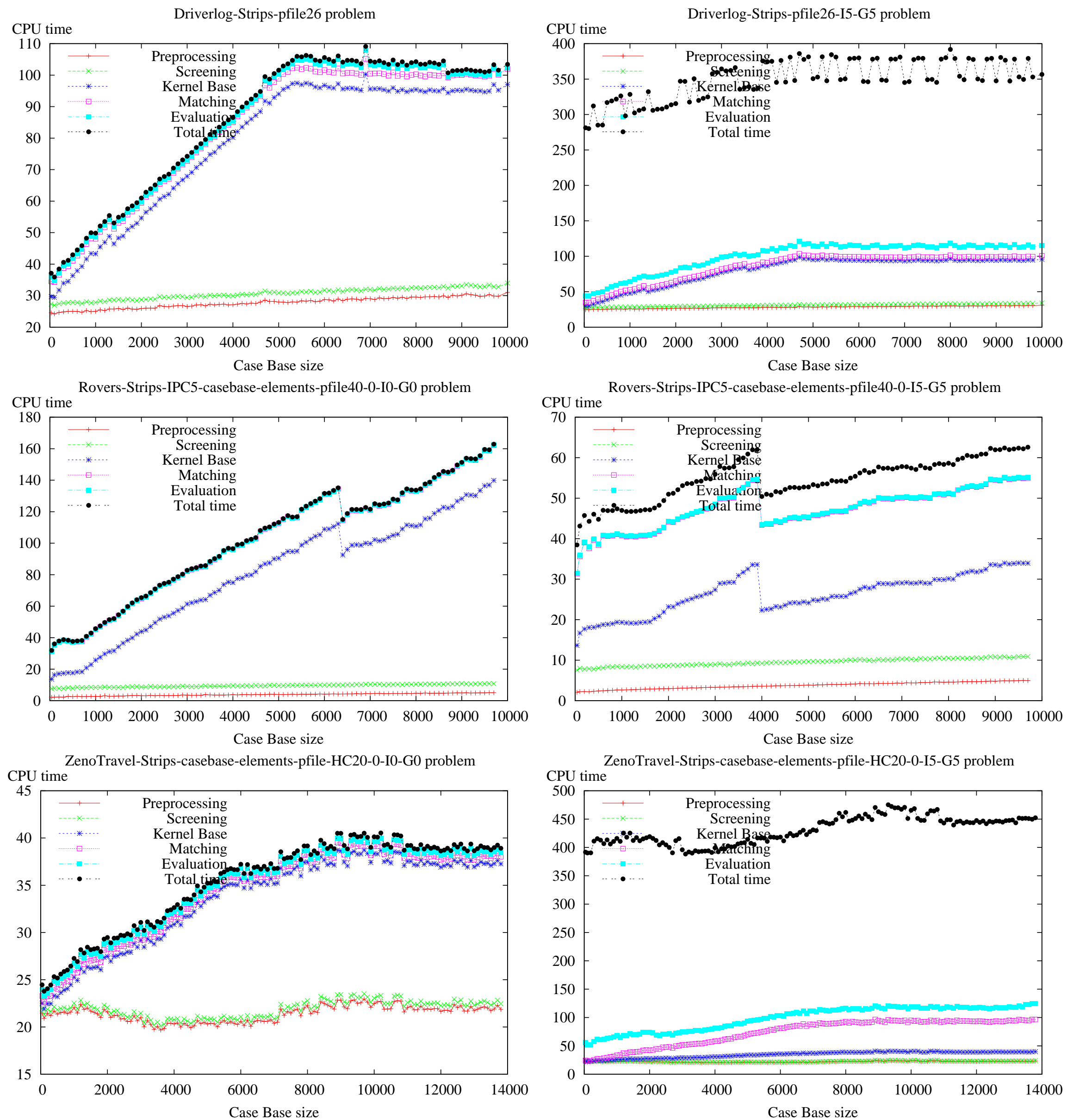


Figure 16: Cumulative CPU-times (seconds) required by the different phases of OAKPLAN-Nns vs. the number of planning cases of the case base.

matching phase stabilises when the case base size is nearly of 5500 cases. This is caused by the maximum number of cases that can be examined at step 2.1 of RETRIEVEPLAN. Besides in the Rovers domain the CPU-time required to find a first solution plan is always very limited and in this case the CPU-times for the computation of the $\mathcal{K}_{base}$ and $\mathcal{K}_{\mathcal{N}}$ kernel functions are comparable. In the DriverLog and in the ZenoTravel domains the matching and evaluation times are clearly dominated by the CPU-time required to find a first solution. Finally note that the first solution produced by LPG-adapt simply represents the first step of a potentially much longer incremental process.

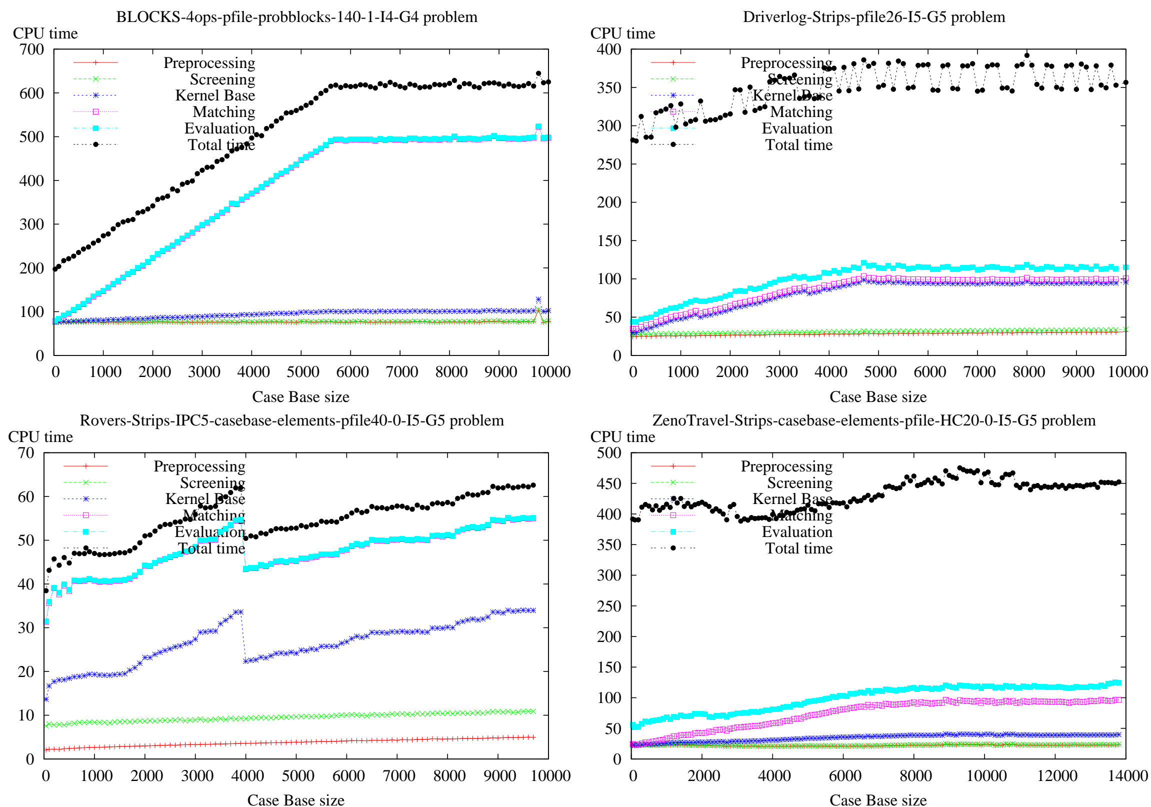


Figure 17: Cumulative CPU-times (seconds) required by the different phases of OAKPLAN-Nns vs. the number of planning cases of the case base.

At last, we can observe a drop in the CPU-time required for the computation of the kernel functions in the Rover domain in Figure 17 when the case base size is close to 4000 cases. This drop is related to an insertion in the case base of a planning case with a high $simil^{ds}$ screening value when the case base size varies from 3900 to 4000 instances. This screening value determines a new $best_ds_simil$ value and consequently the number of cases that satisfy the constraint "$best_ds_simil - simil_i \leq limit$" at step 2.1 of Algorithm RETRIEVEPLAN at page 26 decreases from 600 to 250. So the number of Planning Encoding Graphs that have to be loaded and the number of kernel functions that have to be computed is clearly less than the previous iteration and thus the corresponding CPU-time required for their computation.

4.3 Matching Functions Similarity Results

Here we examine the similarity values obtained with the Neighbourhood kernel function $\mathcal{K}_{\mathcal{N}}$, the Base kernel function $\mathcal{K}_{base}$, the $\mathcal{K}_{node}$ kernel function and the *direct matching* function produced by OAKPLAN for the hardest problems of our benchmarks in relation to a progressive renaming of the domain objects involved. In this way we analyse the effectiveness of our matching processes in comparison to a "direct matching" process simply based on the object names of the planning problems.

In Figures 18-19 we can see how the initial similarity values of the matching functions are equal to 1 for the plots on the left which correspond to the original problems in the case base

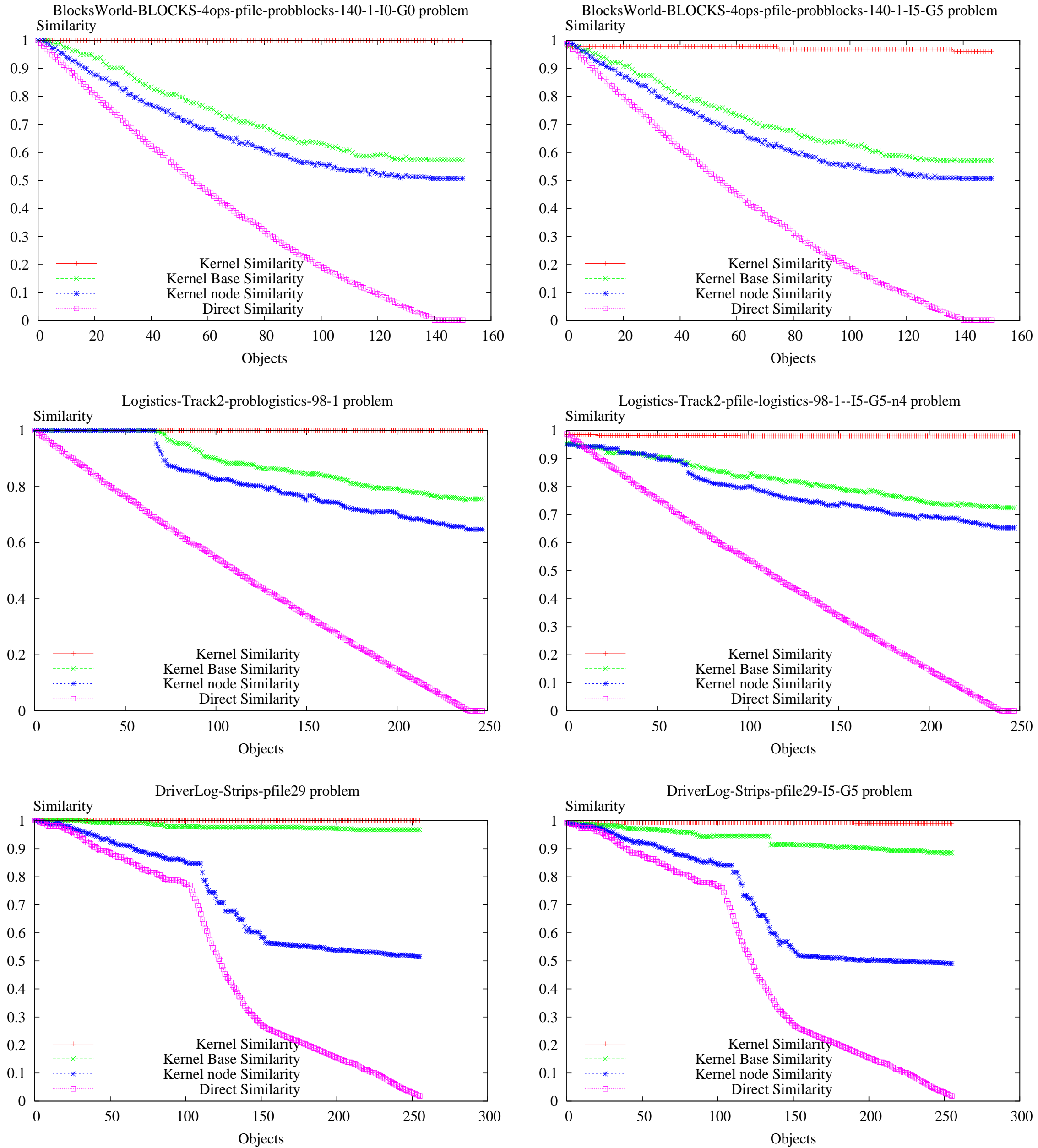


Figure 18: Similarity values for the Neighbourhood kernel function, the Base kernel function, the $\mathcal{K}_{node}$ kernel function and the direct matching for the hardest problems in the BlocksWorld, Logistics and DriverLog domains.

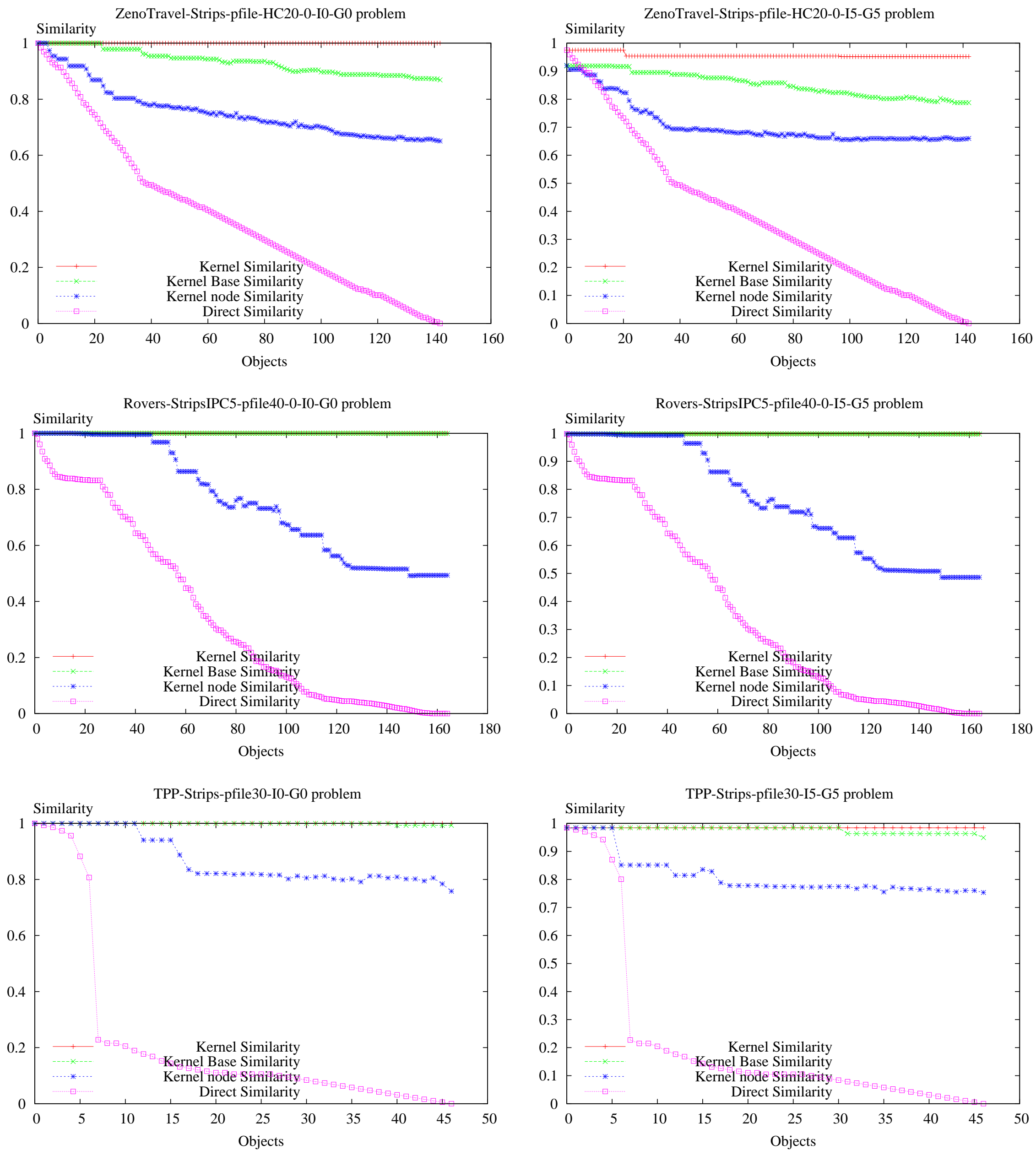


Figure 19: Similarity values for the Neighbourhood kernel function, the Base kernel function, the $\mathcal{K}_{node}$ kernel function and the direct matching for the hardest problems in the ZenoTravel, Rovers-StripsIPC5 and TPP domains.

and close to 1 for the plots on the right where we change 5 initial facts and 5 goal facts with respect to the corresponding element of the case base. As expected the similarity value of the direct matching progressively reduces itself to zero, whereas the similarity value of the $\mathcal{K}_{\mathcal{N}}$ kernel function always remains greater than 0.9, in particular it shows excellent performance in DriverLog, Rovers and TPP. If we consider the $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions we can see that the corresponding similarity values progressively decrease with the increase of the number of renamed objects. This is acceptable and not crucial for OAKPLAN since the $\mathcal{K}_{base}$ function

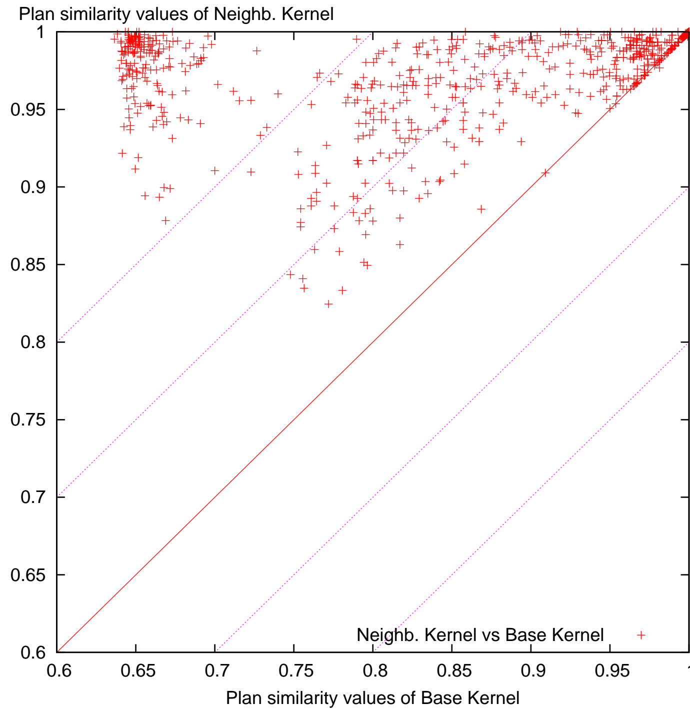


Figure 20: Plan similarity values of $\mathcal{K}_{\mathcal{N}}$ vs. $\mathcal{K}_{base}$

is essentially used in order to reduce the number of cases that must be evaluated accurately with $\mathcal{K}_{\mathcal{N}}$. Considering the plots on the left, which are associated to the original problem in the case base, we can observe that the $simil_{\mu_{\mathcal{N}}}$ values are always equal to 1, showing that $\mathcal{K}_{\mathcal{N}}$ is able to match all the planning problem objects correctly. Moreover it is interesting to note that also $\mathcal{K}_{base}$ performs extremely well in the DriverLog, Rovers and TPP domains.

Finally it is interesting to observe that sometimes the similarity values slightly increases at the increase of the number of renamed objects and this is due to the γ_0 coefficient introduced, as explained previously, in the kernel function in order to guarantee a "greater stability" in the activity assignment which is useful expecially when human agents are handled by the planner.

In order to examine $\mathcal{K}_{\mathcal{N}}$ and $\mathcal{K}_{base}$ more accurately, in Figure 20 we compare their similarity values for all our 1296 benchmark problems using the corresponding case base with all the objects renamed so as to verify the system behaviour when completely new problems are provided to OAKPLAN. Each point corresponds to the similarity values produced by $\mathcal{K}_{\mathcal{N}}$ and $\mathcal{K}_{base}$. If a point is above the solid diagonal, then $\mathcal{K}_{\mathcal{N}}$ performs better than $\mathcal{K}_{base}$ and vice versa. Here we can see that $\mathcal{K}_{\mathcal{N}}$ always performs better than $\mathcal{K}_{base}$ and in only 38 variants $simil_{\mu_{\mathcal{N}}}$ is less than 0.9. It is also important to point out that in all our experiments we obtain a similarity value equal to one for all the test variants in which there is no modifications with respect to the initial and goal facts (which correspond to the $I0$ – $G0$ instances). In fact these planning problems are already "present" in the case base and OAKPLAN identifies a mapping that correctly assigns all the objects of the selected planning case to the objects of the current planning problem.

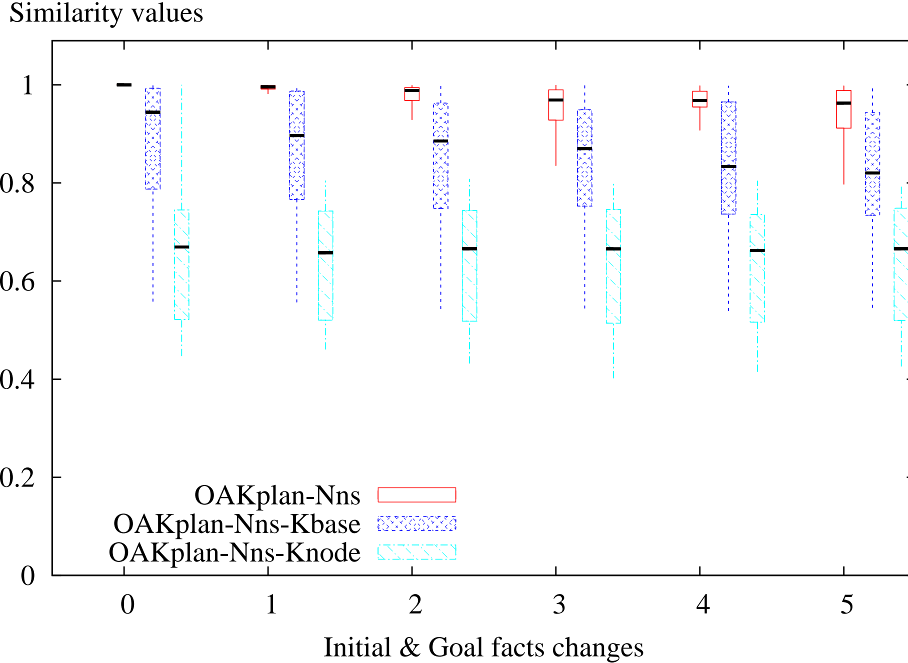


Figure 21: Box & Whiskers plots of the similarity values produced using $\mathcal{K}_{\mathcal{N}}$, $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions in the different benchmark planning problems considering different number of initial and goal changes.

In Figure 21 we can examine the Box & Whiskers plots [30] of the similarity values produced using the $\mathcal{K}_{\mathcal{N}}$, $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions. These results are grouped considering the test set problems with the same number x of initial and goal changes (Ix-Gx variants), this number is reported on the x-axis. Here we can observe that the $\mathcal{K}_{\mathcal{N}}$ kernel function produces high similarity values with a standard deviation which is clearly smaller than the other kernel functions. In particular if we consider the I0-G0 variants the results produced are very close to 1.[31]

4.4 OAKPLAN vs. State of the Art Planners

In this section we analyse the OAKPLAN behaviour with respect to four state-of-the-art planners, showing its effectiveness in different benchmark domains; in particular, we consider METRIC-FF (winner of the 2nd IPC), LPG (winner of the 3rd IPC), DOWNWARD (1st Prize, Suboptimal Propositional Track 4th IPC) and SGPLAN-IPC5 (winner of the 5th IPC).

In Figure 22 we graphically report the number of solutions found, the average CPU-time of the solutions of DOWNWARD, METRIC-FF and SGPLAN-IPC5 and the CPU-time of the

[30] A Box & Whiskers plot is a convenient way of graphically depicting groups of numerical data; the central box represents the values from the first to the third quartile (25 to 75 percentile). The middle line represents the median; the horizontal line extends from the minimum to the maximum value, excluding outside values. An outside value is defined as a value that is smaller than the lower quartile minus 1.5 times the interquartile range, or larger than the upper quartile plus 1.5 times the interquartile range (inner fences).

[31] Additional results are reported in Appendix C on page 131.

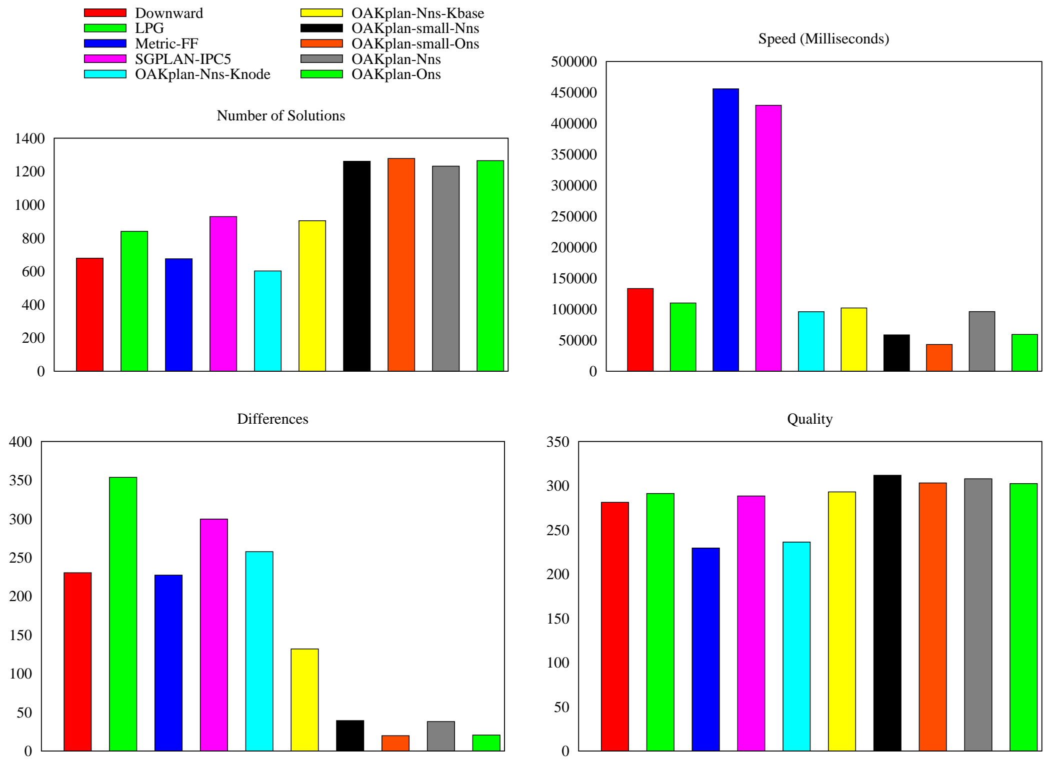


Figure 22: Summary results.

first solutions[32] of LPG and OAKPLAN. Then we consider the average plan difference values expressed as the number of different actions with respect to the solution produced by the corresponding planner on the problems used to generate the variants and the average best plan quality of the solutions generated by considering all benchmark domains.

Here we can see that OAKPLAN can solve the greatest number of variants, followed by SGPLAN-IPC5 and LPG. Regarding the CPU-time, we remark that DOWNWARD, LPG and OAKPLAN present similar computation time, while the CPU-time is more significant in METRIC-FF and SGPLAN-IPC5. However these average values are computed only by considering the problems solved by every single planner. In this case the SGPLAN-IPC5 planner solves 211 variants in the TPP domain requiring 942 seconds for them, whereas these variants only marginally influence the results of LPG. Regarding the difference values we can see that OAKPLAN clearly produces better results than the other planners. With respect to the plan quality we can note that METRIC-FF gives better results whereas OAKPLAN produces the worst results. We would like to point out that in OAKPLAN the optimisation process tries to balance between good quality and low distance values since we are much more interested in generating a plan with a limited number of differences with respect to the target plan than producing solutions of good quality. Moreover OAKPLAN is able to solve much more difficult planning problems than the other planners and these solutions weigh significantly on the average plan quality produced.

In Table 8 we report the summary results of OAKPLAN-Nns compared to the other planners. In the second columns we report the number of the solutions found by the other planners,

[32] Considering the median ones over five runs.

Results for DOWNWARD *and percent errors of* DOWNWARD *vs* OAKPLAN-NNS					
Domain	Solutions	Speed	Quality	Stability	Differences
BlocksWorld	64.0 (-66%)	474335 (+437%)	572 (+155%)	0.60 (-32%)	375 (+634%)
Logistics	198 (-7.0%)	93899 (+19%)	353 (-4.8%)	0.66 (-24%)	242 (+217%)
DriverLog	76.0 (-61%)	41738 (+381%)	78.8 (+14%)	0.45 (-46%)	91.6 (+447%)
ZenoTravel	130 (-38%)	54752 (-27%)	128 (-23%)	0.58 (-29%)	85.7 (+72%)
TPP	211 (+0.47%)	148591 (+444%)	293 (-5.1%)	0.45 (-53%)	315 (+1403%)
TOTAL	679 (-45%)	133420 (+141%)	281 (+5.9%)	0.55 (-38%)	230 (+411%)

Results for LPG *and percent errors of* LPG *vs* OAKPLAN-NNS					
Domain	Solutions	Speed	Quality	Stability	Differences
BlocksWorld	73.0 (-61%)	94078 (+4.3%)	238 (+12%)	0.71 (-19%)	149 (+160%)
Logistics	211 (-0.9%)	139416 (+58%)	451 (+16%)	0.60 (-31%)	396 (+413%)
DriverLog	122 (-38%)	108708 (+412%)	127 (+6.8%)	0.32 (-63%)	199 (+956%)
ZenoTravel	216 (+2.4%)	174570 (+95%)	202 (+2.3%)	0.25 (-70%)	332 (+591%)
Rovers	216 (+0.93%)	19440 (-69%)	335 (-11%)	0.18 (-82%)	489 (+4201%)
TPP	2.00 (-99%)	496110 (+41415%)	393 (+69%)	0.40 (-60%)	468 (+46650%)
TOTAL	840 (-32%)	110054 (+52%)	291 (+3.8%)	0.37 (-58%)	354 (+734%)

Results for METRIC-FF *and percent errors of* METRIC-FF *vs* OAKPLAN-NNS					
Domain	Solutions	Speed	Quality	Stability	Differences
Logistics	171 (-20%)	307669 (+429%)	304 (-8.0%)	0.69 (-20%)	196 (+163%)
DriverLog	65.0 (-67%)	35598 (+345%)	72.2 (+16%)	0.19 (-77%)	124 (+655%)
ZenoTravel	164 (-22%)	219264 (+200%)	123 (-29%)	0.57 (-30%)	97.0 (+98%)
Rovers	198 (-7.5%)	745702 (+1089%)	299 (-19%)	0.16 (-84%)	391 (+3366%)
TPP	77.0 (-63%)	899049 (+7054%)	246 (-3.9%)	0.51 (-47%)	240 (+1202%)
TOTAL	675 (-45%)	455941 (+757%)	229 (-15%)	0.44 (-51%)	227 (+500%)

Results for SGPLAN-IPC5 *and percent errors of* SGPLAN-IPC5 *vs* OAKPLAN-NNS					
Domain	Solutions	Speed	Quality	Stability	Differences
Logistics	216 (+1.4%)	462093 (+404%)	414 (+4.6%)	0.68 (-22%)	268 (+246%)
DriverLog	106 (-46%)	321346 (+2538%)	119 (+31%)	0.18 (-79%)	190 (+971%)
ZenoTravel	180 (-15%)	171353 (+126%)	142 (-23%)	0.53 (-36%)	137 (+175%)
Rovers	216 (+0.93%)	163457 (+162%)	343 (-8.4%)	0.15 (-84%)	467 (+4011%)
TPP	211 (+0.47%)	942278 (+3414%)	314 (+1.6%)	0.41 (-57%)	354 (+1593%)
TOTAL	929 (-25%)	429328 (+644%)	288 (-2.4%)	0.41 (-55%)	300 (+710%)

Table 8: Summary Tables of the different planners examined and a comparison with respect to the corresponding results produced by OAKPLAN-Nns.

in the third columns we report the average speed of the problems solved (in milliseconds), then the average plan qualities produced and finally the average plan stability and the average plan differences with respect to the solutions of the set of target plans produced by every single planner. In the brackets we report the *percent errors* with respect to OAKPLAN-Nns: we consider only the problems solved by both planners for this comparison, except for the column of the solutions found.

Downward cannot solve any problem in the Rovers domain. Globally it can solve 679 problems in comparison with the 1232 solved by OAKPLAN-Nns. DOWNWARD is 141% slower than OAKPLAN-Nns while their plan quality is comparable. The distance values of the plans generated by DOWNWARD with respect to the solutions produced by the same planner on the problems used to generate the variants is 411% greater than OAKPLAN-Nns. This high value is not particularly surprising since DOWNWARD and the other planners do not know the target plans used for this comparison. Moreover the search processes and the solution plans produced by a planner could be significantly different also for two planning instances that only differ in a single initial fact. These distance values are interesting since they are a clear indicator of the good behaviour of OAKPLAN and show that the generative approach is not feasible when we want to preserve the stability of the plans produced.

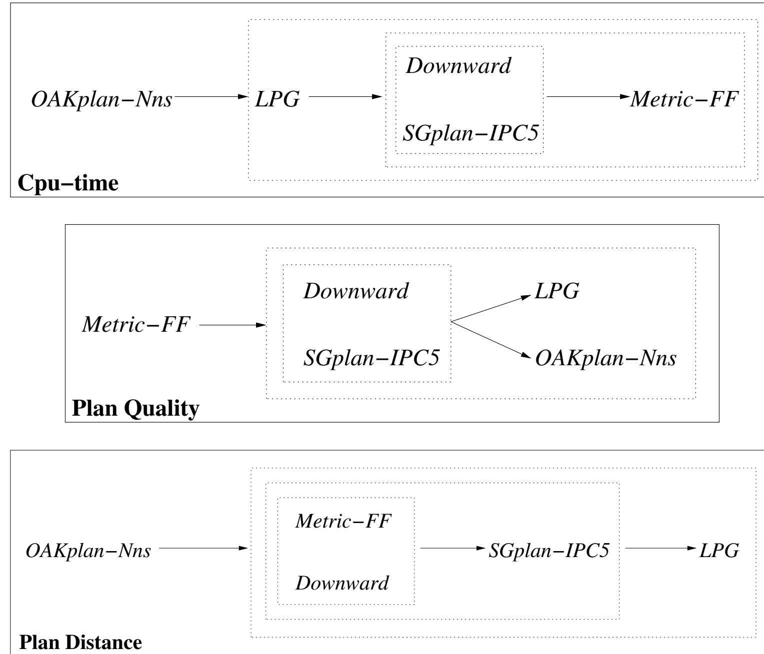


Figure 23: Partial order of the performance of OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5 according to the Wilcoxon signed rank test for our benchmark problems.

LPG can solve 840 of the 1296 variants, requiring 32% CPU-time more than OAKPLAN-Nns and the average distance of the solutions on target planning problems is 354 actions (which corresponds to +734% with respect to OAKPLAN-Nns). It is interesting to remark that the CPU-time needed by LPG to solve the Rovers variants (19.4 seconds) is significantly lower than in OAKPLAN (62.4 seconds) due to the additional CPU-time required by the matching process of OAKPLAN-Nns (53.7 seconds). The distance of the plans generated by LPG in this domain is 4201% greater than OAKPLAN-Nns.

Metric-FF cannot solve any variant in the BlocksWorld domain. Globally it can solve 675 problems and is 757% slower than OAKPLAN-Nns while its plan quality is 15% better. Finally the distance of the plans generated by METRIC-FF with respect to the solutions produced by the same planner on the target problems is 500% greater than OAKPLAN-Nns.

SGPlan-ipc5 planner can solve 929 problems and is 644% slower than OAKPLAN-Nns, the plan qualities are very similar and considering the distance of the plans generated by SGPLAN-IPC5 are on average 710% greater than with OAKPLAN-Nns.

Figure 23 gives a graphical summary of the Wilcoxon results for the relative performance of OAKPLAN-Nns with DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5 in terms of CPU-time, plan quality and difference values for our benchmark problems.[33] Here we can observe that OAKPLAN-Nns is statistically more efficient values than all the other planners in terms of CPU-time and plan distance. On the contrary OAKPLAN-Nns and LPG produce statistically worse plans from the quality point of view than the other planners, while METRIC-FF produces the highest quality plans.

[33]Detailed results are reported in Appendix C at page 89.

Globally we can note that OAKPLAN-Nns is able to solve many more problems than the other planners and the first solution is usually generated in less time. In addition the distance values are significantly lower with respect to the target plans although the quality of the plans produced is slightly worse than that of the plans produced by the other planner; this is also related to the optimisation performed by OAKPLAN where we try to minimise not only the plan quality but also the distance with respect to the solution plan of the planning case selected.

In Figures 24-26 we analyse the CPU-times, plan qualities and plan differences of each planner in the different domains considered in more detail. In general we can note that the majority of the variants where a planner is faster than OAKPLAN-Nns are the smallest in the different domains, where the overhead of the matching phase of OAKPLAN-Nns is proportionally more significant; for example in the Logistics and DriverLog domains the other planners are faster than OAKPLAN-Nns for the first 50 variants, while after these small variants the gap between OAKPLAN-Nns and the other planners progressively increases. Moreover in the Rovers domain the planning encoding graphs of the different planning problems are hefty and the corresponding comparison is particularly demanding in terms of CPU-time; in fact in this domain LPG is always faster than OAKPLAN-Nns. The plan qualities produced by the different planners are very similar except for a very limited number of variants where DOWNWARD performs badly with respect to OAKPLAN-Nns. Finally, considering the distance values we can note the good behaviour of OAKPLAN-Nns which in all but a limited number of cases performs better, with distance values which can be one order, and sometimes two orders, of magnitude better than those of the other planners. It is interesting to observe that, in a limited number of variants, the other planners produce better plans than OAKPLAN-Nns with respect to the distance values. This could be quite easily justified since nothing prevents a planning process from examining the same portion of the search space and to produce similar output plans when a variant is quite similar to the target planning problem used for its generation. Globally we can observe that, except for the simplest variants that are characterised by a limited number of actions in the solution plans, the gap between OAKPLAN-Nns and the other planners progressively becomes more and more significant as the problems become more difficult, except perhaps in the Rovers domain where the distance values of the other planners are always extremely significant.

