Proceedings of the 21th Workshop

Planen, Scheduling und Konfigurieren, Entwerfen (PuK)

Stefan Edelkamp, Jürgen Sauer, Bernd Schattenberg (Editors)

Workshop at

Künstliche Intelligenz 2007

Annual Conference

Osnabrück, Germany

September 10th, 2007

Vorwort

Auch bei der 21. Auflage des PuK-Workshops haben wir wieder den Versuch unternommen, die vor allem in der Fachgruppe Planen/ Scheduling und Konfigurieren/ Entwerfen im FB „Künstliche Intelligenz“ der GI vereinten Forscher und Praktiker zu einem Austausch aktueller Forschungsergebnisse zusammen zu bringen. Wie immer sollen im PUK-Workshop neue Fragestellungen, Lösungskonzepte und realisierte Systeme vorgestellt werden, um die Anwendung intelligenter Technologien in den Themenbereichen der Fachgruppe weiter zu treiben.
In diesen Workshop-Proceedings finden sich daher Papiere aus fast allen betrachteten Bereichen (Planen, Konfigurieren, Scheduling). Sie zeigen einen Querschnitt aus den aktuellen Forschungsgebieten der aktiven Forschungsgruppen in Deutschland.

Organisation

PD Dr. Stefan Edelkamp
Tel. ++49-231-755-5809

stefan.edelkamp@cs.uni-dortmund.de
ls5-www.cs.uni-dortmund.de/~edelkamp

Universität Dortmund
Lehrstuhl V
Fachbereich Informatik
Otto-Hahn-Strasse 14
44227 Dortmund

Apl. Prof. Dr. Jürgen Sauer
Tel. ++49-441-798-4488

juergen.sauer@uni-oldenburg.de
www.wi-ol.de

Universität Oldenburg
Fakultät II, Department
für Informatik
Uhlhornsweg 84
26129 Oldenburg

Dipl.-Inform. Bernd Schattenberg
Tel. ++49-731 50 24259

bernd.schattenberg@uni-ulm.de

Universität Ulm
Institut für
Künstliche Intelligenz
89069 Ulm

Inhalt

Evolution of Configuration Models – a Focus on Consistency

Thorsten Krebs

HITeC e.V. c/o University of Hamburg
Vogt-Kölln-Str. 30, 22527 Hamburg, Germany
krebs@informaik.uni-hamburg.de

Abstract. This paper describes a novel approach that guarantees consistency of structure-based configuration models in an evolution process using a knowledge representation based on description logics. We define the term consistency and discuss different notions of consistency that are domain-dependent or configuration-tool specific. Change operations are introduced as a formal means to capture changes to a configuration model. A set of invariants formally defines what consistency denotes for structure-based configuration and enables an adaptive approach to handle configuration models that rely on different notions of consistency. A model editor implementing this evolution process allows to execute change operations, identifies inconsistency in a configuration model after change execution and suggests repair operations for repairing inconsistent configuration models.

1 Introduction

Modern product manufacturers face the problem of managing a variety of products that change over time. A *product catalog* contains different kinds of products, each of them offering alternative and optional choices to the customer. Such a variety of products is typically needed to satisfy customers with different demands. In order to stay competitive, product manufacturers need to diversify the product variety and also need to improve existing product types over time according to customer demands.

To avoid time-consuming redesign and manual adaptation, complex products are assembled from a set of smaller components. These components are subject to parameterization, i.e. the individual adjustment of their properties according to the context in which they are used. Varying the choice of parts or the structure of assembly leads to high flexibility and adaptability. This helps decreasing the effort of production and increasing the performance and functionality of products [1]. The application of configuration tools is a trend to reach this goal.

1.1 Configuration

Configuration defines the task of building one or more configuration solutions that satisfy a given set of requirements for a desired product from a fixed, predefined set of components [2, 1]. A component is described by a set of properties and ports for connecting it to other components. Constraints describe which kinds of combinations are admissible. A configuration solution contains a description of the components and the connections between them [2]. One of the major benefits the configuration approach offers is that a solution to the given configuration problem is guaranteed to be consistent and complete.

Configuration tools have been used successfully for about three decades. Traditional application areas of configuration are technical or electronic domains, but the approach is not limited to these domains [3]. Recently, configuration was also applied in software domains [4, 5].

Hierarchical configuration approaches employ hierarchical specialization structures and composition structures together in AND/OR graphs [6]. Concepts and relations between concepts are the nodes and edges of such a graph, respectively. Specialization relations are represented through disjunctive relations (the specializations are mutually exclusive) while composition relations are represented through conjunctive relations (all parts of a composite can coexist). A configuration solution is an AND/OR graph containing exactly one of the OR descendants (specializations) and some of the AND descendants (parts), according to cardinality definitions.

Structure-based configuration employs hierarchical modeling facilities to specify the underlying domain knowledge. This kind of knowledge representation is especially appropriate to model a product catalog, because the structure of a set of different but similar products in one configuration model is a typical example for using hierarchical AND/OR graphs. Within such a structure-based configuration model, all potentially configurable products are implicitly represented by describing the components a product can be composed of, component attributes and relations between the components.

1.2 Evolution Framework

The effectiveness of configuration applications depends on the quality of the underlying knowledge, i.e. the configuration model. It is a fundamental tenet that the model unambiguously represents knowledge about components from which products can be assembled. This means that when components of a product domain change, a configuration model representing this domain needs to be changed, too. Changing components have direct influence on the product catalog: certain products may no longer be assembled or new products become possible, existing products may be offered with richer or poorer variability.

Example. A car manufacturer offers a variety of models from each of which a customer can choose between certain alternatives and options. Alternative selections are, among others, the color of the Car and its Seats, the type of Motor (Gasoline or DieselEngine) and its power, whether to have a ManualGearShift or AutomaticTransmission, and so on. Optional selections are, for example, the possibility to have a Sunroof, a Turbocharger, a Hitch, etc. The development of new Motors that consume less fuel, a new design or electronic applications like a ParkingAssistanceSystem, drive the interest of customers in buying a new Car, just to name a few examples. Such developments are not specific to a type of car. Instead, making new developments integratable into all car types is the key to improve return on investment.

It is apparent that sophisticated tool support is needed for managing knowledge about products, components from which products are assembled, dependencies between these components, as well as changes to components and impacts that changes to components have on products. As a first step to reach this goal, in this paper we describe an evolution process that preserves consistency of a configuration model despite its changes. A model editor that implements this evolution process can guarantee to produce only consistent configuration models.

A complete framework describing knowledge management for evolution of configurable products is described in [7]. In this framework, the set of configurable components is represented in a configuration model and the product types are represented by *product models* that refer to concept definitions of the corresponding configuration model. Both the configuration model and product models are defined using modeling facilities from structure-based configuration. A product model in this sense is a subset of the configuration model and a product catalog is a set of product models that refer to the same configuration model. Product models are persistent copies of the corresponding subset of the configuration model. Persistent copies are needed because implicit knowledge about products were lost when evolving the configuration model. Mismatches between product models and the configuration model are identified and give valuable input for managing the product catalog.

In this paper we focus on the notion of consistency and how consistency of configuration models can be preserved despite executed changes.

1.3 Reader's Guide

The remainder of this paper is organized as follows. Section 2 gives a formal definition of the modeling facilities that are used to build configuration models. Section 3 defines the notion of consistency that is used throughout this paper and what constitutes a consistent configuration model. Section 4 gives a formal definition of change operations as a means to capture changes to a configuration model and describes the process of change execution. Section 5 discusses related work and after that Section 6 summarizes and concludes this paper.

2 Knowledge Representation

The main component of a configuration tool is the configuration model. The configuration model consists of both domain knowledge and configuration logic. Domain knowledge is a representation of configurable components and configuration logic describes restrictions on how the components can be combined [8]. Modeling facilities used for structure-based configuration models have been developed from *frames* [9], *semantic networks* [10] and *concept languages* like the KL-ONE system [11].

A configuration model uniquely identifies the components of a domain, their properties and structure. The structure can be represented by a graph whose nodes and edges correspond to concepts and relationships, respectively [12]. A one-to-one correspondence between instances of concept definitions and corresponding real-world objects defines the *declarative semantics* of the configuration model [13]. The structure of the model can be seen as a homomorphic map of the domain structure [14].

2.1 Modeling Facilities

In the following we use a knowledge representation that is based on description logics to formally define all modeling facilities that can be used to represent domain knowledge.[1] The modeling facilities together with the possibilities to form combinations of modeling facilities constitute the specification of the underlying modeling language for defining configuration models.

A *concept* is a description which gathers common features of a set of components. Concepts are interpreted as sets, which means that concept conjunction can be interpreted as set intersection, concept disjunction as set union and negation of concepts as set complement [16]. Concepts are modelled containing two different hierarchical relationships: *is-a* and *has-parts*. The taxonomic *is-a* relation and the partonomic *has-parts* relation are processed differently. The former is concerned with commonalities and differences of the concept definitions while the latter involves spatio-temporal and functional correlation. Every concept carries a unique *name* which identifies it within the domain, specifies exactly one parent concept of which it is a specialization, and an arbitrary number of *attributes* and *composition relations* collectively called *properties*. Concepts are denoted with upper case names, e.g. C, D or Car.

Concepts describe classes of objects from which multiple concept *instances* may be generated during the configuration process. Instances are instance of exactly one concept (e.g. $i \in C$) and inherit all properties from this concept definition. Property values may be partly specified and are only allowed to specify subsets of the original values defined in the concepts of which they are an instance.

The taxonomic hierarchy is defined by $D \subseteq C$. Concept C is called *parent* and concept D is called *child*. The taxonomic relation is *transitive*: concept D is a *direct subconcept* of concept C and an *indirect subconcept* of the parent of C, and so on. A concept D is a subconcept of concept C if and only if every potential instance of D is also an instance of C ($i \in D \Rightarrow i \in C$). Within the taxonomy properties of concepts are monotonically inherited. Additional properties may be defined for more specific concepts and inherited properties may be *overwritten* with more specific values.

Attributes define characteristics of concepts and are denoted as roles with lower case names and a concrete domain as role filler, e.g. $a.\mathsf{Integer}$, $b.\mathsf{String}$ or $\mathsf{color.String}$. The name is uniquely identifiable within the taxonomy. Three

[1] For an overview of description logics we refer the interested read to [15]

value domains are pre-defined for specifying attribute values: integer numbers, real numbers and strings. In implementations, however, the potential values of concrete domains are dictated by the programming language.

Composition relations define the partition of complex concepts into simpler concepts and are denoted as roles with lower case names and concepts as role fillers, e.g. $r.C$, $p.D$ or $\mathsf{hasParts.Motor}$. A composition relation between a *composite* Car and a *part* Motor is denoted by $\mathsf{Car} \Rightarrow \exists\mathsf{hasParts.Motor}$. Composition relations may be assigned a cardinality definition denoting how many instances of the part concept can be instantiated, i.e. a minimum $\exists^{\geq m}\mathsf{hasParts.Motor}$ and a maximum $\exists^{\leq n}\mathsf{hasParts.Motor}$ value $(m \leq n)$; for $n = m$ one can write $\exists^{=n}\mathsf{hasParts.Motor}$. The default cardinality, if nothing else is specified, is ≥ 0.

Interdependencies and restrictions between concepts and their properties are expressed with *constraints*. Constraints represent non-hierarchical dependencies between concepts and concept properties, as well as between the existence of instances of certain concept definitions. A constraint definition consists of an antecedent and a consequent. The antecedent, or *conceptual constraint*, specifies a *pattern* consisting of a conceptual structure, an expression that evaluates to true whenever the pattern matches a corresponding instance structure. The consequent, i.e. a set of *constraint relations*, is executed when the pattern evaluates to true.

For reasons of simplicity we confine ourselves to binary relations that restrict attribute values in the following. A constraint that enforces equal integer values for two attributes a and b of a concept C, for example, is denoted with $C \cap \forall a.\mathsf{Integer} \cap \forall b.\mathsf{Integer} \Rightarrow a = b$.

2.2 Building a Configuration Model

Complex concepts are created by using *conjunction*, i.e. set intersection. For example, specifying that a concept D is a child of a concept C with a string attribute a is achieved by forming their intersection: $C \Rightarrow D \cap \forall a.\mathsf{String}$. *Disjunction*, in contrast, creates set union. For example, the union of all sibling concepts creates a set equivalent to their parent concept. Conjunction and disjunction are both *commutative*: a permutation of the subexpressions does not change the meaning of the compound expression. Concept expressions can be assigned a name by using the := operator, e.g. $\mathsf{Tire} := \mathsf{CarPart} \cap \forall\mathsf{size.Integer}$.

Another operator for creating compound expressions is *negation*. The underlying idea is that a representation is either an adequate description of a situation, or it is not (in the latter case its negation is true). There are no intermediate cases where a representation would be partially correct. However, absence of objects is not explicitly modeled, because configuration models rely on the *closed world assumption* (*CWA*) [17].

Example. In the following we specify an excerpt of concepts from the Car domain. A Car defines the attributes model and color and composition relations with the parts Motor and Tire. A Motor is a CarPart that defines the attributes type, power and fuelConsumption. A Tire also is a CarPart and defines the attributes size and width.

$$\begin{aligned}
\mathsf{Car} &:= \forall \mathsf{model.String} \sqcap \forall \mathsf{color.String} \sqcap \exists^{=1} \mathsf{hasParts.Motor} \sqcap \\
&\quad \exists^{=4} \mathsf{hasParts.Tire} \\
\mathsf{Motor} &:= \mathsf{CarPart} \sqcap \forall \mathsf{type.String} \sqcap \forall \mathsf{power.Integer} \sqcap \\
&\quad \forall \mathsf{fuelConsumption.Integer} \\
\mathsf{Tire} &:= \mathsf{CarPart} \sqcap \forall \mathsf{size.Integer} \sqcap \forall \mathsf{width.Integer}
\end{aligned}$$

2.3 Reasoning

The contents of a configuration model describe premises that are true and are referred to as *predicates*. A *proposition*, in addition, affirms or denies a predicate and is either true or false. Hence, propositions can be seen as interpretations of predicates. There is a subtle difference: let m be the model of concern, then the predicate C states that $m \vDash C$. There is no interpretable freedom in this predicate. A proposition $\exists C$, however, can be affirmed or denied: if, and only if, $m \vDash C$, the proposition is valid.

