Design of a Testbed for Planning Systems

Klaus Varrentrapp and **Ulrich Scholz** and **Patrick Duchstein**
Intellectics Group
Darmstadt University of Technology
Alexanderstraße 10
64283 Darmstadt, Germany
klaus@varrentrapp.de
scholz@informatik.tu-darmstadt.de
patrick@duchstein.com

Abstract

Conducting computational experiments and analyzing their results in a sound manner can be tedious. Experiments have to be organized, i. e. algorithms in various configurations, with several inputs and repetitions have to be run and results have to be analyzed from different perspectives, including statistical evaluation. In this paper we discuss properties of an ideal testbed for experiments and the statistical evaluation thereof, which automates recurring tasks and supports scientific soundness. We present a prototypical testbed, which will realize some of these ideas and which is short of being completed.

Introduction

Currently, we find many proposals for techniques addressing different aspects and phases of the planning process. The right combination of these techniques is necessary to fully exploit their mutual strengths and remedy their mutual weaknesses. However, such combinations mostly have been realized in monolithic blocks resulting in yet another planning algorithm. Conversely, the different phases of planning could be implemented as independent modules with a common interface. With the ability to assemble these building blocks freely, complete planning systems could be designed more easily, yielding greater flexibility and possibly better performance. With a modular structure of planning algorithms, researchers can compare and combine their work.

Experiments mainly comprise configuration and tuning of algorithms and comparison of different algorithms and modules. The process of carrying out computational experiments (experiments in short) of any kind with combinations of algorithms and modules is challenging. Organization and realization of experiments require a lot of practical details with respect of managing the different computation steps, such as providing input, operating the algorithms, storing intermediate data, and analyzing the the results. Besides, appropriate settings of experiments in order to obtain reliable results requires some insight in the problems associated with an empirical analysis of algorithms (McGeoch 1996; 2001; Hooker 1994; 1996; Gent & Walsh 1994; Moret 2002; Rardin & Uzsoy 2001). Hence, it seems desirable to have an easy to use environment for conducting experiments that automatically takes care of most of the recurring tasks.

In order to make scientifically sound decisions, one eventually has to rely on statistical tests. Consequently, an environment for experimentation should include a component for conducting statistical tests, too. Several researchers, mainly from the field of optimization, have addressed the issue of statistical testing and identified it as crucial for future empirical research on algorithms[1] (Cohen 1995; Xu, Chiu, & Glover 1998; Birattari *et al.* 2002; Coy *et al.* 2000). The need for a testbed unifying experiments for different data sets and schemes has been identified in the machine learning community before (Garner 1995; Witten & Frank 2000).

In this paper, we elaborate on the previously mentioned issues of designing a testbed for experimentation with algorithms – with emphasis on planning algorithms. We propose our vision of a general modular architecture that will meet the requirements and address the issue of statistical evaluation. We describe a prototypical implementation of such a testbed for planning algorithms, which is currently under development.

First, we define some basic notions. Experiments with algorithms can consist of a series of runs of one or more algorithms with various parameter settings in an environment that is under control of the experimenter. Combinations of various independent algorithms, say several preprocessing procedures with a planning procedures, can be seen as a (hierarchical) algorithm, too. The purpose of such experiments is, e, g. to find the best algorithm or the optimal configuration of several algorithms, i. e. the best setting of parameters.

From now on, we will denote with *algorithm* any single or composite procedure that is being investigated. With *configuration* we refer to any distinct setting of components and their parameters that yield a running entity. The notion of *experiment* then denotes a coherent conglomeration of running an algorithm in different configurations or comparing a number of algorithms in various configuration in order to gain insight in the behavior and the relationship between algorithms.

[1]www-users.cs.york.ac.uk/~tw/empirical.html

Requirements

In this section we identify several recurring processes and problems that occur when conducting experiments to study algorithms. Based on these basic processes, we propose some requirements for a flexible testbed that automatizes and generalizes the process of experimentation. Subsequently, we briefly mention some of the expected advantages of such a tool.