Finally in Figure 27 we can observe the cumulative distribution of the total number of variants solved by the different planners vs. time. OAKPLAN-Nns is able to solve 1263 variants considering a maximum CPU-time of 1800 seconds even if most solutions are found in the first 800 seconds. A similar behaviour can be observed considering the LPG and DOWNWARD planners although they are able to solve a lower number of variants. On the contrary METRIC-FF and SGPLAN-IPC5 show a constant increment of the number of variants solved, which are in any case less than the variants solved by OAKPLAN. The CPU-time limit of 1800 seconds is used in the International Planning Competitions for the competitors evaluation and we think that it is adequate for the evaluation of the planners used in practical applications.

5 Related Work on Case-Based Planning

In the following section we examine the most relevant case-based planners considering their retrieval, adaptation and storage capabilities. Moreover, we present an empirical comparison of the performance of OAKPLAN vs. the FAR-OFF system and some comments on the advantages of OAKPLAN with respect to other case-based planners.

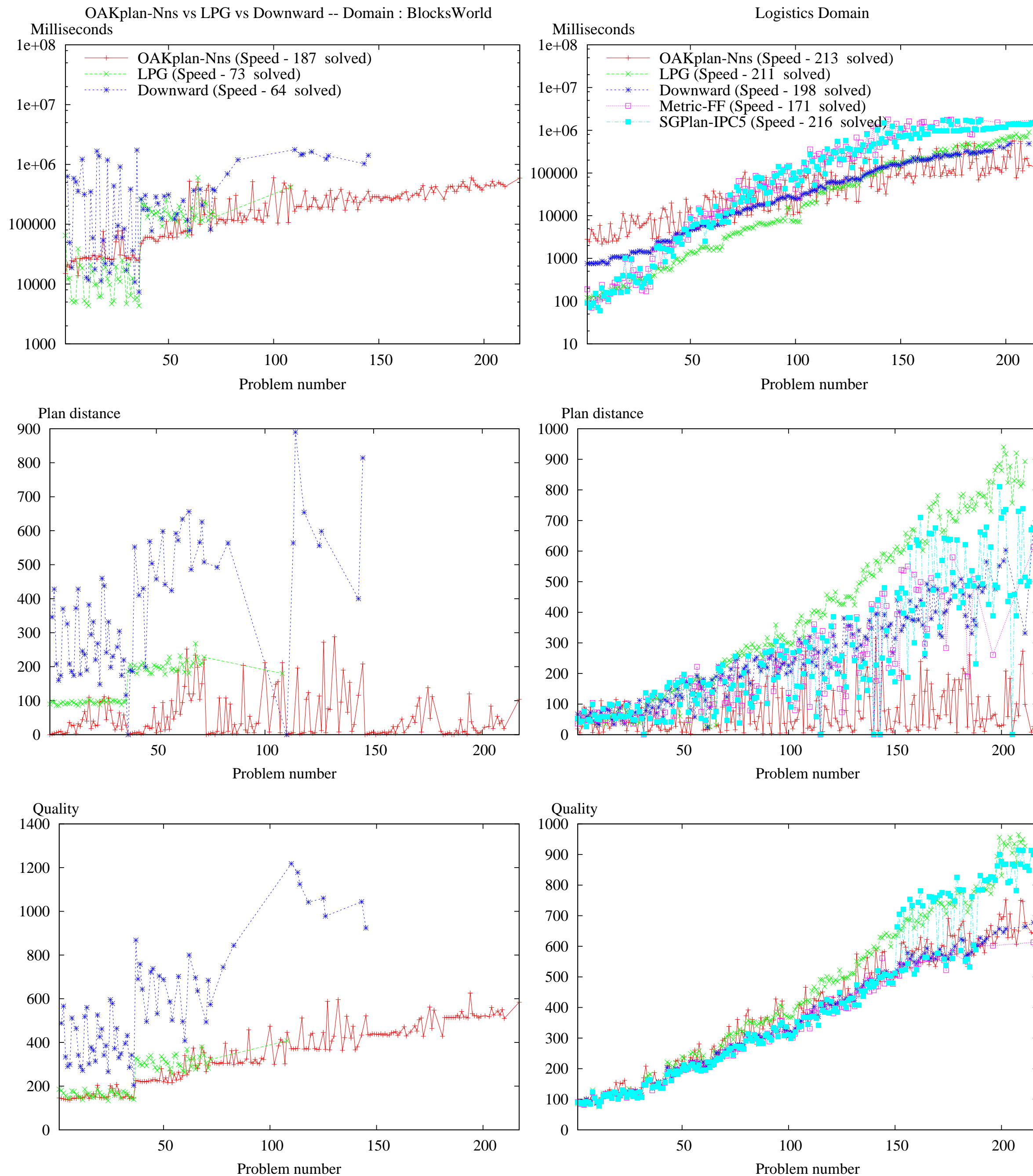


Figure 24: CPU-time (on a logarithmic scale), number of different actions and plan qualities for the BlocksWorld and the Logistics variants. Here we examine OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5.

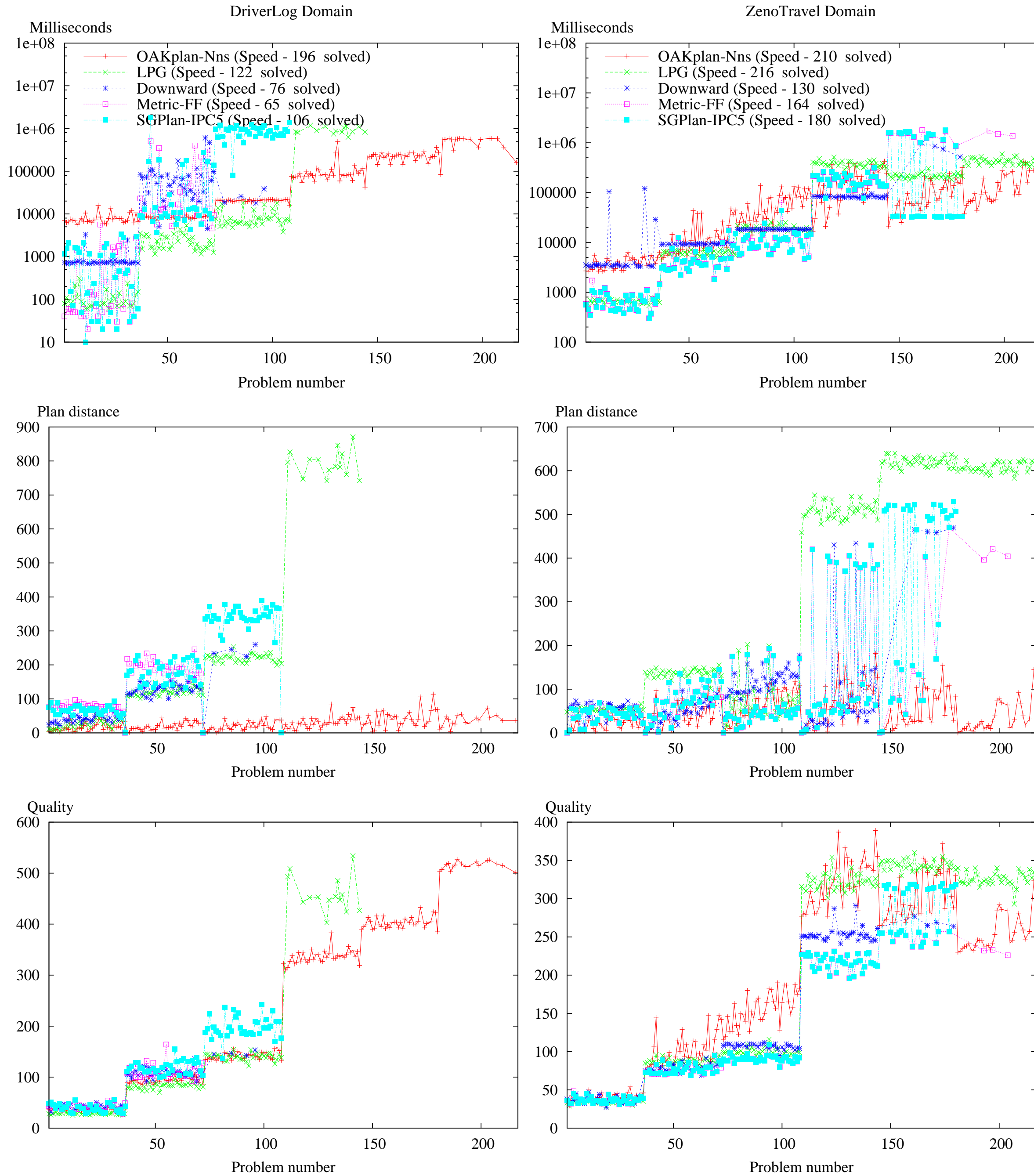


Figure 25: CPU-time (on a logarithmic scale), number of different actions and plan qualities for the DriverLog and the ZenoTravel variants. Here we examine OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5.

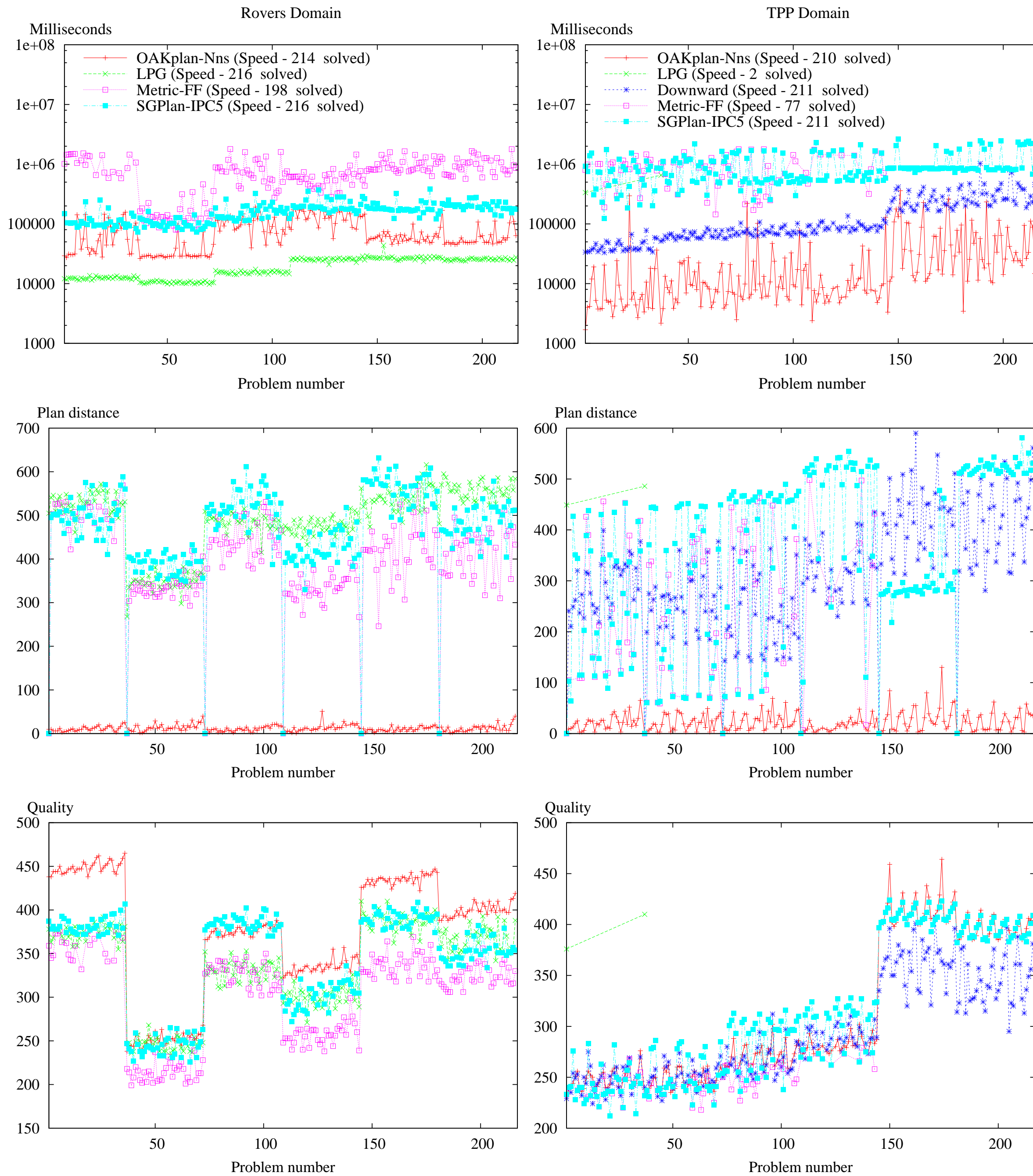


Figure 26: CPU-time (on a logarithmic scale), number of different actions and plan qualities for the Rover and the TPP variants. Here we examine OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5.

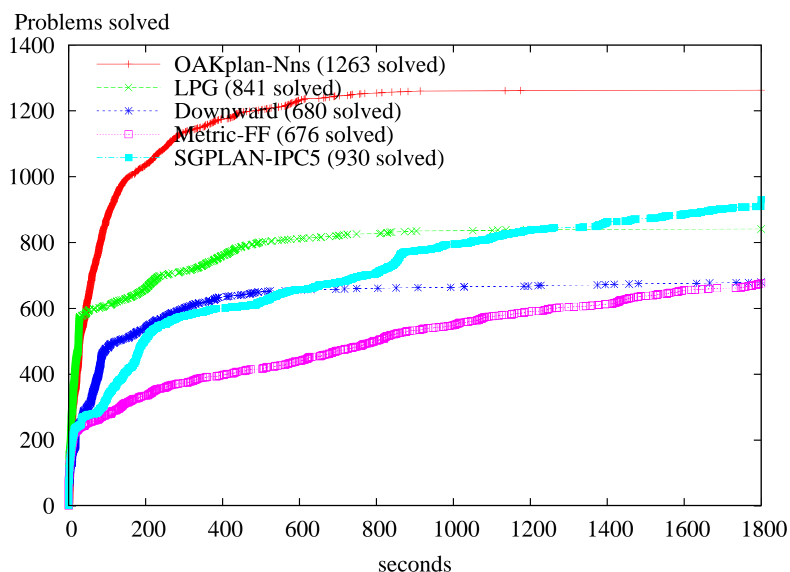


Figure 27: Cumulative distribution of the total number of variants solved by the different planners vs. time.

Some CBP systems designed in the past do not consider any generative planning in their structure, and find a solution only by the cases stored in the case base. These CBP systems are called *reuse-only* systems. As reuse-only systems cannot find any planning solution from scratch, they cannot find a solution unless they find a proper case in the case base that can be adapted through single rules. An alternative approach to reuse-only systems is the *reuse-optional* approach, which uses a generative planning system that is responsible to adapt the retrieved cases. This feature allows a CBP system to solve problems that cannot be solved only by using stored cases and simple rules in the adaptation phase. Empirically, a great number of reuse-optional CBP systems has shown that the use of a case base can permit them to perform better in processing time and in a number of planning solutions than the generative planning that they incorporate.

Obviously the retrieval phase critically affects the systems performance; it must search in a space of cases in order to choose a good one that will allow the system to solve a new problem easily. In order to improve efficiency in the retrieval phase, it is necessary either to reduce the search space or to design an accurate similarity metric. Reducing the search space, only a suitable subset of cases will be available for the search process and an accurate similarity metric will choose the most similar case to decrease the adaptation phase effort. In the literature there are different domain dependent and a few domain independent plan adaptation and case-based planning systems, which mostly use a search engine based on a space of states [36, 39, 79, 80]. An alternative approach to planning with states is that of plan-space planning or hierarchical systems [5] that search in a space of plans and have no goals, but only tasks to be achieved. Since tasks are semantically different from goals, the similarity metric designed for these CBP systems is also different from the similarity rules designed for state-space based CBP systems. For a detailed analysis of case-based and plan adaptation techniques see the papers of Spalazzi [76] and Munoz-Avila & Cox [59].

The CHEF system [37] is the first application of CBR in planning and it is a reuse-only system which is important especially from a historical point of view. It solves problems in the domain of Szechwan cooking and is equipped with a set of plan repair rules that describe how a specific failure can be repaired. Given a goal to produce a dish with particular properties, CHEF first tries to anticipate any problems or conflicts that may arise from the new goal and repairs problems that did not arise in the baseline scenario. It then executes the plan and, if execution results in a failure, a repair algorithm analyses the failure and builds an explanation of the reason why the failure has occurred. This explanation includes a description of the steps and states leading towards the failure as well as the goals that these steps tried to realise. Based on the explanation, a set of plan repair strategies is selected and instantiated to the specific situation of the failure. After choosing the best of these instantiated repair strategies, CHEF implements it and uses the result of the failure analysis to improve the index of this solution so that it will not be retrieved in situations where it will fail again.

Much attention has been given to research that designs suitable similarity metrics. It focuses on choosing the most adaptable case as the most similar one, such as the DIAL [51] and DÉJÀVU [74] systems. The DIAL system is a case-based planner that works in disaster domains where cases are schema-based episodes and uses a similarity assessment approach, called *RCR*, which considers an adaptability estimate to choose cases in the retrieval phase. Our similarity functions differ from the RCR method since they are based on a domain knowledge that is available in action definitions, while the RCR method uses the experience learned from the adaptation of previous utilisation of cases. They also differ in their applicability because the RCR method considers specifically disaster domains while our approach is suitable for domain independent planning.

Similarly, the DÉJÀVU system operates in design domains and uses an *adaptation-guided retrieval* (AGR) procedure to choose cases that are easier to be adapted. The AGR approach in the DÉJÀVU system uses additional domain knowledge, called capability knowledge, which is similar to that used to solve conflicts in partial-order planning systems. This additional knowledge allows to identify the type and the functionality of a set of transformations, which are performed by actions, through a collection of agents called specialists and strategies. It must be well specified so as to maximise the AGR performance. Our similarity functions differ from the AGR approach because we do not use any domain knowledge besides that obtained from actions and states, which is the minimal knowledge required to define a domain for planning systems.

The PLEXUS system [2] confronts with the problem of "adaptive planning", but also addresses the problem of runtime adaptation to plan failure. PLEXUS approaches plan adaptation with a combination of tactical control and situation matching. When plan failure is detected it is classified as either beginning a failing precondition, a failing outcome, a case of differing goals or an out-of-order step. If we ignore how to manage incomplete knowledge, the repair strategy involves the fact of replacing a failed plan step with one that might achieve the same purpose. It uses a semantic network to represent abstraction classes of actions that achieve the same purpose.

The GORDIUS [73] system is a transformational planner that combines small plan fragments for different (hopefully independent) aspects of the current problem. It does not perform an anticipation analysis on the plan, on the contrary it accepts the fact that the retrieved plan will be flawed and counts on its repair heuristics to patch it; in fact, much of the GORDIUS work is devoted to developing a set of repair operators for quantified and metric variables. The previous approaches differ with respect to OAKPLAN fundamentally because they are domain

dependent planners; on the contrary OAKPLAN uses only the domain and planning problems descriptions.

Three interesting works developed at the same time adopt similar assumptions: the PRIAR system [45], the SPA system [39] and the Prodigy/Analogy system [81, 82]. PRIAR uses a variant of Nonlin [78], a hierarchical planner, whereas SPA uses a constraint posting technique similar to Chapman's Tweak [13] as modified by McAllester and Rosenblitt [57]. PRIAR's plan representation and thus its algorithms are more complicated than those of SPA. There are three different types of validations (filter condition, precondition, and phantom goal) as well as different *reduction levels* for the plan that represents a hierarchical decomposition of its structure, along with five different strategies for repairing validation failures. In contrast to this representation the plan representation of SPA consists of causal links and step order constraints. The main idea behind the SPA system that separates it from the systems mentioned above is that the process of plan adaptation is a fairly simple extension of the process of plan generation. In the SPA view, plan generation is just a special case of plan adaptation (one in which there is no retrieved structure to exploit). With respect to our approach that defines a matching function μ from Π to Π' that maximises the similarity function $simil_\mu$, it should be noted that in PRIAR and SPA the conditions for the initial state match are slightly more complicated. In PRIAR the number of *inconsistencies in the validation structure* of the plan library is minimised; in SPA the number of violations of preconditions in the plan library is maximised. Moreover the problem-independent matching strategy implemented in SPA runs in exponential time in the number of objects since it simply evaluates all possible mappings. On the contrary we compute an approximate matching function in polynomial time and use an accurate plan evaluation function on a subset of the plans in the library.

The Prodigy/Analogy system also uses a search oriented approach to planning. A library plan (case) in a transformational or case-based planning framework stores a solution to a prior problem along with a summary of the new problems for which it would be a suitable solution, but it contains little information on the process that generates the solution. On the other hand derivational analogy stores substantial descriptions of the adaptation process decisions in the solution, whereas Veloso's system records more information at each choice point than SPA does, like a list of failed alternatives. An interesting similarity rule in the plan-space approach is presented in the CAPLAN/CBC system [60] which extends the similarity rule introduced by the Prodigy/Analogy system [81, 82] by using feature weights in order to reduce the errors in the retrieval phase. These feature weights are learned and recomputed according to the performance of the previous retrieved cases and we can note that this approach is similar to the RCR method used by the DIAL system in disaster domains. There are two important differences between our approach and the similarity rules of CAPLAN/CBC, one of which is that the former is designed for state-space planning and the latter for plan-space planning. Another difference is that our retrieval function does not need to learn any knowledge to present an accurate estimate: our retrieval method only needs the knowledge that can be extracted from the problem description and the actions of the planning cases.

O-Plan [19, 20] is based on the strategy of using plan repair rules as well. The effects of every action are confirmed while execution is performed. A repair plan formed by additional actions is added to the plan every time a failing effect is necessary in order to execute some other actions. We call repair plans the prebuilt ones which are in a position to repair a series of failure conditions. For instance, we can have repair plans including a plan to replace either a flat tyre or a broken engine. When an erroneous condition is met, the plan is no longer executed but a repair plan is inserted and executed. When the repair plan is complete, the regular plan

is executed once more. Failures are repaired by O-Plan by adding actions. It follows that it does not use either unrefinements or requires a history. However it is not complete and there are some failures which cannot be repaired.

MLR [61] is another case-based system and it is based on a proof system. While retrieving a plan from the library that has to be adapted to the current world state, it makes an effort to employ the retrieval plan as if it were a *proof* to set the goal conditions from the start. Should this happen, there is no need for any iteration to use the plan, otherwise, the outcome is a failed proof that can provide refitting information. On the basis of the failed proof, a plan skeleton is built through a *modification strategy* and it makes use of the failed proof to obtain the parts of the plan that are useful and removes the useless parts. After the computation of this skeleton, gaps are filled through a refinement strategy which makes use of the proof system. Although our object matching function is inspired to the Nebel & Koehler's formalisation, our approach significantly differs from theirs since they do not present an effective domain independent matching function. In fact, their experiments exhibit an exponential run time behaviour for the matching algorithm they use, instead we show that the retrieval and matching processes can be performed efficiently also for huge plan libraries. The matching function formalisation proposed by Nebel & Koehler also tries to maximise first the cardinality of the common goal facts set and second the cardinality of the common initial facts set. On the contrary we try to identify the matching function μ that maximise the $simil_\mu$ similarity value which considers both the initial and goal relevant facts and an accurate evaluation function based on a simulated execution of the candidate plans is used to select the best plan that has to be adapted.

Nebel & Koehler [61] present an interesting comparison of the MLR, SPA and PRIAR performance in the BlocksWorld domain considering planning instances with up to 8 *blocks*. They show that also for these small sized instances and using a *single* reuse candidate the matching costs are already greater than adaptation costs. When the modification tasks become more difficult, since the reuse candidate and the new planning instance are structurally less similar, the savings of plan modification become less predictable and the matching and adaptation effort is higher than the generation from scratch. On the contrary OAKPLAN shows good performance with respect to plan generation and our tests in the BlocksWorld domain consider instances with up to 140 *blocks* and a plan library with *ten thousands* cases.

The LPA* algorithm is used by the SHERPA replanner [53]. This algorithm was originally bound to repair path plan and backtrack to a partial plan having the same heuristic value as before the unexpected changes did in the world using the unrefinement step once. SHERPA is not useful to solve every repair problem, owing to the unrefinement strategy and the single application thereof. Its use is restricted to those problems whose actions are no longer present in the domain description. It follows that through the unrefinement step unavailaible actions are removed.

The Replan [8] model of plans is similar to the plans used in the hierarchical task network (HTN) formalism [21]. A task network is a description of a possible way to fulfil a task by doing some subtasks, or, eventually (primitive) actions. For each task at least one of such task networks exists. A plan is created by choosing the right task networks for each (abstract) task chosen, until each network consists of only (primitive) actions. Throughout this planning process, Replan constructs a derivation tree that includes all tasks chosen, and shows how a plan is derived. Plan repair within Replan is called partialisation. For each invalidated leaf node of the derivation tree, the (smallest) subtree that contains this node is removed. Initially, such an invalid leave node is a primitive action and the root of the corresponding subtree is the task containing this action. Subsequently a new refinement is generated for this task. If the

refinement fails, a new round is started in which task subtrees that are higher in the hierarchy are removed and regenerated. In the worst case, this process continues until the whole derivation tree is discarded.

A very interesting case-based planner is the FAR-OFF[34] (*Fast and Accurate Retrieval on Fast Forward*) system [79]. It uses a generative planning system based on the FF planner [42] to adapt similar cases and a similarity metric, called ADG (Action Distance-Guided), which, like EVALUATEPLAN, determines the adaptation effort by estimating the number of actions that is necessary to transform a case into a solution of the problem. The ADG similarity metric calculates two estimate values of the distance between states. The first value, called *initial similarity value*, estimates the distance between the current initial state I and the initial state of the case I_π building a relaxed plan having I as initial state and I_π as goal state. Similarly the second value, called *goal similarity value*, estimates the distance between the final state of the case and the goals of the current planning problem. Our EVALUATEPLAN procedure evaluates instead every single inconsistency that a case base solution plan determines in the current world state I.

The FAR-OFF system uses a new competence-based method, called *Footprint-based Retrieval* [75], to reduce the space of cases that will be evaluated by ADG. The Footprint-based Retrieval is a competence-based method for determining groups of footprint cases that represent a smaller case base with the same competence of the original one. Each footprint case has a set of similar cases called *Related Set* [75]. The union of footprint cases and Related Set is the original case base. On the contrary OAKPLAN uses a much more simple procedure based on the $simil^{ds}$ function to filter out irrelevant cases. The use of Footprint-based Retrieval techniques and case base maintenance policies in OAKPLAN is left for future work. It is important to point out that the retrieval phase of FAR-OFF does not use any kind of abstraction to match cases and problems.

The FAR-OFF system retrieves the most similar case, or the ordered k most similar cases, and shifts to the adaptation phase. Its adaptation process does not modify the retrieved case, but only completes it; it will only find a plan that begins from the current initial state and then goes to the initial state of the case, and another plan that begins from the state obtained by applying all the actions of the case and goes to a state that satisfies the current goals G. Obviously, the completing of cases leads the FAR-OFF system to find longer solution plans than generative planners, but it avoids wasting time in manipulating case actions in order to find shorter solutions length. To complete cases, the FAR-OFF system uses a FF-based generative planning system, where the solution is obtained by merging both plans that are found by the FF-based generative planning and the solution plan of the planning case selected. On the contrary OAKPLAN uses the LPG-adapt adaptation system, which uses a local search approach and works on the whole input plan so as to adapt and find a solution to the current planning problem.

In Figure 28 we can observe the behaviour of OAKPLAN vs FAR-OFF considering different variants of the greater case bases provided with the FAR-OFF system in the Logistics domain;[35] similar results have been obtained in the BlocksWorld, DriverLog and ZenoTravel domains. Globally, we can observe that FAR-OFF is always faster than OAKPLAN both considering the retrieval and the total adaptation time although also the OAKPLAN CPU-time is always lower

[34] FAR-OFF is available at http://www.fei.edu.br/~flaviot/faroff.

[35] We have used the case bases for the logistics-16-0, logistics-17-0 and logistics-18-0 Logistics IPC2 problems. For each problem considered the FAR-OFF system must have a case base with the same structure to perform tests. More than 700 cases belong to each case base and for each case base we have selected two planning cases and randomly generated 36 variants.

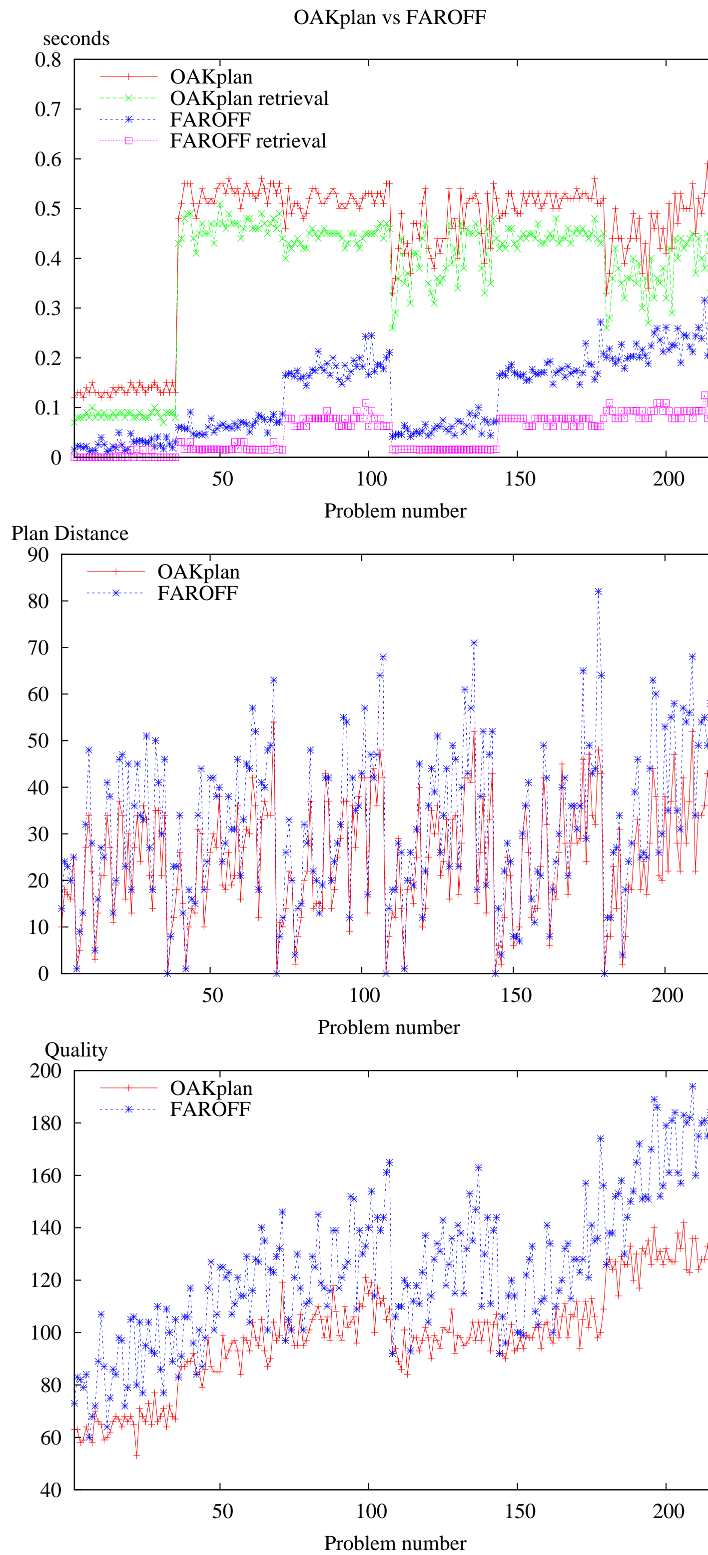


Figure 28: CPU-time, number of different actions and plan qualities for the Logistics variants. Here we examine OAKPLAN vs. FAR-OFF.

than 0.6 seconds. Considering OAKPLAN, most of the CPU-time is devoted to the computation of the matching functions which are not computed by FAR-OFF since it simply considers the *identity* matching function that directly assigns the objects of the case base to those of the current planning problem with the same name. In fact, it does not consider objects which are not already present in the case base and, to overcome this limitation, the variants used in this test are directly obtained by the problems stored in the case bases.

Regarding the plan qualities[36] and the plan distances, it is important to point out that for each variant solved by OAKPLAN we consider only the first solution produced since FAR-OFF does not perform a plan optimisation process. However OAKPLAN is able to obtain better plans both considering the plan quality and the plan distance values. Globally, OAKPLAN is able to find plans with 20% better quality and 24% better plan distances. Moreover further improvements on plan qualities and distance values of OAKPLAN could be obtained by performing the optimisation process of LPG-adapt.

Finally, note that in this experiment we have used the case bases provided by FAR-OFF which contain 700 elements each and the corresponding cases are generated by creating randomly planning problems all with the same configuration: same objects, trucks and airplanes simply disposed in different ways. This kind of experiment is highly unfavourable to OAKPLAN since our first screening procedure cannot filter out a significant number of cases as they all have the same structure. On the contrary, in the experiments described in the previous sections the case bases used by OAKPLAN in the standard configuration (not the “small” versions) are not constrained to a particular planning problem but they have been generated by considering all the different planning problems configurations used in the International Planning Competitions. This is a much more realistic situation, where the cases are added to the case base when the planning problems provided by the users are resolved as time goes by.

6 Summary and Future Work

CBP systems can take advantage of plan reuse where possible. The success of these systems depends on the ability to retrieve old cases that are similar to the target problem and to adapt these cases appropriately. In this paper our aim is to provide a new and effective case-based planner which is able to retrieve planning cases from huge plan libraries efficiently, choose a good candidate and adapt it in order to provide a solution plan which has good plan quality and is similar to the plan retrieved from the case base. We have described a novel case-based planning system, called OAKPLAN, which uses ideas from different research areas showing excellent performance in many standard planning benchmark domains. In this paper we have analysed the main components of our CBP system, which presents significant improvements as to the state of the art especially in the filtering and retrieval phases.

Given a planning problem we encode it as a compact graph structure that we call *Planning Encoding Graph*. This graph representation can give us a detailed description of the topology of the planning problem without requiring any a priori assumptions on the relevance of certain problem descriptors for the whole graph. Although it is possible to formulate the matching of the objects of two planning problem as a MCS problem, its exact resolution is infeasible from a computational point of view also for a limited number of candidate planning cases. Then we describe an approximate evaluation based on *kernel functions* to define a matching among the objects of two planning problems encoded as Planning Encoding Graphs. Experimental

[36]In STRIPS domains the plan quality is obtained by considering the number of actions in the solution plan.

results show the crucial importance of an accurate matching function for the global system performance, not only in order to obtain low distance values but also to solve a reasonable number of planning problems. Anyway, since the retrieval phase could be very expensive from a computational point of view when a huge plan library is used, we developed a screening procedure based on graph *degree sequences* to filter out irrelevant cases and execute the kernel function evaluation only to the most promising planning cases.

Finally an accurate plan evaluation phase is performed to define the "capacity" of the retrieved plans to solve the current planning problem. This evaluation is performed simulating the execution of the plans π_i and evaluating through a relaxed planning graph technique the cost of repairing the inconsistencies corresponding to the unsupported preconditions of the actions of π_i. Moreover, the evaluation of the generation cost allows to choose between an "adaptive" approach and a "generative" approach, if no plan gives an adaptation cost smaller than the generation cost. To the best of our knowledge this is the first case-based planner that performs an efficient domain independent objects matching evaluation on plan libraries with thousands of cases.

We have examined OAKPLAN in comparison with four state of the art plan generation systems showing its extremely good performance in terms of the number of problems solved, CPU time, plan difference values and plan quality. Results are very encouraging and show that the case-based planning approach can be an effective alternative to plan generation when "sufficiently similar" reuse candidates can be chosen. This happens to different practical applications especially when the "world is regular" and the types of problems the agents encounter tend to recur. Moreover this kind of approach could be extremely appealing in situations in which the "stability" of the plan produced is fundamental. This is the case, for example, in mission critical applications where end users do not accept newly generated plans and prefer to use known plans that have already been successful in analogous situations and can be easily validated.

We believe that even more significant results will come from combining our approach with ideas and methods that have been developed in planning, case-based reasoning, graph theory and supervised learning research areas. Specifically, directions we are considering include:

- *Case base maintenance*: the efficiency of the retrieval phase can be improved by using case base maintenance policies and a more thorough evaluation of the competence of the library as proposed by [75, 79].
- *Graph representation*: our current graph representation is based only on the initial and goal states of the planning problem examined; a more accurate representation could try to consider the actions in the solution plans and the domain operators available, or give more importance to the most relevant initial state facts. It could also be very interesting to extend our graph representation to afford temporal and numeric planning problems effectively. In fact, although OAKPLAN can afford temporal and metric domains, the numeric description of the planning problems examined is not actually used in the definition of the Planning Encoding Graph and in the corresponding kernel functions, determining potential low performance.
- *Matching functions*: new and more effective matching functions could be obtained by considering additional information that can be derived from domain analysis such as invariants [23, 32] and symmetries [24]. These functions may also be defined by examining particular structures of the Planning Encoding Graphs like cliques, line-graphs, or using new approaches derived by graph matching and graph edit distance techniques [9, 29, 62, 63, 67].

- *Learning*: these techniques found to be useful in different planning methods; our kernel functions can be plugged into any kernel-based machine learning algorithm, like, e.g., SVMs [18], SVR and Kernel PLS [68] to better classify the planning cases or improve the matching functions themselves.
- *Adaptation*: our retrieval/evaluation/update techniques are independent by the adaptation mechanism adopted and other adaptation methods like ADJUST-PLAN [35, 36] and POPR [80] could be effectively used as well.