Removing concept C from m has the immediate effect that $m \nvDash C$. The proposition $\exists C$ is denied and the proposition $\nexists C$ is affirmed. Knowledge about absent objects is implicit but can be deduced from explicit knowledge. To affirm $\nexists C$, all concept definitions are traversed and it is made sure that C is not among them. We will see later (in Section 4) that propositions are used to describe preconditions and postconditions of change operations.

Example. Although we use an explicit knowledge representation, a configuration model may deduce implicit knowledge. When m contains the concept definitions Motor, $\mathsf{Car} \Rightarrow \exists^{=1}\mathsf{hasParts.Motor}$ and $\mathsf{Mercedes} \Rightarrow \mathsf{Car}$ we know that not only every Car has a Motor, but also every $\mathsf{Mercedes}$ does: $m \vDash \mathsf{Mercedes} \Rightarrow \exists^{=1}\mathsf{hasParts.Motor}$.

3 Consistency

In mathematical logic, a formal system is consistent when none of the facts deducible contradict one another. For knowledge representation systems a concept is consistent when it admits at least one instance. In this sense consistency has nothing to with reality but is rather concerned with the representation of the domain. In the following we concisely define what consistency denotes for configuration models.

3.1 Definition

Coherence is often postulated for consistency. A pervasive tenet of coherence is that truth is primarily a property of whole systems, not of single predicates. This means that consistency is only a significant characteristic of a configuration model when its predicates are coherent, i.e. somehow interrelated. Configuration models are based on ontological conceptualization representing a product domain and we can assume that components of the domain are coherent.

Well-formedness of a configuration model is asserted when it is well-fromed with respect to the underlying language specification. This means that the definition of modelling facilities and their potential interrelations specify what a well-fromed configuration model is.

Consistency, in addition, goes beyond the language specification by semantically interpreting the specified knowledge. A well-formed configuration model may still be inconsistent, which happens, for example, when a constraint relation restricts two attributes in a way that cannot be fulfilled with the specified attributes values.

Example. When the size of Rims is constrained to equal the size of Tires, but the Rim and Tire concepts do not specify overlapping values for their size attributes, the constraint can not be satisfied for any combination of instances of the two concepts.

But the notion of consistency may vary based on the represented domain or the configuration tool. An example for differently understood notions of consistency is the semantics of composition relations, which varies according to three main criteria: whether the relation from the part to the whole is functional or not, whether the parts are homeomerous or not, and whether the part and whole are separable or not. [18] distinguish six major types: (1) component - integral object, (2) member - collection, (3) portion - mass, (4) stuff - object, (5) feature - activity, and (6) place - area. Although product configuration is mainly concerned with type 1, different domains or configuration tools treat composition relations sometimes reflexive or irreflexive, symmetric or antisymmetric and sometimes allow that parts are shared by different composites or not.

Hence, for being able to evolve configuration models independent of the represented domain or the configuration tool that interprets the model, an approach that can adapt to the current semantics of the underlying language specification is needed to preserve consistency of the configuration model when executing changes. Such an approach is described in the following.

3.2 Invariants

Invariants are formulae that have a particular status: they must be guaranteed to hold at every quiescent state of the model, that is before and after a change is executed [19]. Thus, invariants ensure that changes do not introduce inconsistency. If changes follow these invariants, it can be guaranteed that the configuration model remains consistent.

All represented objects need to be unambiguously defined. For the named modeling facilities concept, attribute and conceptual constraint this means that the same name denotes the same definition and no other definition may use that name. For composition relations the name is not the distinctive criteria but its part concept and cardinality.

Concept Unambiguity Invariant All concepts names are unambiguous.

$$\forall C, D \Rightarrow C \neq D$$

Attribute Unambiguity Invariant All attributes names of a concept are unambiguous.

$$\forall C \cap a \cap b \Rightarrow a \neq b$$

Composition Unambiguity Invariant All composition relations of a composite have an unambiguous definition of part concept and cardinality.

$$\forall C \cap \exists^{\geq m} r.D \cap \exists^{\leq n} r.D \cap \exists^{\geq k} p.E \cap \exists^{\leq l} p.E$$
$$\Rightarrow D \neq E \wedge E \subseteq D \Rightarrow m \geq k \wedge n \leq l$$

Constraint Unambiguity Invariant All constraints names are unambiguous.

$$\forall \gamma_1, \gamma_2 \Rightarrow \gamma_1 \neq \gamma_2$$

The taxonomy is a tree. This means that every concept has exactly one parent concept. No concept has more than one parent and no child may be orphaned.

Taxonomy Tree Invariant A concept has exactly one superconcept (if it is not the root concept).

$$\forall D \Rightarrow \exists C \text{ such that } D \subseteq C \wedge |\text{parent}(D)| = 1$$

All represented objects that are referenced by another object are identified by their names. The corresponding definitions with these names need to be defined.

Composition Reference Invariant The part concept referenced in a composition relation has to be defined.

$$\forall C \cap \exists r.P \Rightarrow \exists P$$

Constraint Reference Invariant All concepts defined for the pattern of a constraint need to be defined.

$$\forall \gamma, \forall C \subseteq \text{pattern}(\gamma) \Rightarrow \exists C$$

The cardinality definition of a composition relation needs to specify a correct integer interval.

Composition Cardinality Invariant The minimum cardinality needs to be less than or equal to the maximum cardinality.

$$\exists^{\geq m} r.P \cap \exists^{\leq n} r.P \Rightarrow m \leq n$$

Properties are inherited along the taxonomic hierarchy and may be overwritten by more special concepts. When a property is overwritten, the new value has to specify a subset (or an equal value) of the original value. Vice versa, a more special concept may not overwrite a property with a value that does not specify a subset of the original value.

Attribute Inheritance Invariant An attribute overwriting an inherited attribute may only specify a subset (or equal) value of the original value.

$$\forall D \cap \exists b \subseteq C \cap \exists a \text{ such that } a = b \Rightarrow \text{value}(b) \subseteq \text{value}(a)$$
$$\wedge\ \forall C \cap \exists a, D \cap \exists b \text{ such that } a = b \wedge \text{value}(b) \nsubseteq \text{value}(a) \Rightarrow D \nsubseteq C$$

Composition Inheritance Invariant A composition relation overwriting an inherited composition relation may only specify a child of the original part and a subset of the original cardinality.

$$\begin{aligned}\forall D \cap \exists^{\geq k} p.F \cap \exists^{\leq l} p.F \subseteq C \cap \exists^{\geq m} r.E \cap \exists^{\leq n} r.E \\ \text{such that } F \subseteq E \Rightarrow k \geq m \vee l \leq n \\ \wedge\ \forall C \cap \exists^{\geq m} r.E \cap \exists^{\leq n} r.E, D \cap \exists^{\geq k} p.F \cap \exists^{\leq l} p.F \\ \text{such that } F \subseteq E \wedge (k \ngeq m \vee l \nleq n) \Rightarrow D \nsubseteq C\end{aligned}$$

For being able to adaptively handle different semantics of partonomy, additional invariants are needed for a clear definition. Hence the following invariants are addressed for consistency checking only when appropriate. The basic idea of the adaptive consistency-preserving approach is that these invariants can safely be ignored when not appropriate.

Partonomy Irreflexivity Invariant The partonomy is irreflexive. This means that no concept may be both composite and part for the same composition relation.

$$\forall C \cap \exists r.D \Rightarrow C \cap D = \emptyset$$

Partonomy Antisymmetry Invariant The partonomy is antisymmetric. This means that no concept is reachable via the transitive closure over its composition relations.

$$\forall C \cap \exists r.D \Rightarrow D \cap \exists r.C = \emptyset$$

Finally, the defined constraints need to be satisfiable. Note that constraints are propagated on instances, not on concepts. This means that all potential instances of the concepts that are resticted by a constraint need to be created and evaluated separately.

Constraint Satisfaction Invariant All constraints need to be satisfiable.

$$\forall \gamma, \forall C \subseteq \text{pattern}(\gamma), i \in C \Rightarrow \gamma \cap i \neq \emptyset$$

4 Evolution Process

The process of evolving a configuration model consists of three steps that are run through for every change. A user initiates the evolution process by choosing one of the change operations that are currently applicable. Only change operations for which all preconditions are satisfied are applicable (see below). Selecting a change also includes undo operations. Compound changes are composed of simpler changes, e.g. removing a subtree is composed of removing all the concepts individually. Based on the current state of the configuration model, dependent changes may be needed. For example, when removing a concept that is defined as a part for some other concept, the composition relation cannot persist and needs to be removed, too, or another filler needs to be specified. In case the intended change introduces inconsistency (as a side effect), repair operations are automatically identified and the user can select the most appropriate repair that is added to a compound change. Finally, the compiled change operations are sequentially executed.

Defining the compilation of compound operations is beyond the scope of this paper.[2] Instead, the following two Sections focus on the formal specification of change operations (Section 4.1) and the consistency-preserving approach that identifies which invariants need to be checked for which change operation and how repair operations are identified in case of inconsistency (Section 4.2).

4.1 Specifying Change Operations

For resolving changes to a configuration model, they have to be represented in a suitable format. This representation should capture semantics of a change rather than syntactical changes in some form of textual representation. Within configuration models the same knowledge can be specified with different syntactical means or simply in different order: two knowledge structures can be the same conceptually, but have very different text representations [20]. The textual representation can be implemented by automatic translators.

One of the most important advantages of using change operations that address the semantics of a change to the configuration model instead of a simple text editor is based on the possibility to preserve consistency of the configuration model within a tool-based evolution process. Developing configuration models with a text editor, consistency checking is only possible *after* one or more lines have been changed [21].

Change operations, like any kind of actions, are specified through preconditions and postconditions. The *preconditions* for a change operation encode what the model must be like in order for the change to be applicable. The *postconditions* describe immediate consequences resulting from the change.

Base operations represent elementary changes that cannot be further decomposed into smaller units. This means that a base operation describes an explicit action: the way this action reaches its intended goal does not vary depending on the knowledge specified in the configuration model. There are three types of meta changes: addition, removal and modification. While the first two are concerned with modeling facilities as such, the latter is concerned with properties of modeling facilities. The set of all base operations is the cross-product of modeling

[2] The interested reader is referred to [7] for a definition of the evolution framework that includes compiling compound operations.

facilities and the meta changes. Base operations concerning concepts, for example are addConcept(C), removeConcept(C) and renameConcept(c, “newName”). Addition and removal are defined analogously for all types of modeling facilities. Modifications include, next to renaming, also changing attribute values, composition parts and cardinalities, etc.

Example. The change operation addSubconcept($\mathsf{Motor}, \mathsf{DieselEngine}$) adds a concept $\mathsf{DieselEngine}$ that is a child of Motor. Preconditions of this operation are $\exists\mathsf{Motor}$ and $\nexists\mathsf{DieselEngine}$. Postcondition of this operation is $\exists\mathsf{DieselEngine}$.

Compound operations are compiled from a set of base operations. The way their intended goal is reached may vary based on the current state of the configuration model. Removing a concept C (removeConcept(C)), for example, is a simple operation when C is a leaf node in the taxonomy. When there is a descendant $D \subseteq C$, however, C cannot be simply removed (due to the Taxonomy Tree Invariant). There are two alternative solutions: D (including potential further siblings and children) is also removed (removeSubtree(C)), or D (including potential further siblings) is moved upwards, as a child of the parent of C (removeConceptAndMoveSubconceptsUp(C)).

The preconditions and postconditions of change operations are automatically accumulated for a compound operation, i.e. a compound operation contains all the preconditions and postconsitions its inherent base operations contain. The ability to construct compound operations from a pre-defined set of base operations offers a flexible and extendable way to define new change operations with minimal effort. This helps defining new types of (maybe domain-dependant or tool-specific) changes. Compound operations are treated as *transactions* [22]: either all or none of the operations are executed.

4.2 Preserving Consistency

Invariants are used to check consistency of the configuration model after executing change operations. Different aspects of the language specification are checked by different invariants and different change operations result in changes to different aspects of the language specification. Assuming that the configuration model was consistent before executing a change, not every invariant needs to be checked after change execution but only those invariants that are concerned with aspects of the language specification on which the corresponding change operation performs changes.

Postconditions of a change operation define the result of its execution and thus predict the resulting state of the configuration model. This means that the postconditions of a change operation indicate which invariants need to be checked after change execution.

Variables are used for reasoning about modeling facilities in preconditions and postconditions as well as in invariants. For example, $\exists C$ is a postcondition of adding a concept C to the configuration model (addConcept(C)). It is apparent that there may only be a dependency between two variables used in postcondition and invariant when the type of modeling facility, denoted by the variables match. The postcondition $\exists C$ indicates that all invariants with variables denoting concepts need to be concerned, for example the Concept Unambiguity Invariant or the Taxonomy Tree Invariant. Invariants that only use variables denoting different types of modeling facilities need not be checked. For example, when changing the value of an attribute, the Concept Unambiguity Invariant can safely be discarded.

However, not every match of two variables that denote the same type of modeling facility in both postcondition and invariant necessarily indicates potential inconsistency. For example, when removing a concept the Concept Unambiguity Invariant need not be checked because the the variable denoting concepts in this invariant is universally quantified and cannot be violated by removing a concept. The Taxonomy Tree Invariant, however, needs to be checked because this invariant also contains an existentially quantified variable denoting concepts and thus may be violated by removing a concept.