When studying algorithms empirically, certain recurring general processes appear:

- Provision of problem instances. All algorithms must operate on some problem instances. These must be supplied by either enabling access to benchmark repositories or by including perhaps randomized problem instance generators.
- Operation and synchronization of the algorithms used within the scope of an experiment.
- Ensuring proper flows and potentially storage of all data involved, such as problem instances, algorithm output, data describing the setting of the experiment, the generated test instances, and so on.
- Identification of all data relevant to and produced by an experiment. Delimitation of data from different experiments.
- Processing of raw algorithm output for further analysis. This involves tasks such as plotting graphs and computing statistics like mean, variance, and so on.
- Conducting statistical tests to scientifically verify any statements extracted from the experiment results.
- Supervision, status checking and perhaps modification of the experiment during execution.
- Recovering from partial crashes. The longer the runtime of an experiment, the higher the likeliness of malfunctions. Such disruptions should not corrupt the results. Instead, improper parts of the experiment should be repeatable on demand or even repeated automatically.

Following these general requirements we derive our vision of more specific demands for an ideal testbed:

- Provision of a user friendly interface. This user interface enables the user to specify experiments, supervise them, explore immediate results, trigger statistical evaluation, and visualize any required information in this respect. Ideally, all operations of the experiments can be controlled via the user interface. The interface also enables the user to retrieve any stored data from past experiments, thus allowing easy comparison of earlier results.
- Cooperation with any algorithms and modules and existing analysis tools. The testbed allows any aspect of an algorithm to be subject of investigation. Consequently, a general interface for accessing, controlling and running algorithms with standardized input and output streams will be included. This way, arbitrary algorithms and modules that comply with the interface requirements and stream data formats can be combined. A role model for this approach can be seen in scripts and pipes of Unix. Additionally, provisons for enabling subsequent analysis of data by statistical packages will be integrated.
- Extensibility and adaptability. Users are given the freedom to extent the testbed with individual modules, especially with their own tools, such as scripts for extracting data from output-files for plots. This extention will be at best be feasible without major changes to the testbed. In the ideal case, only interfaces need to be altered.
- Decoupling of the various processes of experimentation. Following this requirement, it seems necessary to give the testbed a fully modular structure as is common in modern software engineering.
- Coherent storage of all experiment relevant data. Special attention has to be given to the issue of storing and labeling of all relevant data describing all aspects of an experiment. In order to reproduce and compare data, it is vital to store data describing the setting, such as the version of operating system and algorithms. Furthermore, it is desirable to facilitate future retrieval of all experiment relevant data, such as raw output data, processed data, and a description of evaluations and tests performed. Altogether, the complete data from any experiment should be accessible cohesively and comfortably.
- Flexible specification of complete experiment settings. All aspects of conducting experiments can be subject to change by the researcher without any confinement. The testbed supports this via a general experiment specification language, which can be seen as a programming language for conducting experiments. It is independent of the algorithms under investigation. A proper design of such a language will be crucial to this venture.

Immediate advantages when using a testbed that meets the previous demands are increased efficiency and the possibility to reproduce and compare experiments. A modular object-oriented design with independent building blocks will enable a heterogeneous and iterative development of the modules. Necessary refinements, generalizations and abstractions of the everyday problems of empirical experimentation can only appear when using it. Hence, being able to independently improve the various processes is of greatest practical use. That way, such an environment can grow with the experience gained with it.

Having an experiment specification language, general templates and procedures for conduction experiments could be extracted and proliferated, putting empirical evaluation on an even more scientifically basis, since non-experts in statistics, now, can use these templates to perform a sound statistical testing. Experimenters can concentrate on the algorithms instead of having to devote energy to the rather complicated and difficult problem of setting up experiments properly.

For example, consider a planning algorithm consisting of several individually parameterized components. The aim is to fine-tune the parameter setting. Dependent on the results of earlier runs with some initial parameter values as validated by statistical tests, later runs can explore more promising areas of parameter settings. This interaction of runs and analysis of results could be expressed within a specification language by some loops and conditional expressions and thus this process could be automated as done in (Birattari *et al.* 2002; Xu, Chiu, & Glover 1998; Coy *et al.* 2000). Each such imperative description of experiment realization can be viewed as a general template,script or program, applicable to other algorithms, too. Generally, any experimental design such as the Taguchi Design (Roy 1990; Tsui 1992) can be regarded as a kind of template or script.