Acknowledgments

This research was supported by the research project "Study and Design of a Prototype of an Intelligent Planning System for the Building of Learning Paths" of the Free University of Bozen. We thank Piergiorgio Bertoli, Alfonso E. Gerevini, Alessandro Saetti and especially the anonymous referees for their helpful comments. The authors would like to thank Flavio Tonidandel and Márcio Rillo for putting their benchmark set and the FAR-OFF planning system at our disposal.

References

[1] A. Aamodt and E. Plaza. Case-based reasoning: foundational issues, methodological variations, and system approaches. *AI Commun.*, 7(1):39–59, March 1994.

[2] R. Alterman. An adaptive planner. In J. Allen, J. Hendler, and A. Tate, editors, *Readings in Planning*, pages 660–664. Kaufmann, San Mateo, CA, 1990.

[3] T. Au, H. Muñoz-Avila, and D. S. Nau. On the complexity of plan adaptation by derivational analogy in a universal classical planning framework. In *Proceedings of the 6th European Conference on Advances in Case-Based Reasoning*, pages 13–27, London, UK, 2002. Springer-Verlag.

[4] F. Bacchus and F. Kabanza. Using temporal logic to express search control knowledge for planning. *Artificial Intelligence*, 116(1-2):123–191, 2000.

[5] R. Bergmann and W. Wilke. Building and refining abstract planning cases by change of representation language. *Journal of Artificial Intelligence Research*, 3:53–118, 1995.

[6] S. Biundo, K. L. Myers, and K. Rajan, editors. *Proceedings of the Fifteenth International Conference on Automated Planning and Scheduling (ICAPS 2005), June 5-10 2005, Monterey, California, USA*. AAAI, 2005.

[7] W. D. Blizard. Multiset theory. *Notre Dame Journal of Formal Logic*, 30(1):36–66, 1989.

[8] G. Boella and R. Damiano. A replanning algorithm for a reactive agent architecture. In D. Scott, editor, *AIMSA*, volume 2443 of *Lecture Notes in Computer Science*, pages 183–192. Springer, 2002.

[9] H. Bunke. Recent developments in graph matching. In *15th International Conference on Pattern Recognition*, volume 2, pages 117–124, 2000.

[10] T. Bylander. An average case analysis of planning. In *Proceedings of the Eleventh National Conference of the American Association for Artificial Intelligence (AAAI-93)*, pages 480–485, Washington, D.C., 1993. AAAI Press/The MIT press.

[11] T. Bylander. The computational complexity of propositional STRIPS planning. *Artificial Intelligence*, 69:165–204, 1994.

[12] T. Bylander. A probabilistic analysis of propositional STRIPS planning. *Artificial Intelligence*, 81(1-2):241–271, 1996.

[13] D. Chapman. Planning for conjunctive goals. *Artificial Intelligence*, 32(3):333–377, 1987.

[14] M. Chein and M. Mugnier. *Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs*. Springer Publishing Company, Incorporated, 2008.

[15] Y. P. Chien, A. Hudli, and M. Palakal. Using many-sorted logic in the object-oriented data model for fast robot task planning. *Journal of Intelligent and Robotic Systems*, 23(1):1–25, 1998.

[16] A. G. Cohn. Many sorted logic=unsorted logic+control? In *Proceedings of Expert Systems '86, The 6Th Annual Technical Conference on Research and development in expert systems III*, pages 184–194, New York, NY, USA, 1987. Cambridge University Press.

[17] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 26(10):1367–1372, 2004.

[18] N. Cristianini and J. Shawe-Taylor. *An Introduction to Support Vector Machines*. Cambridge University Press, 2000.

[19] K. Currie and A. Tate. O-plan: The open planning architecture. *Artificial Intelligence*, 52(1):49–86, 1991.

[20] B. Drabble, J. Dalton, and A. Tate. Repairing plans on the fly, 1997.

[21] Kutluhan Erol, James Hendler, and Dana S. Nau. HTN planning: Complexity and expressivity. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, volume 2, pages 1123–1128, Seattle, Washington, USA, August 1994. AAAI Press/MIT Press.

[22] M. Fox, A. Gerevini, D. Long, and I. Serina. Plan stability: Replanning versus plan repair. In *Proceedings of International Conference on AI Planning and Scheduling (ICAPS)*. AAAI Press, 2006.

[23] M. Fox and D. Long. The automatic inference of state invariants in TIM. *Journal of Artificial Intelligence Research (JAIR)*, 9:367–421, 1998.

[24] M. Fox and D. Long. The detection and exploitation of symmetry in planning problems. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 956–961, 1999.

[25] H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell. Optimal assignment kernels for attributed molecular graphs. In L. De Raedt and S. Wrobel, editors, *ICML*, volume 119 of *ACM International Conference Proceeding Series*, pages 225–232. ACM, 2005.

[26] H. Fröhlich, J. K. Wegner, F. Sieker, and A. Zell. Kernel Functions for Attributed Molecular Graphs – A New Similarity Based Approach To ADME Prediction in Classification and Regression. *QSAR Comb. Sci.*, 25:317–326, 2006.

[27] H. Fröhlich, J. K. Wegner, and A. Zell. Assignment Kernels For Chemical Compounds. In *International Joint Conference on Neural Networks 2005 (IJCNN'05)*, pages 913–918, 2005.

[28] M. R. Garey and D. S. Johnson. *Computers and Intractability : A Guide to the Theory of NP-Completeness (Series of Books in the Mathematical Sciences)*. W. H. Freeman, 1979.

[29] T. Gärtner. A survey of kernels for structured data. *SIGKDD Explor. Newsl.*, 5(1):49–58, 2003.

[30] D. Gentner. The mechanisms of analogical learning. In B. G. Buchanan and D. C. Wilkins, editors, *Readings in Knowledge Acquisition and Learning: Automating the Construction and Improvement of Expert Systems*, pages 673–694. Kaufmann, San Mateo, CA, 1993.

[31] A. Gerevini, A. Saetti, and I. Serina. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research (JAIR)*, 20:pp. 239–290, 2003.

[32] A. Gerevini and L. Schubert. On point-based temporal disjointness. *Artificial Intelligence*, 70:347–361, 1994.

[33] A. Gerevini and L. Schubert. Accelerating Partial-Order Planners: Some Techniques for Effective Search Control and Pruning. *Journal of Artificial Intelligence Research (JAIR)*, 5:95–137, Sept. 1996.

[34] A. Gerevini and I. Serina. Fast planning through greedy action graphs. In *Proceedings of the 16th National Conference of the American Association for Artificial Intelligence (AAAI-99)*, pages 503–510. AAAI Press/MIT Press, July 1999.

[35] A. Gerevini and I. Serina. Plan adaptation through planning graph analysis. In *Lecture Notes in Artificial Intelligence (AI*IA 99)*, pages 356–367. Springer-Verlag, 1999.

[36] A. Gerevini and I. Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *Proceedings of the 5th International Conference on Artificial Intelligence Planning and Scheduling (AIPS-00)*, pages 112–121. AAAI Press/MIT Press, 2000.

[37] K. Hammond. Explaining and repairing plans that fail. *Artificial Intelligence*, 45:173–228, 1990.

[38] S. Hanks and D. S. Weld. Systematic adaptation for case-based planning. In J. Hendler, editor, *AIPS-92: Proc. of the First International Conference on Artificial Intelligence Planning Systems*, pages 96–105. Kaufmann, San Mateo, CA, 1992.

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

[40] P. Hansen. Upper bounds for the stability number of a graph. *Rev. Roumaine Math. Pures Appl.*, 24:1195–1199, 1979.

[41] D. Haussler. Convolution kernels on discrete structures. Technical Report UCS-CRL-99-10, UC Santa Cruz, 1999.

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

[43] M. Johnson. *Relating metrics, lines and variables defined on graphs to problems in medicinal chemistry*. John Wiley & Sons, Inc., New York, NY, USA, 1985.

[44] S. Kambhampati. A theory of plan modification. In *Proceedings of the Eighth National Conference on Artificial Intelligence (AAAI-90)*, pages 176–182, Boston, Massachusetts, USA, July 1990. AAAI Press/MIT Press.

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

[46] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled graphs. In T. Fawcett and N. Mishra, editors, *ICML*, pages 321–328. AAAI Press, 2003.

[47] H.A. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In Howard Shrobe and Ted Senator, editors, *Proceedings of the Thirteenth National Conference of the American Association for Artificial Intelligence (AAAI-96)*, pages 1194–1201. AAAI Press, 1996.

[48] V. Kuchibatla and H. Muñoz-Avila. An analysis on transformational analogy: General framework and complexity. In *ECCBR*, volume 4106 of *Lecture Notes in Computer Science*, pages 458–473. Springer, 2006.

[49] H. W. Kuhn. The Hungarian method for the assignment problem. *Naval Research Logistic Quarterly*, 2:83–97, 1955.

[50] D. B. Leake, editor. *Case-Based Reasoning*. The MIT Press, Cambridge, Massachusetts, 1996.

[51] D. B. Leake, A. Kinley, and D. C. Wilson. Case-based similarity assessment: Estimating adaptability from experience. In *Proceedings of the 14th National Conference on Artificial Intelligence -AAAI'97*, pages 674–679, Menlo Park, CA, USA, 1997. AAAI Press.

[52] P. Liberatore. On the complexity of case-based planning. *Journal of Experimental & Theoretical Artificial Intelligence*, 17(3):283–295, 2005.

[53] M. Likhachev and S. Koenig. A generalized framework for lifelong planning a* search. In Biundo et al. [6], pages 99–108.

[54] D. Lin. An information-theoretic definition of similarity. In J. W. Shavlik, editor, *ICML*, pages 296–304. Morgan Kaufmann, 1998.

[55] R. Y. Liu. An upper bound on the chromatic number of a graph. *J. Xinjiang Univ. Natur. Sci.*, 6:24–27, 1989.

[56] D. Long and M. Fox. The 3rd international planning competition: Results and analysis. *Journal of Artificial Intelligence Research (JAIR)*, 10:1–59, 2003.

[57] D. McAllester and D. Rosenblitt. Systematic nonlinear planning. In *Proceedings of the Ninth National Conference on Artificial Intelligence (AAAI-91)*, pages 634–639, July 1991.

[58] J. Mercer. Functions of positive and negative type and their connection with the theory of integral equations. *Philos. Trans. Roy. Soc. London*, A 209:415–446, 1909.

[59] H. Muñoz-Avila and M. Cox. Case-based plan adaptation: An analysis and review. *IEEE Intelligent Systems*, 23(4):75–81, 2008.

[60] H. Muñoz-Avila and J. Hüllen. Feature weighting by explaining case-based planning episodes. In *EWCBR '96: Proceedings of the Third European Workshop on Advances in Case-Based Reasoning*, pages 280–294, London, UK, 1996. Springer-Verlag.

[61] B. Nebel and J. Koehler. Plan reuse versus plan generation: A complexity-theoretic perspective. *Artificial Intelligence- Special Issue on Planning and Scheduling*, 76:427–454, 1995.

[62] M. Neuhaus and H. Bunke. A convolution edit kernel for error-tolerant graph matching. volume 4, pages 220–223, Washington, DC, USA, 2006. IEEE Computer Society.

[63] M. Neuhaus and H. Bunke. *Bridging the gap between Graph Edit Distance and Kernel Machines*. World Scientific, 2007.

[64] A. N. Papadopoulos and Y. Manolopoulos. Structure-based similarity search with graph histograms. In *In Proceedings of the 10th International Workshop on Database & Expert Systems Applications*, pages 174–178. IEEE Computer Society Press, 1999.

[65] M. E. Pollack, D. Joslin, and M. Paolucci. Flaw selection strategies for partial-order planning. *Journal of Artificial Intelligence Research (JAIR)*, 6:223–262, 1997.

[66] J. W. Raymond, E. J. Gardiner, and P. Willett. Rascal: Calculation of graph similarity using maximum common edge subgraphs. *The Computer Journal*, 45(6):631–644, June 2002.

[67] K. Riesen and H. Bunke. Approximate graph edit distance computation by means of bipartite graph matching. *Image Vision Comput.*, 27(7):950–959, 2009.

[68] R. Rosipal and L. J. Trejo. Kernel partial least squares regression in reproducing kernel hilbert space. *J. Mach. Learn. Res.*, 2:97–123, 2002.

[69] B. H. Ross. Some psychological results on case-based reasoning. In *Proc. of a Workshop on Case-Based Reasoning*, pages 144–147, Pensacola Beach, FL, 1989.

[70] F. Ruskey, R. Cohen, P. Eades, and A. Scott. Alley cats in search of good homes. *Twenty-fifth Southeastern Conference on Combinatorics, Graph Theory and Computing*, 102:97–110, 1994.

[71] B. Scholkopf and A. J. Smola. *Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond*. MIT Press, Cambridge, MA, USA, 2001.

[72] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference of the American Association for Artificial Intelligence (AAAI-94)*, pages 337–343, Seattle, WA, 1994. Morgan Kaufmann.

[73] R. G. Simmons. A theory of debugging plans and interpretations. In *Proc. of AAAI-88*, pages 94–99, St. Paul, MN, 1988.

[74] B. Smyth and M. T. Keane. Adaptation-guided retrieval: Questioning the similarity assumption in reasoning. *Artificial Intelligence*, 102(2):249–293, 1998.

[75] B. Smyth and E. McKenna. Footprint-based retrieval. In K. D. Althoff, R. Bergmann, and K. Branting, editors, *ICCBR*, volume 1650 of *Lecture Notes in Computer Science*, pages 343–357. Springer, 1999.

[76] L. Spalazzi. A survey on case-based planning. *Artificial Intelligence Review*, 16(1):3–36, 2001.

[77] B. Srivastava, T. A. Nguyen, A. Gerevini, S. Kambhampati, M. B. Do, and I. Serina. Domain independent approaches for finding diverse plans. In M. M. Veloso, editor, *IJCAI*, pages 2016–2022, 2007.

[78] A. Tate. Generating project networks. In *Proceedings of the Fifth International Joint Conference on Artificial Intelligence (IJCAI-77)*, pages 888–889, Cambridge, MA, 1977. MIT.

[79] F. Tonidandel and M. Rillo. The FAR-OFF system: A heuristic search case-based planning. In M. Ghallab, J. Hertzberg, and P. Traverso, editors, *AIPS*, pages 302–311. AAAI, 2002.

[80] R. van der Krogt and M Weerdt. Plan repair as an extension of planning. In Biundo et al. [6], pages 161–170.

[81] M. Veloso. Learning by analogical reasoning in general problem solving. Technical report, CMU-CS-92-174, Department of Computer Science, Carnegie Mellon University, 1992.

[82] M. Veloso. *Planning and Learning by Analogical Reasoning*, volume 886 of *Lecture Notes in Artificial Intelligence and Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, USA, 1994.

[83] S. V. N. Vishwanathan and A. J. Smola. Fast kernels for string and tree matching. In S. Becker, S. Thrun, and K. Obermayer, editors, *NIPS*, pages 569–576. MIT Press, 2002.

[84] S. Vosniadou and A. Ortony, editors. *Similarity and analogical reasoning*. Cambridge University Press, New York, NY, USA, 1989.

[85] C. Walther. A mechanical solution of Schubert's steamroller by many-sorted resolution. *Artificial Intelligence*, 26(2):217–224, 1985.

[86] F. Wilcoxon and R. A. Wilcox. *Some Rapid Approximate Statistical Procedures*. American Cyanamid Co., Pearl River, NY, USA., 1964.

A Proofs

Theorem 3 `obj_match`is NP-hard.
Proof. Similarly to the Nebel and Koehler's analysis [61], NP-hardness is proved by a polynomial transformation from the subgraph isomorphism problem for directed graphs, which is NP-Complete ([28] p. 202), to `obj_match`. The subgraph isomorphism problem is defined as follows:
Instance: Two directed Graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$.
Question: Does G_2 contain a subgraph isomorphic to G_1, i.e., do there exist a subsets $V \subseteq V_2$ and a subset $E \subseteq E_2$ such that $|V| = |V_1|$, $|E| = |E_1|$, and there exists a one-to-one function $\mu : V_1 \rightarrow V$ satisfying $(u, v) \in E_1$ if and only if $(\mu(u), \mu(v)) \in E$?

Given an instance of the subgraph isomorphism problem, we construct an instance of `obj_match` as follows; let

$$\Pi_1 = \langle \mathcal{P}_r(\mathbf{O}_1, \mathbf{P}_1), \mathcal{I}_1, \mathcal{G}_1, \mathcal{O}p_1 \rangle$$

$$\Pi_2 = \langle \mathcal{P}_r(\mathbf{O}_2, \mathbf{P}_2), \mathcal{I}_2, \mathcal{G}_2, \mathcal{O}p_2 \rangle$$

be two planning instances such that

$$\mathbf{O}_1 = \mathbf{O}_2 = V_1 \cup V_2$$

$$\mathbf{P}_1 = \mathbf{P}_2 = \{p\}$$

$$\mathcal{I}_1 = \mathcal{I}_2 = \varnothing$$

$$\mathcal{G}_1 = \{p(u, v) | (u, v) \in E_1\}$$

$$\mathcal{G}_2 = \{p(u, v) | (u, v) \in E_2\}$$

$$\mathcal{O}p_1 = \mathcal{O}p_2$$

Now G_2 contains a subgraph isomorphic to G_1 iff there exists a mapping $\mu(\cdot)$ such that $|\mu(\mathcal{G}_1) \cap \mathcal{G}_2| = |E_1|$ and

$$simil_\mu(\Pi_1, \Pi_2) = \frac{|\mu(\mathcal{G}_1) \cap \mathcal{G}_2| + |\mu(\mathcal{I}_1) \cap \mathcal{I}_2|}{|\mathcal{G}_2| + |\mu(\mathcal{I}_1)|} = \frac{|E_1| + |\varnothing|}{|E_2|}$$

Note that G_2 is isomorphic to G_1 iff there exists a mapping $\mu(\cdot)$ such that $|\mu(\mathcal{G}_1) \cap \mathcal{G}_2| = |E_1| = |E_2|$ and $simil_\mu(\Pi_1, \Pi_2) = 1$

□

Theorem 4 R_1 is a kernel function.
Proof. We have to show that given a set of patterns $\mathbf{x}^1, ..., \mathbf{x}^n$ the kernel matrix $\mathbf{R} = (R_1(\mathbf{x}^i, \mathbf{x}^j))_{i,j}$ is symmetric and positive semidefinite [71].

Clearly, R_1 is symmetric, because of the definition. Let ζ denote a permutation of an n-subset of natural numbers $1, ..., m$, or a permutation of an m-subset of natural numbers $1, ..., n$, respectively; then for any ζ it is

$$k_v(n_{\zeta(1)}(\mathbf{x}), n_1(\mathbf{x})) \cdot k_e(e_{\zeta(1)}(\mathbf{x}), e_1(\mathbf{x})) + ... + k_v(n_{\zeta(n)}(\mathbf{x}), n_n(\mathbf{x})) \cdot k_e(e_{\zeta(n)}(\mathbf{x}), e_n(\mathbf{x})) \quad (15)$$

$$\leq \frac{1}{2}(k_v(n_1(\mathbf{x}), n_1(\mathbf{x})) \cdot k_e(e_1(\mathbf{x}), e_1(\mathbf{x})) + k_v(n_{\zeta(1)}(\mathbf{x}), n_{\zeta(1)}(\mathbf{x})) \cdot k_e(e_{\zeta(1)}(\mathbf{x}), e_{\zeta(1)}(\mathbf{x})) + ... \quad (16)$$

$$... + k_v(n_n(\mathbf{x}), n_n(\mathbf{x})) \cdot k_e(e_n(\mathbf{x}), e_n(\mathbf{x})) + k_v(n_{\zeta(n)}(\mathbf{x}), n_{\zeta(n)}(\mathbf{x})) \cdot k_e(e_{\zeta(n)}(\mathbf{x}), e_{\zeta(n)}(\mathbf{x})))$$

$$= \sum_i k_v(n_i(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_i(\mathbf{x}), e_i(\mathbf{x})) \quad (17)$$

because for any i

$$2 \cdot k_v(n_{\zeta(i)}(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_{\zeta(i)}(\mathbf{x}), e_i(\mathbf{x})) \leq$$
$$\leq k_v(n_i(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_i(\mathbf{x}), e_i(\mathbf{x})) + k_v(n_{\zeta(i)}(\mathbf{x}), n_{\zeta(i)}(\mathbf{x})) \cdot k_e(e_{\zeta(i)}(\mathbf{x}), e_{\zeta(i)}(\mathbf{x}))$$

since k_v and k_e are positive semidefinite kernel functions and the product of two kernel functions is a kernel function. Now, if we take the maximum over all ζ then $R_1(\mathbf{x}, \mathbf{x})$ = (15)=(16)=(17).

Similarly $R_1(\mathbf{y}, \mathbf{y}) = \sum_j k_v(n_j(\mathbf{y}), n_j(\mathbf{y})) \cdot k_e(e_j(\mathbf{y}), e_j(\mathbf{y}))$. Without loss of generality we can assume that $|\mathbf{y}| \geq |\mathbf{x}|$. Further it holds for all $\alpha, \beta \in I\!R$ and i, j

$$2\alpha\beta k_v(n_i(\mathbf{x}), n_j(\mathbf{y})) \cdot k_e(e_i(\mathbf{x}), e_j(\mathbf{y})) \leq \quad (18)$$
$$\alpha^2 k_v(n_i(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_i(\mathbf{x}), e_i(\mathbf{x})) + \beta^2 k_v(n_j(\mathbf{y}), n_j(\mathbf{y})) \cdot k_e(e_j(\mathbf{y}), e_j(\mathbf{y}))$$

because k_v and k_e are positive semidefinite kernel functions. It is

$$\alpha^2 R_1(\mathbf{x}, \mathbf{x}) - 2\alpha\beta R_1(\mathbf{x}, \mathbf{y}) + \beta^2 R_1(\mathbf{y}, \mathbf{y}) = \quad (19)$$
$$\alpha^2 \sum_i k_v(n_i(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_i(\mathbf{x}), e_i(\mathbf{x})) -$$
$$-2\alpha\beta max_\pi \sum_i k_v(n_i(\mathbf{x}), n_{\pi(i)}(\mathbf{y})) \cdot k_e(e_i(\mathbf{x}), e_{\pi(i)}(\mathbf{y})) +$$
$$+\beta^2 \sum_j k_v(n_j(\mathbf{y}), n_j(\mathbf{y})) \cdot k_e(e_j(\mathbf{y}), e_j(\mathbf{y}))$$

By definition of R_1 the second sum of previous equation has $min(|\mathbf{x}|, |\mathbf{y}|) = |\mathbf{x}|$ addends. Using (18) we have

$$(19) \geq \sum_{i=1}^{|\mathbf{x}|} (\, \alpha^2 k_v(n_i(\mathbf{x}), n_i(\mathbf{x})) \cdot k_e(e_i(\mathbf{x}), e_i(\mathbf{x}) - 2\alpha\beta k_v(n_i(\mathbf{x}), n_{\zeta(i)}(\mathbf{y})) \cdot k_e(e_i(\mathbf{x}), e_{\zeta(i)}(\mathbf{y})) +$$
$$+\beta^2 k_v(n_{\zeta(i)}(\mathbf{y}), n_{\zeta(i)}(\mathbf{y})) \cdot k_e(e_{\zeta(i)}(\mathbf{y}), e_{\zeta(i)}(\mathbf{y})\,) \geq 0. \quad (20)$$

This proofs the positive semidefiniteness of each 2×2 kernel matrix. From this we can generalise the result to $n \times n$ matrices by induction using the assumption that k_v and k_e are non-negative. Suppose we already know that each $n \times n$ kernel matrix $\mathbf{R} = (R_1(\mathbf{x}^i, \mathbf{x}^j))_{i,j}$ for a set of objects $\mathbf{x}^1, ..., \mathbf{x}^n$ is positive semidefinite. Now assume we extend the matrix to size $n + 1 \times n + 1$ by adding an object $\mathbf{x}^{n+1}$. It is

$$\sum_{i,j=1}^{n+1} \mathbf{v}_i \mathbf{v}_j \mathbf{R}_{i,j} = \sum_{i,j=1}^{n} \mathbf{v}_i \mathbf{v}_j \mathbf{R}_{i,j} + 2 \sum_{j=1}^{n} \mathbf{v}_{n+1} \mathbf{v}_j \mathbf{R}_{n+1,j} + \mathbf{v}_{n+1}^2 \mathbf{R}_{n+1,n+1} \quad (21)$$

By induction assumption we know the first part of (21) to be non-negative. Furthermore, by definition k_v and k_e are non-negative and thus also R_1 is non-negative. Hence, we have $\mathbf{v}_{n+1}^2 \mathbf{R}_{n+1,n+1} \geq 0$.

Therefore, in order to make $(21) < 0$ we have to suppose $2 \cdot \sum_{j=1}^n \mathbf{v}_{n+1} \mathbf{v}_j \mathbf{R}_{n+1,j} < 0$ and similarly to (18) we have that

$$2 \cdot \mathbf{v}_{n+1} \mathbf{v}_j \mathbf{R}_{n+1,j} \leq \mathbf{v}_{n+1}^2 \mathbf{R}_{n+1,n+1} + \mathbf{v}_j^2 \mathbf{R}_{j,j}$$

this leads to

$$2 \cdot \sum_{j=1}^{n} \mathbf{v}_{n+1}\mathbf{v}_j\mathbf{R}_{n+1,j} \leq \sum_{j=1}^{n} \left(\mathbf{v}_{n+1}^2\mathbf{R}_{n+1,n+1} + \mathbf{v}_j^2\mathbf{R}_{j,j}\right) < 0$$

which is a contradiction to the non-negativity of R_1 . Hence, it is $(21) \geq 0$, which proofs the theorem. □

Theorem 5 Let $\gamma(l) = p^l$ with $p \in (0, 0.5)$. If there exists a $C \in \mathbb{R}^+$, such that $k_v(v_1, v_2) \leq C$ for all v_1, v_2 and $k_e(e_1, e_2) \leq 1$ for all e_1, e_2 then (eq. 10) converges for $L \to \infty$.

Proof. It is

$$R_1(u,v) \leq \frac{1}{max(|E(u)|,|E(v)|)} \sum_{i=1}^{min(|E(u)|,|E(v)|)} max_{v_1,v_2} k_v(v_1,v_2) \cdot max_{e_1,e_2} k_e(e_1,e_2) \leq$$

$$\leq \frac{min(|E(u)|,|E(v)|)}{max(|E(u)|,|E(v)|)} C \leq C$$

and thus

$R_2(u,v) \leq$

$$\frac{1}{|\mathcal{N}^i(v)| \cdot |\mathcal{N}^i(u)|} \sum_{h,h'} max_{n_h^i(v), n_{h'}^i(u)} R_1\left(n_h^i(v), n_{h'}^i(u)\right) \cdot max_{e_h^i(v), e_{h'}^i(u)} k_e\left(e_h(v), e_{h'}^i(u)\right) +$$

$$+ \frac{1}{|\mathcal{N}^o(v)| \cdot |\mathcal{N}^o(u)|} \sum_{h,h'} max_{n_h^o(v), n_{h'}^o(u)} R_1\left(n_h^o(v), n_{h'}^o(u)\right) \cdot max_{e_h^o(v), e_{h'}^o(u)} k_e\left(e_h^o(v), e_{h'}^o(u)\right) \leq$$

$$\leq \frac{1}{|\mathcal{N}^i(v)| \cdot |\mathcal{N}^i(u)|} \sum_{i=1}^{|\mathcal{N}^i(v)|} \sum_{j=1}^{|\mathcal{N}^i(u)|} C + \frac{1}{|\mathcal{N}^o(v)| \cdot |\mathcal{N}^o(u)|} \sum_{i=1}^{|\mathcal{N}^o(v)|} \sum_{j=1}^{|\mathcal{N}^o(u)|} C = 2 \cdot C$$

Similarly we can show that $R_l(u,v) \leq 2^{l-1}C$ for $l = 3, ..., L$. Therefore we have (eq. 10)$\leq C + pC + p^2 2C + ... + p^L 2^{L-1} C \leq C + \sum_{l=1}^{L} (2p)^l C$

which converges for $L \to \infty$ if $p \in (0, 0.5)$.

□

B Variants

As previously described (see page 35), our tests have been conducted on a series of variants of problems from different standard benchmark domains of the 2nd, 3rd and 5th International Planning Competitions.[37] The variant problems have been generated by taking six problems from each benchmark test suite (except for the Logistics domain) and then randomly modifying the initial state and goal states for a total of 216 planning problems for each domain. The problems considered are:

- probblocks-40-0, probblocks-60-0, probblocks-80-0, probblocks-100-0, probblocks-120-1 and probblocks-140-1 for BlocksWorld Additionals;
- randomly selected from logistics-16-0 to logistics-100-1 for Logistics Additionals Track2;
- pfile14, pfile17, pfile20, pfile-HC03, pfile-HC06, pfile-HC09 for DriverLog ;
- pfile14, pfile17, pfile20, pfile-HC14, pfile-HC17, pfile-HC20 for ZenoTravel;
- pfile35, pfile36, pfile37, pfile38, pfile39, pfile40 for Rovers-IPC5;
- pfile25, pfile26, pfile27, pfile28, pfile29, pfile30 for TPP.

In the following we present a brief description of the domains and of the operators used in order to modify the initial and the goal states of a base problem. In order to modify the initial state, we randomly choose a completely instantiated "noisy" operator among those with all the preconditions satisfied; the effects of the "noisy" operator determine a new initial state.

With respect to the goals, we propagate the effects of the actions of the solution plan of the base problem in order to define a complete goal state; then we randomly choose a completely instantiated "noisy" operator among those with all the preconditions satisfied and at least one goal of the base problem that belongs to them. The effects of the "noisy" operator change the goal state; in particular the negative effects delete the corresponding goals, while the positive effects are added to the goal set.

BlocksWorld

The BlocksWorld is a standard AI planning benchmark. In this domain, there is a Table on which a number of blocks are stacked. A block can be placed either on the Table or on top of another block, but at most one block can be on top of any given block. There is a robot arm that picks up and places the blocks, so only one block can be moved at a time.

The `NOISE-falldown` noisy operator is used to randomly split a pile of blocks into piles; on the contrary the `NOISE-pile-up` operator is used to randomly pile up a pile of blocks on the top of another one:

```
(:action NOISE-FALLDOWN
:parameters (?x ?y)
:precondition (and (not (= ?x ?y)) (on ?x ?y) (handempty))
:effect
(and (clear ?y) (not (on ?x ?y)) (ontable ?x)))

(:action NOISE-PILE-UP
:parameters (?x ?y)
```

[37] The IPCs test problems that we have used are available at the following websites:
For IPC2, `http://www.cs.toronto.edu/aips2000/`,
For IPC3, `http://planning.cis.strath.ac.uk/competition/`,
For IPC5, `http://ipc5.ing.unibs.it`.

```
:precondition (and (not (= ?x ?y))(ontable ?x) (clear ?y) (handempty))
:effect
(and (not (ontable ?x)) (not (clear ?y)) (on ?x ?y)))
```

DriverLog

This domain involves driving trucks around for delivering packages between locations. The complication is that the trucks require drivers who must walk between trucks in order to drive them. The paths for walking and the roads for driving from different locations are specified in the problem description.

The `NOISE-move-package` and `NOISE-move-driver` noisy operators are used to randomly change the location of a *package* and of a *driver* respectively:

```
(:action NOISE-MOVE-PACKAGE
:parameters
(?obj - obj ?loc-from - location ?loc-to - location ?loc-control
- location)
:precondition
(and (at ?obj ?loc-from) (link ?loc-control ?loc-to))
:effect
(and (not (at ?obj ?loc-from)) (at ?obj ?loc-to)))

(:action NOISE-MOVE-DRIVER
:parameters
(?driver - driver ?loc-from - location ?loc-to - location ?loc-control
- location)
:precondition
(and (at ?driver ?loc-from) (path ?loc-control ?loc-to))
:effect
(and (not (at ?driver ?loc-from)) (at ?driver ?loc-to)))
```

Logistics

The logistics domain is a standard benchmark in AI planning. In this domain, there are several cities and in each city a number of places. The goal is to move a number of packages from their initial places to their respective destinations. Packages can be loaded into (and of course unloaded from) vehicles of different kinds (trucks or airplanes). Trucks can only move between places in the same city, while airplanes can only move between places of a particular kind, namely airports.

The `NOISE-move-package` and the `NOISE-fly-airplane` noisy operators are used to change the location of a package and of an airplane respectively:

```
(:action NOISE-MOVE-PACKAGE
:parameters (?obj ?loc-from ?loc-to)
:precondition
(and (package ?obj) (location ?loc-from) (location ?loc-to)
```

```
(at ?obj ?loc-from))
:effect
(and (not (at ?obj ?loc-from)) (at ?obj ?loc-to)))

(:action NOISE-DELETE-CONN
:parameters (?loc-from ?loc-to)
:precondition
(and (conn ?loc-from ?loc-to) (airport ?loc-from) (airport ?loc-to)
)
:effect
(and (not (conn ?loc-from ?loc-to))))

(:action NOISE-fly-airplane
:parameters (?airplane ?loc-from ?loc-to)
:precondition
(and (airplane ?airplane) (airport ?loc-from) (airport ?loc-to)
(at ?airplane ?loc-from) (conn ?loc-from ?loc-to))
:effect
(and (not (at ?airplane ?loc-from)) (at ?airplane ?loc-to)))
```

Rovers

This domain requires that a collection of rovers navigates a planet surface, finding samples, analyse them and communicating the data back to a lander. In addition, a minor subtlety in the encoding is used to prevent parallel communication between rovers and the lander. Because deletes occur before adds, this has the overall effect of leaving the channel free, but it makes the fact a "moving target" which prevents a concurrent action from using the fact. Some planners find this mechanism difficult to handle.

The `NOISE-communicate_soil_data`, the `NOISE-communicate_rock_data` and the `NOISE-communicate_image_data` noisy operators are used to change the status (either "*communicated*" or "*not communicated*") of *soil* data, *rock* data and *image* data respectively from a waypoint x to a waypoint y:

```
(:action NOISE-communicate_soil_data
:parameters (?r - rover ?x - waypoint ?y - waypoint)
:precondition
(and (communicated_soil_data ?x) (at_soil_sample ?y) )
:effect
(and (not (communicated_soil_data ?x) ) (communicated_soil_data
?y) ) )

(:action NOISE-communicate_rock_data
:parameters (?r - rover ?x - waypoint ?y - waypoint)
:precondition
(and (communicated_rock_data ?x) (at_rock_sample ?y) )
:effect
(and (not (communicated_rock_data ?x) ) (communicated_rock_data
```

```
?y) ) )

(:action NOISE-communicate_image_data
:parameters ( ?x - objective ?y - objective ?m - mode ?i - camera)
:precondition
(and (communicated_image_data ?x ?m) (calibration_target ?i ?y)
)
:effect
(and (not (communicated_image_data ?x ?m) ) (communicated_image_data
?y ?m) ) )
```

TPP

This is a relatively recent planning domain that has been investigated in Operations Research (OR) for several years. The Travelling Purchase Problem (TPP) is a known generalisation of the Travelling Salesman Problem. We have a set of products and a set of markets. Each market is provided with a limited amount of each product at a known price. The TPP consists in selecting a subset of markets such that a given demand of each product can be purchased, minimising the routing cost and the purchasing cost.

The `NOISE_drive` and the `NOISE_on-sale` noisy operators are used to randomly change the location of a truck and to randomly change the sale conditions of some goods respectively:

```
(:action NOISE_drive
:parameters (?t - truck ?from ?to - place)
:precondition
(and (at ?t ?from))
:effect
(and (not (at ?t ?from)) (at ?t ?to)))

(:action NOISE_on-sale
:parameters (?g - goods ?m1 - market ?l1 - level ?m2 - market ?l2
- level )
:precondition
(and (on-sale ?g ?m1 ?l1) (on-sale ?g ?m2 ?l2) )
:effect
(and (on-sale ?g ?m1 ?l2) (not (on-sale ?g ?m1 ?l1)) (on-sale ?g
?m2 ?l1) (not (on-sale ?g ?m2 ?l2))
))
```

ZenoTravel

This transportation domain involves transporting people around on planes, using two different modes of movement: fast and slow. The key to this domain is that, where the expressive power of the numeric tracks is used, the fast movement consumes fuel faster than slow movement, making the search for a good quality plan much harder.

For this domain, we defined five noisy operators:

- the `NOISE-fuel` operator can be used to randomly change the fuel level of an aircraft;
- the `NOISE-fly` and the `NOISE-zoom` operators can be used to randomly modify the location of an aircraft using different amount of fuel;
- the `NOISE-move-package` operator can be used to randomly change the location of a package;
- the `NOISE-debark` operator can be used to randomly modify the objects inside an aircraft.

```
(:action NOISE-refuel
:parameters (?a - aircraft ?c - city ?l - flevel ?l1 - flevel)
:precondition
(and (fuel-level ?a ?l) (next ?l1 ?l) (at ?a ?c))
:effect
(and (fuel-level ?a ?l1) (not (fuel-level ?a ?l))))

(:action NOISE-fly
:parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 - flevel)
:precondition
(and (at ?a ?c1) (fuel-level ?a ?l1) (next ?l2 ?l1))
:effect
(and (not (at ?a ?c1)) (at ?a ?c2) (not (fuel-level ?a ?l1)) (fuel-level
?a ?l2)))

(:action NOISE-zoom
:parameters (?a - aircraft ?c1 ?c2 - city ?l1 ?l2 ?l3 - flevel)
:precondition
(and (at ?a ?c1) (fuel-level ?a ?l1) (next ?l2 ?l1) (next ?l3 ?l2)
)
:effect
(and (not (at ?a ?c1)) (at ?a ?c2) (not (fuel-level ?a ?l1)) (fuel-level
?a ?l3) ) )

(:action NOISE-MOVE-PACKAGE
:parameters (?p - person ?c1 - city ?c2 - city)
:precondition
(and (at ?p ?c1) )
:effect
(and (not (at ?p ?c1)) (at ?p ?c2) ))

(:action NOISE-debark
:parameters (?p - person ?a - aircraft ?c - city)
:precondition
(and (in ?p ?a) (at ?a ?c))
:effect
(and (not (in ?p ?a)) (at ?p ?c)))
```

C Additional results

In this section we report additional results regarding OAKPLAN.