When specifying results of change operations as postconditions, variables may be existentially quantified (e.g. $\exists C$ or $\nexists C$). Additionally, a characteristic of some modeling facility may be assigned a specific value (e.g. name(C) = Car). Variables that appear in invariants may be bound in two ways: universally quantified (e.g. $\forall C$) or existentially quantified (e.g. $\exists C$ or $\nexists C$). Additionally, a characteristic of some modeling facility may be assigned a specific value (e.g. name(C) = Car). Note that assigning a specific value to a characteristic of a variable, an existing instance of the corresponding modeling facility needs to be ensured, by universal of existential quantification (e.g. $\exists C \cap \text{name}(C) = \mathsf{Car}$).

Postcondition	**Potentially Violated Invariant**
$\exists C$	$\forall C$, $\nexists C$
$\nexists C$	$\exists C$
$\exists C$ and property(C) = value	$\forall C$, $\exists C$ or $\nexists C$ and property(C) = value

Table 1. Postconditions of change operations indicate which invariants need to be checked after change execution.

Table 1 shows which invariants need to be checked after executing changes with specific postconditions. The left column of the table lists the different ways in which a variable may appear in a postcondition to specify the result of a change operation. The right column of the table lists the ways in which a variable denoting a modeling facility of the same type may appear in an invariant. A row specifies a combination of variables such that the corresponding invariant needs to be checked.

Example. Configuration model m defines $\mathsf{TurboDiesel} \subseteq \mathsf{DieselEngine}$ and $\mathsf{Car} \Rightarrow \exists^{=1}\mathsf{hasParts.DieselEngine}$. Now let us assume the car vendor no longer produces diesel engines. We need to remove the $\mathsf{DieselEngine}$ concept and all its descendants (removeSubtree($\mathsf{DieselEngine}$)). This change operation consists of the base operations removeConcept($\mathsf{DieselEngine}$) and removeConcept($\mathsf{TurboDiesel}$), and has the preconditions $\exists\mathsf{DieselEngine}$ and $\exists\mathsf{TurboDiesel}$, and the postconditions $\nexists$DieselEngine and $\nexists$TurboDiesel. The following invariants need to be checked because they contain an existentially quantified concept variable.

– Composition Reference Invariant: the DieselEngine concept is missing for the definition of the Car concept. A *Part Concept is Missing Violation* is identified.
– Constraint Reference Invariant: no violation is identified because no constraint definition restricts the DieselEngine concept.
– Taxonomy Tree Invariant: no violation is identified because the complete subtree of DieselEngine was removed and no orphaned concepts remain.

Basically, an inconsistency is identified when two affirmed propositions express facts that are logically inconsistent. There are exactly two alternatives to repair an inconsistency: replace either of the two inconsistent propositions with a proposition expressing the exact opposite.

Example. A *Part Concept is Missing Violation* is identified between $\exists\mathsf{Car} \Rightarrow \exists^{=1}\mathsf{hasParts.DieselEngine}$ and $\nexists$DieselEngine. The first repair alternative is removing the composition relation with the missing part concept. The second repair alternative is adding a concept definition with the name DieselEngine. For the given example it is obvious that adding a concept with the name DieselEngine is not reasonable due to removing a concept with this name caused the inconsistency. However, simply removing the composition relation $\exists^{=1}\mathsf{hasParts.DieselEngine}$ from the definition of the Car concept also may not be reasonable. The user has to decide whether the Car concept is worth existing without the DieselEngine: when it is, removing the composition relation repairs the inconsistency, when it is not, additional repair alternatives are removing the definition of Car or finding a substitute for the DieselEngine (maybe its parent, the general Motor).

It it apparent that an inconsistency where an object is missing can be repaired by adding this object, e.g. $\nexists C$ can be repaired by addConcept(C). Vice versa, an inconsistency where an existing object should not exist can be repaired by removing this object, e.g. $\exists C$ can be repaired by removeConcept(C). Inconsistencies where characteristics of an object are assigned invalid values can be repaired by changing that value, this includes renaming concepts or properties of concepts, changing attribute values, and so on. While some changes to object characteristics like renaming a concept cannot be automated with tool support, for example consistent attribute values can be computed and suggested.

Example. A Mercedes concept is defined as a child of Car that specifies, among others, an attribute $\mathsf{color.\{black; white\}}$. The Mercedes should be offered in silver, so the concept is defined as $\mathsf{Mercedes} \Rightarrow \forall\mathsf{color.silver}$. The Attribute Inheritance Invariant does not allow this attribute value since the color attribute of the child concept (i.e. Mercedes) does not specify a subset of that value of the parent concept (i.e. Car). But a subset of the value domain can easily be computed: either the Mercedes species color.black or color.white or both $\mathsf{color.\{black; white\}}$. Vice versa, the Car may accommodate silver as a potential candidate and specify $\mathsf{color.\{black; white; silver\}}$. For this example, obviously the latter choice is more appropriate.

5 Related Work

Configuration tools have been developed in the last decades. Two structure-based configuration tools employ a very similar representation of configuration models: KONWERK [23] and EngCon [24]. No model editor is publically available so that configuration models have to be created and maintained using text editors, which is error-prone, of course.

The consistency-preserving evolution process described in this paper is part of a framework for knowledge management for evolution of configurable products [7]. In this framework a configuration model represents the set of configurable components and different product types are represented by so-called product models that refer to concepts in this configuration model. Mismatches between product models and the configuration model are identified, explained to the user and serve as input for managing the product catalog.

Invariants are known from knowledge base and database communities, typically called *integrity constraints* [25]. To distinguish this domain-independent language restricting notion from the domain-dependent problem solving notion of constraints introduced in Section 2, we stick to the terms *invariant* and *constraint*, respectively in this paper. To the knowledge of the author, invariants were first introduced in [26] to ensure consistent evolution of database schemata. [27] and [28] also define invariants to preserve consistency in the course of ontology evolution. Ontology editors are readily available but only handle a portion of the knowledge needed for configuration.

6 Summary and Outlook

This paper describes an evolution process that preserves consistency of structure-based configuration models. In this process a user executes changes to a configuration model and tool support ensures the configuration model remains consistent despite its changes. A set of invariants defines what denotes a consistent configuration model. The necessary invariants are checked after change execution and preserve consistency of the configuration model. In case inconsistency is identified, the tool support suggests repair operations from which the user can select the most appropriate repair. With this help the repair process is partially automated.

An adaptive approach to selecting invariants based on the semantics of a configuration model allows to manage configuration models that represent different domains or will be used in different configuration tools. A prototypical *Language-independent Model Editor* (*LiMEd*) is currently developed that is able to handle configuration models from different product domains and different configuration tools. The model editor offers a pre-defined set of change operations from which a user can choose in a drop-down menu. Only applicable operations are offered. Invariants defined for executed changes are evaluated. Those invariants that are not part of the current language specification need not be checked and can safely be discarded. First results show that it is feasible to handle configuration models of reasonable size and from different sources. A test model containing knowledge about a fictious car domain and an EngCon [24] model containing knowledge about a car periphery supervision system [5], created in the ConIPF project (Configuration in Industrial Product Families)[3], were used for extensive testing. The car domain contains 52 concepts, 24 attributes, 33 composition relations and 9 constraints. The car periphery supervision domain contains 72 concepts, 107 attributes, 34 composition relations and 62 constraints. All consistency tests computed in less than a second. Scalability tests with hundreds or thousands of concepts still have to be done, but the first results look very promising.

Future work includes investigating whether the defined set of invariants sufficiently covers the intended notion(s) of consistency and identifying "intelligent" repair operations, i.e. not just adding or removing predicates but, for example, computing potential values for constrained attributes. This will be done by testing larger configuration models with thousands or tens of thousands of concepts and more complex constraints. Another topic of future work is to test configuration models from different product domains and configuration tools or importing standard formats like *Web Ontology Language* (*OWL*)[4] ontologies, for which the currently implemented modeling facilities and invariants may need to be extended.

References

1. Stumptner, M.: An overview of knowledge-based configuration. AI Communications **10**(2) (1997) 111–126
2. Mittal, S., Frayman, F.: Towards a generic model of configuration tasks. In: Proceedings of Eleventh International Joint Conference on AI. (1989) 1395–1401
3. Hotz, L., Krebs, T.: Configuration – state of the art and new challenges. In: Proceedings of 17. Workshop Planen, Scheduling und Konfigurieren, Entwerfen (PuK2003) – KI 2003 Workshop. (2003) 145–157
4. Männistö, T., Soininen, T., Sulonen, R.: Product configuration view to software product families. In: Software Configuration Management (SCM-10) – Papers from the ICSE Workshop. (2001)
5. Hotz, L., Krebs, T., Wolter, K., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J.: Configuration in Industrial Product Families - The ConIPF Methodology. AKA Verlag (2006)
6. Nilsson, N.J.: Principles of Artificial Intelligence. Morgan Kaufmann (1980)
7. Krebs, T.: Kowledge management for evolution of configurable products. In: Twenty-seventh SGAI International Conference on Artificial Intelligence (AI-2007), Cambridge, England, Springer Verlag (December 2007)
8. Blecker, T., Abdelkafi, N., Kreuter, G., Friedrich, G.: Product configuration systems: State of the art, conceptualization and extensions. In: Eight Maghrebian Conference on Software Engineering and Artificial Intelligence (MCSEAI 2004), Sousse, Tunisia (2004) 25–36
9. Minsky, M.: A framework for representing knowledge. Technical Report 306, Massachusetts Institute of Technology, Cambridge, MA, USA (June 1974)

[3] http://www.conipf.org

[4] http://www.w3.org/TR/webont-req/

10. Quillian, M.R.: Semantic Memory. In: Semantic Information Processing. MIT Press (1968) 227–270
11. Brachman, R.J., Schmolze, J.G.: An overview of the KL-ONE knowledge representation system. Cognitive Science **9** (1985) 171–216
12. Conradi, R., Westfechtel, B.: Version models for software configuration management. ACM Computing Surveys (CSUR) archive **30**(2) (1998) 232–282
13. Mylopoulos, J.: Object-orientation and knowledge representation. In Meersman, R., Kent, W., Khosla, S., eds.: Object-Oriented Databases: Analysis, Design & Construction (DS-4), Proceedings of the IFIP TC2/WG 2.6 Working Conference on Object-Oriented Databases: Analysis, Design & Construction, Windermere, UK, North-Holland (1990) 23–37
14. Heylighen, F.: Bootstrapping knowledge representations: From entailment meshes via semantic nets to learning webs. Kybernetes **30**(5/6) (2001) 691–722
15. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F.: The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press (2003)
16. Baader, F., Hollunder, B.: Kris: Knowledge representation and inference system. SIGART Bulletin **2**(3) (1991) 8–14
17. Reiter, R. In: On Closed World Data Bases. Plenum (1978) 119–140
18. Winston, M.E., Chaffin, R., Herrmann, D.: A taxonomy of part-whole relations. Cognitive Science **11**(4) (1987) 417–444
19. Herzig, A., Rifi, O.: Propositional belief base update and minimal change. Artificial Intelligence **115**(1) (1999) 107–138
20. Visser, P.R.S., Jones, D.M., Bench-Capon, T.J.M., Shave, M.J.R.: An analysis of ontological mismatches: Heterogeneity versus interoperability. In: AAAI 1997 Spring Symposium on Ontological Engineering, Stanford, USA (1997)
21. Sindt, T.: Formal operations for ontology evolution. In: Proceedings of the International Conference on Emerging Technologies (ICET'03), Minneapolis, Minnesota (USA) (2003)
22. Gray, J.: The transaction concept: Virtues and limitations (invited paper). In: Very Large Data Bases, 7th International Conference, Cannes, France (1981) 144–154
23. Günter, A., Hotz, L.: KONWERK - a domain independent configuration tool. In: Proceedings of Configuration (AAAI Workshop), Orlando, FL, USA, AAAI Press (1999) 10–19
24. Hollmann, O., Wagner, T., Gnter, A.: EngCon: A flexible domain-independent configuration engine. In: Proceedings Configuration (ECAI 2000-Workshop). (2000) 94–96
25. Gray, P.M., Embury, S.M., Hui, K.Y., Kemp, G.J.: The evolving role of constraints in the functional data model. Journal of Intelligent Information Systems **12**(2-3) (1999) 113–137
26. Banerjee, J., Kim, W., Kim, H.J., Korth, H.F.: Semantics and implementation of schema evolution in object-oriented databases. In: Proceedings of the 1987 ACM SIGMOD International Conference on Management of Data (SIGMOD 1987), New York, NY, USA, ACM Press (1987) 311–322
27. Maedche, A., Motik, B., Stojanovic, L., Studer, R., Volz, R.: An infrastructure for searching, reusing and evolving distributed ontologies. In: Proceedings of the 12th international conference on World Wide Web (WWW '03), New York, NY, USA, ACM Press (2003) 439–448
28. Stojanovic, L.: Methods and Tools for Ontology Evolution. PhD thesis, Universität Karlsruhe (2004)

Augmenting JSHOP2 Planning with OWL-DL

Ronny Hartanto[*]
Bonn-Rhein-Sieg Univ. of Applied Sciences
53757 Sankt Augustin, Germany
ronny.hartanto@fh-bonn-rhein-sieg.de

Joachim Hertzberg[**]
University of Osnabrück
49069 Osnabrück, Germany
hertzberg@informatik.uni-osnabrueck.de

Abstract

The paper describes an approach for combining a reasoner based on description logic (concretely, OWL-DL) and a planner, in this case an off-the-shelf HTN planner (concretely, JSHOP2). The domain representation is formulated in the DL regime, which can also used for logical reasoning about state. Domain and problem descriptions are then generated automatically from the DL representation as needed, plans are generated on the planner's side, and transformed back into the DL representation. A working example is given for a simulated robot navigation domain. In particular, we show that the automatic planning problem generation leads to skipping over parts of the domain description deduced to be irrelevant for solving the concrete planning problem at hand, thereby reducing substantially the size of the problem description for the planner and speeding up the planning process.

Introduction

Knowledge representation and reasoning play very important roles in field of artificial intelligence (AI). They enable an intelligent agent to store information and reason about it. This knowledge needs to be stored in a specific manner, such that the reasoning engines can apply their algorithms to it.