Proposed Architecture

In this subsection we further elaborate on our preceeding reflections and propose a general architecture for a testbed for experimentation with algorithms, depicted in figure 1.

The envisioned testbed consists of several components or modules for the specific tasks of experimentation. These modules are independent to such a degree that they can be exchanged with newer versions without having to change the other parts of the testbed. Special care has to be addressed to the interfaces of theses modules. By interface we primarily mean part of a module that handles communication with other modules. Only secondly we denote by interface the fixed communication protocol used between a module and one of its interface parts. Interfaces in the primary meaning are needed to flexibly separate modules. The independent modules are all controlled by a central control unit (CCU) with standardized interfaces and with no or little interaction between the modules. When changing a module, only the module and perhaps its interface to the CCU has to be changed.

In case any component fails, the CCU can detect such a crash and recover it, e. g. by repeating it or by notifying the user. Modules need not be one single entity but can be a collection of loosely related small tools that are transparent to the CCU by means of the interface that controls them directly. In the simplest case, an interface only relays commands and data. Additionally, we propose to direct all flow of data via databases. By this, all data including intermediate data can be stored efficiently and safely. One database, which will store the experiment specification, can also store pointers to all experiment relevant data, so this data can be collected from the various databases and thus be retrieved afterwards.

The CCU manages all operations that have to be done in order to conduct the experiment as a whole. Its various interfaces comprise:

- Interface to the user interface (IUI).
- Interface to the unit that runs and controls the algorithms (IRU).
- Interface that reads a new specification of an experiment (IES).
- Interface for accessing a database where experiment specifications and pointers/keys to all relevant experiment data is stored (DBI).
- Interface to the module that performs statistical processing such as data analysis and statistical testing (ISU).
- Interface to the module computing all displays and graphics presented to the user (IDU).

The modules that belong to the previously listed interfaces are described next:

Run Control Unit (RCU) This module operates the algorithm in terms of the combination of algorithm components subject to the experiment. It invokes the components with correct parameter settings, ensuring proper flow of data between the components, provision of problem instances as input, and storage of any output data. As determined by the experiment specification, the CCU invokes the components in proper sequence with actual parameter values on the specified instances. This module again has certain interfaces to ensure flexibility with respect to different problem inputs and algorithm components. These are:

- Interface to the problem generation unit (IGU). The problem generation unit (PGU) typically is either a potentially randomized problem instance generator or retrieves its problem instances from a benchmark repository. Generally, the PGU is the module that provides the experiment input.
- Interface to algorithm components (IAC). Heterogeneous algorithm components might have very individual modes of control and in/output data formats. In order to give enough flexibility, this interface can be exchanged and tailored to the concrete algorithm components at hand, e, g. by means of wrappers for single components.
- Database Interface (DBI): This interface stores the output data to the appropriate database.

Display Unit (DU) The DU is responsible for computing presentations of any data regarding the experimentation results. These comprise the data from statistics such as means, variances, confidence intervals, results from statistical tests such as tables from analysis of variance procedures, and any plots and graphs illustrating aspects of the experiment such as runtime vs. solution-quality trade-off curves. Typically, this unit will employ plotting programs such as Gnu-plot.

Statistical Unit (SU) This module computes any statistics needed by the user. Additionally, it takes care of conducting any statistical testing. The data it needs to accomplish this is retrieved from the database where the ARU stores the raw output data from the applied algorithms. Typically, this module comprises a statistical package and some added tools.

User Interface (UI) The UI provides easy management and overview of the experiment to the user. It transmits orders from the user to the CCU and displays feedback from the CCU. The UI can include provisions for easy editing and processing of experiment specifications. Additionally, the presentation of results of any kind will be relayed through the UI from the DU and SU, respectively. Implementing the UI as a web interface permits a central experimentation facility equipped with appropriate hardware that is shared via the Internet.