Tables 9–11 (see page 79) show the results of the Wilcoxon signed rank test comparing OAKPLAN-Nns with OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$. Each cell in Table 9 gives the result of a comparison between the performance of OAKPLAN-Nns and another tested planner in terms of CPU-time. In this Table, as well as in the next ones concerning an analysis based on the Wilcoxon test, the T-distribution used by the Wilcoxon test is approximately a normal distribution when the number of samples is sufficiently large. Therefore, the cells of the Figure contain the z-value and the p-value characterising the normal distribution. The higher the z-value is, the more significant the difference of the performance is. The p-value represents the level of significance in the performance gap. We use a confidence level of 99.9%; hence, if the p-value is lower than 0.001, then the performance of the two planners is statistically different. The third value in each cell is the number of problems solved by at least one planner. An up arrow in these cells indicates that the first planner named in the title performs worse than the other planner compared, on the contrary a down arrow indicates that the first planner performs better than the other planner compared. We considered all the test problems that can be attempted by both the compared planners and that are solved by at least one of them. When a planner does not solve a problem, the corresponding CPU-time is set to 1800 seconds. Tables 10 and 11 (see page 88) give the results of the Wilcoxon signed rank test about the plan quality and distance values for OAKPLAN-Nns and the other tested planners. The third value in each cell is the number of problems solved by both the planners compared.

Tables 12–14 (see page 88) show the results of the Wilcoxon signed rank test comparing OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons . Each cell in Table 12 gives the result of a comparison between the performance of OAKPLAN-Nns and another tested planner in terms of CPU-time. Similarly Tables 10 and 11 give the results of the Wilcoxon signed rank test about the plan quality and distance values for OAKPLAN-Nns and the other tested planners.

Moreover, Table 15 (see page 89) shows the results of the Wilcoxon signed rank test comparing OAKPLAN-Nns with DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5. Each cell in Table 15 gives the result of a comparison between the performance of OAKPLAN-Nns and another tested planner in terms of CPU-time. We considered all the test problems that can be attempted by both the planners compared and that are solved by at least one of them. When a planner does not solve a problem, the corresponding CPU-time is set to 1800 seconds (i.e., the maximum CPU-time limit). The third value in each cell is the number of problems solved by at least one planner. Tables 16 and 17 (see page 90) give the results of the Wilcoxon signed rank test about the plan quality and distance values for OAKPLAN-Nns and the other planners tested. The third value in each cell is the number of problems solved by both the planners compared.

In Figures 29 – 33 (see page 80) we compare OAKPLAN-Ons vs. OAKPLAN-small-Ons in the different planning benchmark domains considered; in particular we show the CPU time (on a logarithmic scale) required to find the first solution, the number of different actions with respect to the input plan of the adaptation process and the plan qualities considering different case bases. Similarly in Figure 34 – 39 (see page 85) we compare OAKPLAN-Nns vs. OAKPLAN-small-Nns. In Figures 40 – 45 (see page 94) we examine the behaviour of OAKPLAN-Ons vs OAKPLAN-Nns and in Figures 46 – 51 (see page 100) the behaviour of

CPU-time Analysis - OAKPLAN variants					
	Nns *vs* Nns-$\mathcal{K}_{base}$	Nns *vs* Nns-$\mathcal{K}_{node}$	Nns *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-adapt-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−20.2328	−29.0856	−26.8957	−29.7681	−16.3912
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 1233	↓ 1234	↓ 1232	↓ 1232	↓ 1004

CPU-time Analysis - OAKPLAN variants					
	Nns-$\mathcal{K}_{base}$ *vs* Nns-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−23.9940	−7.3736	−8.2873	−25.1029	−20.7260
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 890	↑ 974	↓ 991	↑ 975	↑ 753

Table 9: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ in terms of CPU-time for our benchmark problems.

OAKPLAN-small-Ons vs OAKPLAN-small-Nns. Moreover in Figures 52 – 63 (see page 106) we compare OAKPLAN-Nns and OAKPLAN-Ons vs the other state of the art planners examined.

In Figure 64 (see page 118) we give a graphical representation of the performance of OAKPLAN-Nns compared to the other planners in terms of CPU-time, plan qualities and distance values. Each point corresponds to a problem solved by OAKPLAN-Nns and another reference planner. If a point is above the solid diagonal, then OAKPLAN-Nns performs better than the other planner and vice versa. The distance of a point from the main diagonal indicates the performance gap (the greater the distance, the greater the gap). It easy to see that, in general, OAKPLAN-Nns finds a plan more quickly, with a lower plan distance and similar plan quality with respect to the other planners. Examining the CPU-time we can see that the other planners are faster than OAKPLAN-Nns in a number of variants; this usually happens in the simplest variants of the benchmark problems where the matching process is the most significant part of the total adaptation process of OAKPLAN-Nns and, considering the LPG planner, in all the variants of the Rovers domain.
Then in Figures 65 – 76 (see page 119) we examine the scatterplots produced by OAKPLAN-Nns and OAKPLAN-Ons vs the other planners in the different planning benchmark domains.

Finally Figure 77 (see page 131) reports the Box & Whiskers plots of the similarity values produced using the $\mathcal{K}_{\mathcal{N}}$, $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions. These results are grouped considering the test set problems (Ik-Gx variants) with k initial changes (reported on the top of the Figures) and x goal changes reported on the x-axis.

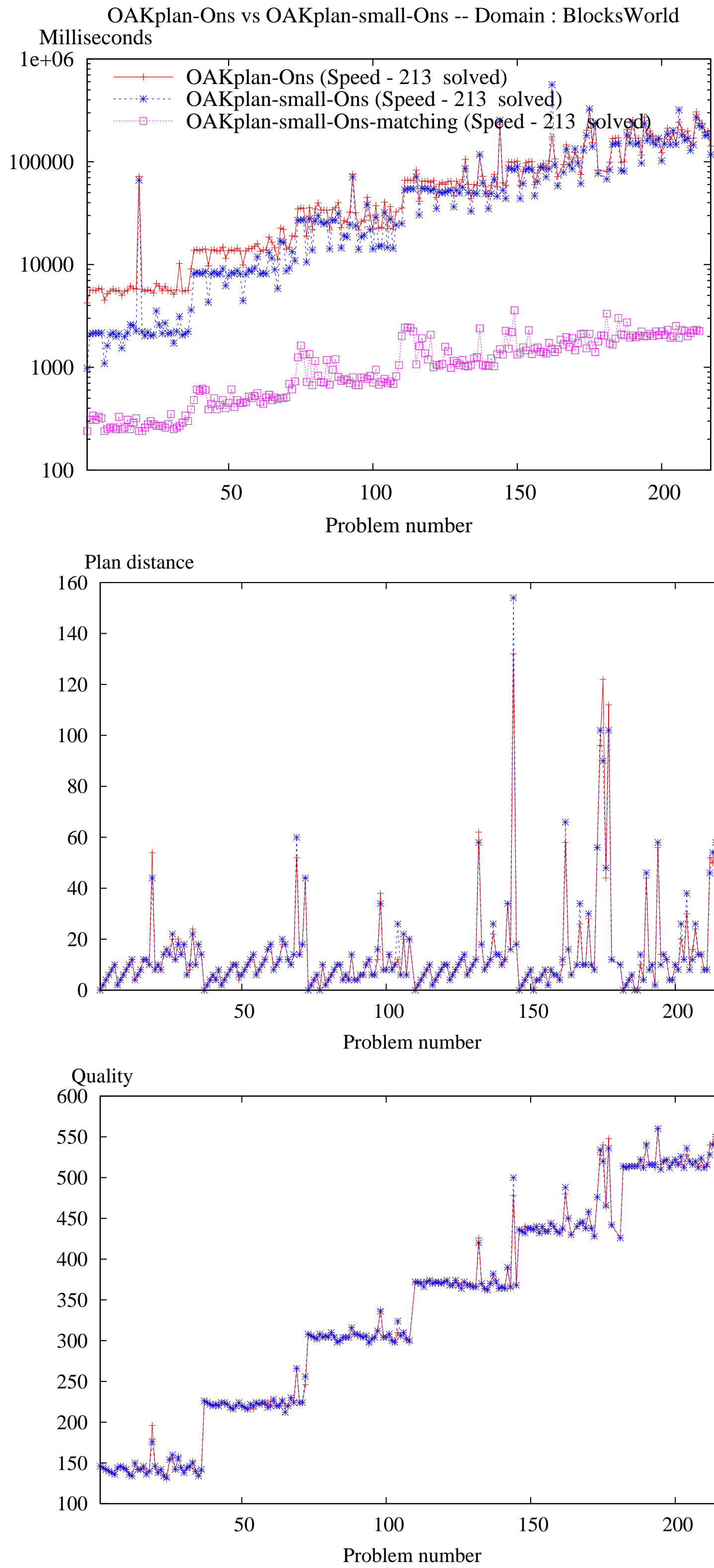
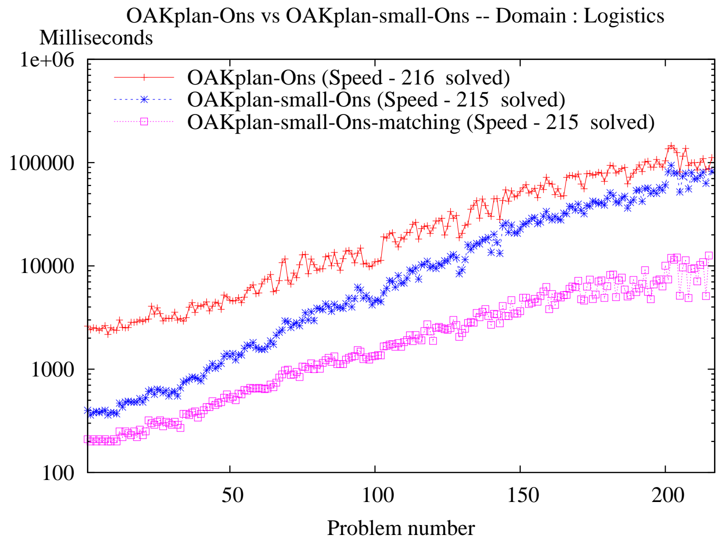
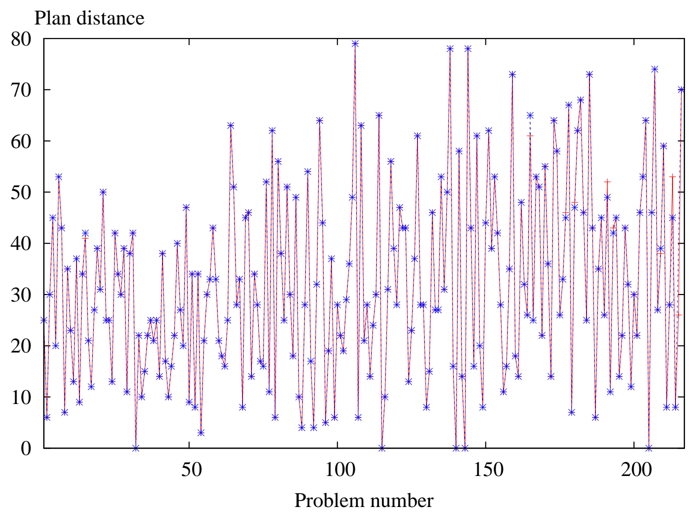


Figure 29: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-Ons vs OAKplan-small-Ons.





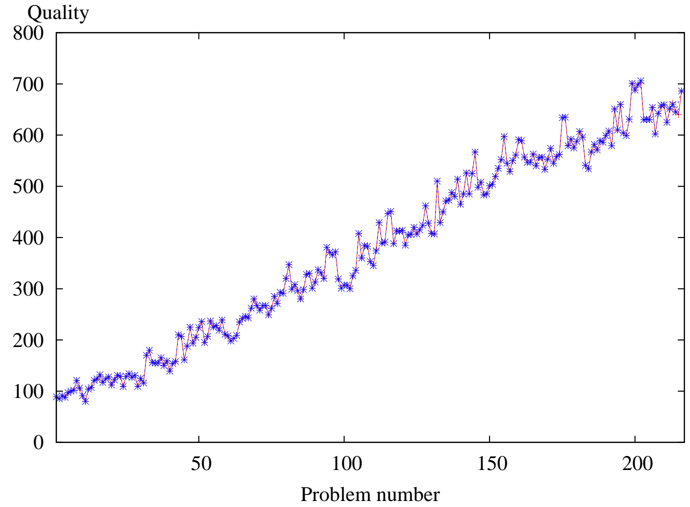


Figure 30: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-Ons vs OAKplan-small-Ons.

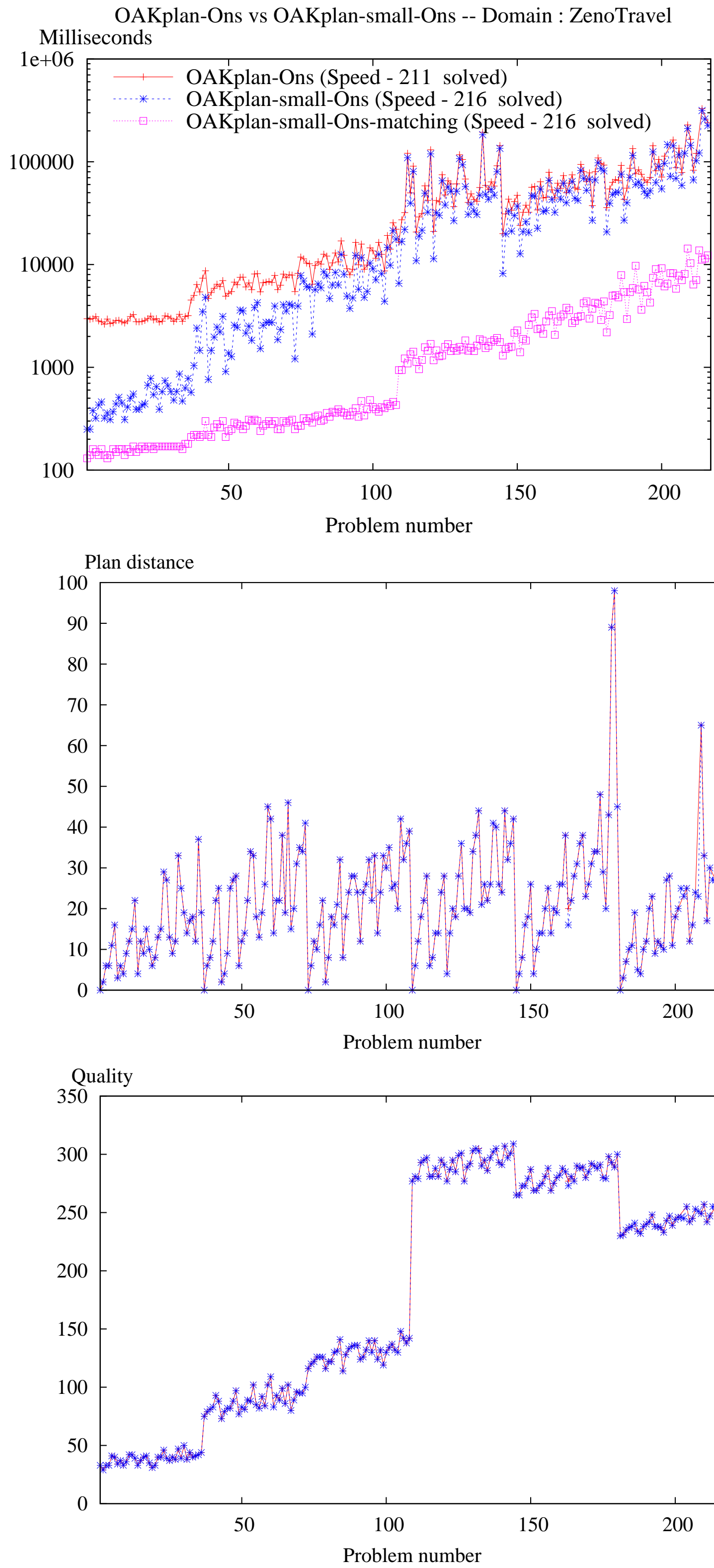


Figure 31: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-Ons vs OAKplan-small-Ons.

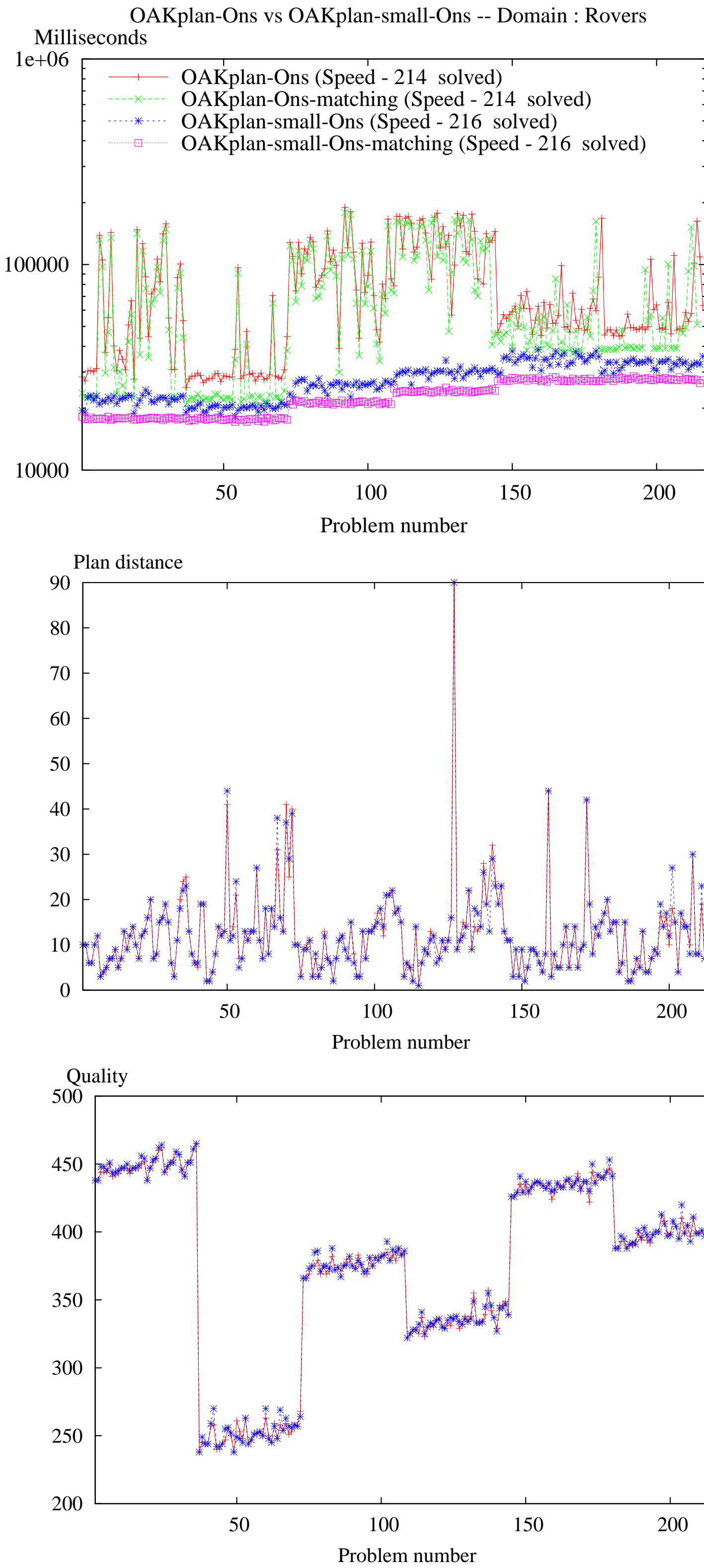


Figure 32: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-Ons vs OAKplan-small-Ons.

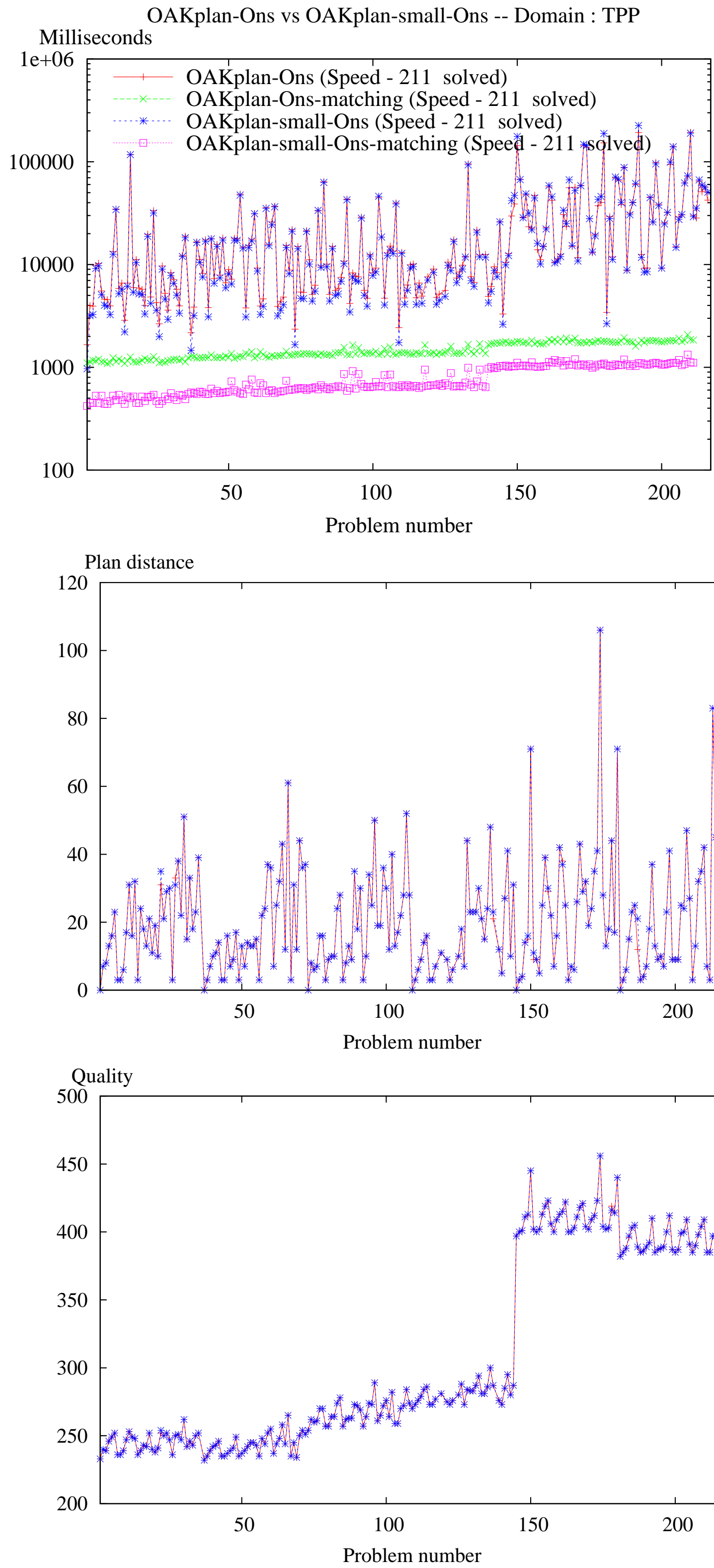


Figure 33: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-Ons vs OAKplan-small-Ons.

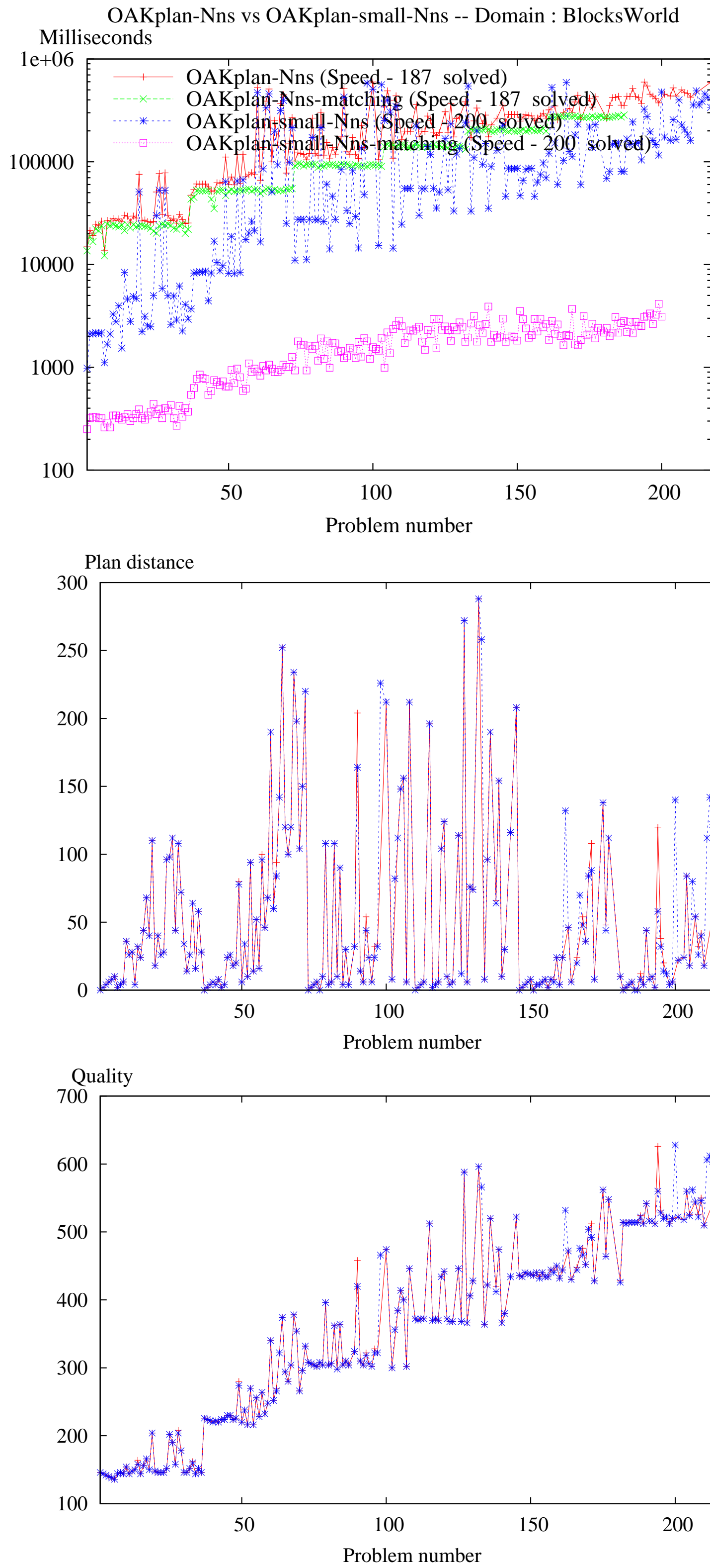


Figure 34: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.

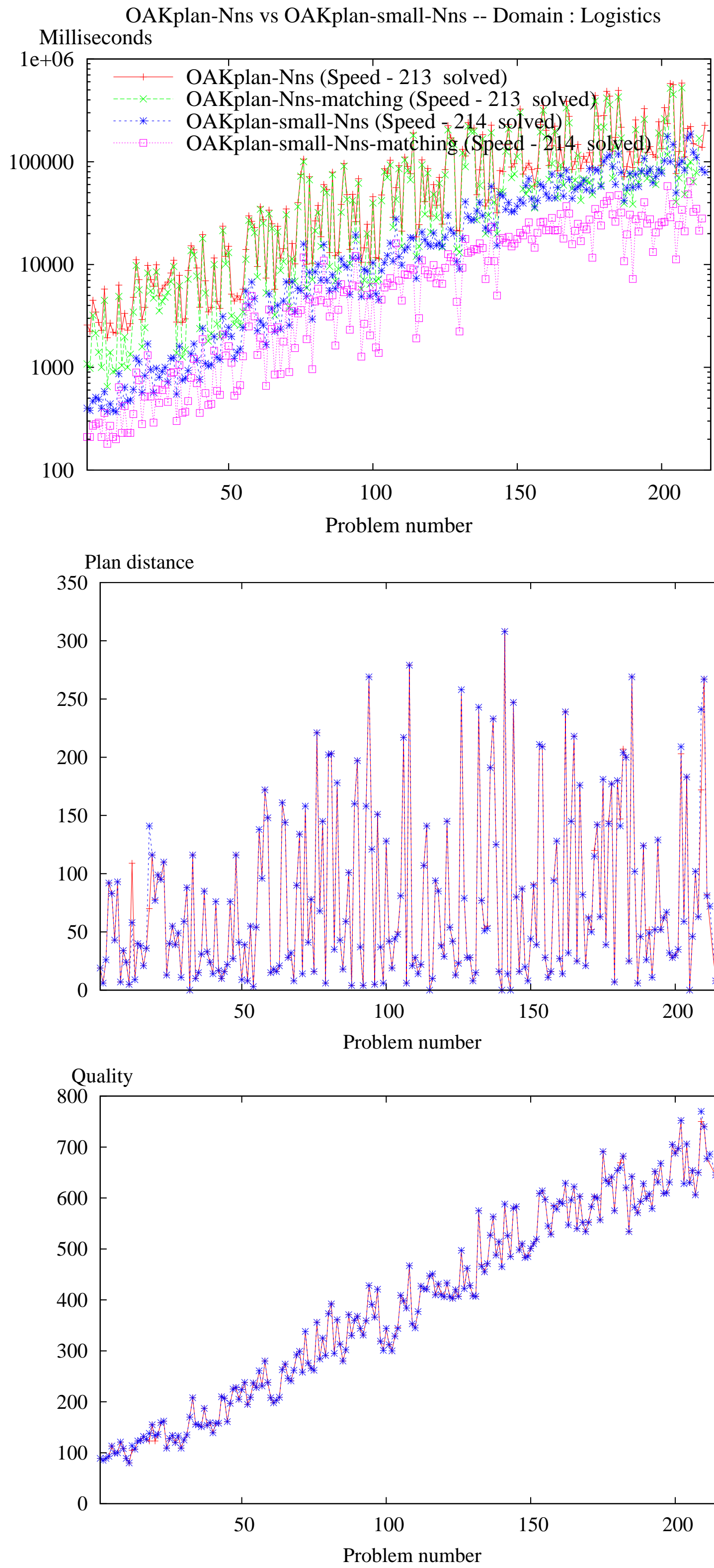
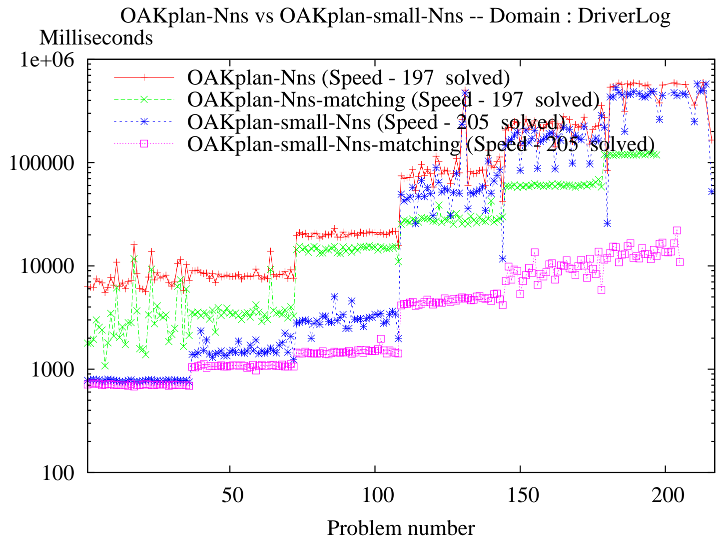
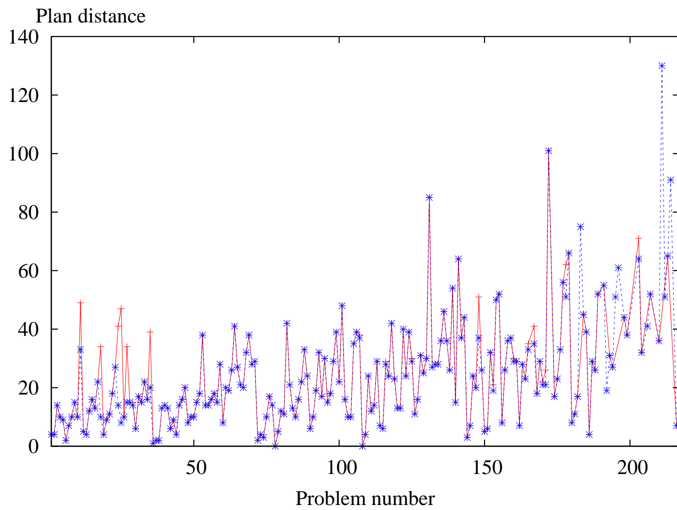


Figure 35: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.





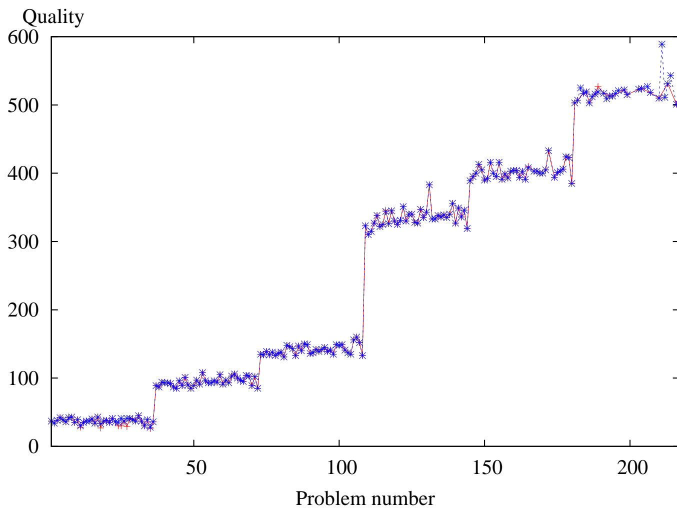


Figure 36: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the DriverLog variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.

Plan Quality Analysis - OAKPLAN variants					
	Nns *vs* Nns-$\mathcal{K}_{base}$	Nns *vs* Nns-$\mathcal{K}_{node}$	Nns *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-adapt-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−16.0029	−4.6283	−18.2599	−7.3752	−7.7294
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 888	↑ 368	↓ 973	↓ 751	↑ 720

Plan Quality Analysis - OAKPLAN variants					
	Nns-$\mathcal{K}_{base}$ *vs* Nns-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−9.3523	−1.2574	−6.0694	−8.4452	−11.7015
p-value	< 0.001	0.2086	< 0.001	< 0.001	< 0.001
num.	↑ 369	888	↑ 649	↓ 368	↓ 368

Table 10: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ in terms of plan quality for our benchmark problems.

Plan Distance Analysis - OAKPLAN variants					
	Nns *vs* Nns-$\mathcal{K}_{base}$	Nns *vs* Nns-$\mathcal{K}_{node}$	Nns *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-adapt-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−25.6188	−16.5937	−26.8364	−23.7026	−19.6781
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 888	↓ 368	↓ 973	↓ 751	↓ 720

Plan Distance Analysis - OAKPLAN variants					
	Nns-$\mathcal{K}_{base}$ *vs* Nns-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{base}$ *vs* Nns-adapt-$\mathcal{K}_{node}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{base}$	Nns-$\mathcal{K}_{node}$ *vs* Nns-adapt-$\mathcal{K}_{node}$
z-value	−13.6294	−3.7804	−19.2003	−13.4415	−0.7513
p-value	< 0.001	< 0.001	< 0.001	< 0.001	0.4525
num.	↓ 369	↓ 888	↓ 649	↑ 368	368

Table 11: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Nns-$\mathcal{K}_{base}$, OAKPLAN-Nns-$\mathcal{K}_{node}$, OAKPLAN-Nns-adapt-$\mathcal{K}_{base}$ and OAKPLAN-Nns-adapt-$\mathcal{K}_{node}$ in terms of plan distance values for our benchmark problems.

CPU-time Analysis - OAKPLAN Case Bases						
	Nns *vs* Ons	Nns *vs* small-Nns	Nns *vs* small-Ons	Ons *vs* small-Nns	Ons *vs* small-Ons	small-Nns *vs* small-Ons
z-value	−25.6136	−30.1762	−11.9419	−13.3277	−11.8104	−3.3715
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↑ 1265	↑ 1260	↑ 1268	↑ 1277	↑ 1272	↑ 1277

Table 12: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons in terms of CPU-time for our benchmark problems.

Plan Quality Analysis - OAKPLAN Case Bases						
	Nns *vs* Ons	Nns *vs* small-Nns	Nns *vs* small-Ons	Ons *vs* small-Nns	Ons *vs* small-Ons	small-Nns *vs* small-Ons
z-value	−17.4870	−3.6466	−16.3177	−17.5653	−3.2450	−16.5138
p-value	< 0.001	< 0.001	< 0.001	< 0.001	0.0012	< 0.001
num.	↑ 1232	↑ 1232	↑ 1037	↓ 1248	↓ 1066	↑ 1056

Table 13: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons in terms of plan quality for our benchmark problems.

Plan Distance Analysis - OAKPLAN Case Bases						
	Nns *vs* Ons	Nns *vs* small-Nns	Nns *vs* small-Ons	Ons *vs* small-Nns	Ons *vs* small-Ons	small-Nns *vs* small-Ons
z-value	−17.9790	−4.3791	−17.6283	−17.2818	−0.1225	−17.9139
p-value	< 0.001	< 0.001	< 0.001	< 0.001	0.9025	< 0.001
num.	↑ 1232	↑ 1232	↑ 1037	↓ 1248	1066	↑ 1056

Table 14: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, OAKPLAN-Ons, OAKPLAN-small-Nns and OAKPLAN-small-Ons in terms of plan distance for our benchmark problems.