There are several methods and/or languages for representing knowledge on a computer, for example Logic-Based Representation Languages, Rule-Based Representation Languages, Visual Languages, etc. These methods depend on the purpose of the system and the kind of information that needs to be stored. Logic-based languages are important for intelligent agents due to their natural semantics that make them suitable for application to a machine implementation.

Some examples of logic-based languages are propositional logic, first-order logic, Knowledge Interchange Format (KIF) and description logic. The description logic is state of the art for knowledge representation and reasoning. The description logic provides some model checking methods, for example consistency, satisfiability and subsumption. It also offers the possibility to query for objects of interest.

[*]Thanks to Mrs. Iman Awaad for her support on this work.

[**]This research was partly funded by the European Commission's 6th Framework Programme IST Project MACS under contract/grant number FP6-004381. The Commission's support is gratefully acknowledged.

A planner is a reasoning system that produces a sequence of actions to achieve a goal. Hence, a planner also needs knowledge about possible actions, effects, etc. Each domain contains its own set of possible actions. In planning field, the knowledge describing a domain's possible actions and their effects is often referred to as the domain description. However this might cause a problem in planning. Planning is an NP-hard problem. The more information the agent has, the more detailed the plan and the more costly is the plan generation process. Most planning domains are optimized only to solve a specific planning problem in order to limit the amount of information that needs to be processed.

There are several ways to represent planning domains, for example situation calculus, STRIPS, graph-planning, to name a few. However, these representations are not as powerful as the one used in description logic. In the planning field, the domain is described in such so as to enable the planner to generate the plan with minimal cost.

To summarize, the dilemma facing an intelligent agent is how to produce a domain description efficiently such that the planning process is not inundated with unnecessary information. In the example presented here, the agent is an autonomous intelligent robot. Its knowledge base may contain a priori knowledge to which it may add knowledge it collects as it performs its tasks. Our robot also has a planner that plans its action. In order to circumvent the problem mention above, the robot might use two or more knowledge bases or a filter which removes information which is unrelated to the current domain. At present, these are the two most commonly used solutions to address the problem.

This paper presents an approach which allows the description logic-based knowledge representation and the planner to work synergistically. All information is kept in one single knowledge base. The description of the planning domain is generated from the information within the knowledge base. The description contains all the information that is required for generating a plan that will achieve the goal.

In our approach, a hierarchical planning method is used. The approach is versatile enough to allow the use of planner which use different methods for representing the knowledge than that used in description logic.

With the approach presenting here, the agent can continue to collect as much information as possible while still maintaining a reasonable size of planning domain description.

This approach has been tested on a robotic agent, however it should also be possible to implemented within other planning domains.

This paper is organized in the following manner: the following section describes the knowledge base and planner in brief. After that, the design of the proposed approach is explained, showing how the knowledge base (KB) is modeled, the connection between the KB and planner, how the planning domain is generated and the process of writing back the plans to the KB. This is followed by the implementation within the navigation domain of a mobile robot is shown. The results are then presented. The conclusion briefly summarizes this paper. Finally the future work section shows possible extensions of this work.

The Knowledge Base and Planner

As previously mentioned, description logic is state of the art in knowledge representation and reasoning. The Ontology Web Language (OWL) is a state of the art language for representing ontologies. It was developed for use in the semantic web. However, due to its success, many other fields have applied it within their domains.

The Knowledge Base

There are several frameworks of semantic web, namely the Resource Description Framework (RDF), Ontology Inference Layer (OIL) and DARPA Agent Markup Language (DAML). OWL adopts the semantics of DAML+OIL. There are three different versions of OWL, namely OWL-Lite, OWL-DL and OWL-Full (Dean *et al.* 2004). OWL-DL is a higly expressive description logic that is related to $\mathcal{SHOIN}$(**D**) (Horrocks, Patel-Schneider, & van Harmelen 2003). OWL-Lite is $\mathcal{SHIF}$(**D**) and as such constitute a subset of OWL-DL. The OWL-Full is a superset of OWL-DL (Horrocks, Patel-Schneider, & van Harmelen 2003).

In this work the OWL-DL is used for representing knowledge. The choice of OWL-DL as opposed to OWL-Lite or OWL-Full has several motivating factors. OWL-Lite's inference process in worst case takes exponential time (EXPTIME), however the expressive power is still below that of the OWL-DL (Horrocks, Patel-Schneider, & van Harmelen 2003). In contrast, OWL-Full's expressiveness goes far beyond that of description-logic, thus the reasoning time is undecideable. OWL-DL can express quite complex things and has the complexity of NEXPTIME (Horrocks, Patel-Schneider, & van Harmelen 2003).

There are several inference engines that can be used with OWL-DL. These are Pellet (an open source OWL-DL reasoner in java) (Sirin *et al.* 2007), FaCT++ (new generation of FaCT - Fast Classification of Terminologies) (Tsarkov & Horrocks 2006) and RacerPro (RACER - Renamed ABox and Concept Expression Reasoner) (Haarslev & Möller 2003).

The Planner

There are several planning paradigms available within the research community. These paradigms includes classical planning, neoclassical planning, heuristic planning, hierarchical task planning, an so on (Ghallab, Nau, & Traverso 2004). Currently, there are many planners available. A Hierarchical Task Network (HTN) has been used in practical applications more than any other planner as it provides a convenient way to write problem-solving "recipes" that correspond to how a human domain expert might approach the problem (Ghallab, Nau, & Traverso 2004). HTN planning provides the possibility to have tasks as goal instead of robot specific actions. Thus it is really helpful for the users to define their goal among these tasks. Hence, a HTN planner is used in our approach.

Before presenting our design, the concept of HTN planning will be covered here as background information. The HTN planning domain is defined by two tuples $\mathcal{D} = (O, M)$ where O is a set of operators and M is a set of methods. The HTN planning problem is defined by four tuples $\mathcal{P} = (s_0, w, O, M)$, where the s_0 is the initial state, w is the initial task network, O and M represents the domain $\mathcal{D}$ (Ghallab, Nau, & Traverso 2004). The goal in HTN planning is represented as some a set of tasks (methods).

There are several implementations of a HTN planner, such as Nonlin, SIPE-2 (System for Interactive Planning and Execution), O-Plan (Open Planning Architecture), UMCP (Universal Method Composition Planner) and SHOP (Simple Hierarchical Ordered Planner). There are four variants of SHOP available, namely SHOP, JSHOP, SHOP2 and JSHOP2.

The SHOP2 planner is used in our approach mainly for two reasons. Firstly, it won one of the top four prizes at the 2002 International Planning Competition. Secondly, it has a Java implementation, namely JSHOP2. As our approach is implemented in Java, having a planner in the same programming language provides some advantages. Nevertheless, programming language should not be a problem for combining the planner with the knowledge base. In previous work, a web-service technology has been used to encapsulate the planner, such that the planner can be executed from any machine with any operating systems and programming languages that support web-services (Hartanto & Hertzberg 2005). Moreover, the JSHOP2 planner compiles each domain description into separate domain-specific planner, thus a significant performance increase in the speed of planning is gained (Ilghami & Nau 2003).

Design

The architecture of our approach is shown in figure 1. It shows how a user interacts with the system and how the planner and the KB are connected. Currently the contents of the KB are specified manually, however in the future, an automatic specification will be addressed.

The KB contains states of the world, methods, operators and plans. The user can query a specific method or goal to be achieved. In this case the system will generate the planning-domain based on the requested method or goal, such that it becomes a complete domain description for the JSHOP2 planner. This process will be explained in more detail in the following section. In figure 1 the inference engine is embedded within the KB box.

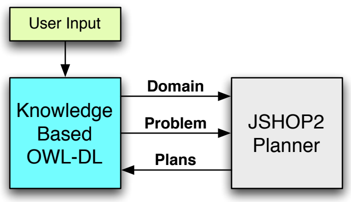


Figure 1: OWL-DL and JSHOP2 architecture

Modelling the KB

The modeling of the KB is of great importance. There are guidelines that show how an ontology should be designed (Noy & McGuinness 2002). One of these guideline deals with an iterative process, in which the ontology evolves during the modeling process. In this work, a starting model, that is based on the planning domain, is proposed. Figure 2 shows the proposed planning ontology. This ontology is a straight forward one. It has a $Planning\text{-}Domain$, a $Planning\text{-}Problem$, a $Method$ and a $Operator$ as its components.

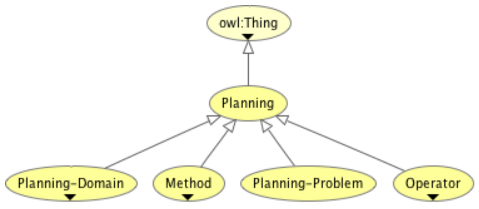


Figure 2: Planning Ontology

Figure 2 shows five classes (without owl:Thing) that depict the planning components. The $Planning$ class is the superclass of the $Planning\text{-}Domain$, $Planning\text{-}Problem$, $Method$ and $Operator$. This planning ontology is based on the HTN planning-domain. It maps the HTN planning-domain to the OWL-DL classes. The instances of these classes will be used later for generating the JSHOP2 planning-domain.

Here is more detailed specification of the upper picture of the planning ontology given in description logic syntax:

$Planning\text{-}Domain \sqsubseteq Planning$
  $\exists hasMethod.Method$
  $\exists hasOperator.Operator$
$Planning\text{-}Problem \sqsubseteq Planning$
  $\exists hasDomain.Planning\text{-}Domain$
$Method \sqsubseteq Planning$
  $\exists hasMethod.Method$
  $\exists hasOperator.Operator$
  $\leqslant 1\ shop2code$
  $\geqslant 1\ useState$
$Operator \sqsubseteq Planning$
  $\exists hasOperator.Operator$
  $\leqslant 1\ shop2code$
  $\geqslant 1\ useState$
$Planning\text{-}Domain \sqcap Planning\text{-}Problem \sqcap Method \sqcap Operator \sqsubseteq \bot$

The $shop2code$ is a datatype property with type string. Each method and operator will have $shop2code$ with the JSHOP2 specific syntax. The $useState$ property contains the required states for the corresponding method or operator, written in query string format.

The Connection between the KB and the planner

Figure 1 also shows how the KB is connected with the JSHOP2 planner. Three connections can be seen: domain, problem and plans. The domain and problem are required by the JSHOP2 planner to generate a plan. The generated plan is then inserted into the KB.

The planning ontology uses the HTN planning description, nevertheless the domain is not described using the JSHOP2 domain description syntax, which is described in the JSHOP2 manual (Ilghami 2006). Thus a mechanism which passes the information to the planner is required. The $shop2code$ property provides this feature. It acts as a bridge between the KB and planner. The next subsection describes how the planning-domain and planning-problem descriptions for JSHOP2 are generated.

Generating the Domain and Problem Descriptions for the JSHOP2

The planning-domain description can be generated in two ways. Either the planning-domain description can be generated from the $Planning\text{-}Domain$ instance or it can be generated from the an instance of $Method$.

The $Planning\text{-}Domain$ instance consists of sets of domains and operators, that are used in a specific domain. Algorithm 1 shows the steps needed to generate a list of methods and operators from the $Planning\text{-}Domain$ instance. This process is a straight forward one, as it simply adds the contents of the $Planning\text{-}Domain$ into the list.

Algorithm 1 Generate Domain from $Planning\text{-}Domain$

```
Require: $pd = instance\text{-}of\,Planning\text{-}Domain$
Ensure: $out = hashMap < Method, Operator >$
  for all $method$ from $pd$ do
    $out \Leftarrow +method$
  end for
  for all $operator$ from $pd$ do
    $out \Leftarrow +operator$
  end for
```

The second approach, which generates the planning-domain description from an instance of $Method$, does so by performing the steps described in algorithm 2. The algorithm shows a recursive method for generating a list of methods and operators from a $Method$ instance. Methods in HTN planning are implicitly the goals, that the planner need to achieve(Ghallab, Nau, & Traverso 2004). The $Method$ instance has the required information, such as which operators or methods are needed in order to achieve the given goal. Operators in HTN planning are implicitly actions that can be executed. In the same way as a $Method$, an $Operator$ consists of the operators which needed to perform the given action. When generating the planning-domain description, this algorithm is called with a $Method$ instance as parameter.

Algorithm 2 genDomain($input$)

```
Require: $input = instance\text{-}of\ Method\ or\ Operator$
Ensure: $out = hashMap < Method, Operator >$
  if $input\ isInstanceOf\ Method$ then
    for all $method$ from $input$ do
      $out \Leftarrow +genDomain(method)$
    end for
  end if
  for all $operator$ from $input$ do
    $out \Leftarrow +genDomain(operator)$
  end for
```

The JSHOP2-specific domain description is generated by iterating through each entry in the generated list. Each entry has a $shop2code$ property which describes the methods or operators in JSHOP2-specific syntax. At the beginning and the end of the domain descriptor, a header and a footer will be added such that it complies the JSHOP2 syntax.

The planning-problem consists of operators, methods, initial states and goals. Once the planning-domain is generated, it's already halfway to having a complete planning-problem for the JSHOP2 planner. The missing parts are the initial states and the goals. A goal is defined by choosing a method from the planning-domain. The initial states are inferred from the KB based on the current world model. Each $Operator$ or $Method$ has $useState$ property(ies). These properties contain query strings of the states that are needed to perform these operations or methods. These states are defined explicitly in the precondition of methods and operators. The user can perform an additional query in addition to these queries, so as to minimize the planning-problem. This can be seen in detail in the section example domain which follows.

Once the planning-problem is completely generated, it is fed into the JSHOP2 planner. The JSHOP2 planner will generate the one or several plan(s) from the given problem description and return all possible plans.

Inserting the Plans into the KB

Not only can the JSHOP2 planner retrieve domain and problem descriptions from the KB, the computed plans are inserted into the KB as well. Thus, the KB serves as a single source which the agent consults. The KB stores all the generated plans, thus minimizing the demand for re-planning in case of failure during a plan's execution. The sequencing layer executes the plan. If a problem is encountered during a plan's execution, the sequencer will try to execute the next plan until the goal is achieved or all the plans have been tried. In case that all the plans execute without successfully achieving the goal, re-planning take into consideration the current world model proceeds or an error message is generated.