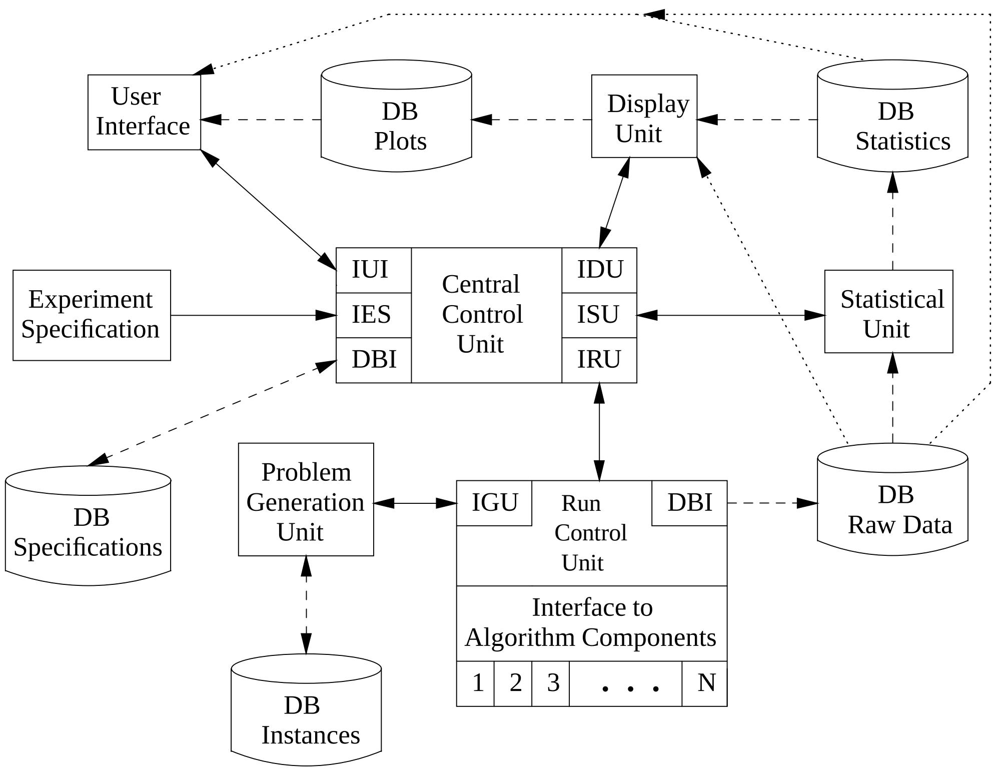

Figure 1: **Architecture of a testbed for conducting empirical experiments.** (Full lines represent control information, dashed lines indicate flow of data, and dotted lines indicate optional flow of data.) The central control unit (CCU) controls all processes of the testbed. It reads the experiment specification through its experiment specification interface (IES) and subsequently instructs the run control unit (RCU) running the algorithms, the statistical unit (SU) processing the results, and the display unit (DU) computing display such as plots and tables presented to the user via the user interface (UI). These units are accessed by the run control unit interface (IRU), the statistical unit interface (ISU), the display unit interface (IDU), and the interface to the user interface (IUI), respectively. The RCU retrieves problem instances through its problem generation unit interface (IGU) from the problem generator unit (PGU) which either accesses a database of instances or employs a problem instance generator. All intermediate data is stored persistently in various databases (DB).

This architecture is by no means fixed. It has to be tested and adapted in practical use of the testbed. Particularly, a proper design of the interfaces can only be achieved by experience. So far, the architecture mainly reflects the insight in the necessity of separating replaceable parts of the testbed.

Statistical Testing

Tuning and comparing algorithms are important aspects when designing algorithms. Usually, these tasks are performed by running algorithms with different parameter settings on a collection of problem instances. These instances either come from a benchmark repository or they are generated on demand, often randomly. On the basis of such experimentation results, postulations about the behavior and performance of the tested algorithms are made.

However, empirical results can be quite misleading. When using benchmarks as instances, for example, exact postulations can only be formulated with respect to the set of benchmarks used or with respect to the set of instances that can be generated by the problem instance generator used. Generalizations to a bigger set of problems (instances) are subject to immanent uncertainty. There is always the danger of tuning for the benchmarks or the problem generator (Hooker 1996). In fact, the excerpt of instances used for any experimentation need not be representative of all possible or of all relevant instances. When using randomly generated problems, it could happen, that, by chance, the instances generated are quite easy, suggesting the algorithm tested is very good. However, in this case, the results certainly should not be generalized, since the tested instances are too easy, which, unfortunately, is unknown to the experimenter.