CPU-time Analysis					
	OAKPLAN-Nns *vs* DOWNWARD	OAKPLAN-Nns *vs* LPG	OAKPLAN-Nns *vs* METRIC-FF	OAKPLAN-Nns *vs* SGPLAN-IPC5	METRIC-FF *vs* SGPLAN-IPC5
z-value	−25.1799	−13.7264	−27.5647	−25.8335	−17.6837
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 1236	↓ 1240	↓ 1235	↓ 1239	↑ 933

CPU-time Analysis					
	DOWNWARD *vs* LPG	DOWNWARD *vs* METRIC-FF	DOWNWARD *vs* SGPLAN-IPC5	LPG *vs* METRIC-FF	LPG *vs* SGPLAN-IPC5
z-value	−10.1772	−6.8887	−1.9748	−17.9592	−8.4884
p-value	< 0.001	< 0.001	0.0483	< 0.001	< 0.001
num.	↑ 1061	↓ 914	995	↓ 916	↓ 1054

Table 15: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5 in terms of CPU-time for our benchmark problems.

Plan Quality Analysis					
	OAKPLAN-Nns *vs* DOWNWARD	OAKPLAN-Nns *vs* LPG	OAKPLAN-Nns *vs* METRIC-FF	OAKPLAN-Nns *vs* SGPLAN-IPC5	METRIC-FF *vs* SGPLAN-IPC5
z-value	−4.9790	−0.2059	−19.4010	−8.0375	−13.2636
p-value	< 0.001	0.8368	< 0.001	< 0.001	< 0.001
num.	↑ 675	832	↑ 672	↑ 922	↓ 671

Plan Quality Analysis					
	DOWNWARD *vs* LPG	DOWNWARD *vs* METRIC-FF	DOWNWARD *vs* SGPLAN-IPC5	LPG *vs* METRIC-FF	LPG *vs* SGPLAN-IPC5
z-value	−3.4888	−12.4853	−2.2506	−16.5751	−5.8731
p-value	< 0.001	< 0.001	0.0244	< 0.001	< 0.001
num.	↓ 458	↑ 440	613	↑ 599	↑ 715

Table 16: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5 in terms of plan quality for our benchmark problems.

Plan Distance Analysis					
	OAKPLAN-Nns *vs* DOWNWARD	OAKPLAN-Nns *vs* LPG	OAKPLAN-Nns *vs* METRIC-FF	OAKPLAN-Nns *vs* SGPLAN-IPC5	METRIC-FF *vs* SGPLAN-IPC5
z-value	−21.1373	−24.3809	−20.1907	−24.6913	−8.3423
p-value	< 0.001	< 0.001	< 0.001	< 0.001	< 0.001
num.	↓ 675	↓ 832	↓ 672	↓ 922	↓ 671

Plan Distance Analysis					
	DOWNWARD *vs* LPG	DOWNWARD *vs* METRIC-FF	DOWNWARD *vs* SGPLAN-IPC5	LPG *vs* METRIC-FF	LPG *vs* SGPLAN-IPC5
z-value	−6.4373	−0.3193	−3.8209	−14.9301	−10.1334
p-value	< 0.001	0.7495	< 0.001	< 0.001	< 0.001
num.	↓ 458	440	↓ 613	↑ 599	↑ 715

Table 17: Results of the Wilcoxon signed rank test about the performance of OAKPLAN-Nns, DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5 in terms of plan distance values for our benchmark problems.

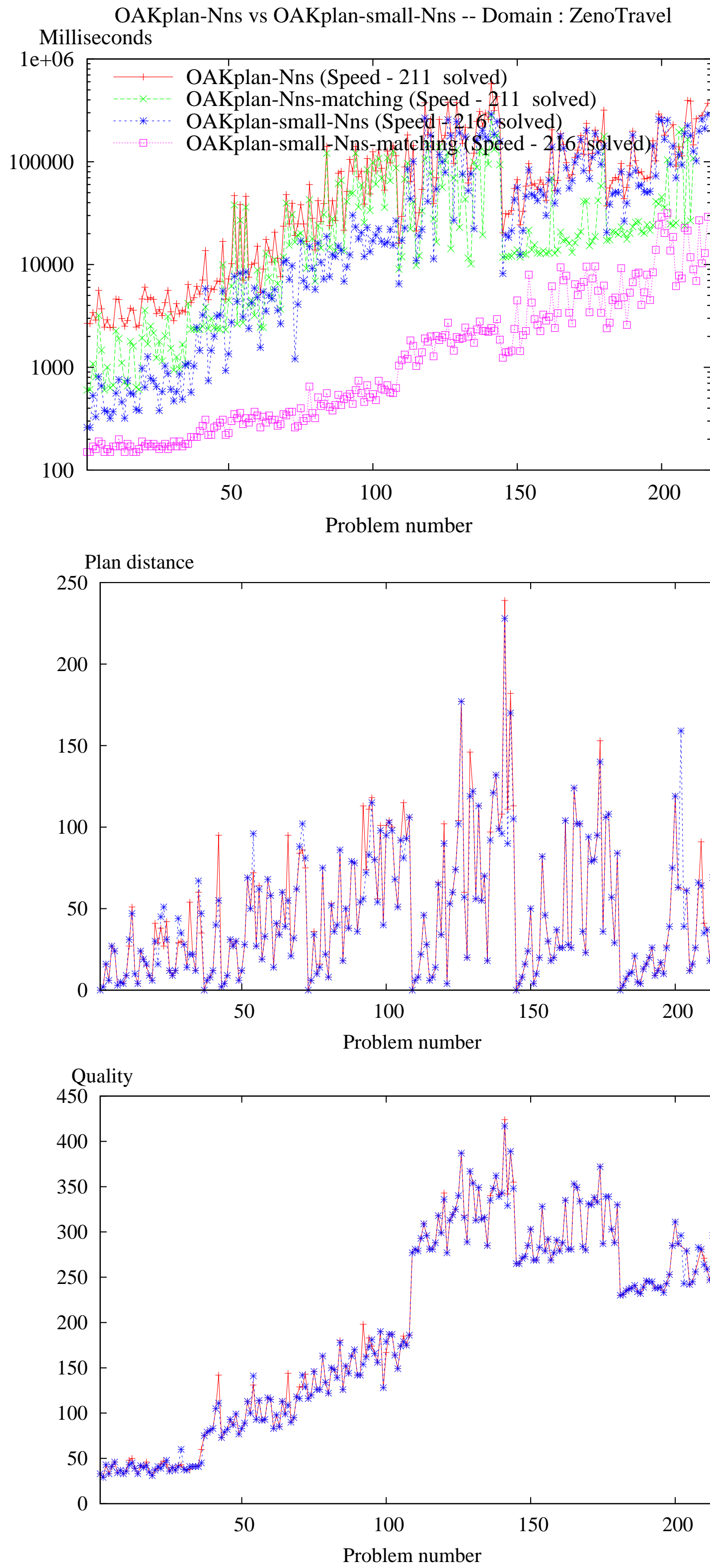
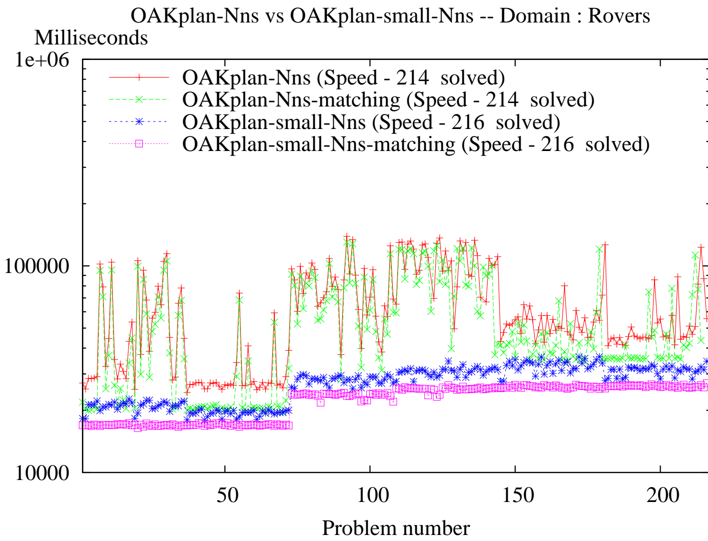
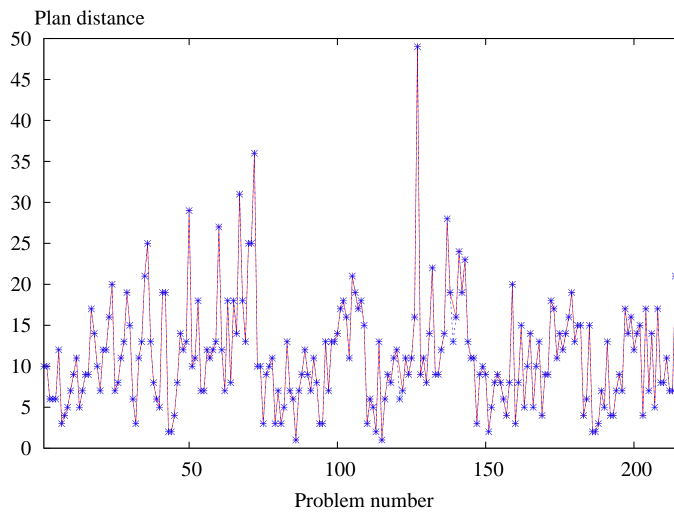


Figure 37: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.





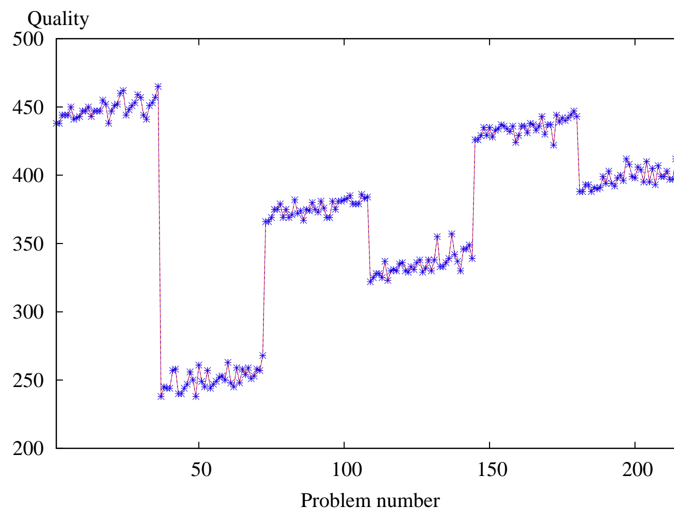


Figure 38: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.

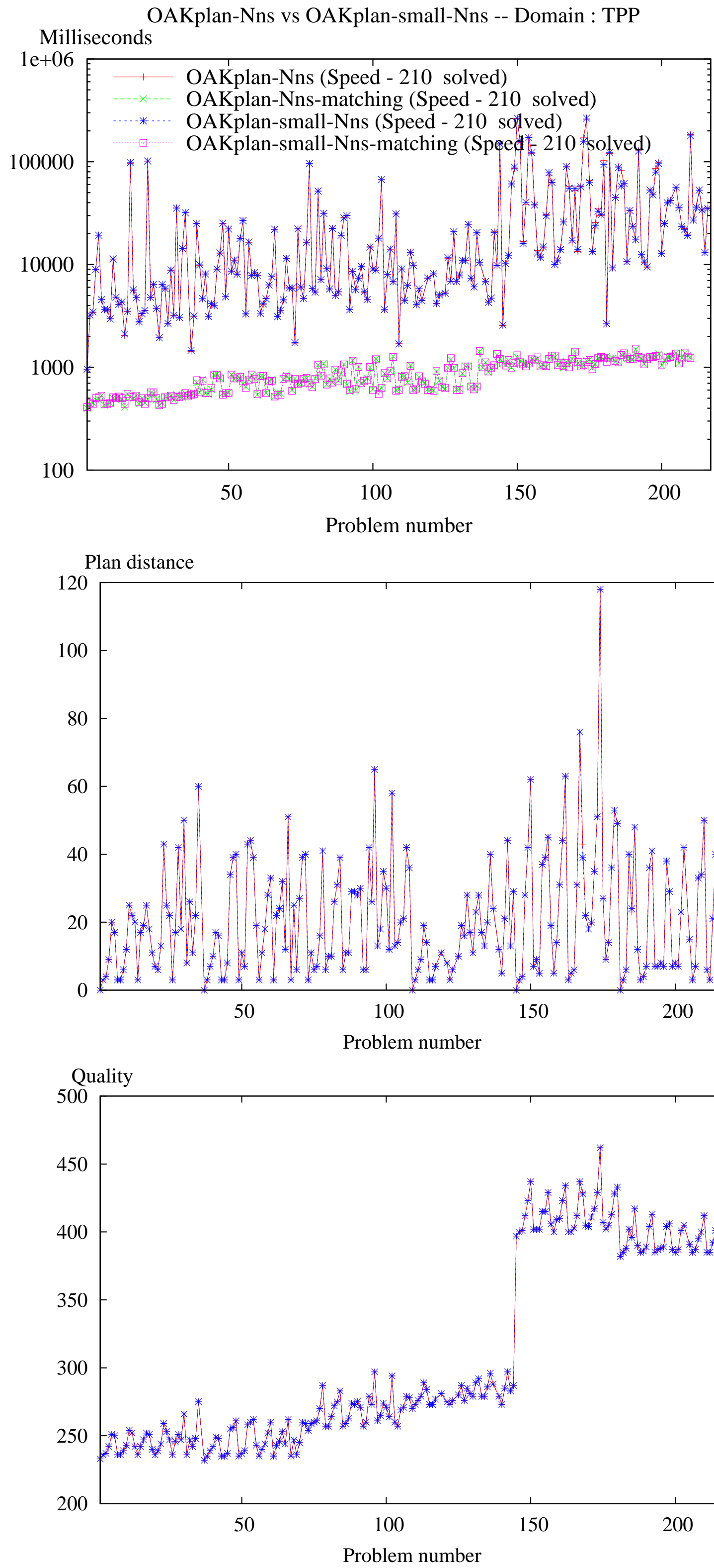


Figure 39: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-Nns vs OAKplan-small-Nns.

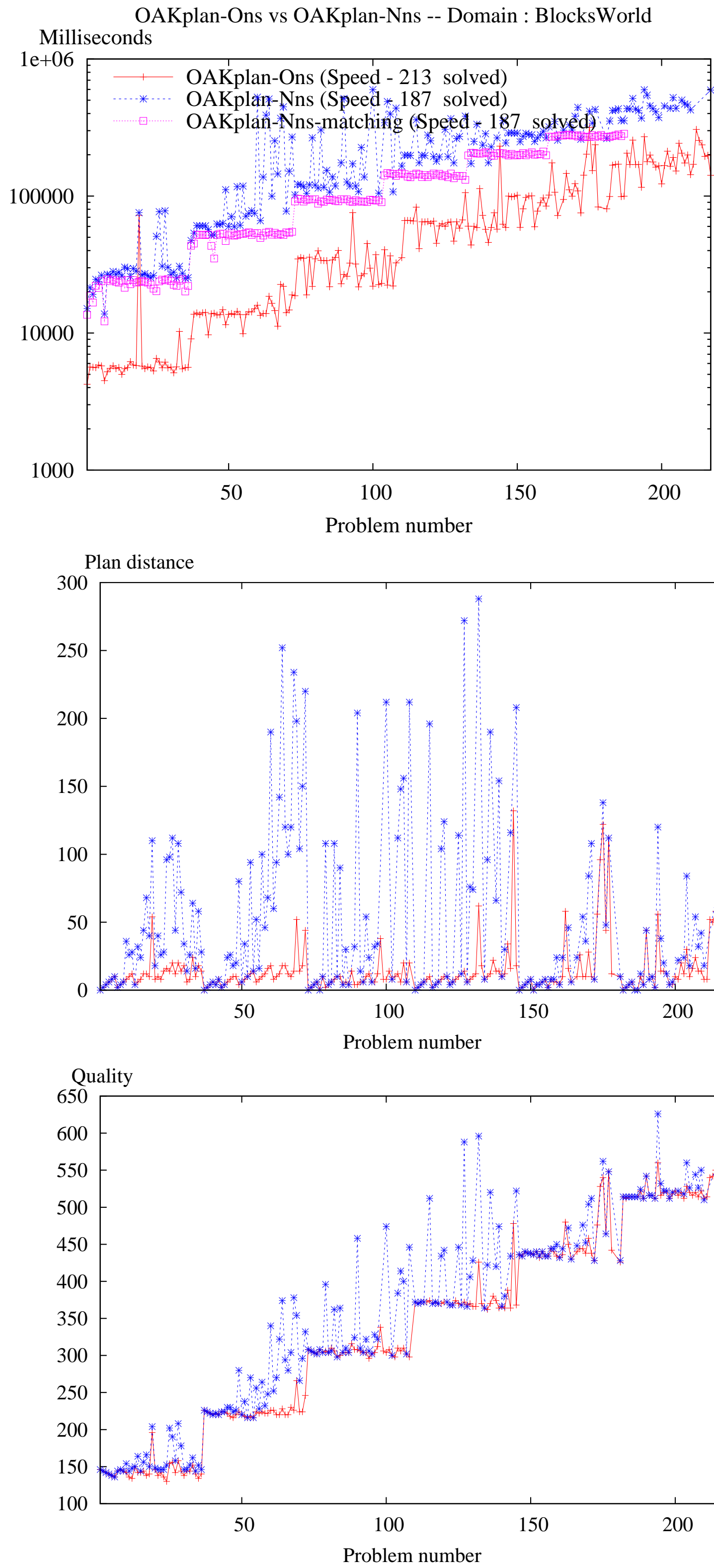


Figure 40: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-Ons vs OAKplan-Nns.

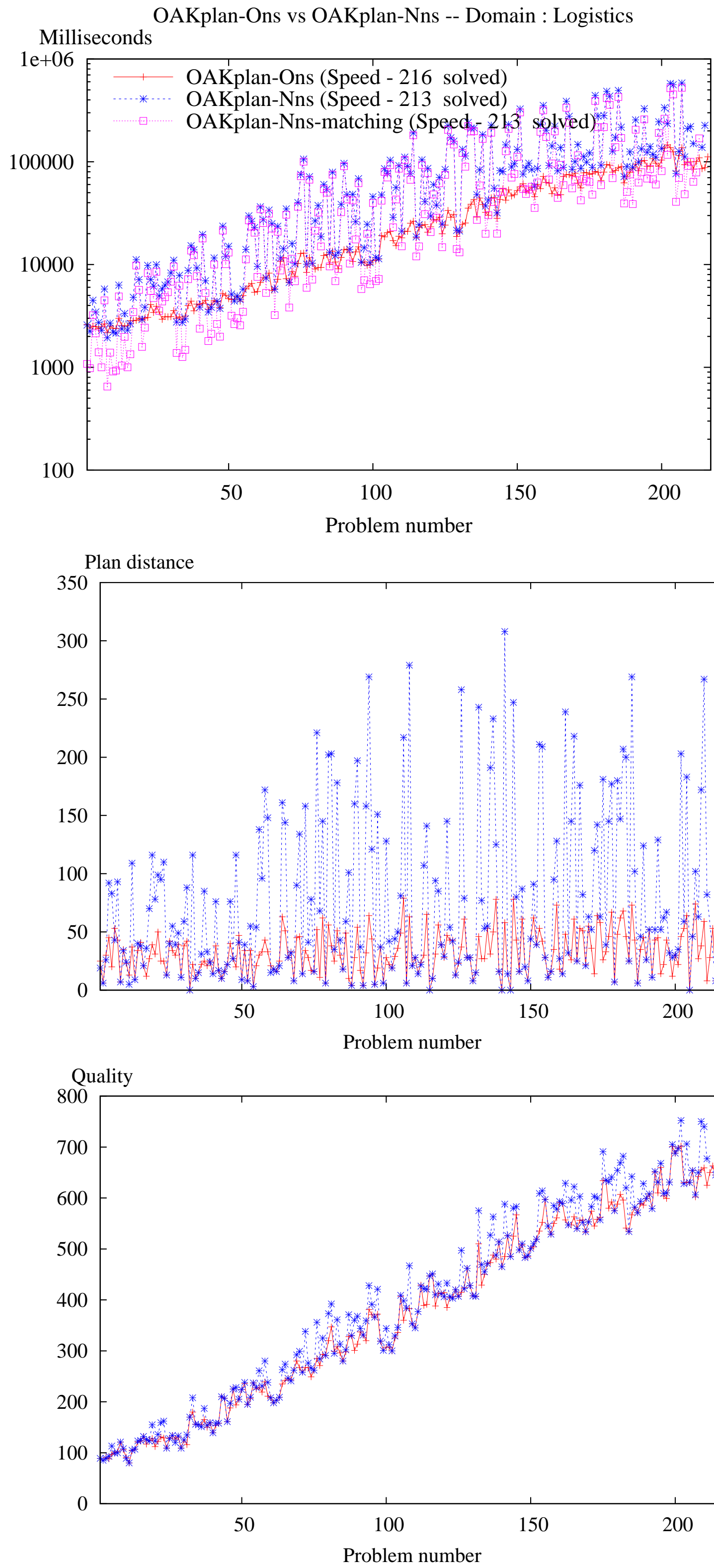


Figure 41: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-Ons vs OAKplan-Nns.

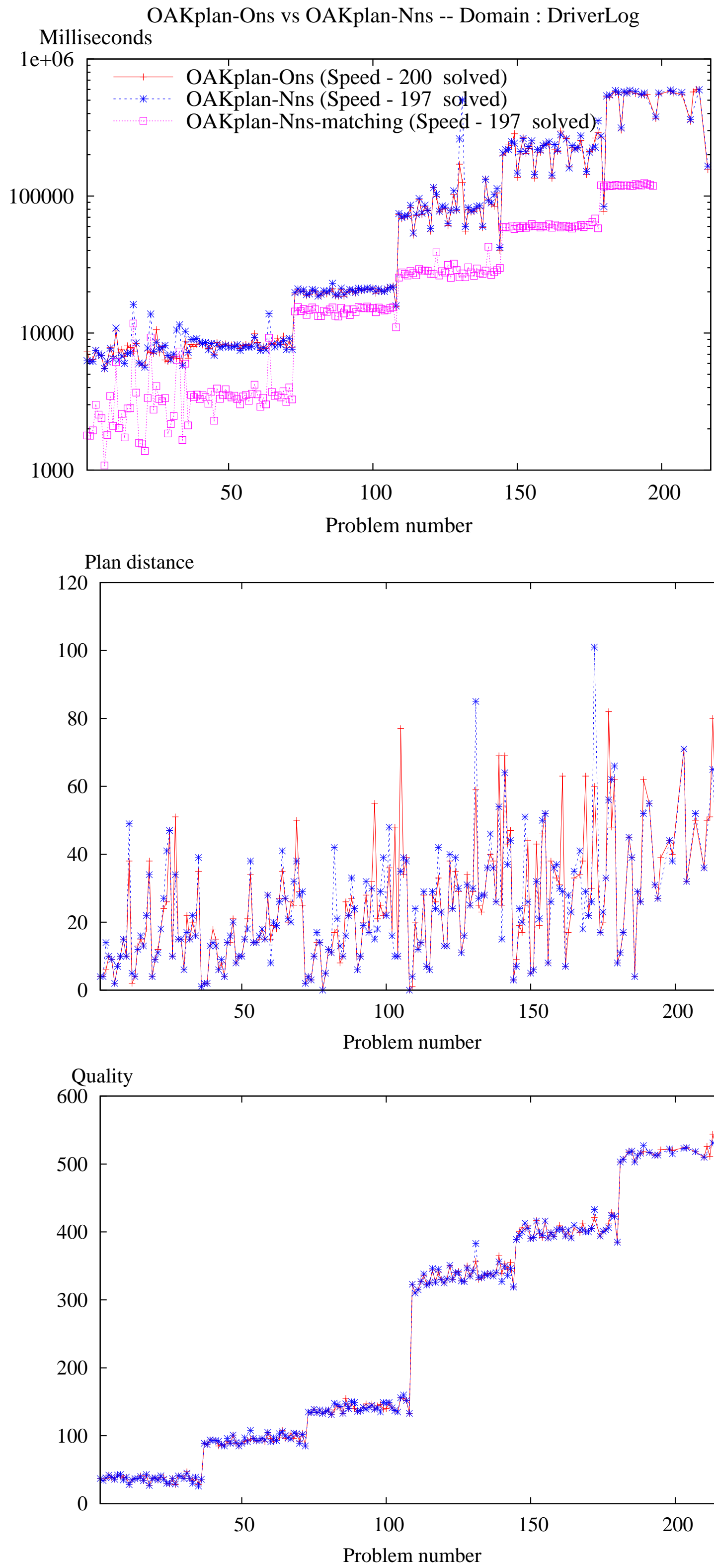


Figure 42: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the DriverLog variants. Here we compare OAKplan-Ons vs OAKplan-Nns.

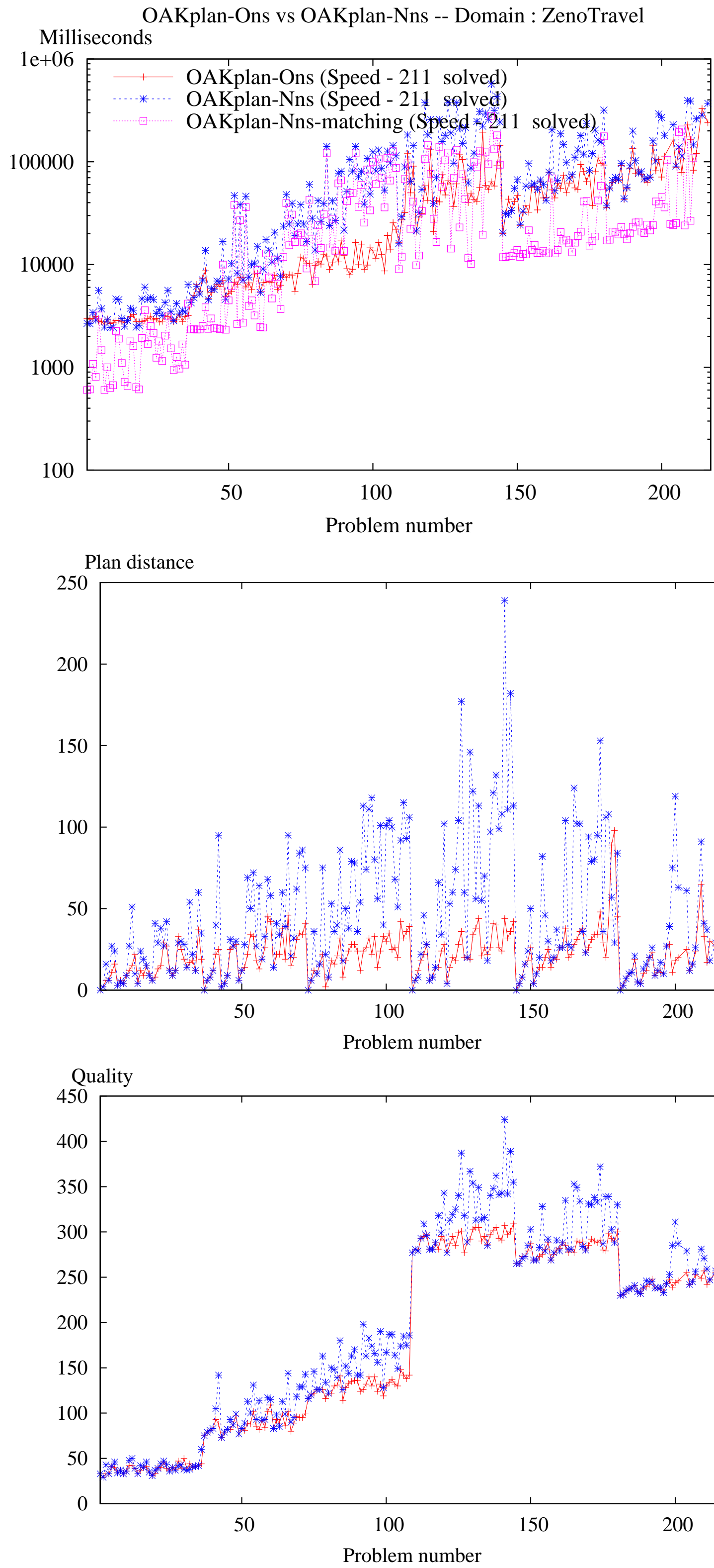


Figure 43: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-Ons vs OAKplan-Nns.

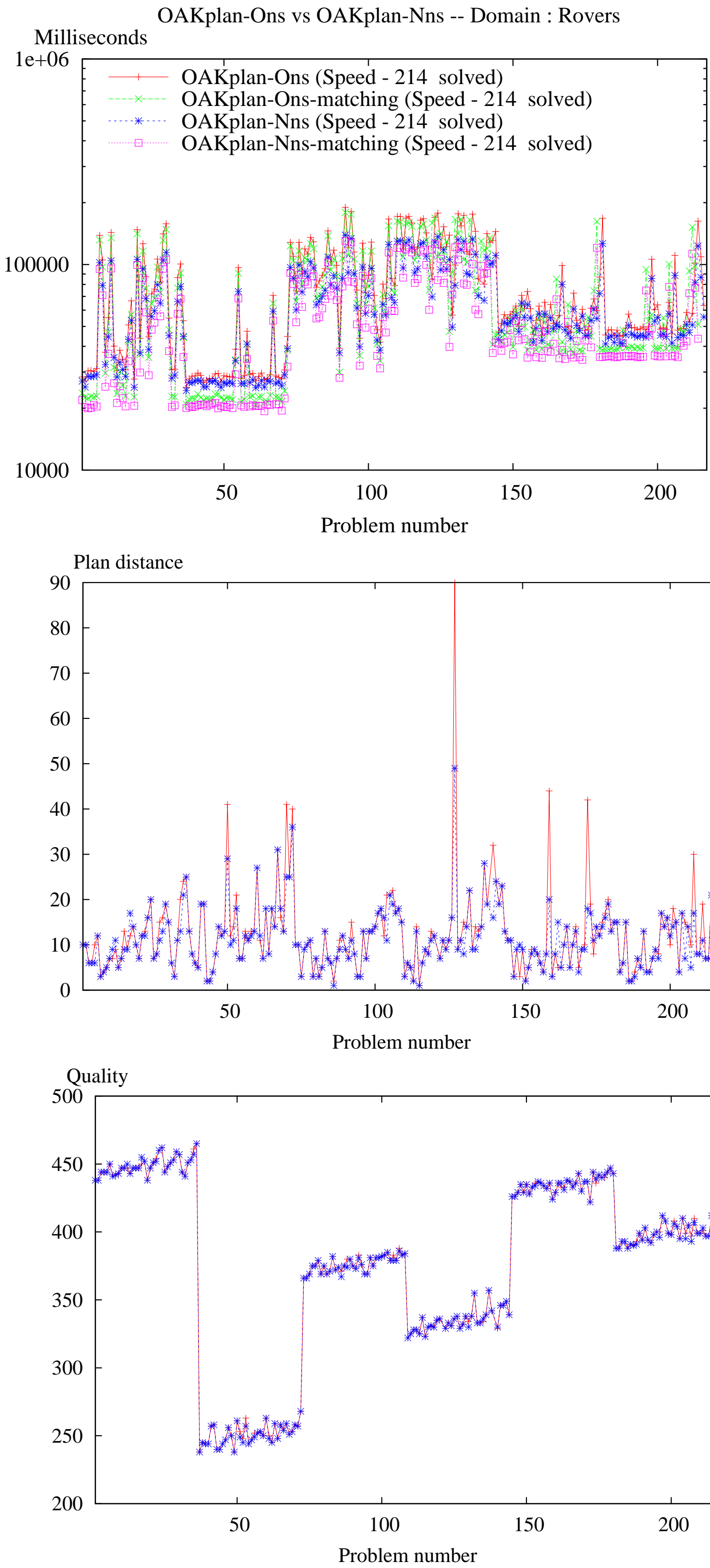


Figure 44: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-Ons vs OAKplan-Nns.

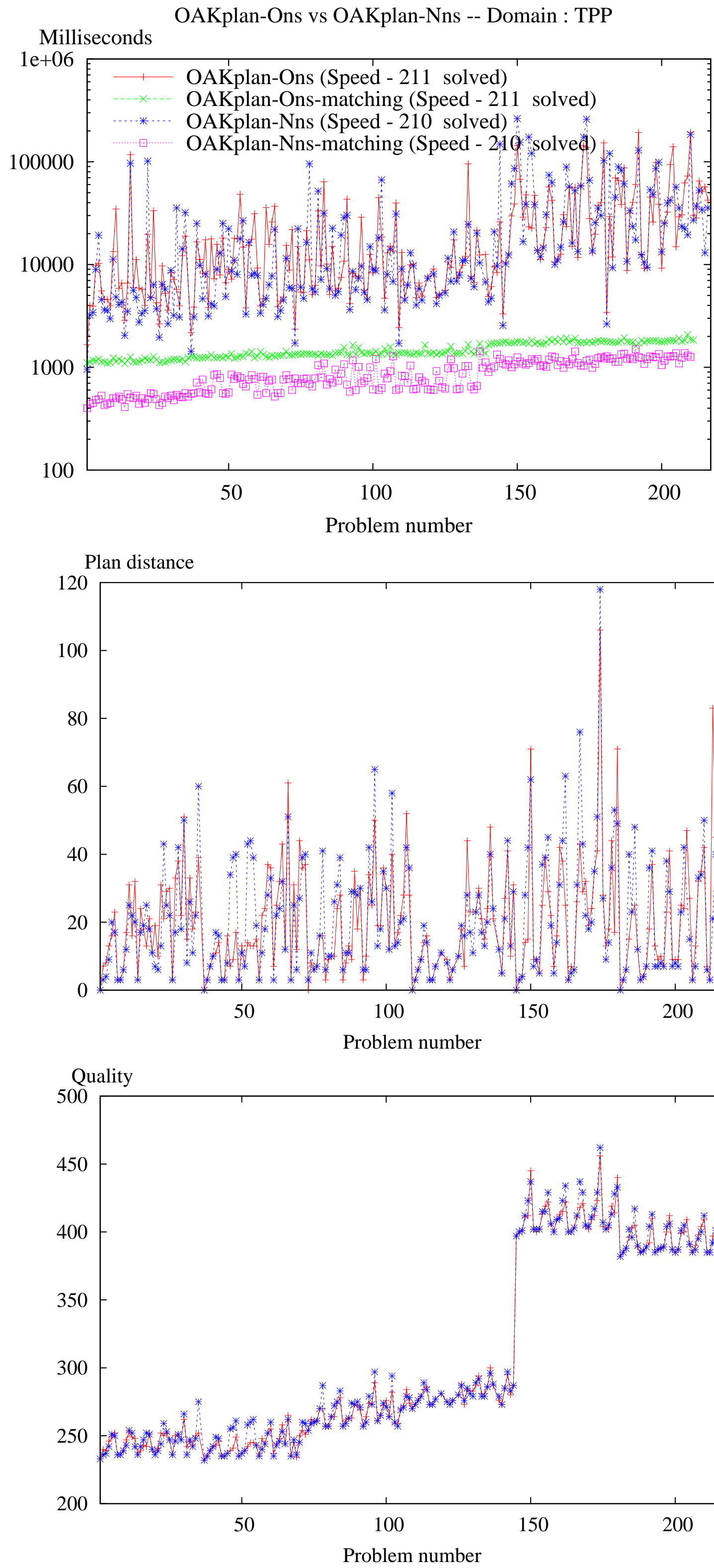
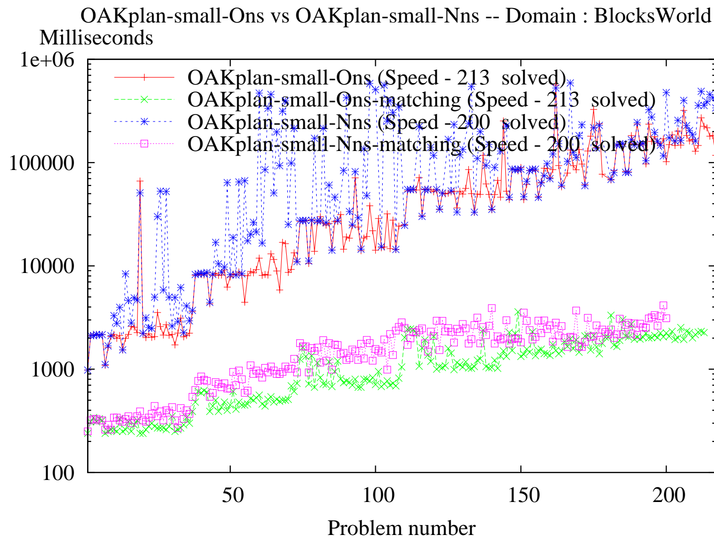
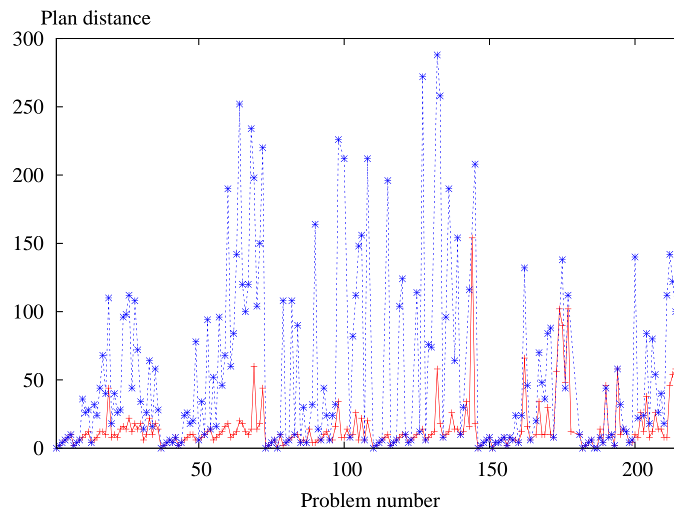


Figure 45: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-Ons vs OAKplan-Nns.