A Plan is a sequence of actions, which the agent needs to perform in sequence. OWL doesn't support ordering naturally as the constructors of these (rdf:List and rdf:nil) are unavailable due to serialization (Drummond *et al.* 2006). In this work the OWL-List approach from Drummond et. al. is used for representing the plans in the KB. Figure 3 shows the model of the $Plan$ class and its relationship with the $OWLList$ class.

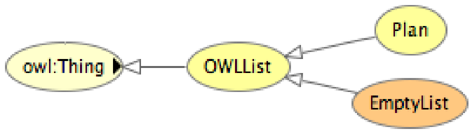


Figure 3: Plan & OWLList Ontology

Implementation

In this section an example that uses our approach is presented as well as the tools that are used. This approach was tested with the navigation problem in the mobile-robotics domain.

Used Tools

Our approach is implemented in Java. The JSHOP2 planner is closely coupled with the KB in our approach, to enable our application to get the Java-objects that are generated by JSHOP2. The application is developed within Eclipse.

The ontology is built and designed with the help of Protege, a free open source ontology editor and knowledge base framework. Two inference engines have been tested with our application, namely FaCT++ and Pellet. A commercial inference engine, namely RacerPro, should also work with our approach.

Example Domain

As mentioned previously, our approach was tested within the mobile-robotics navigation domain. In our example the mobile-robotics domain contain three operators and two methods. These operators are $!drive\text{-}robot$, $!visit$ and $!unvisit$. The exclamation sign in the front of operators' name means that these operators are ground operators (Ilghami 2006). Two of these operators are dummy operators, which do not correspond to real actions of the robot, namely $!visit$ and $!unvisit$. These operators are required by the JSHOP2 planner in order to mark a visited place, such that the search algorithm is tractable.

Two methods with the same name, $navigate$, serve the same purpose with different variables. These methods are $(navigate\ ?robot\ ?to)$ and $(navigate\ ?robot\ ?from\ ?to)$. With the first method, one can give a command such as *navigate all robots to room5*. The KB will generate the initial states for every available robots consistent with their current positions. Without using explicitly the current positions of the robots. In the second method the positions of the robots are used.

The following code shows the $!drive\text{-}robot$ operator and one of the navigate methods.

```
(:operator (!drive-robot ?robot ?loc-from
    ?loc-to)
  ((at ?robot ?loc-from))
  ((at ?robot ?loc-from))
  ((at ?robot ?loc-to)))

(:method  (navigate ?robot ?from ?to)
Case1 ((at ?robot ?to))
  ()
Case2 ((adjacentto ?from ?to))
  ((!drive-robot ?robot ?from ?to))
Case3 ((room ?room) (adjacentto ?from ?room)
    (not (visited ?room)))
  ((!drive-robot ?robot ?from ?room)
    (!visit ?room)(navigate ?robot ?room ?to)
    (!unvisit ?room))
)
```

From these code snippets, we can extract the required states, such as *(at ?robot ?location)*, *(room ?location)*, *(adjacentto ?location ?location)* and *(visited ?location)*. All these states are written on the $useState$ property in the corresponding method or operator. The *(visited ?location)* is a state, that is generated on-line during computation. Thus, it is not included in the $useState$ property.

Results

The example presented above was implemented and tested within an office environment consisting of two buildings that are connected to each other by a corridor. Each building has six rooms and a corridor as shown in figure 4. These rooms are connected to each other through doors. However, to simplify the domain the doors are not described in the ontology, as our actor can not currently manipulate a door handle. Each room has an object property, namely $adjacentto$, for example $room$-2 has two object properties $adjacentto$ $corridor$-1 and $adjacentto$ $room$-4.

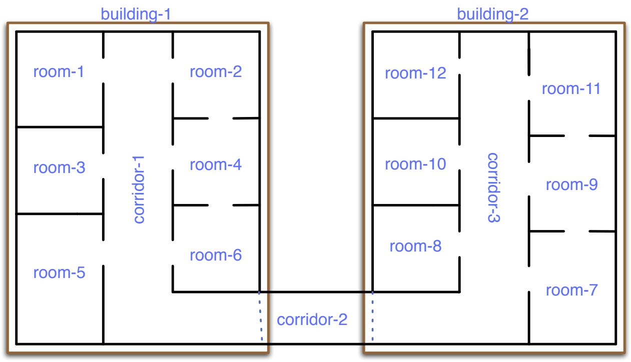


Figure 4: Simple Map of the Building

Figure 5 shows an ontology containing an actor and a fixed-object. In this example the actor is a mobile robot. It has the property $\exists at.Room$, that represents its current spatial position in the domain. The $Fixed\text{-}Object$ is the superclass of $Building$. The $Building$ is the superclass of $Room$. The $Fixed\text{-}Object$ is the container of all objects that cannot be manipulated by the actor.

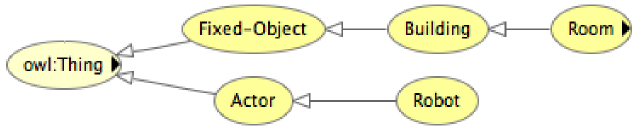


Figure 5: Actor & Fixed-Object Ontology

Having $Building$ as a super-class of $Room$ enables the users to query the instances later. As shown in figure 4, the office-structure consists of two buildings and twelve rooms. Putting an additional query on top of the initial states query, we can minimize the size of the planning-problem. Let us assume that the robot is in $room$-1 and the goal is to navigate to $room$-6. Normally the KB provides all the rooms and their connectivity. This domain grows and increases linearly with the KB, the more information within the KB the bigger the size of a generated planning-problem. In addition more time is required to compute the plan. By adding an additional query, we can limit the size of the planning-problem. In this case, we can limit our query to include only $building$-1 ($\forall Room \sqsubseteq building$-1). In our example, the generated initial-states are reduced from 15 objects into eight objects.

The following code shows an example query in SPARQL (Prud'hommeaux & Seaborne 2007) to retrieve all `rooms` in `building-1` from the KB (the prefixes are omitted).

```
SELECT DISTINCT ?Room
WHERE
{ ?Room :inBuilding ?Building;
  rdf:type :Room.
  FILTER(?Building = :building-1)
}
```

Generated Plan

The concepts, which are presented here, are sufficient to build a complete planning-problem for the JSHOP2 planner. Table 1 shows the results from the given example whose goal was to navigate $robot1$ from an initial-position to a destination. It shows the number of all possible plans and the required time to compute them. The last column shows whether the query is applied (only building 1) or not (buildings 1 and 2).

Table 1: Generated Plans from Navigate $robot1$

Init.-Pos	Dest	#Plans*	Time (ms)	buildings
room-1	room-4	1	4	1 and 2
room-1	room-4	1	3	1
room-1	room-9	1	15	1 and 2
room-4	room-1	7	14	1 and 2
room-4	room-1	7	10	1
room-4	room-9	7	36	1 and 2

*Note that JSHOP2 will deliver one or several solutions for a given problem, but not the complete set of plans.

The following sequence is one example of a generated plan to navigate $robot1$ from $room$-1 to $room$-9 with $cost = 10$:

```
(!drive-robot robot1 room-1 corridor-1)
(!visit corridor-1)
(!drive-robot robot1 corridor-1 corridor-2)
(!visit corridor-2)
(!drive-robot robot1 corridor-2 corridor-3)
(!visit corridor-3)
(!drive-robot robot1 corridor-3 room-9)
(!unvisit corridor-3)
(!unvisit corridor-2)
(!unvisit corridor-1)
```

This plan has ten actions, however the actual robot actions are only four due to the dummy operators ($!visit$ and $!unvisit$).

Conclusion

In this paper, an approach which couples a knowledge base system with a planning system is presented. It solves the problem that an intelligent agent faces in dealing with large amounts of information which may or may not be useful in generating a plan to achieve a goal. Our approach allows the agent to collect information from its surroundings and insert them into the KB as it maintains a small size for the planning-domain description. The planning system will only get the information from the knowledge base which is required to solve the current problem with minimum computation cost. This is done while still using a single KB.

In addition, this approach allows the automatic generation of the planning domain description from the KB. It includes the full capability of an inference engine in the description logic framework and the advantages of available planning techniques.

The knowledge base system is implemented in OWL-DL, due to its support of description logic. It has the expressive power of description logic and its compatibility with several inference engines. These inference engines are RacerPro, FaCT++ and Pellet. In addition, the HTN planning technique, as implemented in JSHOP2, is used as the planning component.

An example which uses the approach in navigating a mobile robot is also presented. It shows how the planning system benefits from being coupled with the KB. The user can query the possible tasks which can be achieved by the robot. After that the KB generates the necessary methods and operators for the domain description. Besides that the current world information will also be inferred from the KB. This information is part of the problem description. Finally, the planning system computes the plan from these domain and problem descriptions.

The results section shows the performance benefit of the proposed planning system. The KB generates complete domain and problem descriptions. In addition, the user can restrict the problem description is sufficient to generate complete plans.

Future Work

Automatic insertion of the perception data of a robot into the KB will be addressed. This ensures that the knowledge within the KB remains current and up to date.

The sequencing layer which controls plan execution will be implemented. The sequencing layer must not only coordinate the execution of actions within a plan but it must handle exceptions and failures encountered during the plan execution. In the event that a failure occurs, a heuristic such as try executing the next best plan or re-planning may be used.

Applying the concept of affordance to the KB and the robot specification will also be our future work. This will enable the system to query objects that can be manipulated. It will offer the user an object specific goal, such as grab the plate, wash the plate or bring a cup of coffee.

Additional support for other planners should also be explored. One way of approaching this will be to support PDDL which can then be fed to planners which support it.

References

Dean, M.; Schreiber, G.; Bechhofer, S.; van Harmelen, F.; Hendler, J.; Horrocks, I.; McGuinness, D.; Patel-Schneider, P.; and Stein, L. 2004. OWL web ontology language reference. W3C Recommendation. http://www.w3.org/TR/owl-ref/.

Drummond, N.; Rector, A.; Stevens, R.; Moulton, G.; Horridge, M.; Wang, H. H.; and Seidenberg, J. 2006. Putting OWL in Order: Patterns for Sequences in OWL. In *Workshop on OWL: Experiences and Directions 2006*.

Ghallab, M.; Nau, D.; and Traverso, P. 2004. *Automated Planning: Theory and Practice*. Morgan Kaufmann.

Haarslev, V., and Möller, R. 2003. Racer: An OWL Reasoning Agent for the Semantic Web. In *Proceedings of the International Workshop on Applications, Products and Services of Web-based Support Systems, in conjunction with the 2003 IEEE/WIC International Conference on Web Intelligence,Ê Halifax, Canada, October 13*, 91–95.

Hartanto, R., and Hertzberg, J. 2005. Offering Existing AI Planners as Web Services. In *19. Workshop "Planen, Scheduling und Konfigurieren, Entwerfen"*.

Horrocks, I.; Patel-Schneider, P. F.; and van Harmelen, F. 2003. From SHIQ and RDF to OWL: The Making of a Web Ontology Language. In *JWS 03*.

Ilghami, O., and Nau, D. S. 2003. A General Approach to Synthesize Problem-Specific Planners. Technical report, Department of Computer Science, Institute for Systems Research, and Institute for Advanced Computer Studies University of Maryland.

Ilghami, O. 2006. Documentation for JSHOP2. Technical report, Department of Computer Science University of Maryland.

Noy, N. F., and McGuinness, D. L. 2002. *Ontology Development 101 : A Guide to Creating Your First Ontology*.

Prud'hommeaux, E., and Seaborne, A. 2007. SPARQL Query Language for RDF. W3C Candidate Recommendation. http://www.w3.org/TR/rdf-sparql-query/.

Sirin, E.; Parsia, B.; Grau, B. C.; Kalyanpur, A.; and Katz, Y. 2007. Pellet: A Practical OWL-DL Reasoner. *Journal of Web Semantics* 5(2).

Tsarkov, D., and Horrocks, I. 2006. FaCT++ Description Logic Reasoner: System Description. In *Proc. of the Int. Joint Conf. on Automated Reasoning (IJCAR 2006)*, volume 4130 of *Lecture Notes in Artificial Intelligence*, 292–297. Springer.

Extending Plant Packing

Matthias Postina, René Schumann, Sonja Aust and Jan Behrens

OFFIS, Escherweg 2, 26121 Oldenburg Germany
`matthias.postina|rene.schumann|sonja.aust|jan.behrens@offis.de`

Abstract. The transportation of plants is expensive, since plants have a comparable high volume in relation to their value. Therefore effective plant packing is an important issue, as the transportation costs are directly dependent on the number of needed trolleys. The potted plant packing problem was introduced in [1] and a first solution was presented. It turned out that the generated plans were not sufficient in comparison to trolleys packed by humans. By observing human packers, we identified plant stacking as a usual habit to increase compactness of the packing. Thus we extended our algorithms, adding the option of stacking plants. In this paper we present the techniques implementing stacking strategies. It is shown that stacking of plants increases the performance of the packing algorithm significantly.

1 Introduction

The transportation of plants is expensive, since plants often have a comparable high volume in relation to their value. For standardized transport, potted plants are loaded on transport trolleys (shown in Figure 1). The cost of transportation depends on the number of these trolleys. In order to minimize transportation costs, effective packing of trolleys is necessary. Thus the potted plant packing problem was presented in [2] and steps towards a solution were shown in [1]. Based on the evaluation of test data, it turned out that the computed solutions could not compete with the number of needed trolleys packed by humans. Computed plans need up to 75% more trolleys (see section 4 for details). Different reasons could be identified for this lack of performance.

- Humans commonly use to stack plants on trolley layers.
- The algorithm works on pessimistic data. The size of plants is given in an interval. The algorithm uses always the maximal size, which is a pessimistic assumption.
- Humans pack several layers in parallel.