There is always some kind of uncertainty about the results of experiments due to the randomness involved. Generally, there are two sources of randomness in many computational experiments. First, any choice of the test set of problem instances effectively is drawn from the underlying set of all valid or possible problem instances and as such a random experiment. Second, algorithms can themselves be randomized and hence each run is a random experiment, too. It should be clear that generalizations from the results of random experiments have to be handled with care.

A random experiment draws a number of individuals, called the sample, from a set of possible outcomes, called population. The aim of statistical inference is to control the danger of making wrong generalizations beyond the sample, since generalization from the sample is only valid when the sample is representative of the population.

The law of probabilities which governs the distribution of the observed values from a random experiment is called probability distribution. The random experiment of repeatedly throwing a coin observing heads vs. tails, for example, is distributed according to the binomial distribution with parameter p indicating the probability of observing heads. The set of all binomial distributions different only in the value of p is called the family of binomial distributions. Generally, families of distribution functions are called parameterized distributions.

Statistical inference exploits fundamental relations between the size of a sample, the variance of the observed values, and the confidence of the conclusions, following the intuition that the smaller the variance and the bigger the size of the sample, the more confidence we can have in our generalizations. Three main questions emerge when analyzing random experiments. All questions seek confident information about the true value of a parameter such as p previously or about the true value of a function of some parameters such as p^2 assuming some fixed family of probability distribution:

1. **Parameter estimation:** What is the value of a parameter or function?
2. **Confidence intervals:** What are the bounds of the value of a parameter or function given a certain confidence is expected?
3. **Hypothesis testing:** Does a parameter or function value fall into a certain interval? How high is the probability it does?

Consider running two planning algorithms pairwise on the same problem instances measuring the difference of the length of the computed plans. Assuming this difference is distributed according to a normal distribution, examples of the previous problems are:

1. What exactly is the mean difference for all problem instances? Particularly, is the mean difference positive or negative (indicating superiority of an algorithm)?
2. What are the bounds of the mean difference with a probability of 95%?
3. Is the mean difference below 0? How high is the probability it is?

Statistical inference further can be employed to answer questions about the magnitude of deviations of observed results from the expected mean. Further on, statistical inference can be employed to test whether a set of algorithms performs equally well or to regress the behavior of an algorithm depending on the problem instance size. Sometimes it is interesting to hypothesize on the underlying family of probability distribution. Finally, the influence of certain factors such as different parameter settings can be tested. One can test for mutual independence vs. correlation of a set of factors and whether a certain factor has an influence at all (Larson 1982; Lehmann 1997). When no assumption about the underlying probability distribution can be made, these questions can still be answered with so called non-parametric tests (Siegel 1956). However, they are less powerful. Existing standard statics package such as R (Ihaka & Gentleman 1996; Cribari-Neto & Zarkos 1999; Venables & Ripley 1999) supply these tests and can be used when employing statistical testing for a experimentation.

When deciding to employ statistical testing to validate conclusions, a proper design of the experiment in advance is crucial. Experimental design is concerned with proper planning of experiments involving statistical evaluation. The goal is to draw valid and objective conclusions with minimal effort. As such, the main issues of experimental design are planning of experiments to collect appropriate data and to properly analyze these data by statistical methods (Cohen 1995; Mason, Gunst, & Hess 1989; Montgomery 1991; Dean & Voss 1999). There are several methods that describe how to properly design efficient experiments such as the Taguchi method (Roy 1990; Tsui 1992). These methods can be viewed as kind of template for conducting experiments as referred to previously. A language for specifying experiments could provide a standardized means to reusably describe such methods.

Planning Problems and Planning Systems

By now, we presented general properties of testbeds that are common for a variety of application domains. To demonstrate these ideas, we are currently working on a testbed for a specific problem domain: planning. At the time of writing, the work is about to be completed.

In planning the task is to execute actions in a world to bring it in a desired state. A world, also called domain, is defined by the set of its properties and the set of executable actions. A planning problem is a domain together with an initial world state and a set of goal states. Its solution is a sequence of actions that change the initial state into one of the goal states.