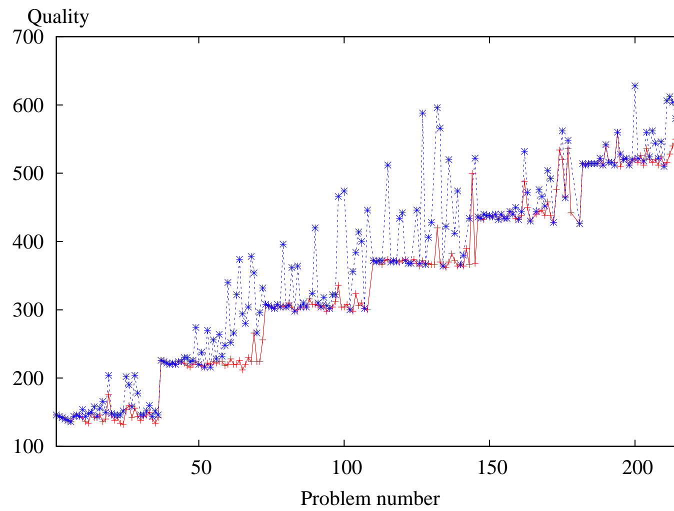
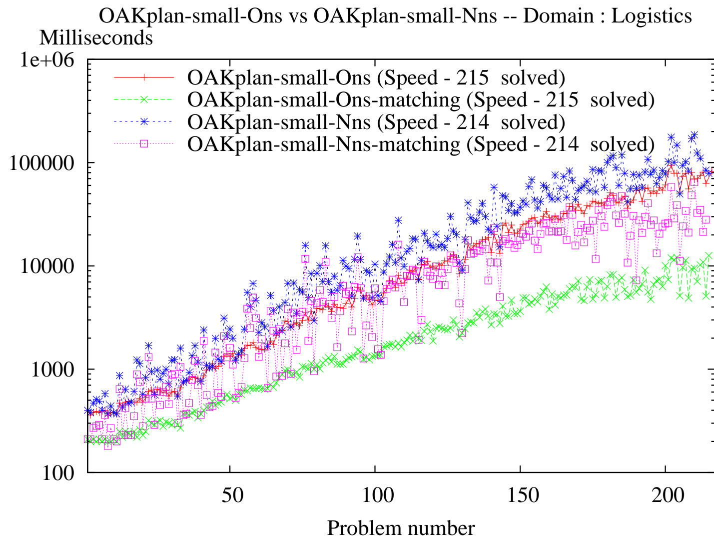
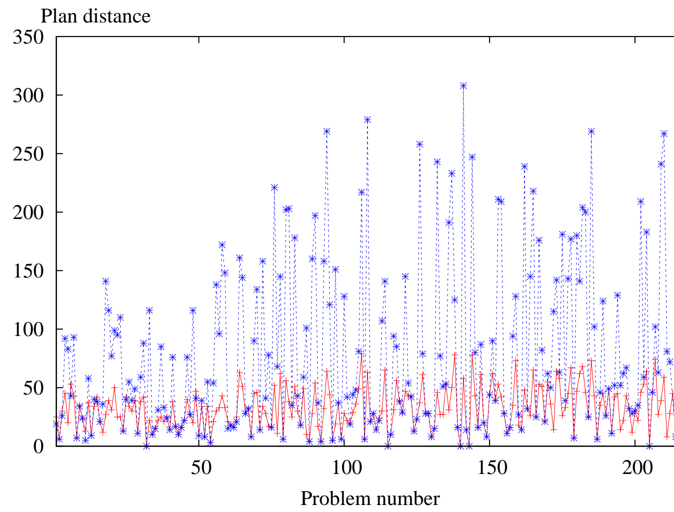


Figure 46: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.





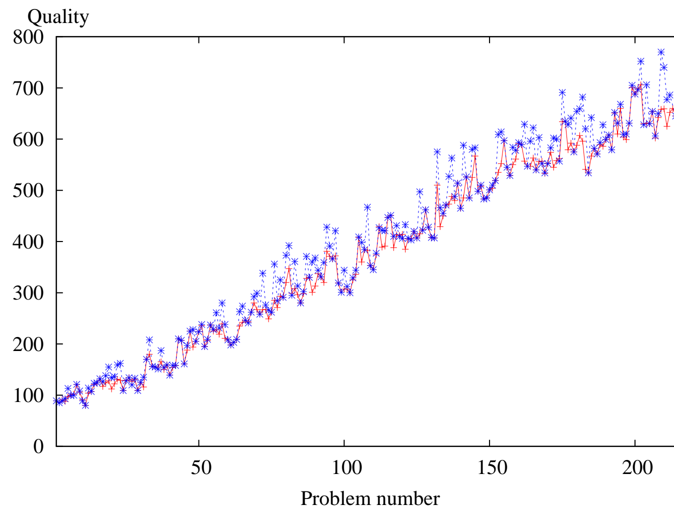
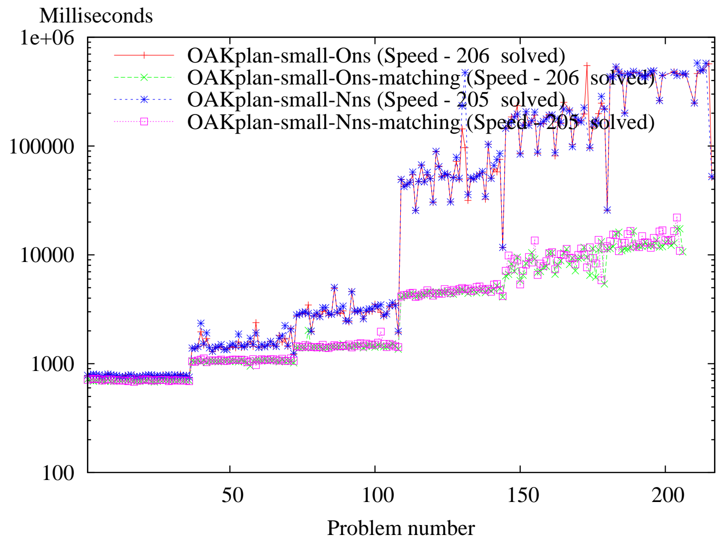
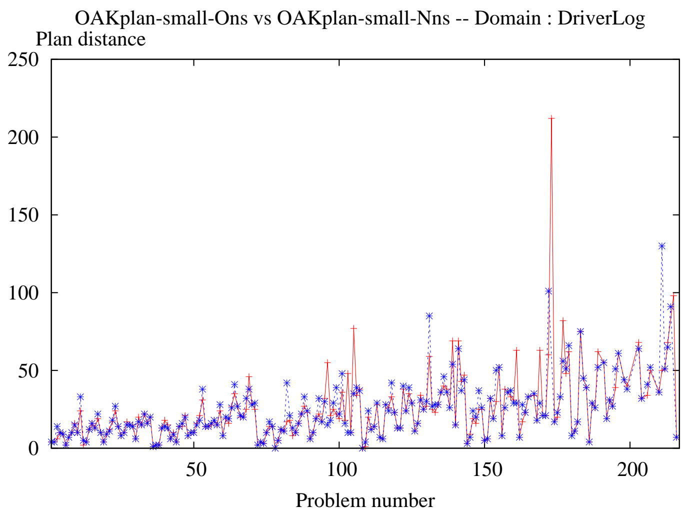


Figure 47: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.





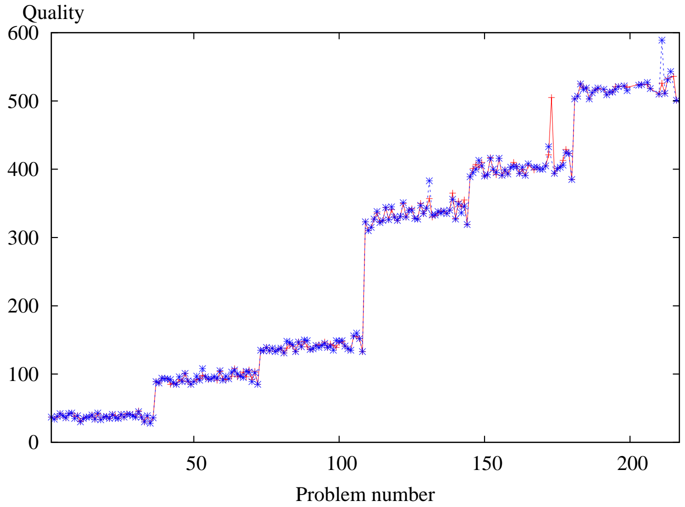


Figure 48: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the DriverLog variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.

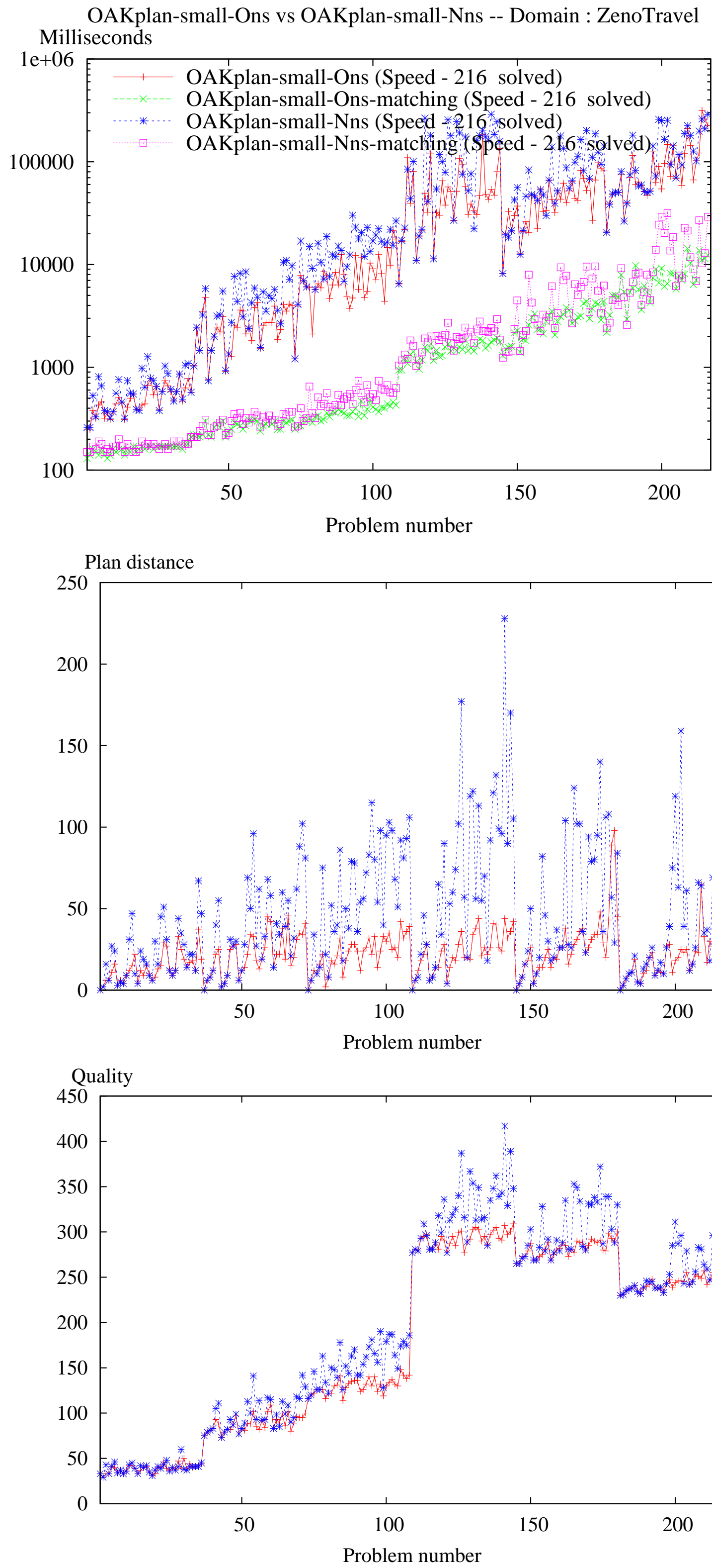


Figure 49: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.

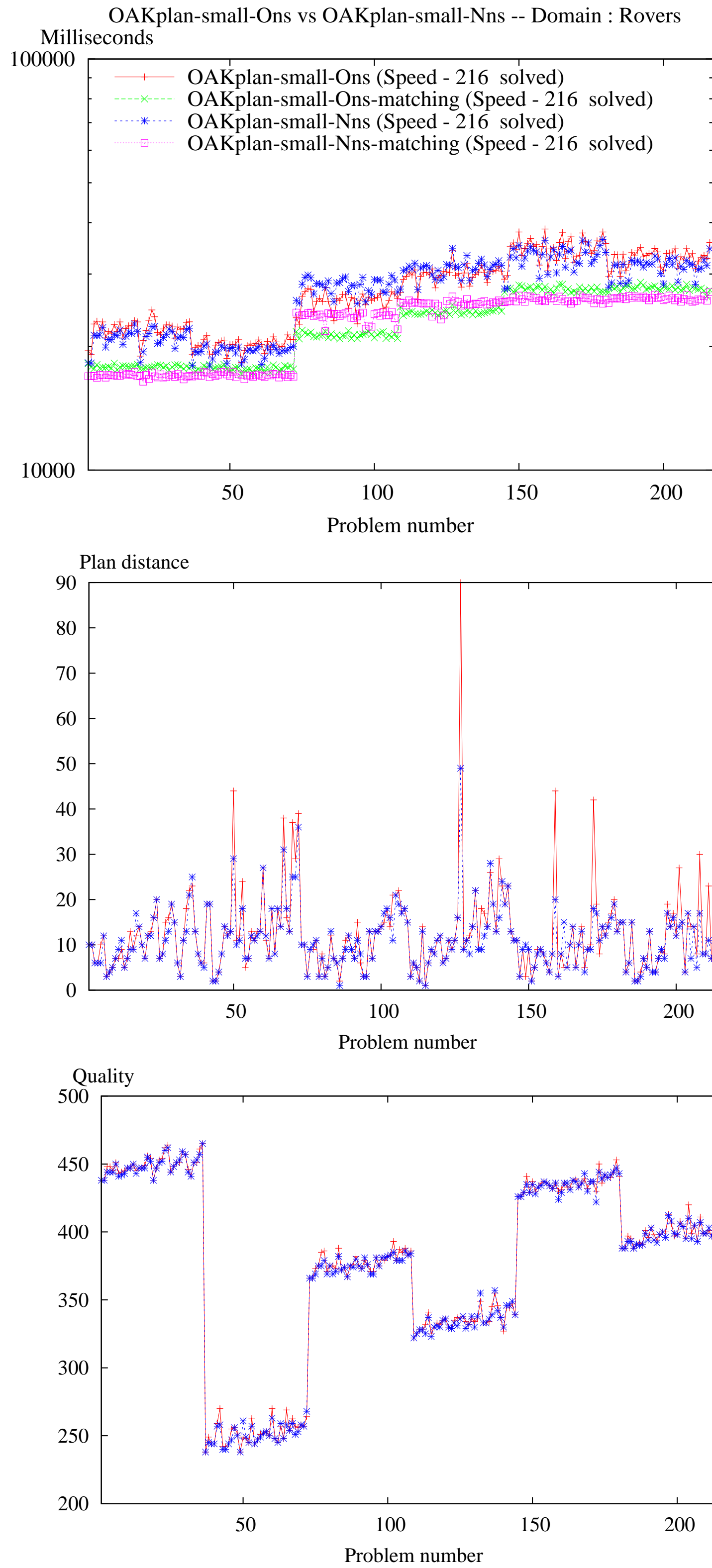


Figure 50: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.

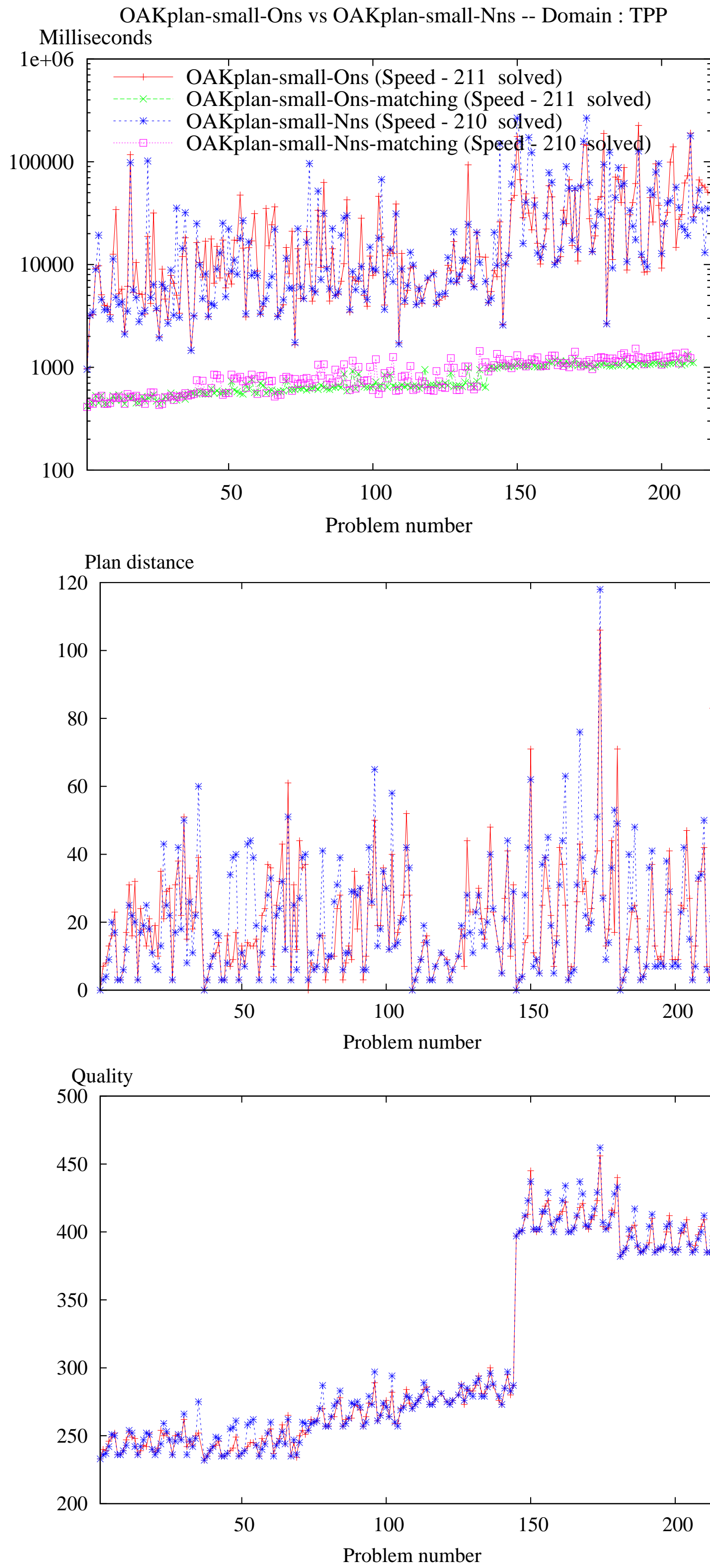


Figure 51: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-small-Ons vs OAKplan-small-Nns.

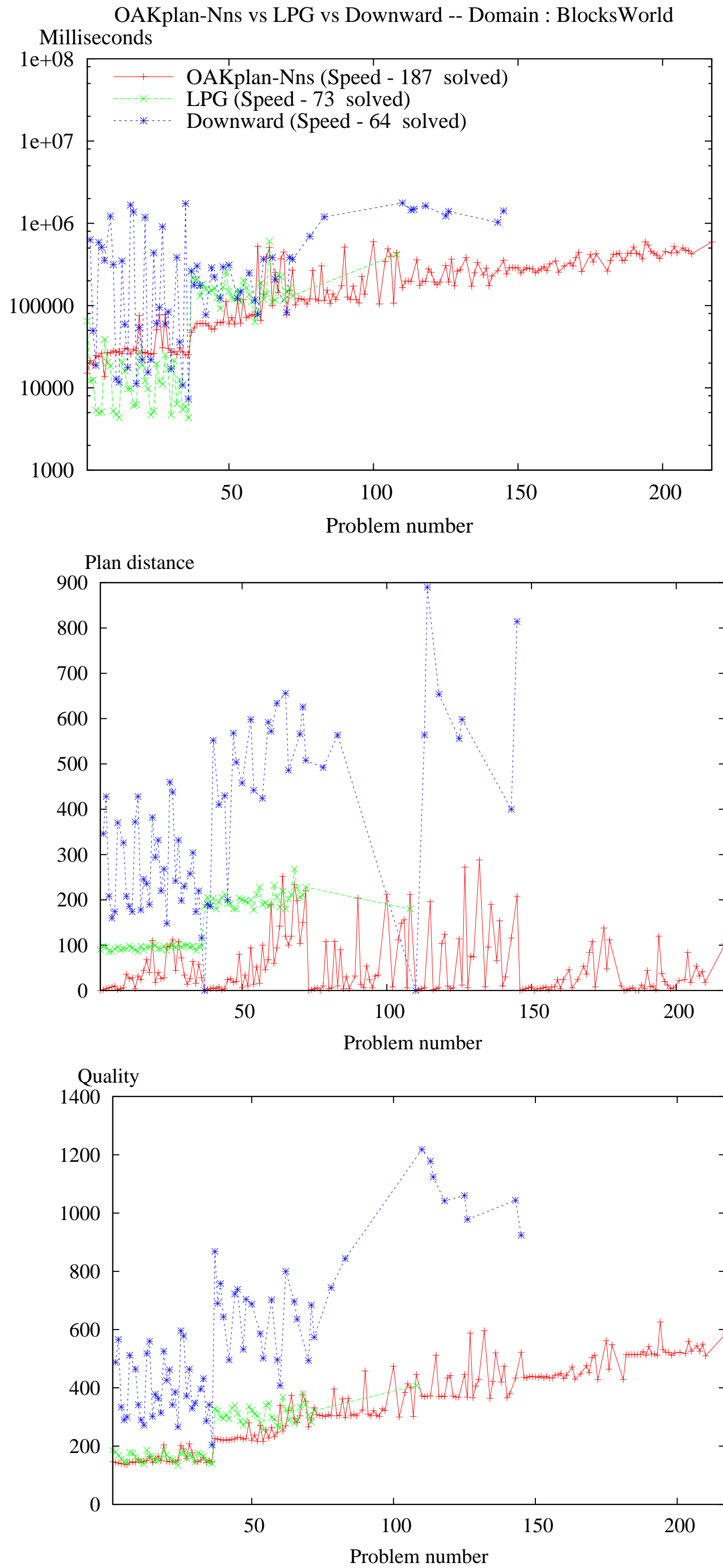
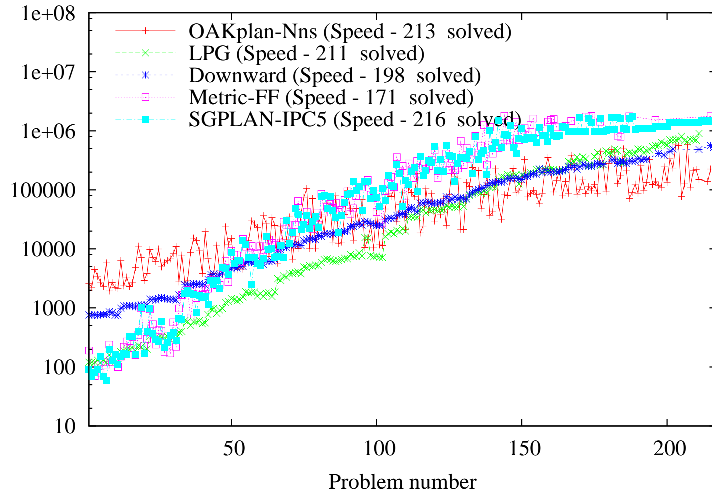
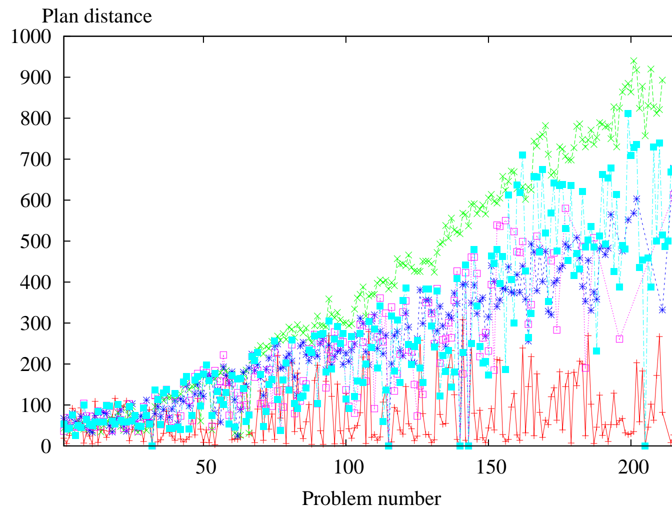


Figure 52: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-Nns vs LPG vs Downward.





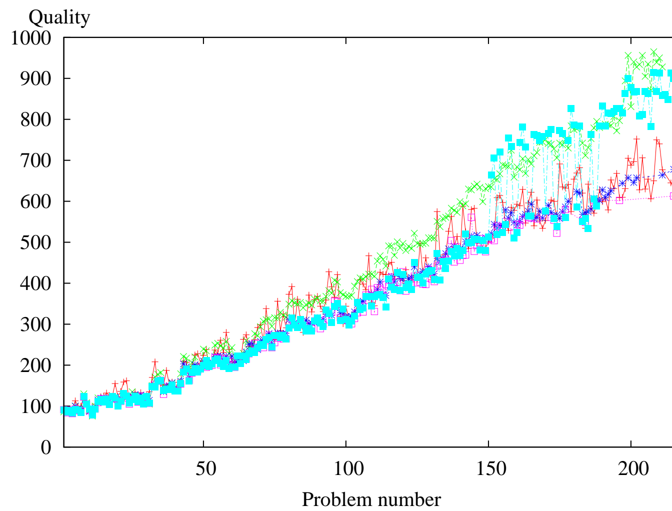
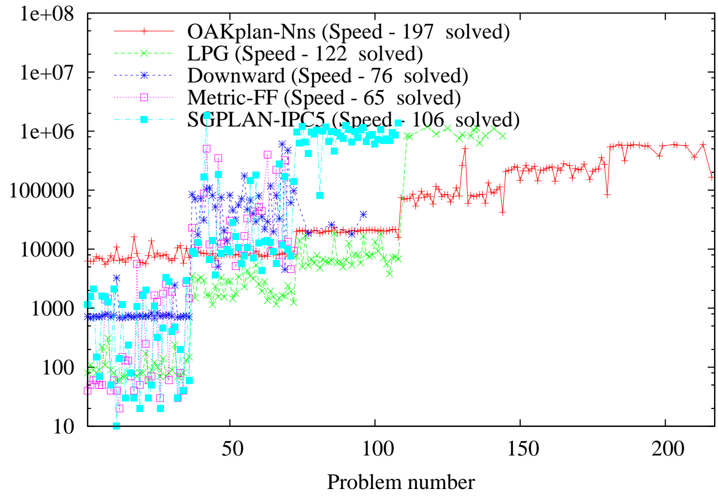
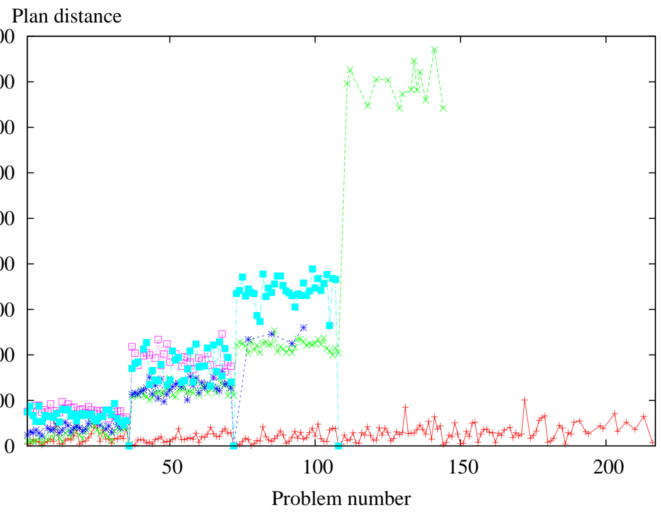


Figure 53: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-Nns vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.





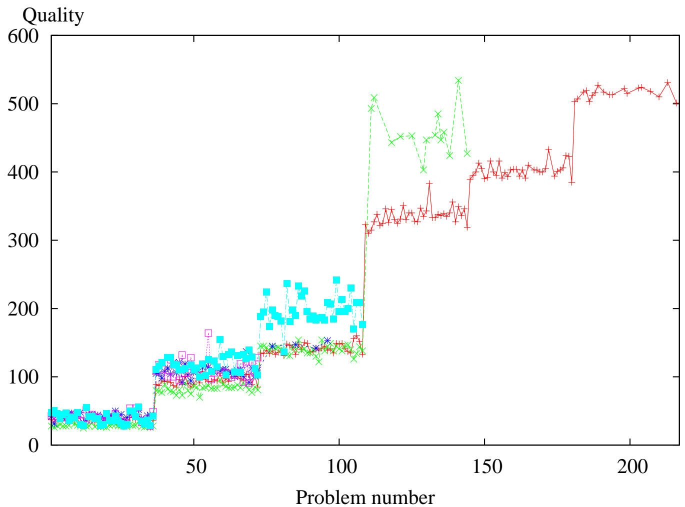
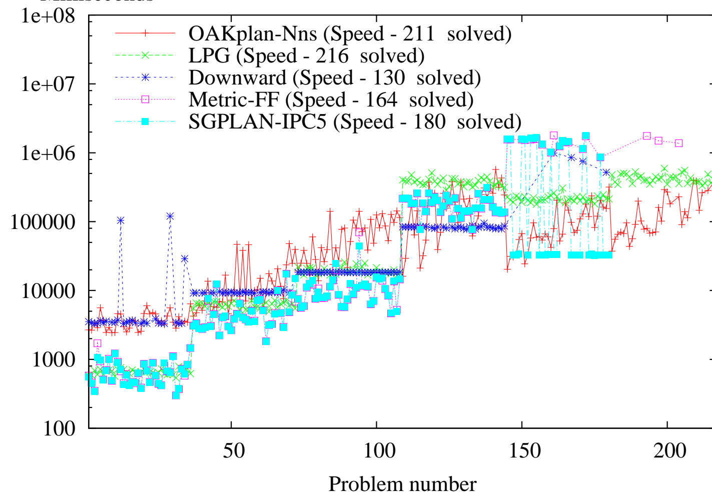
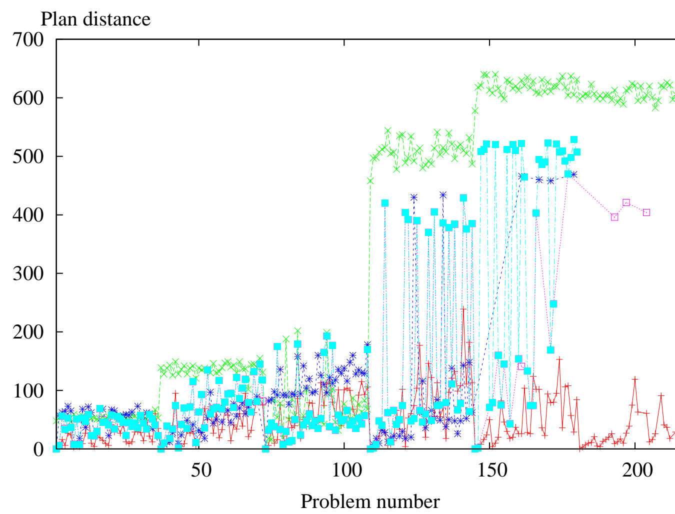


Figure 54: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the DriverLog variants. Here we compare OAKplan-Nns vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.





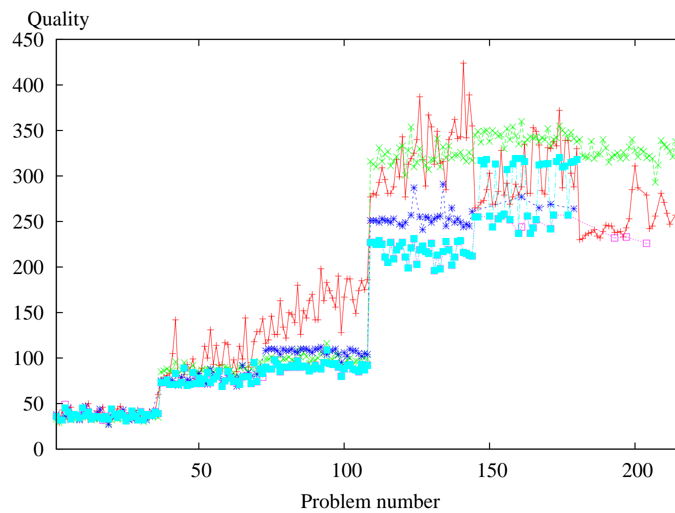


Figure 55: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-Nns vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.

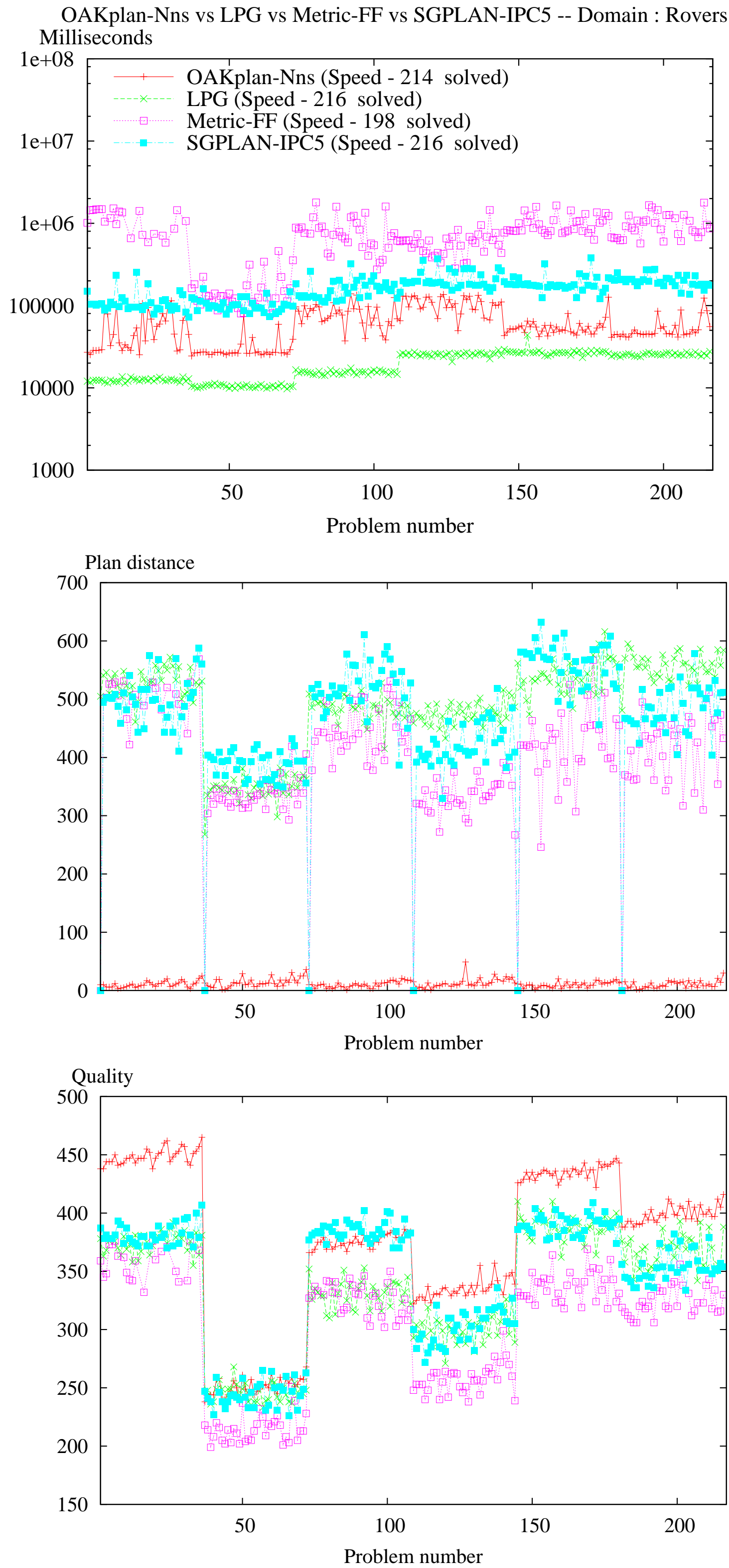
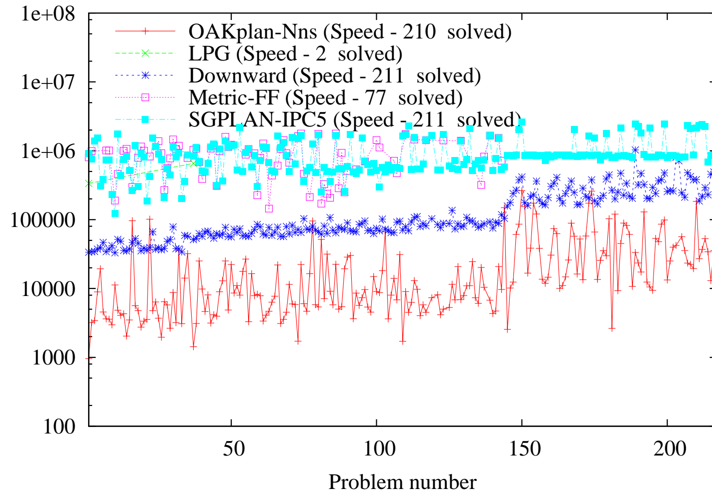
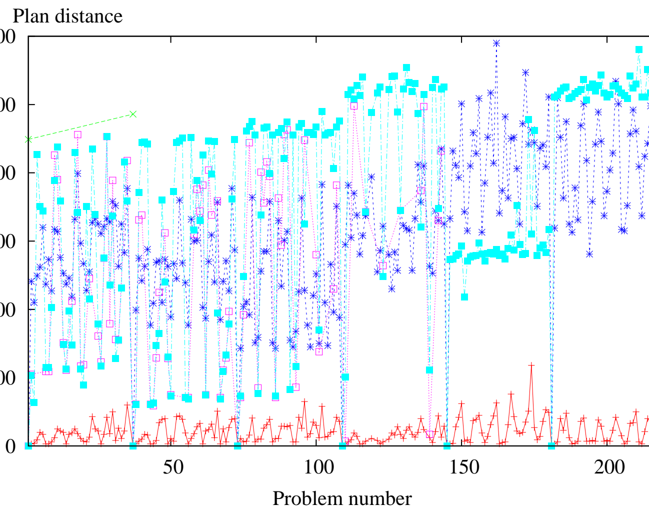


Figure 56: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-Nns vs LPG vs Metric-FF vs SGPLAN-IPC5.