Based on these results it is obviously necessary to scale the performance of the packing program. In fact stacking of plants is a very efficient way to increase the packing density. For instance the capacity for 10 l pots can be increased by more then 40% using stacking (cf. Figure 5). For other pot sizes the results are similar. So in fact stacking of plants seems to be a promising technique to improve results.

However, the pessimistic assumption about the actual size of the plants can not be relaxed completely. But as plants are a flexible good, this allows to place plants more densely. In consequence the assumptions about the plants dimensions can be relaxed partially. It is assumed that the size of each plant is the mean value of the given interval. When this general rule is not applicable - e.g. during the anthesis of the plants - we assume further that this can be specified for each type of plant. Of course the planning quality as well as the data quality has to be approved and each plant has to be checked to that effect.

The usage of average values for the plants size leads to a more realistic model for packing plants. Moreover the model of plants has to be detailed, to remove some oversimplifications. In our first approach plants and especially their pots were modeled as cylindric objects. It turned out that this model hinders the computation of realistic plans. Real pots are conic, their base has a lower diameter then their top. This allows that pots can be placed more densely at the border area of the trolley layers. This effect is significant for realistic packing. In consequence our model has to be expanded to respect the correct placement of conic pots - hence the base diameter is used for the calculation of the placement at the border area of the trolley layer.
The packing of parallel layers is actually not tackled and remains as future work. But an improvement that avoids to create only marginal used layers does already exist. If such a layer is detected, an algorithm tries to distribute the plants to other layers. This strategy helps to avoid nearly empty layers in general.
Actually two improvements were implemented. First basic data is updated, and the over pessimistic assumptions about the plant size is reduced. Second, stacking of plants is supported.

As the packing problem was described in [2] and a basic solution was presented in [1], this article focuses on the extended feature of stacking and it's integration into the existing tool. Thus the article is structured as follows. In the next section the potted plant packing problem and a solution approach is briefly described. In section 3 the stacking of plants is presented in detail. It is discussed what different techniques exist for stacking and when they can be applied. Then in section 4 case studies are presented, where the computed solutions are compared with human generated packings. Finally we conclude and discuss future work.

2 The potted plant packing problem

2.1 Problem statement

The major task is to compute a valid packing instruction for a given transportation order. Such a packing instruction must contain directives for the exact placement of each and every plant that is part of the order. Any such directive holds information about the plant's designated place - and each layer's exact placement (mounting height) within the trolley. To clarify the problem, a trolley is shown in Figure 1. A number of further constraints and additional rules have

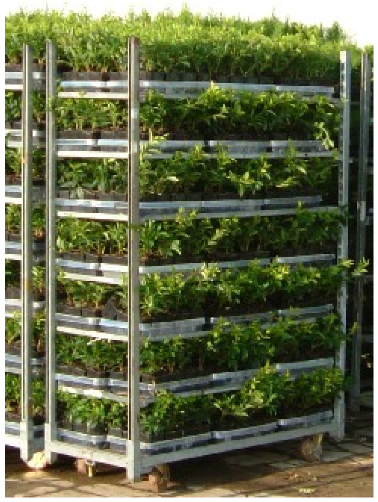


Fig. 1. packed trolleys

to be observed as part of the problem. For example: it is allowed to stack plants on a trolley layer, the placement of layers into trolleys has to respect the stability of trolleys, and the total trolley height usually has to be less than the available internal truck height. Modeling aspects focusing these issues were discussed in [2] and [1].

2.2 Existing solution approach

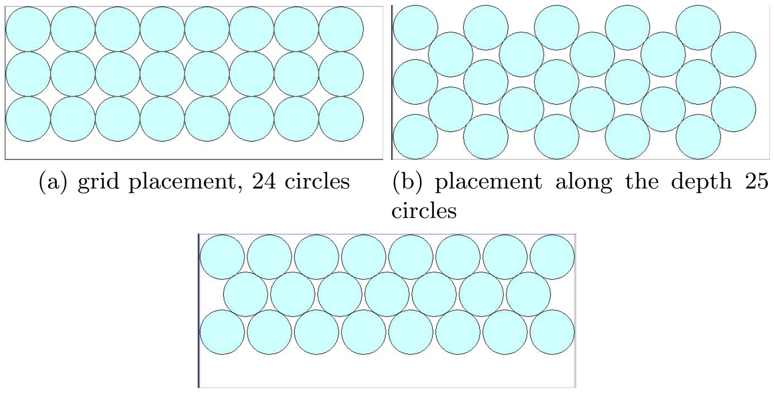
For the sake of simplicity it is assumed that for each and every plant a cylinder can be computed, that contains the plant. To solve the potted plant packing problem it is decomposed into two sub-problems:

- Distribution of plants on trolley layers
- Distribution of layers on trolleys

Packing of layers The packing of trolley layers corresponds to the packing of circles into a rectangle, since the height of the plants can be omitted. The packing of trolley layers can itself be subdivided into two different tasks.

- packing of equal circles into a rectangle
- packing of unequal circles into a rectangle

The packing of equal circles into a rectangle is a standard geometrical problem. Even though optimal packings are known for a growing number of circles (e.g. in [3]) this is an NP-hard task. However, for the packing of equal circles simple heuristics can be applied, that offers already a good solution quality. The circles are placed along the horizontal or vertical of the layer, or are placed in as a grid. Unfortunately it depends on the size of the circles and the rectangle which heuristic performs best. The results of the different placement strategies are shown in Figure 2.



(a) grid placement, 24 circles

(b) placement along the depth 25 circles

(c) placement along the width, 23 circles

Fig. 2. regular circle placements

The placement of unequal circles into a rectangle is a quite more difficult task. Only few publications have yet dealt with this problem, for example [4], [5] and [6]. We adopted the maximum hole degree algorithm (B1.0) presented in [6]. The main idea of this algorithm is the subsequent placement of circles into corners. A corner is defined by two sides of the rectangle, a rectangular side and a circle, or two circles. The first two circles are placed by a simple placement strategy. Then for each circle not placed already within the rectangle, all possible corner placements are computed. The circle being associated with the placement having the minimal distance to another circle or side is chosen next. This is repeated until no more valid corners are found or all elements have been placed. A detailed description of the algorithm as well as a complexity analysis can be found in [6]. An illustration showing the packing of unequal circles into a rectangle is sketched in Figure 3.

Even if the trolley layers are computed step by step the design decisions within the algorithm are taken into a direction that the overall summed height of all layers is minimized. This is an heuristic approach aiming at the minimization of the number of needed trolleys. Consequently the objective function regards the global context even though this computation is separated into partial blocks.

Packing of trolleys The problem of distributing the layers to trolleys is a classical bin packing problem. But additionally the position of each layer within a trolley has to be computed. Trolley layers are hooked into mounting points, which are generally found at 5 cm intervals from a base of 20 cm up to a height

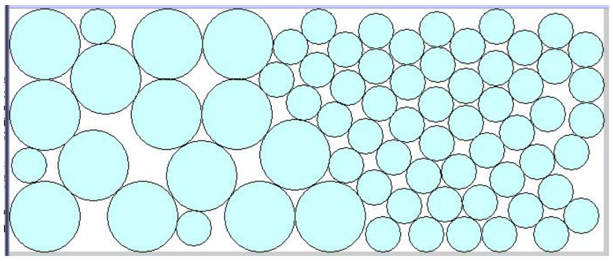


Fig. 3. placement computed by the maximum hole degree algorithm

of 190 cm. The placement of layers within a trolley follows a simple strategy: The tallest trolley layer is hooked into the topmost mounting point that still ensures the adherence of all other constraints, especially the maximum allowed height. All remaining trolley layers are then sorted in ascending order by their weight and inserted top to bottom into the trolley. This strategy aims at two goals. It tries to

– maximize the usage of available space on the truck and
– lower the center of gravity to the nethermost point for stability reasons.

3 Stacking plants

As already mentioned solutions for the potted plant packing problem generated by humans are outperforming computed solutions. Contrary to programs, human packers have an intuitive understanding of spatial optimization, so stacking plants was observed as a typical human packing habit. To imitate such stacking is consequently the first approach to improve the packing algorithm. This section categorizes the plants stacking problem as the stacking on homogeneous patterns and as the stacking on heterogeneous patterns.

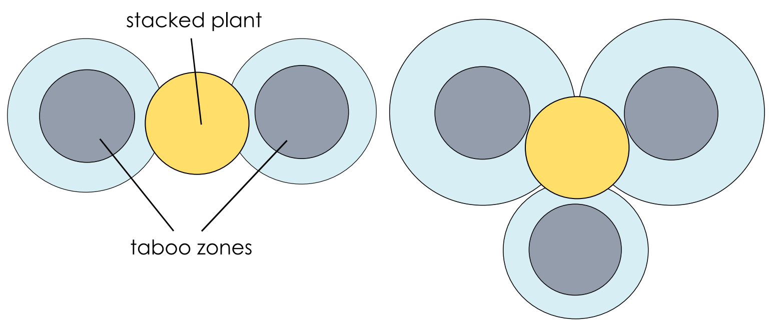
3.1 Stacking on homogeneous patterns

The terms homogeneous and heterogeneous pattern are related to a layer of a trolley which is already initially packed. Whenever a layer of a trolley is homogeneously packed - which means only plants of the same category with exactly the same pot and plant size were used to fill the basic layer - we say this layer follows a homogeneous pattern. Figure 2 has already shown such homogeneous patterns. In order to support stacking, we need to upgrade the planning algorithms and we have to regard stacking in more detail.

Definition 1:
We call a position a *stackable place* whenever a pot could be placed on the top of a minimum set of three neighbored pots having the same height.

We chose this minimal set of three plants as bed for a stacked plant since plants are fragile goods and falling over leads to a total loss of plant value. So the configuration pictured in Figure 4 (a) would be an illegal configuration according to our definition. For the same reason stacking needs to take care of the underlying plants. Keeping this in mind, we defined a taboo zone for each plant reflecting its actual shape. These taboo zones have to be respected when looking for valid stackable places. Since we focus on the packing of circular objects it is sufficient to store the center and the radius to indicate a stackable place. Furthermore we need a statically defined minimal footprint to guarantee a sufficient contact surface and also a maximal boundary to prevent damage from the underlying plants. On the strength of stability we also restrict the stack height to a maximum of two per trolley layer. Basically this reflects the packing habit of human packers.

Definition 2:
We say a stackable place is *valid* for a certain pot category, if the footprint of pots assigned to this category guarantees a sufficient contact surface without violating the taboo zones of the underlying plants.



(a) Non stackable place.

(b) Valid stackable place for the stacked plant (assuming same pot hight).

Fig. 4. Visualization of Definitions

It is easy to recognize the set of stackable places when the basic layer follows a homogeneous pattern. One needs to identify the size of the radius, the size of the taboo zone and the used placement strategy (cf. Fig. 2). By means of this information one can calculate an offset along a direction vector pointing out of each underlying circle center. So finding the center of the stackable places could be seen as a pattern/raster shift of the underlying layer. Such a pattern shift is shown in Figure 5 where the center of each stackable place is moved along the direction vector $\overrightarrow{v}$.

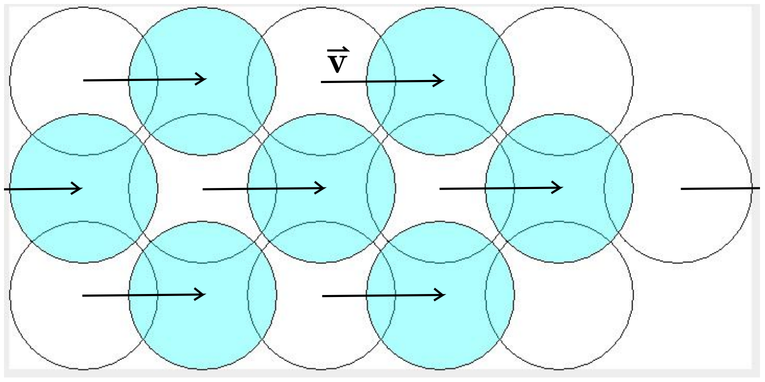


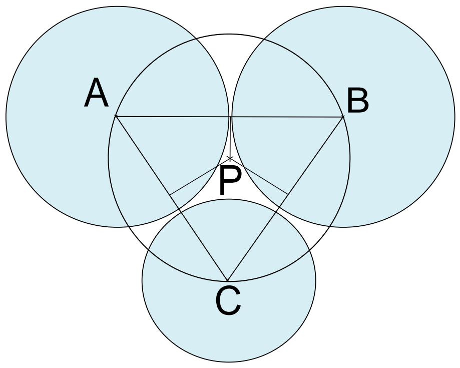
Fig. 5. Pattern postponement along a direction vector v (10 l pots following a homogeneous pattern)

3.2 Stacking on heterogeneous patterns

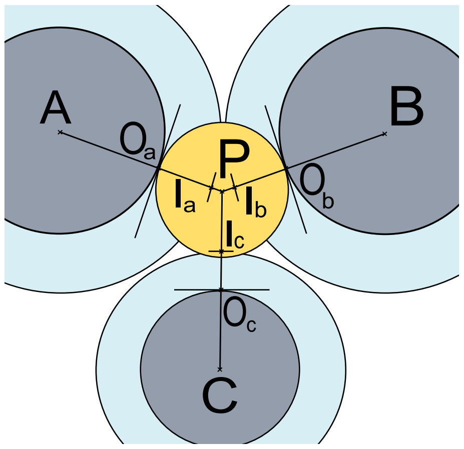
In analogy to homogeneous pattern, we say a layer follows a heterogeneous pattern, when plants of different categories (with miscellaneous pot diameters) and different pot heights were used to fill the basic layer of a trolley. Figure 3 shows such a heterogeneous arrangement.