Problem descriptions commonly specify only the bare minimum of knowledge: The variable features of the domain and the available operators. In the problem domains most often addressed, however, there tends to be a rich structure "hidden" in the domain description. A planning system, now, consists of several algorithms that address different aspects of this structure. The output of one such algorithm is the input of the next and the flow of information from the initial problem to the solution can be quite complex.

To allow the modularization of planning systems, we use the Domain Knowledge Exchange Language DKEL (Scholz & Haslum 2000). It is designed to be part of domain and problem definitions in PDDL (Ghallab *et al.* 1998; Bacchus 2000), a widespread language to state planning problems. DKEL allows to decouple the extraction of domain knowledge from its use by augmenting the original domain or problem description with domain knowledge, rather than altering or reducing it right away. This way, the effect of a single module can be subject of investigation.

DKEL consists of five "general" knowledge forms which are modest generalizations of the forms of knowledge produced and used by domain analyzers and planners in existence today. The following is an example of a knowledge item for the three-operator blocksworld:

```
(:replaceable
   :optimal
     (:parallel-length :nb-operators)
   :vars (?x ?y ?z)
   :replaced ((move-from-table ?x ?y)
              (move ?x ?y ?z))
   :replacing
     (:empty (move-from-table ?x ?z)))
```

It states that it is always possible to replace the sequence of moving a block from the table onto a block and immediately onto another block by moving it directly onto the second location, at the time step of the second move. This replacement does not increase the number of operators nor does it increase plan length for parallel plans.

Prototype of a Testbed for Planning

The prototype testbed is concerned with planning algorithms composed from a number of planning modules in specific order. The testbed consists of a central control unit (CCU), a user interface (UI), one relational database for storing all relevant data in several tables, and a statistical unit. The CCU is directly responsible for handling any modules by starting the modules in proper sequence and configuration, collecting the results from the runs, and database storage of the results. Currently, all storage in the database is persistently. The raw algorithm output is stored in a table of the database, too, and accessed directly by the CCU to feed the statistical unit, in our case the R package[2] (Ihaka & Gentleman 1996; Cribari-Neto & Zarkos 1999; Venables & Ripley 1999). The results are displayed by a web user interface. As problem generators we use the generators of the FF domain collection.[3] All generated instances are stored in a table.

Any information concerning parameters of modules such as parameter flags and range of parameter values of a planning module is stored in a table. This table acts as a description of the standardized command interface as required by the testbed. New modules can easily be incorporated by adding a database entry for the new algorithm or module and placing the executable into the file system – the only prerequisite is that an algorithm provides a standardized command interface in the style of Unix. If necessary a wrapper, which can be regarded as kind of interface, has to be written for a module to adapt to the previous requirements. The testbed executes the modules via a system call. It calls the modules in proper sequence and directs the in- and outputs of the various modules via files in pipelining fashion, starting with the input from the table containing the instances, ending with the table containing the results of a run.The user interface currently accesses the database directly to read in the available modules and their parameters. So far, the prototype does not recover from partial crashes of any module.

Performing experiments is divided into three independent parts:

Instance generation The user can choose a domain and generate new problem instances by specifying several parameters defining the instance such as the size. Any instance generated that way is stored in a table for future use.

Specification and Execution The user can specify an algorithm and a configuration to run on a set of problem instances. The results and specifications of each distinct run are stored in a table and are identified by the name for the algorithm in its actual configuration, the problem instance identification and a timestamp. Algorithms can be specified by selecting a number and sequence of modules and configuring parameter values for each modules. Each

[2]www.r-project.org

[3]www.informatik.uni-freiburg.de/~hoffmann/ff-domains.html

algorithm and configuration specification created this way is named and stored in a table. Instead of specifying a new algorithm and configuration, the user can choose among the stored algorithm-configuration pairs. Problem instances can be selected from all stored instances generated in the past.

Statistical evaluation Applying statistical tests is performed by first choosing two or more named algorithm-configuration pairs. Next, a set of run results on a number of possibly common instances are picked for each pair. Then, a statistical test has to be choosen which will compare the selected algorithm-configuration pairs on the selected results. Depending on whether the user requests a test with repeated measures, i. e., tests on the same set of instances for all tested algorithm-configuration pairs, or without repeated measures, the tests are performed on the intersection of the sets of runs with respect to the same instances, or not, respectively. If the user selected more than two algorithm-configuration pairs for testing with a t-test, all possible pairwise combinations are tested. The results of the tests are displayed by the UI.