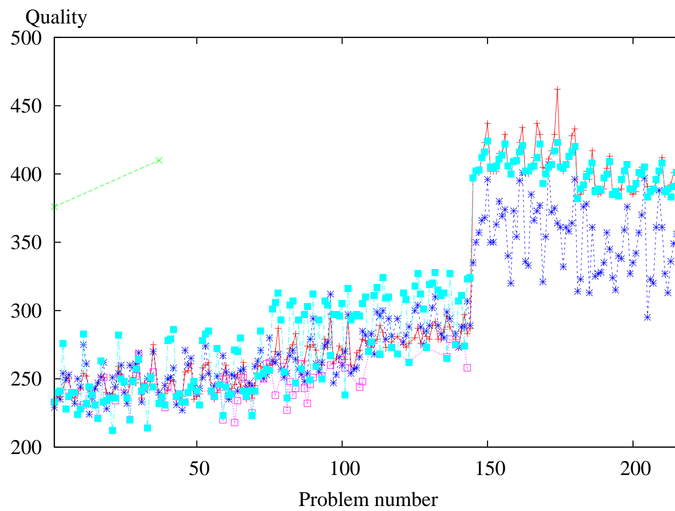


Figure 57: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-Nns vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.

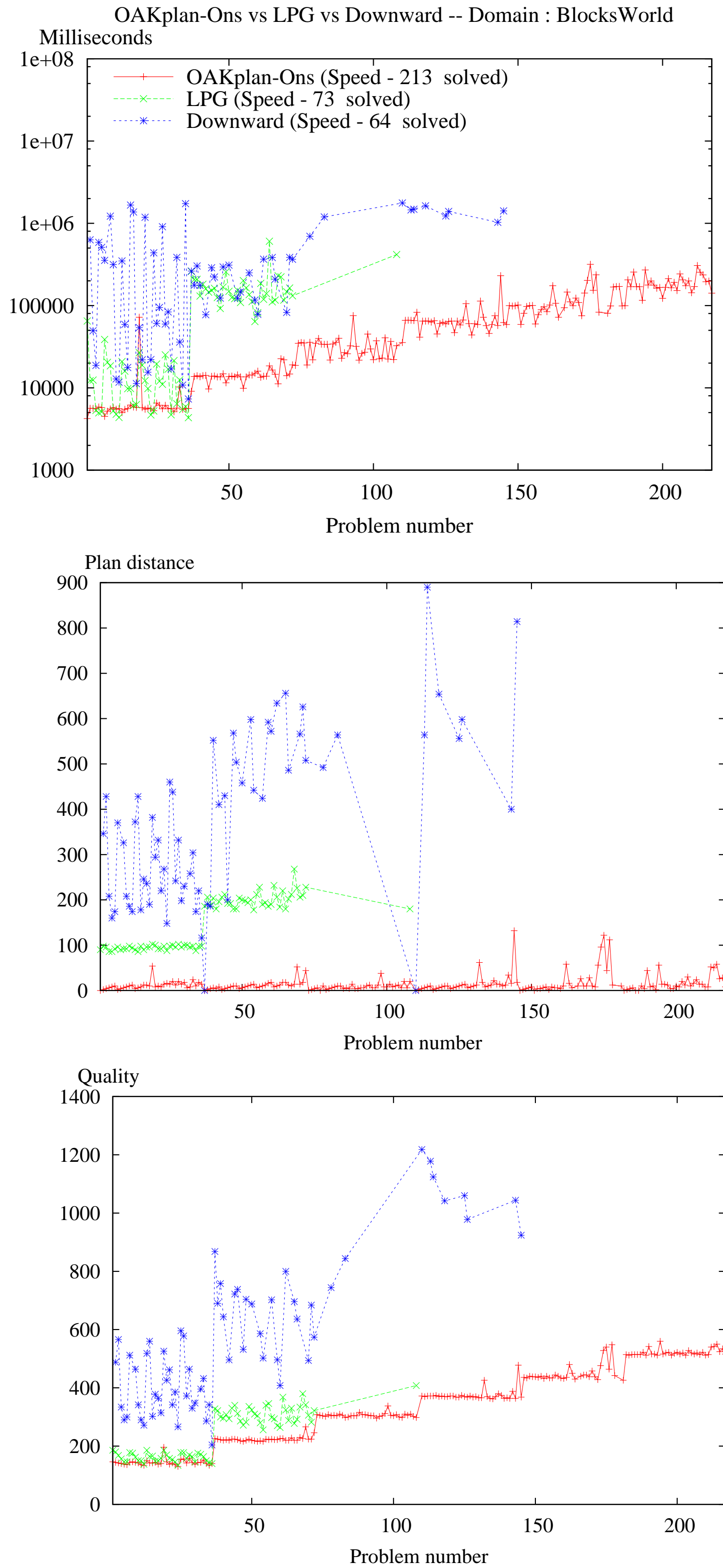
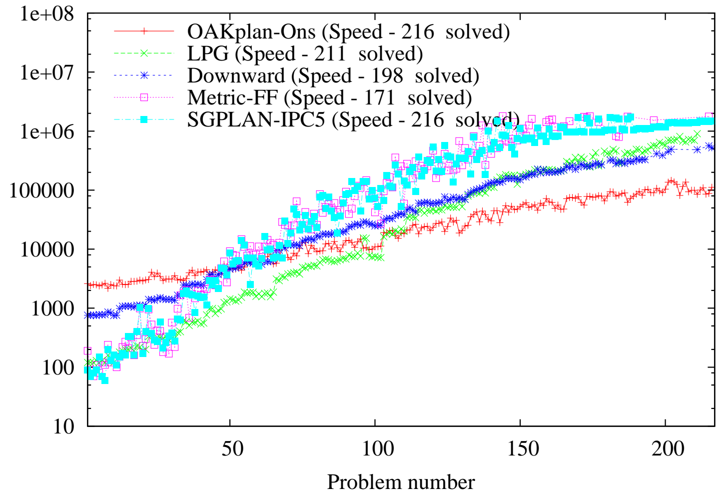
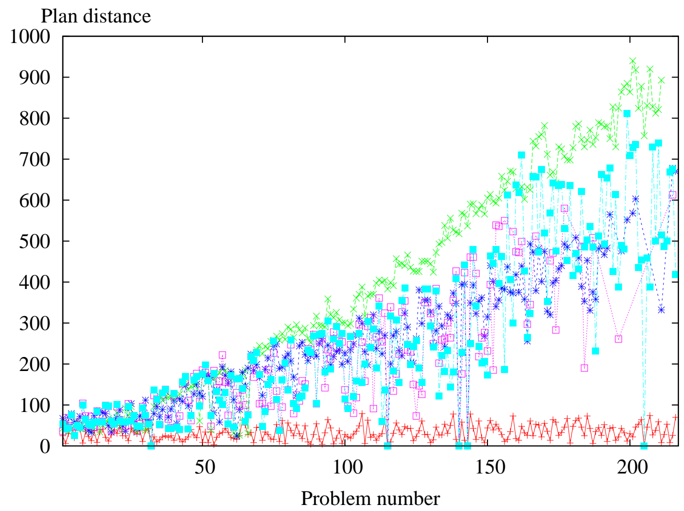


Figure 58: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the BlocksWorld variants. Here we compare OAKplan-Ons vs LPG vs Downward.





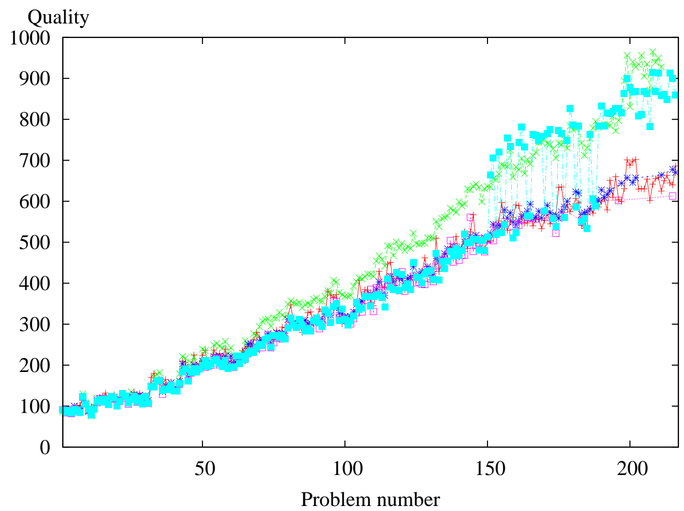
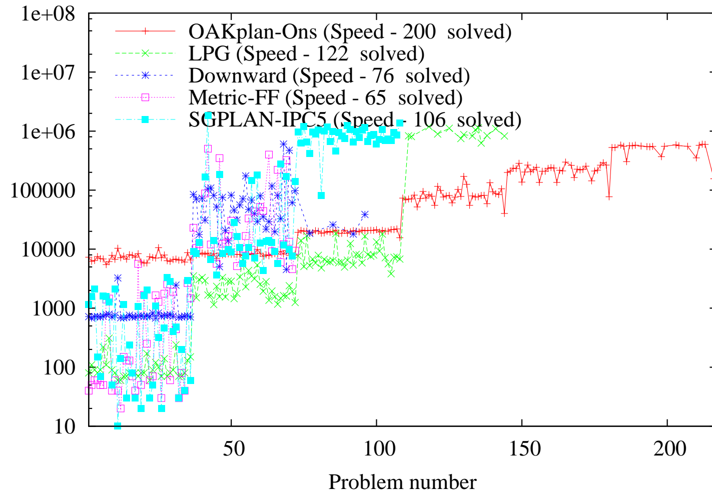
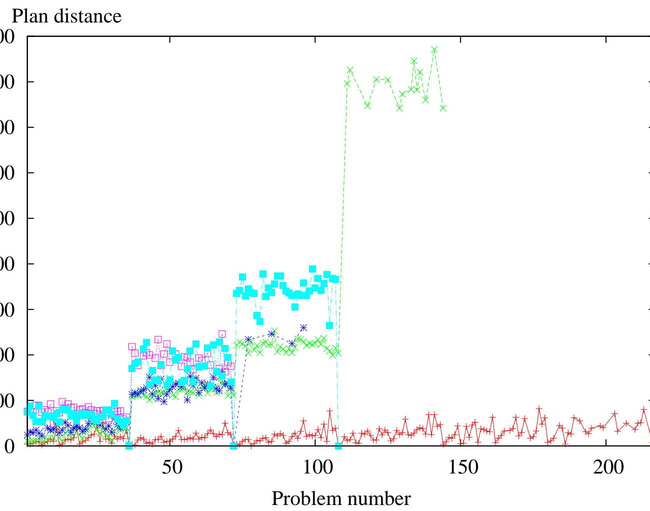


Figure 59: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Logistics variants. Here we compare OAKplan-Ons vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.





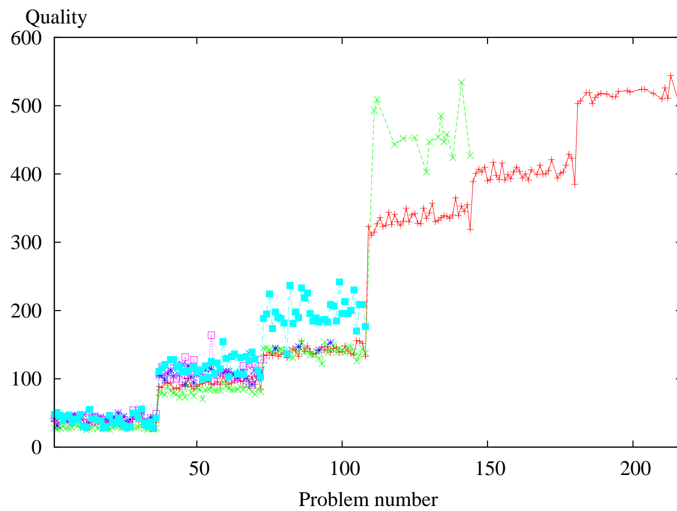
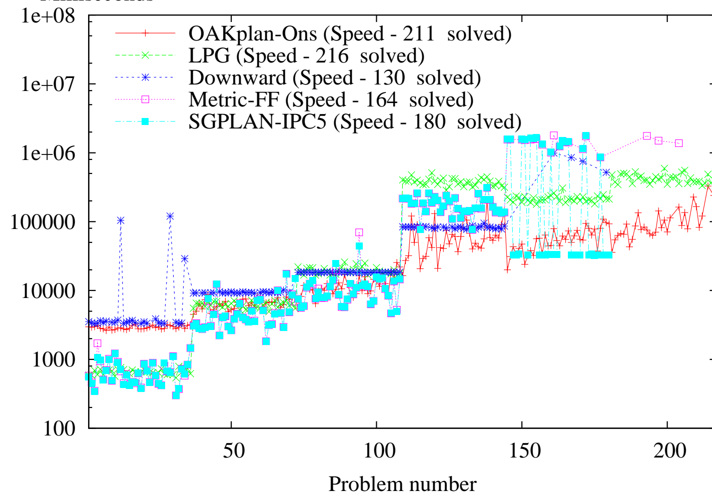
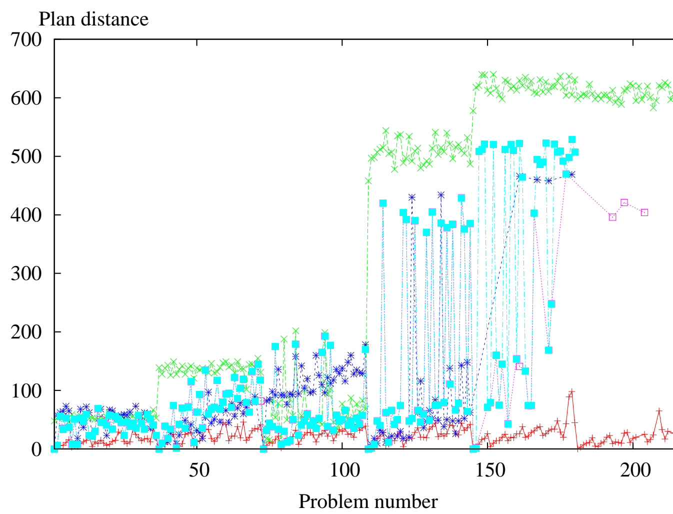


Figure 60: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the DriverLog variants. Here we compare OAKplan-Ons vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.





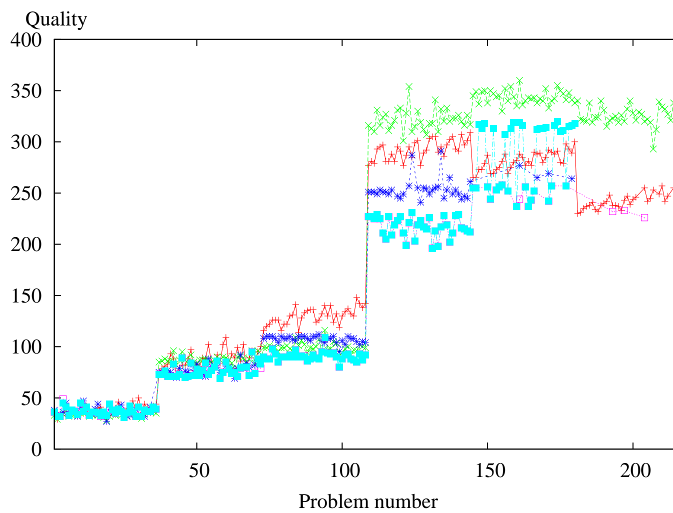


Figure 61: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the ZenoTravel variants. Here we compare OAKplan-Ons vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.

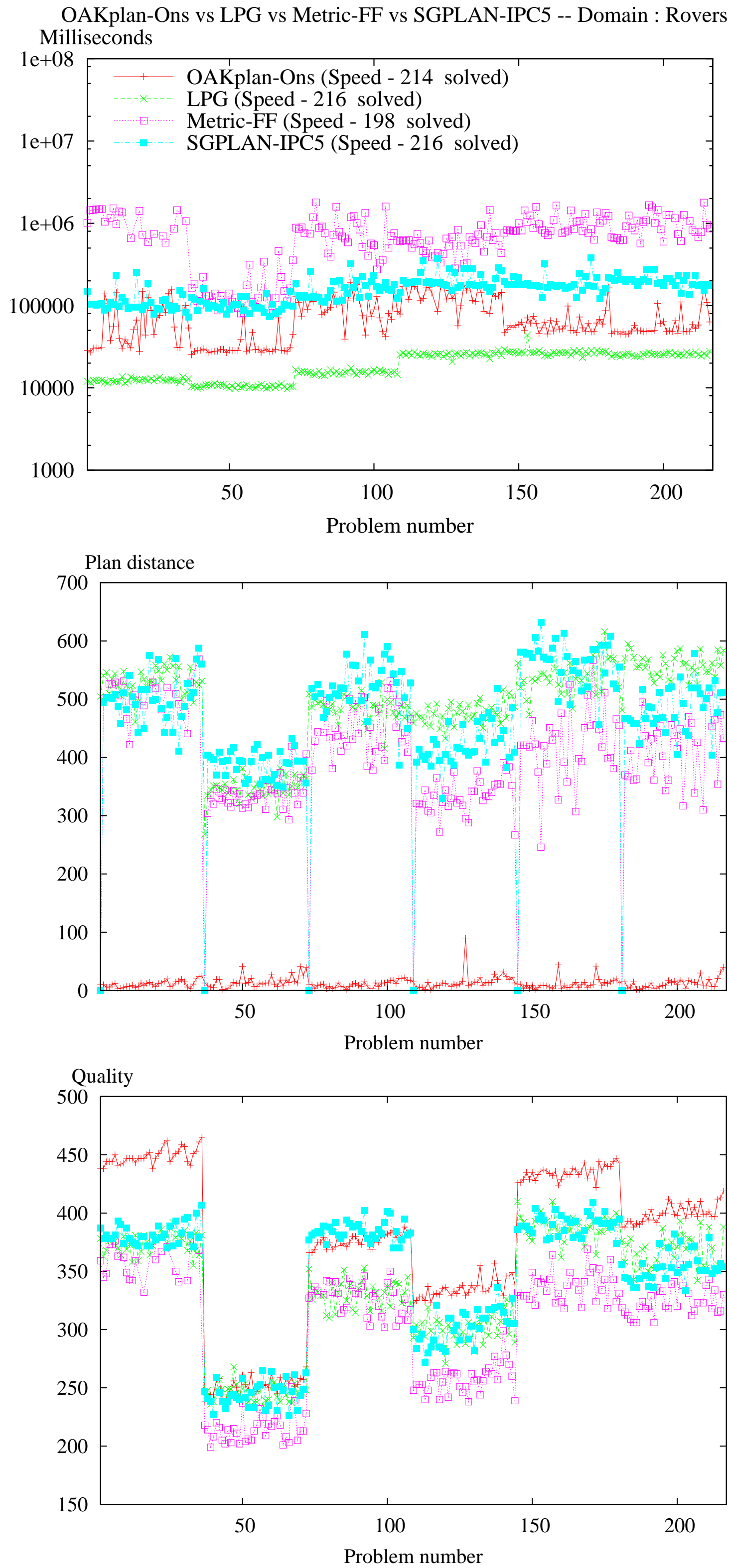
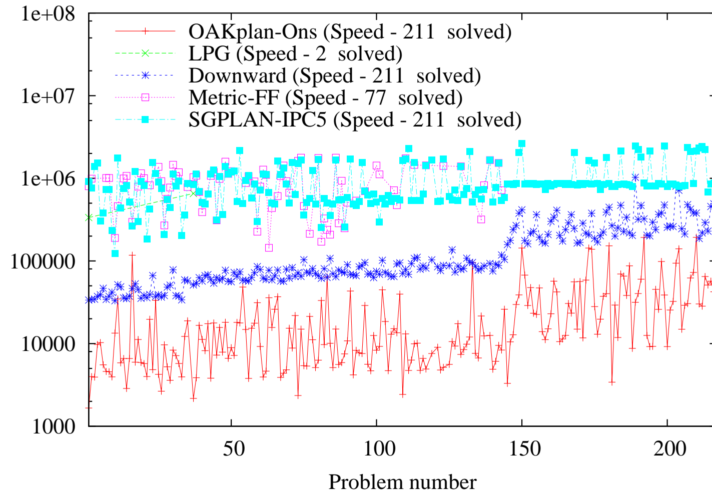
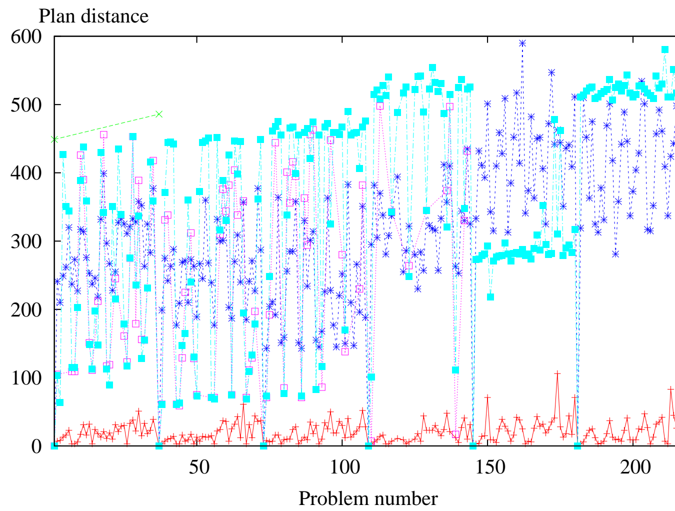


Figure 62: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the Rovers variants. Here we compare OAKplan-Ons vs LPG vs Metric-FF vs SGPLAN-IPC5.





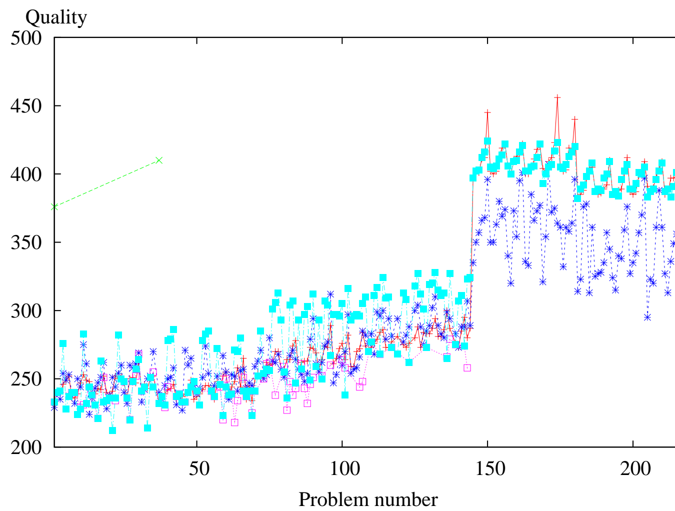


Figure 63: CPU time (on a logarithmic scale), number of different actions with respect to the input plan of the adaptation process and plan qualities for the TPP variants. Here we compare OAKplan-Ons vs LPG vs Downward vs Metric-FF vs SGPLAN-IPC5.

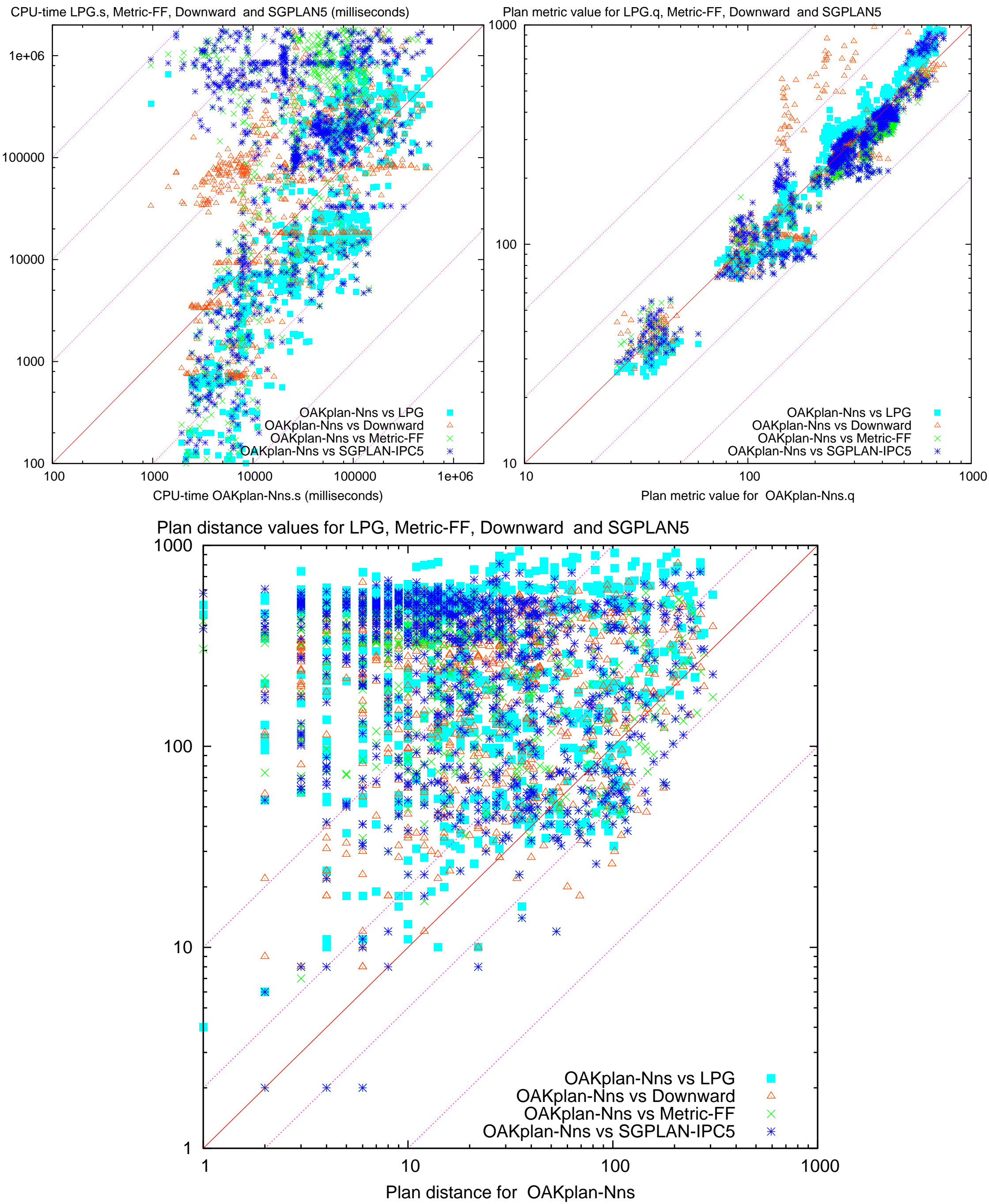


Figure 64: CPU-time, plan qualities and plan differences of OAKPLAN-Nns vs DOWNWARD, LPG, METRIC-FF and SGPLAN-IPC5.

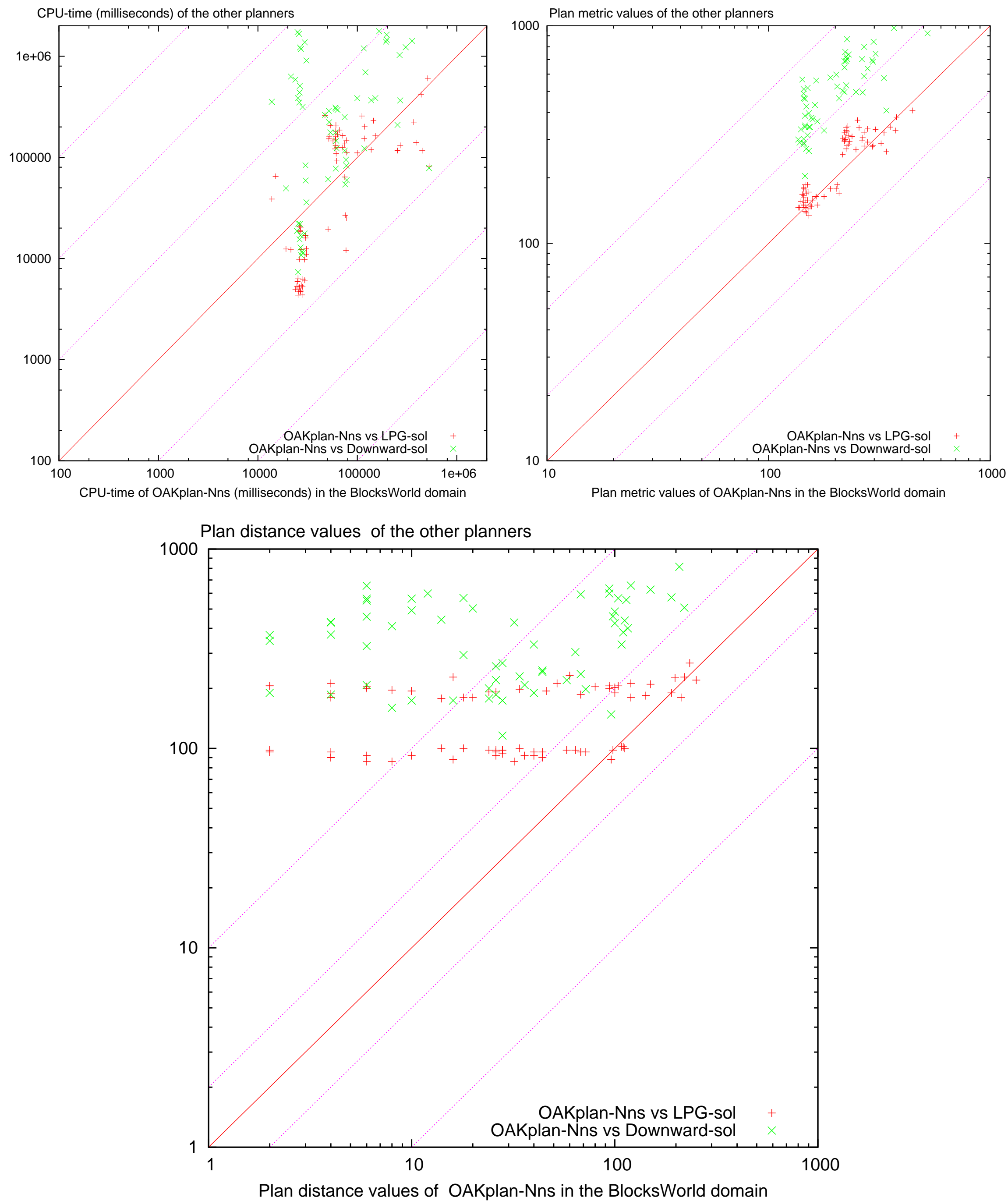


Figure 65: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the BlocksWorld domain.

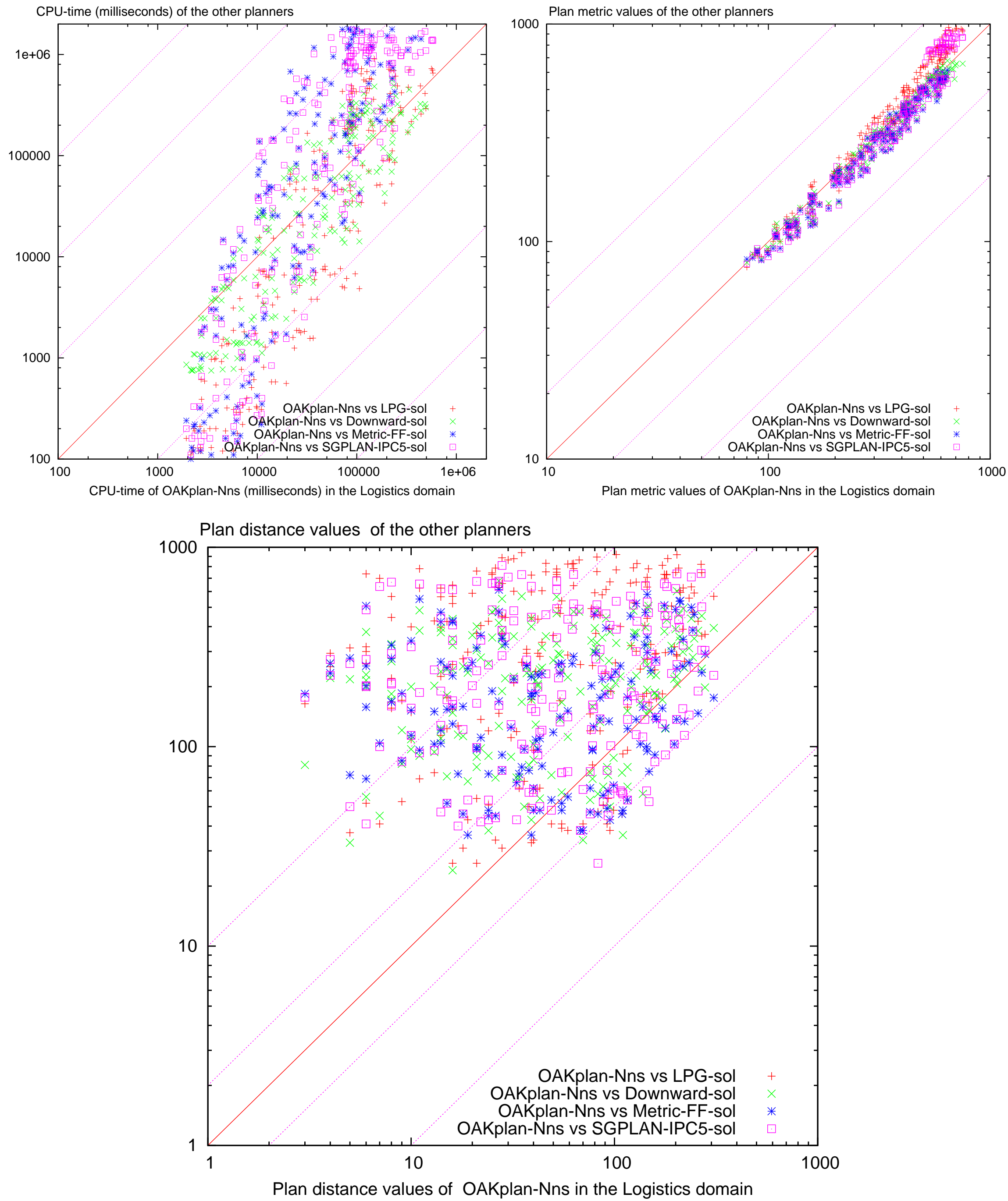


Figure 66: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the Logistics domain.

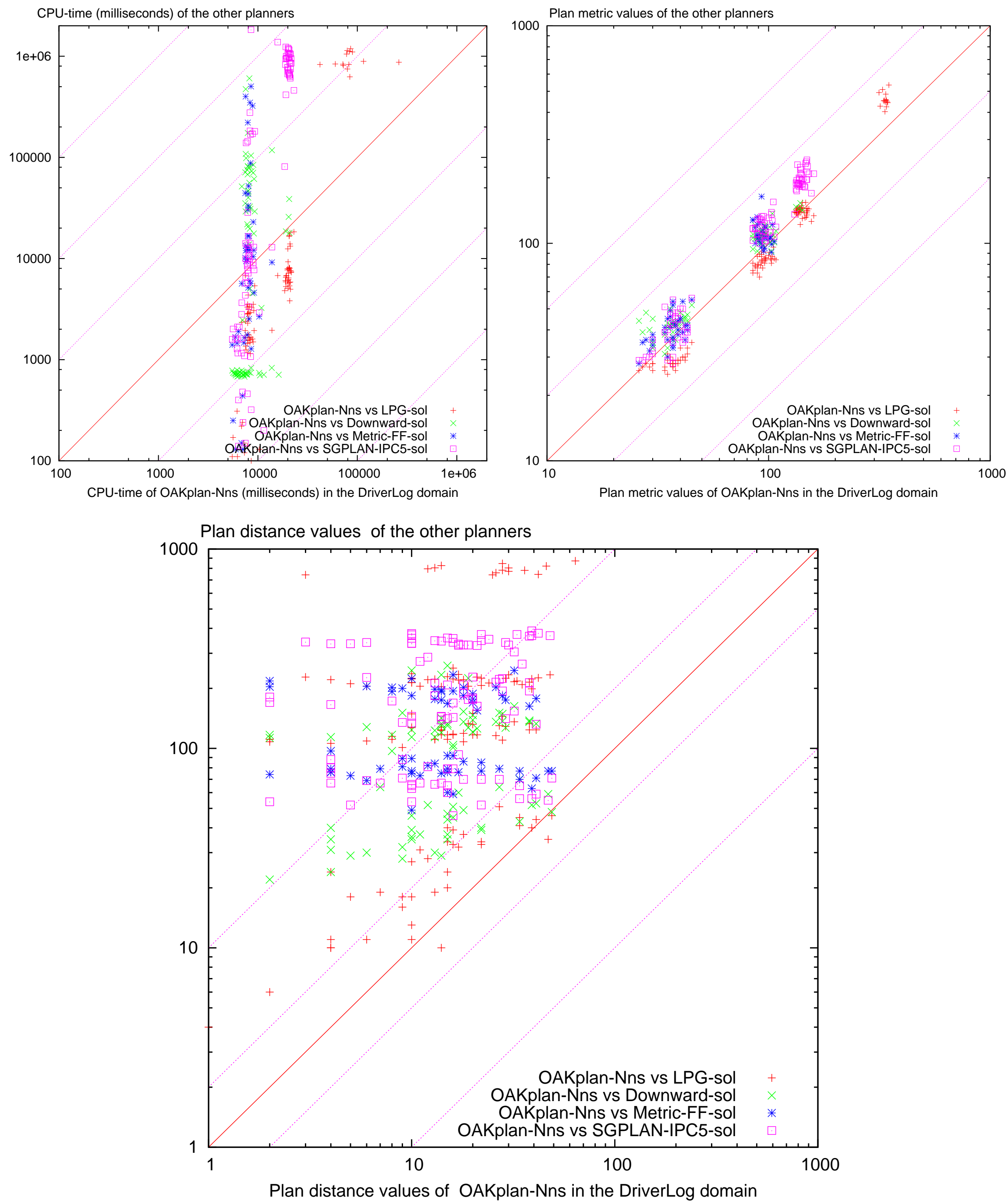


Figure 67: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the DriverLog domain.