Obviously it is much harder to spot stackable places on the top of a layer following a heterogeneous pattern. Since pot heights may vary one need to identify regions in the layer where at least three neighbored pots are having the same pot height. Furthermore these three pots need to be grouped in a way that offers a valid stackable place - which is close enough to allow stacking but wide enough to respect taboo zones of the underlying plants. Moreover one needs to calculate the centers of the stackable places for each possible triangular constellation individually. Such a center of a stackable place could be calculated as follows:

When regarding a troika of pots of the same height, the centers A, B and C of these three circular objects are forming a triangle. We want to find a point P having the same distance to each of A, B and C. Such P is the center of the circumscribed circle of the triangle which is the intersection point of the perpendicular bisectors of the sides. This is shown in Figure 6 (a). Now, one need to show that the identified stackable place having P as center is a valid stackable place for a certain pot category according to Definition 2. In order to do so, we



(a) Finding P as center of a possible stackable place



(b) Identifying the category of the stackable place

Fig. 6. Finding stackable places

will formalize Definition 2 based on the sketch in Figure 6 (b).

Given are the points A, B and C as centers of the three neighbored pots C_a, C_b, C_c with known radii. All of these pots are planted and each taboo zone is known and identified by a taboo radius. We already calculated point P and define $\overline{PA}, \overline{PB}$ and $\overline{PC}$ as straight lines - each going through P and one of the centers A, B and C. Further we name the intersection points of these lines with the taboo circles O_i and the intersection points of these lines with the pot bounding circles I_i $(i \in a, b, c)$.

To reduce search space we introduced the concept of categories in [1]. Still following this concept, each plant is classified as member of a specific cylindric category. Such a category could be seen as a virtual box hosting plants with similar dimensions. The radius R_c of the cylinder is taken to define a valid stackable place for a certain pot category.

We say the stackable place having P as center is a valid stackable place for a certain pot category Cat_n $(n \in \mathbb{N})$, if

$R_c \leq \min[|\overline{PO_a}|, |\overline{PO_b}|, |\overline{PO_c}|]$ *and*

$R_c \geq \max[|\overline{PI_a}|, |\overline{PI_b}|, |\overline{PI_c}|]$.

We define the set of stackable places S. Valid stackable places are assigned to pot categories so that each category holds a subset V_n of S, $V_n \subseteq S$ $(n \in \mathbb{N}$

and n is corresponding to the index of the categories Cat_n). The data structure SP is a hash holding the key value pairs of categories and subsets V_n of S :

$$\begin{array}{lll} SP = \{ & Cat_1 => V_1 & , \\ & Cat_2 => V_2 & , \\ & ... & , \\ & Cat_n => V_n & \} \end{array}$$

Further we have a second hash CS holding the key value pairs of categories C_n and corresponding plant sets $Pl_n(n \in \mathbb{N}\ )$ which are subsets of all non placed plants:

$$\begin{array}{lll} CS = \{ & Cat_1 => Pl_1 & , \\ & Cat_2 => Pl_2 & , \\ & ... & , \\ & Cat_n => Pl_n & \} \end{array}$$

The stacking algorithm formalized in Algorithm 1 works as follows: Given the hashes SP and CS, the algorithm starts to assign plants in ascending order with respect to their plant category (which means highest dimension first, ordered by the height) to stackable places ($assign(...)$). During this processing, the hash SP - representing valid stackable places per category - is updated permanently ($updateV_x(...)$). This leads to a shrinking set of valid stackable places. Whenever a plant is assigned to a stackable place, such plant is deleted from CS ($removeElementFromPl_j(...)$). The algorithm stops the stacking for a trolley layer, when the hash SP is empty or when all plants were placed. Figure 7 shows a graphical representation of an exemplary calculation.

```
1. $SP$ // hash, given as described
2. $CS$ // hash, given as described
3. $V_i$ // subset of $S$
4. $Pl_i$ // subset of all non placed plants
5. WHILE ( $\exists i \in \mathbb{N} : V_i \neq \emptyset \vee Pl_i \neq \emptyset$ )
    – $j = \min\{i \in \mathbb{N} : V_i \neq \emptyset \wedge Pl_i \neq \emptyset\}$
    – $assign(v_{jx}, pl_{jy})$ // ( $x, y \in \mathbb{N}, v_{jx} \in V_j, pl_{jy} \in PL_j$ )
    – $removeElementFromPl_j(pl_{jx})$
    – $updateV_x(v_{jy})$
```

Algorithm 1: Stacking on heterogeneous patterns

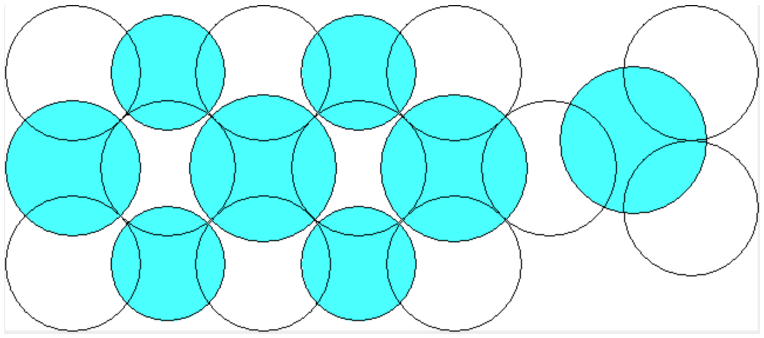


Fig. 7. Stacking on heterogeneous patterns

4 On the influence on stacking

After we have implemented the aforementioned stacking strategies, we compare the computed solutions with and without stacking to solutions generated by humans. As we improved only one aspect of the previous mentioned it is not surprisingly, that humans still need fewer trolleys. But as can be seen in table 1 the quality of the solutions with stacking increases.
Scenario 1 comprises homogeneously and heterogeneously packed layers. Scenario 2 is more difficult, as there exist no homogeneous layers. Due to the aforementioned model extension concerning the form of pots, the homogeneously packed layers are equal either packed by hand or computed. So the quality difference can be explained by different packed heterogeneous layers and differences between the assumed and real height of the layers. For this reason the quality of the schedules in scenario 1 is better then in scenario 2.
Nevertheless in both cases stacking can improve the computed results drastically. Of course the usage of percent values is problematic, as the number of CC-trolleys is a discrete value. But the improvement by stacking is as significant in absolute as in relative numbers. Analyzing the trolleys packed by humans we

scenario	**#CC needed**	**without stacking**			**with stacking**		
		#CCs	abs. delta	delta in %	#CCs	abs. delta	delta in %
1	5	7	2	40%	6	1	20%
2	8	14	6	75%	10	2	25%

Table 1. Results of the Case Study of effective packing

had to state that human packers take advantage of the fact that they are breaking rules the planning strategy has to respect. For example plants were stacked up to three levels per trolley layer or plants were stacked on the base of only two plants. So it was possible to place more plants on one layer and to pack the plants more densely. Of course such constraint violation is not thinkable for the algorithm. It is actually in discussion that the packing rules should be applied for humans as well to avoid plant damages.

5 Related Work

The plant packing problem could be considered as an exotic problem with respect to its special domain. The complex structure of the problem makes it hard to assign this problem to a certain research area. As a variation of the 3D respectively 2D bin packing problem it belongs to the field of combinatorial optimization which is part of the operations research. Otherwise the problem of finding optimal patterns for putting circular objects into rectangular shapes is also regarded in the field of computational geometry as well as the 3D instance of this problem in the area of sphere packing. Furthermore a number of constraints is limiting the search space and the defined data structure containing pairs of categories and feasible places reminds of modeling for a constraint satisfaction problem.
To the best knowledge of the authors a problem like this was never published which hinders general benchmarking with similar solutions. Nonetheless benchmark analysis of certain parts of the problem stays open as future work.

6 Summary and Future work

When observing human packers it turned out that plant stacking reflects a usual habit to increase compactness of the packing. By know our planning algorithm was not implementing such a case. This paper shows our approach to enable the algorithm to support plant stacking. We distinguished between stacking on homogeneous and heterogeneous patterns and indicated solutions for both of them. It is assumed that stacking will have a relevant impact on the quality of packing solutions. First test results are reinforcing such assumption.

For the sake of simplicity we are dealing with plant categories rather than individual plants. However, the categories are either boxes or cylinders whereas a plant pot is usually tapered. So contrary to cylinders the top and bottom diameters of the pots are unequal. Further research will have to detect if such a simplification leads to proper results or if the data model needs to be updated. Early tests using such data models and limited to homogeneous patterns are indicating, that computed solutions were on a par with solutions generated by human packers in such cases.

Inspired by human packing habits for a second time, the parallel packing of trolley layers is another field of further research. As described, the objective function (minimization of the overall height of all trolleys) is already designed globally but the algorithm works in layers. Using parallel packing one may assume better results, but with respect to complexity and the corresponding runtime of the algorithms we would like to watch more real life results of this approach first. This is because the program based on the planning algorithms is also used in a real time context: In order to be able to calculate the associated shipping costs the dispatcher needs to know how many trolleys are necessary to fulfill an order, which is given by phone at that moment.

References

1. Schumann, R., Behrens, J.: The potted plant packing problem: Towards a practical solution. In Sauer, J., Edelkamp, S., eds.: 20. Workshop Planen, Scheduling und Konfigurieren, Entwerfen (PUK), Bremen, Universität Bremen (2006) 37 – 48
2. Schumann, R., Behrens, J., Siemer, J.: The potted plant packing problem. In Sauer, J., ed.: 19. Workshop Plan, Scheduling und Konfigurieren / Entwerfen (PUK). Fachberichte Informatik, Koblenz, Universität Koblenz-Landau Fachbereich Informatik (2005)
3. Nurmela, K.J., Östergård, P.R.J.: Packing up to 50 equal circles in a square. Discrete and Computational Geometry **18** (1997) 111 – 120
4. George, J.A., George, J.M., Lamar, B.W.: Packing different sized circles into a rectangular container. European Journal of Operational Research (EJOR) **84** (1995) 693 – 712
5. Correia, M.H., Oliveira, J.F., Ferreira, J.S.: Cylinder packing by simulated annealing. Pesquisa Operacional **20** (2000) 269 – 286
6. Huang, W.Q., Li, Y., Akeb, H., Li, C.M.: Greedy algorithms for packing unequal circles into a rectangular container. Journal of the Operational Research Society (JORS) **56** (2005) 539 – 548

Optimal Infinite-State Planning with Presburger Automata

Björn Borowsky and Stefan Edelkamp

Computer Science Department
Otto-Hahn-Str. 14
University of Dortmund

Abstract. The paper proposes a new approach to infinite-state action planning with automata theory. State sets and actions are encoded as Presburger formulae and represented using minimized automata. The symbolic exploration that contributes to the planning via model checking paradigm repeatedly applies partitioned images on such automata to compute the automata for the successors of the current state set. The implementation adapts a library for automata manipulation.

1 Introduction

With recent developments of the problem domain description language PDDL [9], modern action planning features arithmetic expressions. Despite that numerical planning is undecidable in general [13], in recent planning competitions, explicit-state planners have reported considerable success in solving benchmark domains with numbers. As these planners all rely on heuristic or local search, there is no guarantee on the plan quality.

For propositional planning, several optimal planning approaches are known. Planners like Satplan [18] and Graphplan [3] optimize the parallel plan length, while other planners optimize the number of steps [21]. For propositional planning problems with preferences [10], where the degree of satisfaction is computed in a (linear) cost metric, optimal plans have been generated [7].

As resources in time and space are limited, compact symbolic representations like BDDs [5] have been exploited to explore the planning state space more efficiently. They represent sets of states and actions uniquely and compactly. State sets can either be generated due to the non-deterministic structure of the problem, in which case the set represents the current belief [2]. In deterministic planning, our focus, state sets are generated during the exploration, like the set of states in a certain breadth-first layer.

One drawback of BDDs is that they refer to a fixed-sized binary state encoding. For propositional domains, a minimized multi-variate variable encoding of a planning problem can be inferred [14], but due to their structural limitations BDDs cannot handle unbounded numbers.

This work proposes a novel metric planning approach for symbolic exploration based on automata theory, which generates step- or cost-optimal plans in domains that contain numerical expressions. The language expressiveness includes linear expressions in the preconditions and effects. Each state set is represented as a set of linear constraints. Therefore, the approach can cope with infinite sets of states.

The paper is structured as follows. First, we introduce to Presburger arithmetics and present an equivalent finite state automata representation, which together with the basic operations allows to prove theorems on first-order expression. Next we consider how to represent state sets and actions in the formalism, making explicit, which fragment of PDDL the approach is designed for. Three different encoding strategies for propositions are discussed. We then formalize the automated translation process of a planning problem into Presburger formulae and automata. Next we propose exploration algorithms that are capable of finding shortest and cost-optimal plans even in infinite state spaces. Finally, we provide an implementation of the planner in *Java* and draw conclusion.

2 Presburger Arithmetics

Presburger arithmetic is the first-order theory of addition and ordering over the integers[1]. Terms in Presburger arithmetic consist of constants 0 and 1 and sums of terms. For example, $x + x + 1 + 1 + 1 = 2x + 3$

[1] Usually the Presburger arithmetic is defined on natural numbers, but negative numbers can be realized using the 2-complement.

is a term. Equations and inequations over terms are atomic formulae. The first-order theory over the natural numbers with addition are the set of all sentences, which are true in first-order predicate logic over atomic formulae. Given a successor function the Presburger arithmetic expressions can be formalized by a set of axioms and a schema for induction.

The automata encoding for Presburger formulae exploits the fact that integer numbers can be expressed by words over the alphabet $\{0, 1\}$, using the 2-complement representation. Here, the binary representation is written from left to right, such that the most significant bit is located left. The representation is not unique as the bit string can be prefixed with the an arbitrary number of zeros without changing its interpretation. In general, vectors of integer numbers can be represented as terms over $\{0, 1\}^n$. For example, the pair $(4, 13)$ generates the vectors

$$\binom{00100}{01101} = \binom{000100}{001101} = \binom{0000100}{0001101} = \dots$$

The automata representation accepts all solutions to a Presburger equation or inequation. For example, an automaton for $x - 3y = 1$ has 5 states (see Figure 1): The initial state a whose outgoing transitions interpret the sign bits correctly, b for $x - 3y = 0$, c for $x - 3y = 2$, d for $x - 3y = 1$ (accepting) and e, which the automaton takes if the word already read is not a prefix of some solution. The transition from a to b is labeled by $\binom{0}{0}$. For $\binom{0}{1}$ we go to e, since for $x = 0$ and $y = -1$, $x - 3y$ has a value of 3 which cannot become smaller when reading further bitvectors, as x cannot become smaller and y cannot become larger. For a similar reason we go to e when reading $\binom{1}{0}$. For $\binom{1}{1}$ we go to c.