The planning modules under construction can be divided in three classes: Preprocessing techniques, which exhibit knowledge about the planning problem prior to the search for a plan, planners, and a module to connect and manipulate the input and output of the other modules in various ways. The considered planners are a reimplementation of GRT (Refanidis & Vlahavas 2001) and the planner FF (Hoffmann & Nebel 2001), adapted for use with the testbed.

We are implementing three preprocessing modules for our testbed: RIFO (Nebel, Dimopoulos, & Koehler 1997), TIM (Fox & Long 1998), and a technique to order goals (Koehler & Hoffmann 2000). RIFO eliminates irrelevant operators and TIM infers types and invariants. These techniques enrich the planning problem with knowledge which can be used by subsequent stages of the planning problem. The technique to order the goals of a planning problem is somehow different. It gives information how to divide the planning problem in to smaller ones and how to combine the partial solutions to a solution of the initial problem. This information is independent of the planning algorithm used. To account for this independence, we provide a wrapper for planners that uses the goal-ordering knowledge in a way transparent for the planner.

The use of a common language for input and output does not suffice in combining planning modules to a planning system. Each module has specific requirements that have to be met by its input, and changing these requirements can amount to a redesign of its algorithm. To account for these requirements, we provide a module, called DeDKEL, to manipulate planning problems in various ways: It allows to eliminate DKEL statements by either encoding it in the problem description or by just removing them. Furthermore, it can reduce the complexity of a problem description by eliminating specific features of PDDL, like negation or quantification.

DeDKEL has additional features that are useful for testing planing systems: concealment and randomization. In order to conceal the structure of a planning problem, e. g. for competitions, it can be helpful to replace identifiers by meaningless strings. This has been done for the mystery domain, a concealed logistics domain, which was used for the first AIPS planning competition. Randomization can be useful to test whether the performance of a planning system depends on the order of interchangeable features of the problem description.

Conclusion

In this paper, we discussed properties of an ideal environment to conduct and evaluate computational experiments. Some of these ideas are about to be implemented in a testbed for planning systems with a stress on reusability, extensibility and applicability to a wide range of problem domains.

This is not the first attempt to build such an environment: Anyone who has conducted a number of experiments on a collection of test sets will have felt the urge to automate this task. Likewise, the need for reusability of planning and preprocessing techniques has long been noticed. For example the TIM API[4] allows to easily include TIM into a planning system.

For the future, we plan to iteratively extent the testbed to meet further needs. The general direction was already outlined during this paper. In this process, the experience gained with the prototype will yield further insight in which features are necessary, which degrees of abstraction and generalization of experimentation processes are useful, and how much is necessary. This, hopefully, will also furnish us with more insight about needed features for an experimentation specification language.

This paper is inspired by our work in the meta-heuristics network,[5] which investigates heuristic search algorithms that solve combinatorial problems. This network has gained expertise in experimentation and statistical evaluation and feels the need for a tool that resembles the properties amplified within this paper. We hope that our prototype in the context of planning helps us on the way to a general purpose testbed for computational experiments.

References

Bacchus, F. 2000. Subset of PDDL for the AIPS 2000 planning competition. http://www.cs.toronto.edu/~aips2000/pddl-subset.ps.

Birattari, M.; Stützle, T.; Paquete, L.; and Varrentrapp, K. 2002. A Racing Algorithm for Configuring Metaheuristics. Technical Report AIDA-02-01, Darmstadt University of Technology.

Cohen, P. R. 1995. *Empirical Methods for Artificial Intelligence*. Cambridge, Massachusetts: The MIT Press.

Coy, S. P.; Golden, B. L.; Runger, G.; and Wasil, E. A. 2000. Using Experimental Design to Find Effective Paramterer Settings for Heuristics. *Journal of Heuristics* 7:77–97.

[4]www.dur.ac.uk/~dcs0www/research/stanstuff/TIMAPI/int.html

[5]www.metaheuristics.org

Cribari-Neto, F., and Zarkos, S. G. 1999. R: Yet another Econometric Programming Environment. *Journal of Applied Econometrics* 14:319–329.