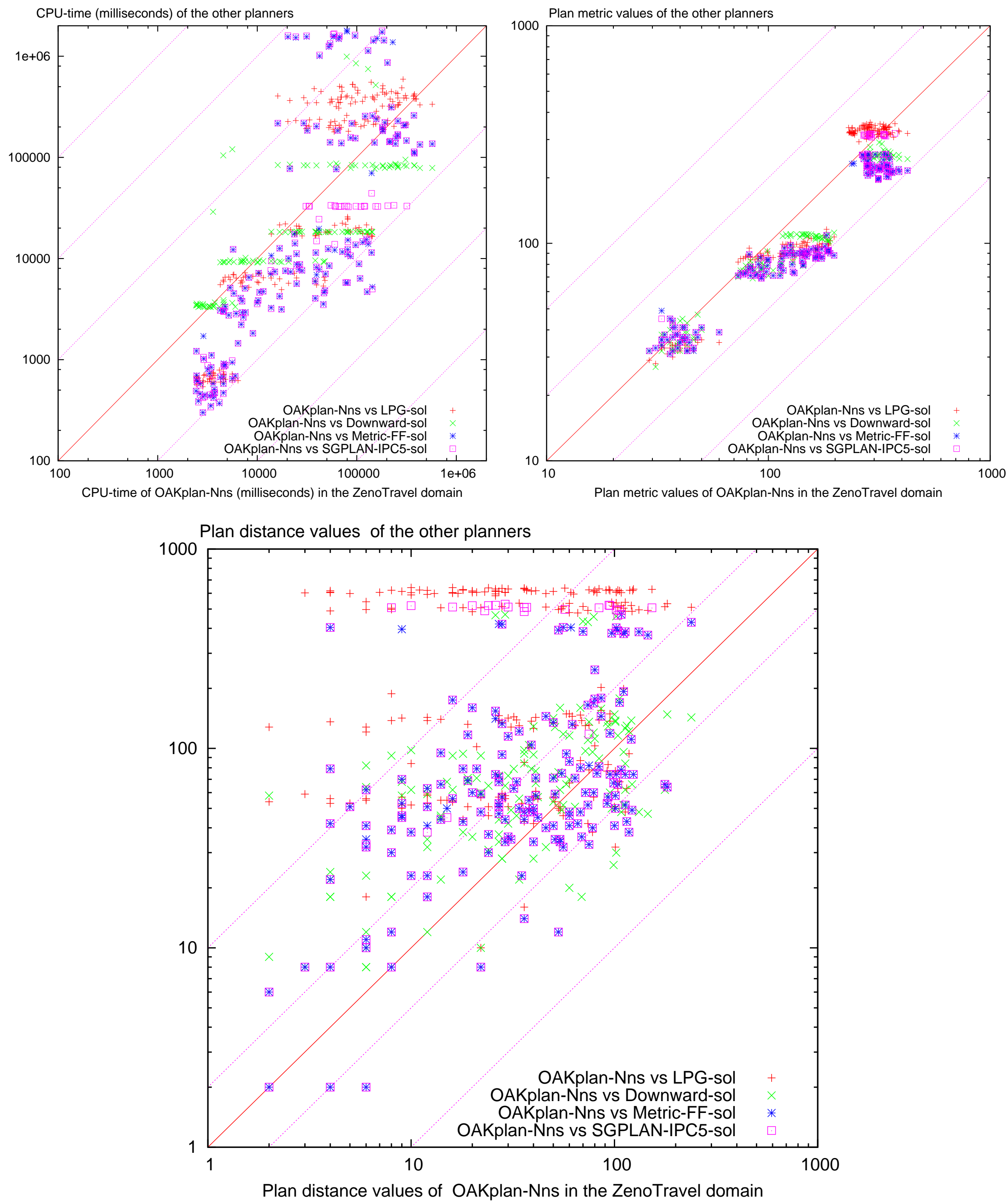


Figure 68: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the ZenoTravel domain.

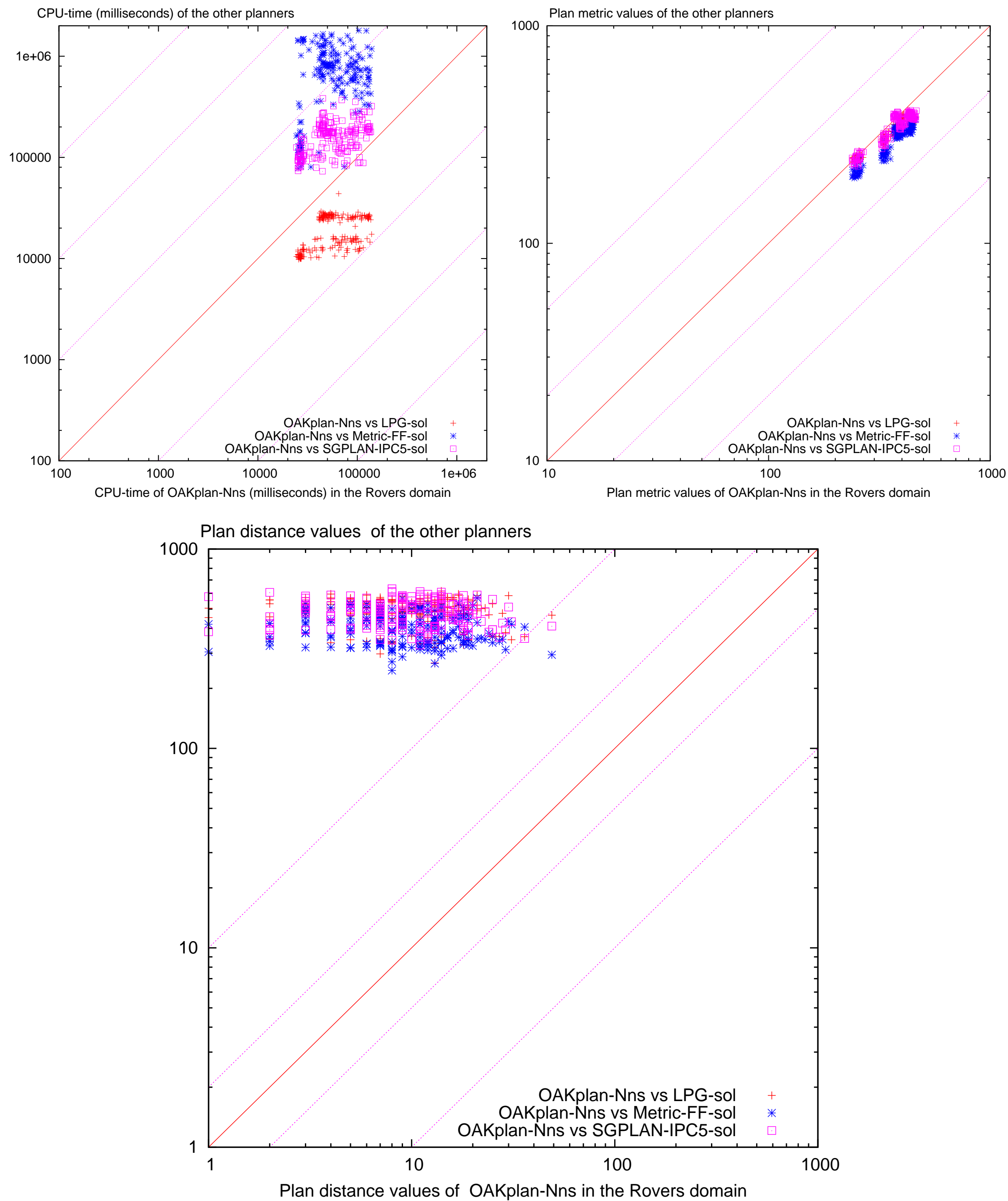


Figure 69: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the Rovers domain.

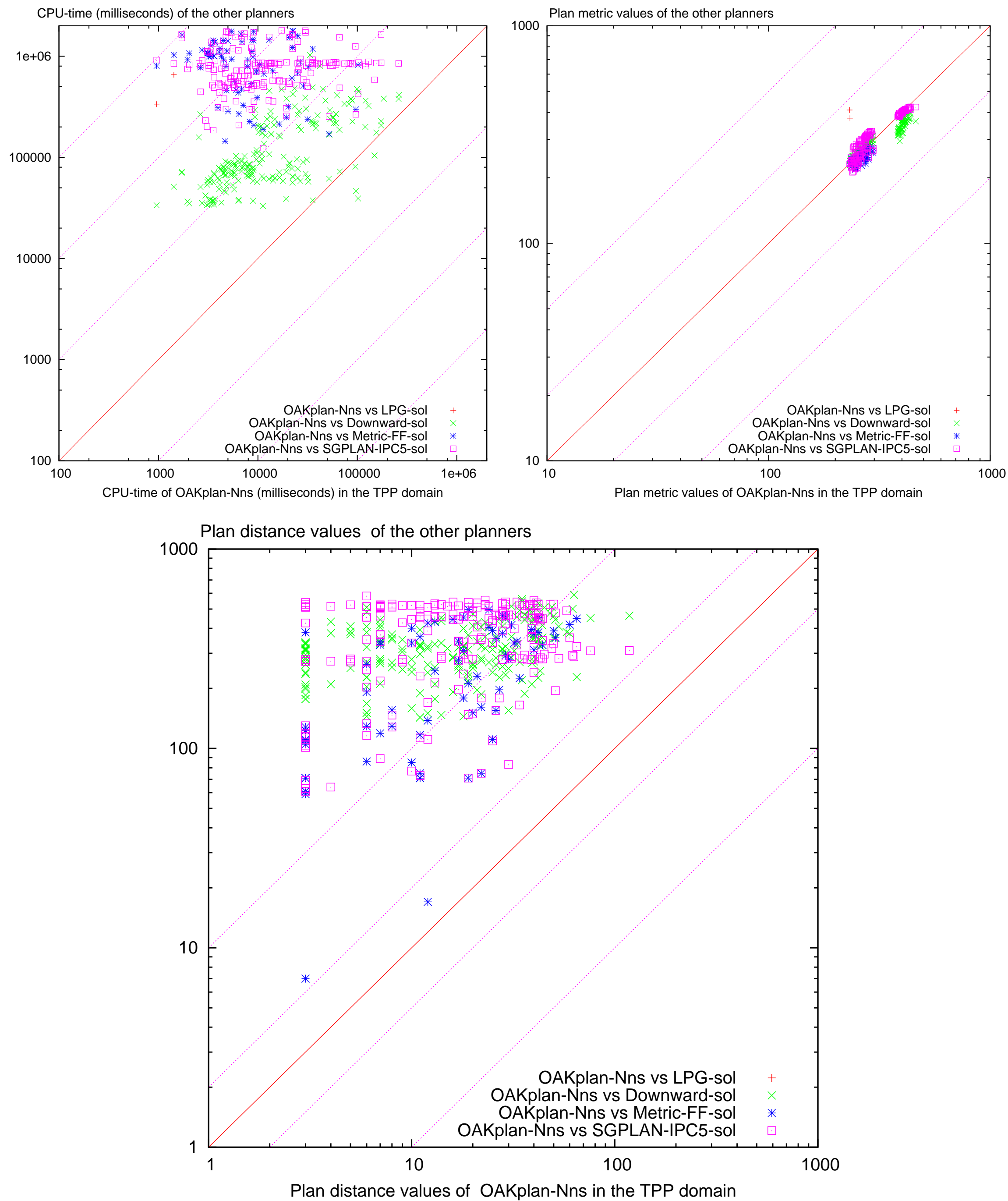


Figure 70: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Nns and the other planners in the TPP domain.

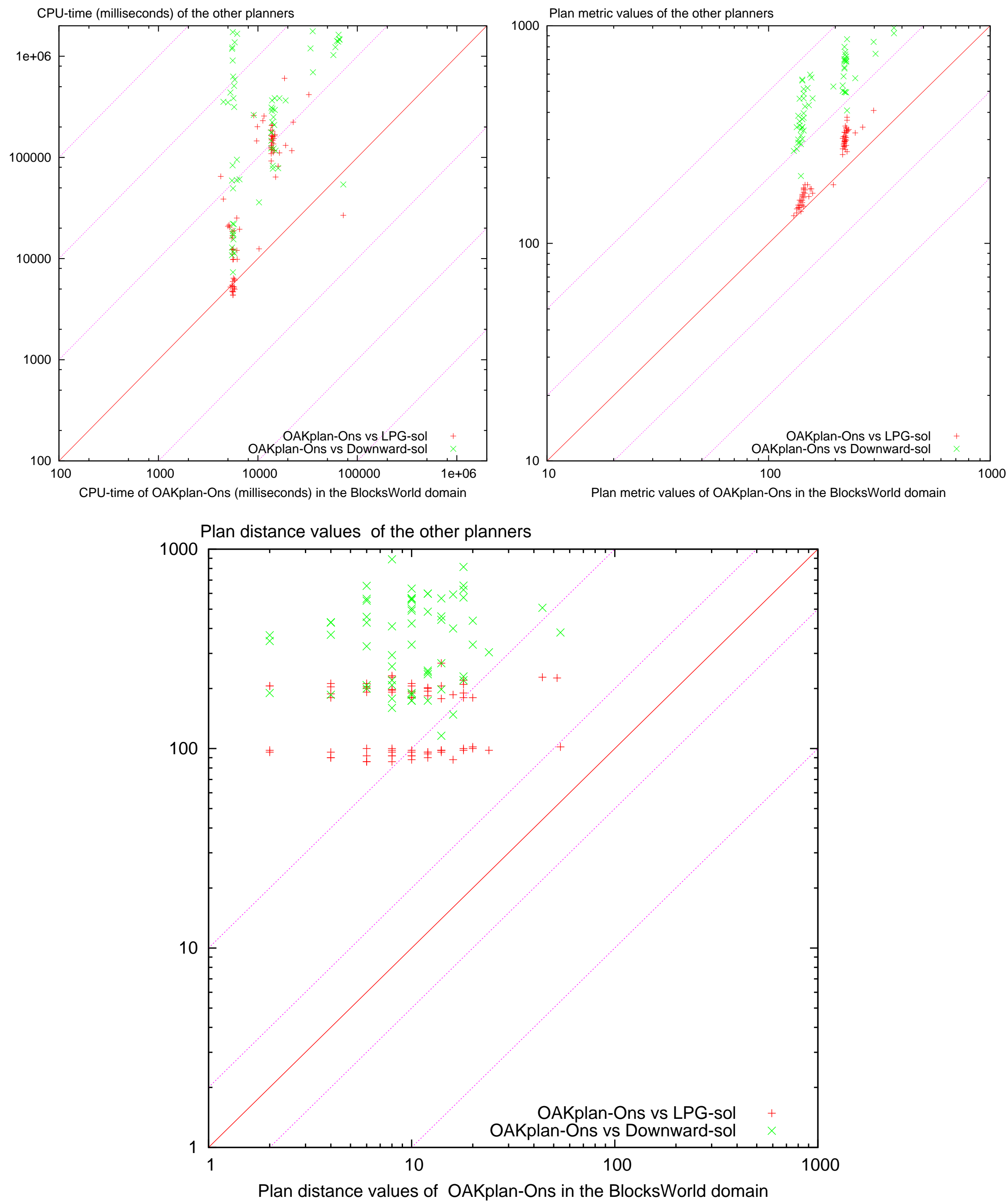


Figure 71: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the BlocksWorld domain.

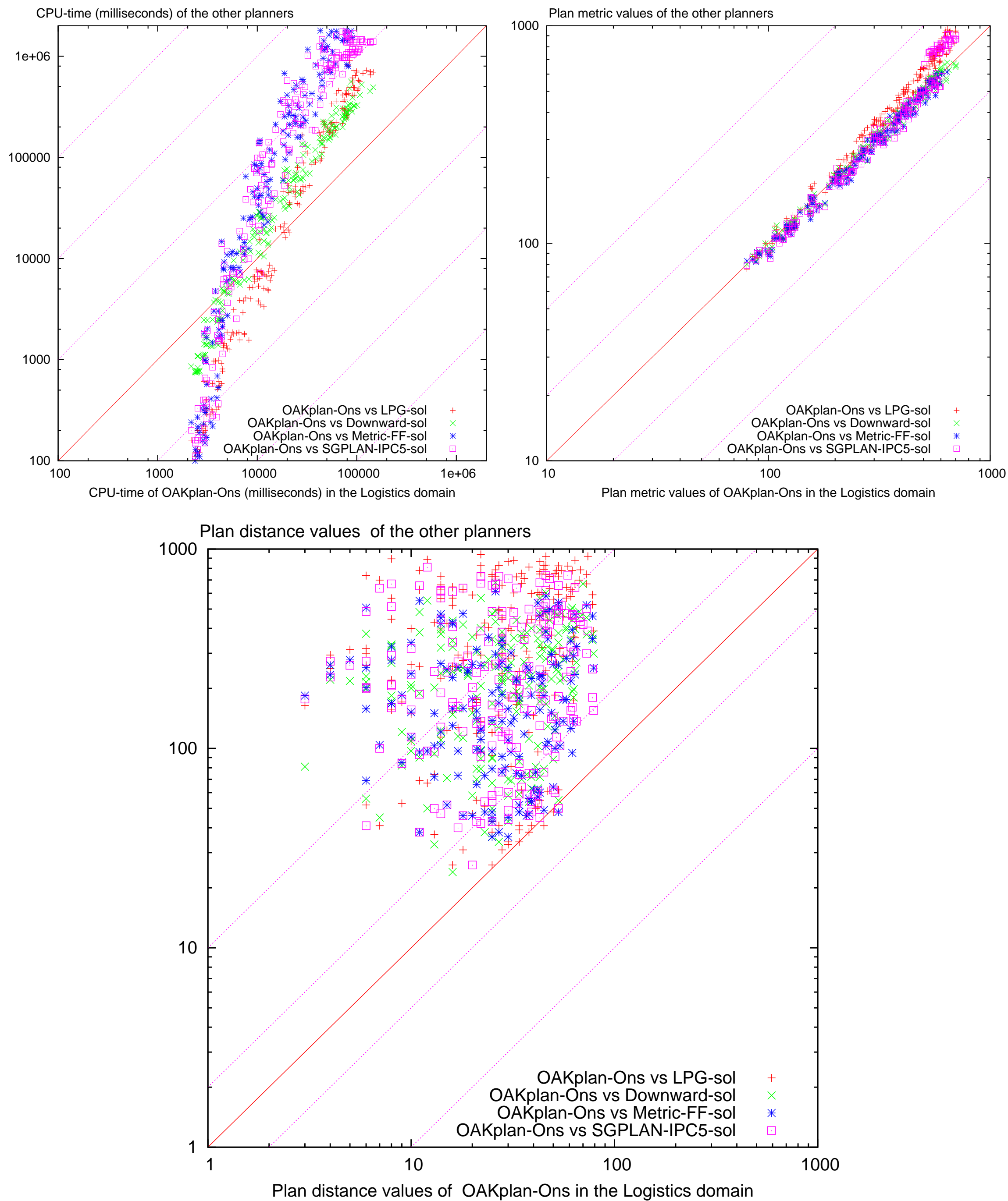


Figure 72: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the Logistics domain.

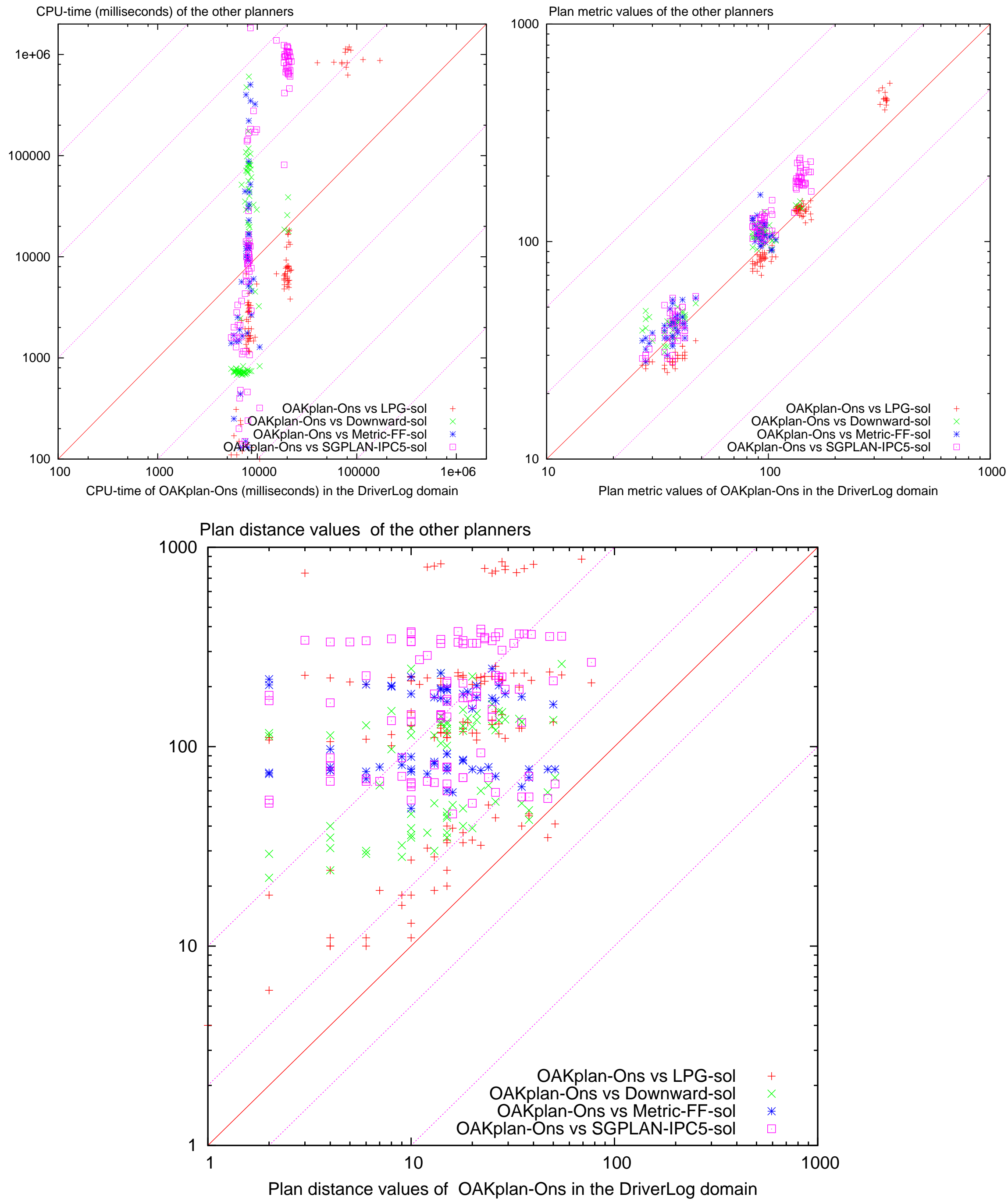


Figure 73: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the DriverLog domain.

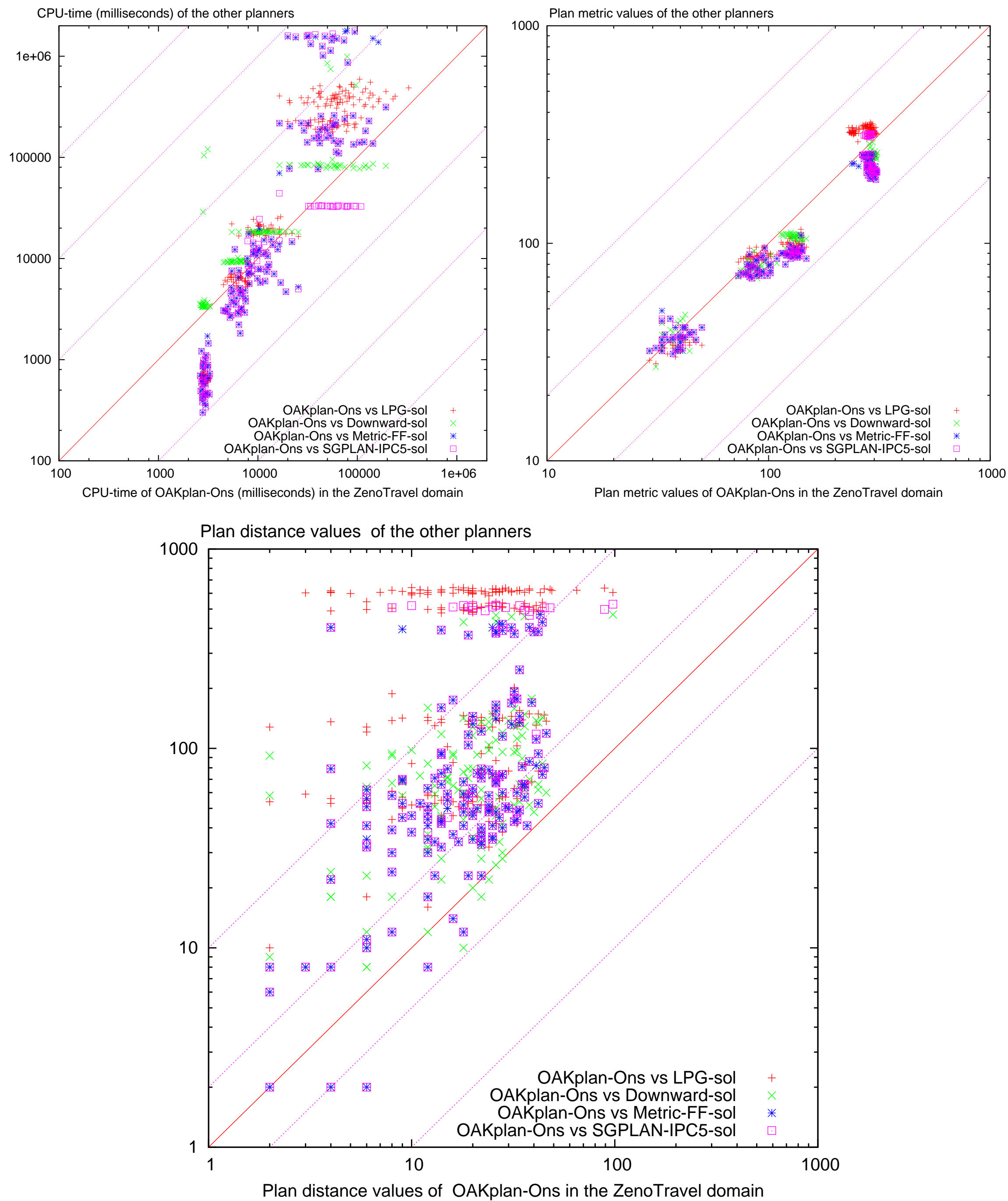


Figure 74: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the ZenoTravel domain.

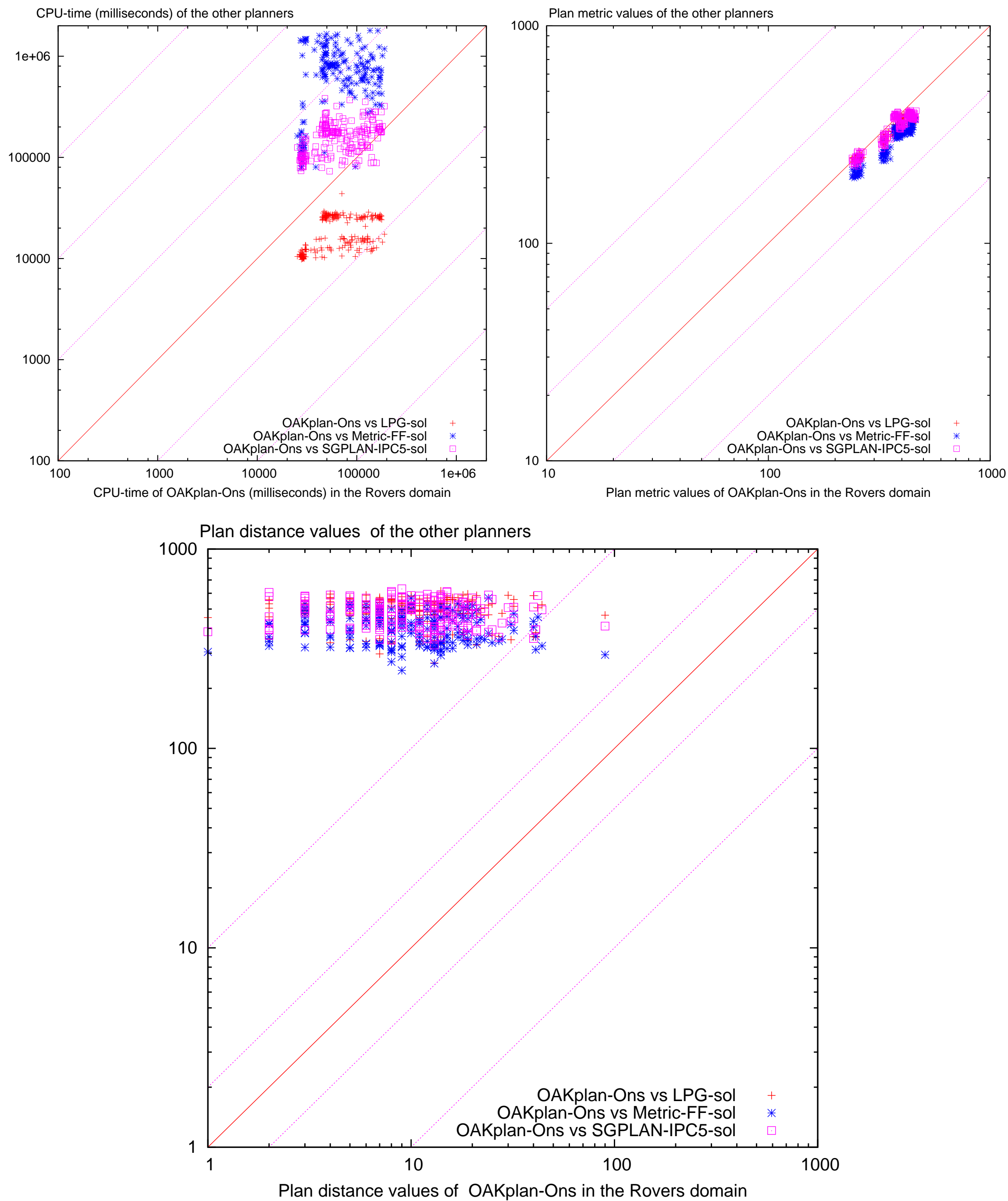


Figure 75: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the Rovers domain.

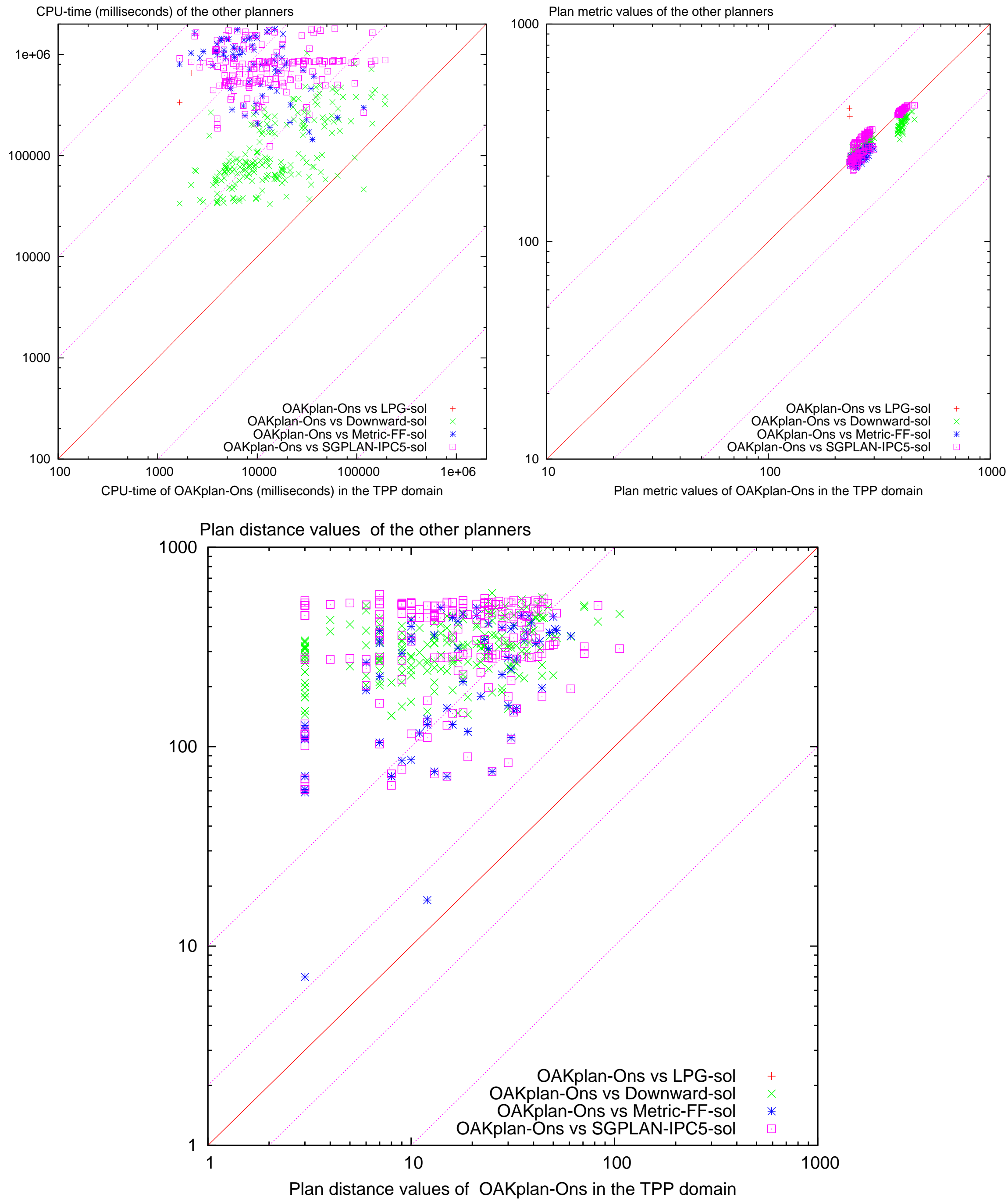


Figure 76: CPU time, plan qualities and number of different actions with respect to the target plans for OAKplan-Ons and the other planners in the TPP domain.

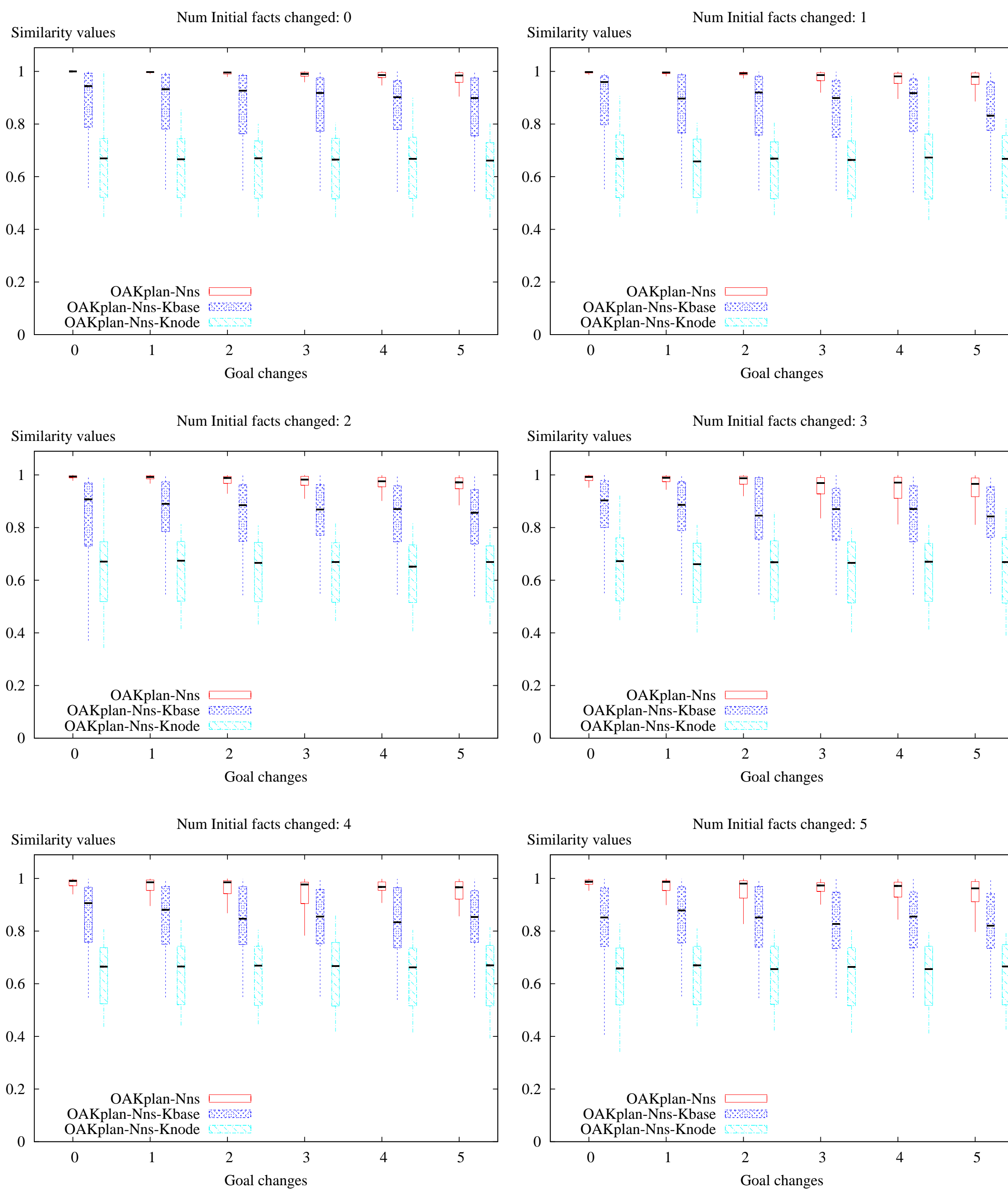


Figure 77: Box & Whiskers plots of the similarity values produced using $\mathcal{K}_{\mathcal{N}}$, $\mathcal{K}_{base}$ and $\mathcal{K}_{node}$ kernel functions in the different benchmark planning problems considering different numbers of initial changes (reported on the top of the figures) and different numbers of goal changes (reported on the x-axis of the figures).