Dean, A., and Voss, D. 1999. *Design and Analysis of Experiments*. New York, NY: Springer Verlag.

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

Garner, S. 1995. Weka: The Waikato Environment for Knowledge Analysis. In *Proceedings of the New Zealand Computer Science Research Students Conference*, 57–64.

Gent, I. P., and Walsh, T. 1994. How Not To Do It. Technical Report 714, University of Leeds.

Ghallab, M.; Howe, A.; Knoblock, C. A.; McDermott, D.; Ram, A.; Veloso, M.; Weld, D.; and Wikins, D. 1998. PDDL - the planning domain definition language. Technical Report CVC TR-98-003/DCS TR-1165, Yale Center for Computational Vision and Control.

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

Hooker, J. 1994. Needed: An Empirical Science of Algorithms. *Operations Research* 42(2):201–212.

Hooker, J. 1996. Testing Heuristics: We have it all wrong. *Journal of Heuristics* 1:33–42.

Ihaka, R., and Gentleman, R. 1996. R: A language for data analysis and graphics. *Journal of Computational and Graphical Statistics* 5(3):299–314.

Koehler, J., and Hoffmann, J. 2000. On reasonable and forced goal orderings and their use in an agenda-driven planning algorithm. *Journal of Artificial Intelligence Research* 12:338–386.

Larson, H. 1982. *Introduction to Probability Theory and Statistical Inference*. New York, NY: John Wiley & Sons, Inc.

Lehmann, E. 1997. *Testing Statistical Hypothesis*. Springer Verlag.

Mason, R.; Gunst, R.; and Hess, J. 1989. *Statistical Design and Analysis of Experiments*. New York, NY: John Wiley & Sons, Inc.

McGeoch, C. C. 1996. Towards an Experimental Method for Algorithm Simulation. *INFORMS Journal of Computing* 8:1–15.

McGeoch, C. C. 2001. Experimental Analysis of Algorithms. In Pardalos, P., and Romeijn, E., eds., *Handbook of Global Optimization, Volume 2: Heurstic Approaches*. Kluwer Academic.

Montgomery, D. 1991. *Design and Analysis of Experiments*. New York, NY: John Wiley & Sons, Inc., 3rd edition.

Moret, B. 2002. Towards a Discipline of Experimental Algorithmics. In *DIMACS Monograph in Discrete Mathematics and Theoretical Computer Science*. Forthcoming. AMS Press.

Nebel, B.; Dimopoulos, Y.; and Koehler, J. 1997. Ignoring irrelevant facts and operators in plan generation. In *Proceedings of the European Conference on Planning*, 338–350.

Rardin, R., and Uzsoy, R. 2001. Experimental Evaluation of Heuristic Optimization Algorithms: A Tutorial. *Journal of Heuristics* 7:262–304.

Refanidis, I., and Vlahavas, I. 2001. The GRT planning system: Backward heuristic construction in forward state-space planning. *Journal of Artificial Intelligence Research* 15:115–161.

Roy, R. 1990. *A Primer on the Taguchi Method*. Van Nostrand Reinhold.

Scholz, U., and Haslum, P. 2000. Enable your planner! Decoupling domain analysis and planning. Technical Report AIDA-00-05, Darmstadt University of Technology.

Siegel, S. 1956. *Nonparametric Statistics for the Behavioral Sciences*. McGraw-Hill.

Tsui, K. L. 1992. An Overview of Taguchi Method and Newly Developed Statistical Methods for Robust Design. *IIE Transactions* 24(5):44 – 57.

Venables, W. N., and Ripley, B. D. 1999. *Modern Applied Statistics with S-Plus*. Springer, 3rd edition. ISBN 0-387-98825-4.

Witten, I., and Frank, E. 2000. *Data mining: Practical Machine Learning Tools and Techniques with Java Implementations*. San Francisco: Morgan Kaufmann.

Xu, J.; Chiu, S.; and Glover, F. 1998. Fine-Tuning a Tabu Search Algorithm with Statistical Tests. *International Transactions in Operational Research* 5(3):233 – 244.