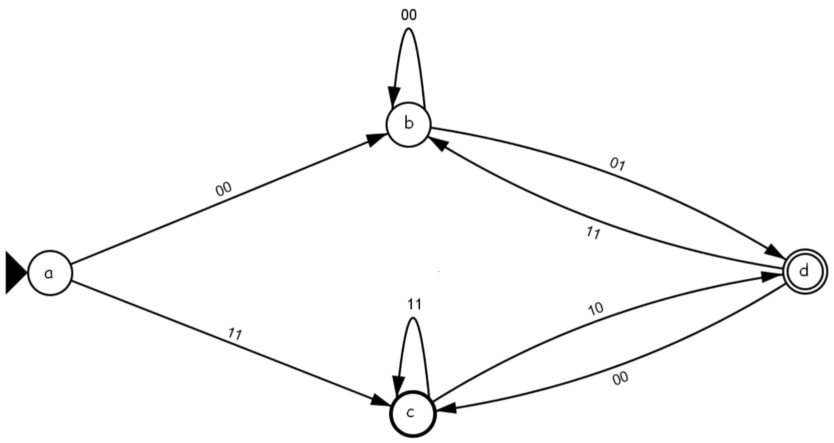


Fig. 1. Sample automaton for $x - 3y = 1$ (e omitted).

As the automaton for a Presburger formula is not unique, minimization is needed. One of the main advantages of Presburger formulae compared to most other representation formalisms in planning is that they can concisely represent infinite sets of states. For example the set of all odd numbers is represented by the equation $\exists k : x = 2k + 1$ and leads to an automaton with two states.

The conjunction of two atomic formulae is realized via intersecting the two automata. The disjunction of two atomic formulae might yield a non-deterministic automaton (that can be determinized with a possible exponential blow up). Negation requires to represent the complement of the language accepted by the automata, which (in case it is deterministic) simply toggles the acceptance condition. The projection of one variable is a linear-time operation, that also may lead to a non-deterministic automaton.

To prove the correctness of a theorem in Presburger arithmetic, one simply has to construct the automaton for it, and check, if its language is not empty. A lower bound for such decisions is triple exponential in the size of the formula [8], and an matching upper bound with automata has been given by [19].

3 Representing State Sets and Actions

This section describes how the semantics of a problem and its domain can be represented as a set of minimized finite automata. Henceforth, we assume that both the problem and the domain have been fully-instantiated. Our approach supports many of the elements of PDDL2.2 [16]. We impose the following restrictions (**R**):

1. Functions have integer values only.
2. Numerical expressions are linear.
3. Linear expressions used in preconditions or effects do not contain fractional numbers or divisions.
4. Scalars used in `scale-up` or in `scale-down` assignments are constant expressions.
5. There are no temporal actions.
6. The domain does not contain conditional effects.
7. The problem does not specify timed initial literals.

In the simplest case, a problem is given by one initial state and a set of goal states. An action can be modeled as a (finite or infinite) set of transitions within the state space, where a transition t is a pair (s_1, s_2) of a predecessor state s_1 and a successor state s_2.

3.1 Encoding the State Space

The state space induced by the domain is encoded using integer variables such that sets of states as well as sets of transitions can be characterized by Presburger formulae.

Numerical components A planning domain defines a set $\mathcal{F} = \{f_1, \ldots, f_n\}$ of zero-arity functions and a set $\mathcal{P} = \{p_1, \ldots, p_m\}$ of zero-arity predicates, with $n, m \in \mathbb{N}_0$. We map each function f_i, $i = 1, \ldots, n$, onto a variable x_i, in the following referred to as a *current state variable*. The variables x_i in a Presburger formula describe properties of the numerical components of those states that are members of the state set to be characterized by the formula. Since x_i can be assigned to arbitrary integer, we are able to encode infinite sets of states. To be capable of characterizing transition sets, too, for each current state variable x_i we create a *successor state variable* x'_i, which denotes the value of x_i in the successor state.

Propositional components Predicates can be represented in a similar straightforward way, but it may be rewarding to choose a more advanced encoding. In our work we have considered the following three encodings.

Definition 1 *(Binary Encoding) A* binary encoding *introduces a current state variable y_j and a successor state variable y'_j for $j = 1, \ldots, m$. If predicate p_j is currently* true*, then y_j is assigned to -1. The fact that p_j is currently* false *is expressed through $y_j = 0$. The current state variable and the successor state variable belonging to a predicate p are denoted by* $\mathrm{curr}(p)$ *and* $\mathrm{succ}(p)$*, respectively.*

Definition 2 *(Prime Number Encoding) Let $G_1, \ldots, G_l$ be the blocks of a partition of $\mathcal{P}$. For each $p_j \in \mathcal{P}$, let* $\mathrm{group}(p_j)$ *be the index $k \in \{1, \ldots, l\}$ for which $p_j \in G_k$ holds. A* prime number encoding *introduces a current state variable g_k and a successor state variable g'_k for each group G_k of predicates. Each $p_j \in \mathcal{P}$ is assigned to a prime number* $\mathrm{prime}(p_j)$*, which is unique within $G_{\mathrm{group}(p_j)}$. The domain of g_k and g'_k is the set of all products of prime numbers assigned to the members of G_k, where the prime factors of each product must be pairwise different. Predicate p_j is* true *if, and only if,* $\mathrm{prime}(p_j)$ *is a prime factor of $g_{\mathrm{group}(p_j)}$. The current state variable and the successor state variable belonging to a group G are denoted by* $\mathrm{curr}(G)$ *and* $\mathrm{succ}(G)$*, respectively.*

Definition 3 *(Mutex Group Encoding) Let $G_1, \ldots, G_l$ be the blocks of a partition of $\mathcal{P}$, where all predicates being the member of the same group G_k are mutual exclusive, i.e., at most one predicate in G_k can be* true *at any point in time. Again, for each $p_j \in \mathcal{P}$, let* $\mathrm{group}(p_j)$ *be the index $k \in \{1, \ldots, l\}$ for which*

$p_j \in G_k$ *holds. A* mutex group encoding *introduces a current state variable* g_k *and a successor state variable* g'_k *for each predicate group* G_k*. All members of a group* G_k *are numbered uniquely from* 1 *to* $|G_k|$*. Let* $\text{num}(p_j)$ *be the number of* p_j *within* $G_{\text{group}(p_j)}$*. The domain of* g_k *and* g'_k *is* $\{0, \dots, |G_k|\}$*. Predicate* p_j *is* true *if, and only if,* $g_{\text{group}(p_j)} = \text{num}(p_j)$*. The fact that no predicate in* G_k *is currently true is expressed through* $g_k = 0$*. The current state variable and the successor state variable belonging to a group* G *are also denoted by* $\text{curr}(G)$ *and* $\text{succ}(G)$*, respectively.*

The binary encoding straightforward, but it doesn't make any attempts to reduce the dimensionality of the state space. For prime number encodings, moderate group sizes should be selected, as products grow very fast for an increasing number of factors. The larger the amount of places at which a predicate p_j occurs, the lower $\textit{prime}(p_j)$ should be. Groups should be sized as similar as possible. For many domains it is possible to automatically identify groups of mutual exclusive predicates [14]. If such a partitioning is known, a corresponding mutex group encoding should be chosen.

3.2 Linear Expressions and Numerical Effects

Linear expressions may occur in effects as the right side (rvalue) of assignments (like `increase`), and in preconditions as operands of binary comparisons (like <=). Each linear expression is simplified such that the result is of the form $a_0 + a_1 z_1 + \cdots + a_d z_d$, where $a_0, \dots, a_d$ are non-zero integers and $z_1, \dots, z_d$ are pairwise different current state variables. We can do that because we require that linear expressions occurring in effects or preconditions do not make use of non-integer numbers or divisions. For a linear PDDL2.2 expression e let $\textit{lin}(e)$ be this simplified representation.

Let $\textit{trans}(P)$ be a Presburger formula describing a PDDL2.2 segment P. For a function f_i, linear expressions e, e_1 and e_2, a constant expression c and a relation $\prec \in \{=, <=, <, >=, >\}$, we define

- $\textit{trans}(\texttt{(assign } f_i \; e\texttt{)}) := (x'_i = \textit{lin}(e))$
- $\textit{trans}(\texttt{(increase } f_i \; e\texttt{)}) := (x'_i = x_i + \textit{lin}(e))$
- $\textit{trans}(\texttt{(decrease } f_i \; e\texttt{)}) := (x'_i = x_i - \textit{lin}(e))$
- $\textit{trans}(\texttt{(scale-up } f_i \; c\texttt{)}) := (x'_i = \textit{lin}(c) \cdot x_i)$
- $\textit{trans}(\texttt{(} \prec \; e_1 \; e_2\texttt{)}) := (\textit{lin}(e_1) \prec \textit{lin}(e_2))$

Obviously, for an assignment a the formula $\textit{trans}(a)$ describes exactly the set of transitions which correctly update the lvalue of a in dependence on the old content of the lvalue and the rvalue, evaluated before applying a. Functions other than the one updated by a can be changed arbitrarily. The translation of binary comparisons should be self-explanatory.

Due to the restriction on integers, our semantics of `scale-down` differs from the PDDL2.2 semantics:

- $\textit{trans}(\texttt{(scale-down } f_i \; c\texttt{)}) :=$
  $((x_i \geq 0 \land \exists r : (x_i = \textit{lin}(c) \cdot x'_i + r \land 0 \leq r < |\textit{lin}(c)|))\lor$
  $(x_i < 0 \land \exists r : (x_i = \textit{lin}(c) \cdot x'_i - r \land 0 \leq r < |\textit{lin}(c)|)))$

The result of a division is the non-fractional part of the result a precise division would yield. The formula is a distinction of two cases. For one state exactly one disjunct is satisfiable. If x_i is not negative, then x'_i will be equal to $\frac{\textit{lin}(c)}{|\textit{lin}(c)|} \frac{x_i - (x_i \, \textit{mod} \, |\textit{lin}(c)|)}{|\textit{lin}(c)|}$, which is equivalent to $x_i = \textit{lin}(c) \cdot x'_i + (x_i \, \textit{mod} \, |\textit{lin}(c)|)$. For $x_i < 0$, we have $x'_i = -\frac{\textit{lin}(c)}{|\textit{lin}(c)|} \frac{-x_i - ((-x_i) \, \textit{mod} \, |\textit{lin}(c)|)}{|\textit{lin}(c)|}$, which is equivalent to $x_i = \textit{lin}(c) \cdot x'_i - (x_i \, \textit{mod} \, |\textit{lin}(c)|)$.

3.3 Propositional Effects and Access to Predicates

Propositional preconditions access predicates in a reading, whereas propositional effects access predicates in a writing manner. As the translation of these accesses is also determined by the encoding of the logical state space, we will discuss special aspects of the translation process for each encoding introduced above.

For $p \in \mathcal{P}$, let *read*(p) be a Presburger formula which is *true* in a state s, if, and only if, p is *true* in s according to the state encoding.

If one pair of Presburger variables describes a group of predicates instead of a single predicate, then the computation of the value for the successor state variable may require complete knowledge about all updates to some member of the group performed by an effect. For this reason, the translation of an effect is more complex than the translation of a read access. For $D \subseteq \mathcal{P}$ and $A \subseteq \mathcal{P}$, *write*(D, A) is a Presburger formula, which is *true* for a transition $t = (s_1, s_2)$, if, and only if,

- all predicates $p \in D$ are *false* in the successor state s_2,
- all predicates $p \in A$ are *true* in s_2, and
- all predicates $p \in \mathcal{P} \backslash (D \cup A)$ are *true* in s_2 (if p is *true* in s_1).

For $D \cap A = \emptyset$, *write*(D, A) is undefined. The formula *write*(D, A) can be used to encode the logical part of an effect by choosing for A the set of all predicates that occur in a positive literal and choosing for D the set of all predicates that occur in a negative literal, but not in a positive literal (PDDL2.2 first executes the deletions, and then executes the addings). Note that $(D, A) = (\emptyset, \emptyset)$ is a valid choice.

The initial state could be described by a goal description, which is translated into a Presburger formula that reads the predicates, but as we will see later, negations may cause the acceptance of invalid states. On the one hand, the invalid states could be removed by building the conjunct of the formula with another formula that characterizes the whole state space. On the other hand, it is possible to describe an initial state by a much more elegant formula, because due to the fact that all predicate valuations are known, the formula needs not to read the current state variables. This motivates the definition of *init*. For a set $T \subseteq \mathcal{P}$, *init*(T) is a Presburger formula, which accepts only one logical state, namely the state in which, according to the state encoding, $p \in \mathcal{P}$ is interpreted as *true* if, and only if, $p \in T$.

In the following, we provide definitions of *read*, *write* and *init* for all three encodings.

Binary Encoding Per construction, a predicate $p \in \mathcal{P}$ is to be interpreted as being *true*, if, and only if, *curr*(p) is equal to -1. Obviously, for the following definitions of $read_{bin}$, $write_{bin}$ and $init_{bin}$, the axioms of *read*, *write* and init hold with respect to the semantics of the binary encoding:

$$read_{bin}(p) := (curr(p) = -1)$$
$$write_{bin}(D, A) := (\textstyle\bigwedge_{d \in D}(\text{succ}(d) = 0) \wedge$$
$$\bigwedge_{a \in A} (succ(a) = -1) \wedge \bigwedge_{p \in \mathcal{P} \backslash (D \cup A)} (succ(d) = curr(d)))$$
$$init_{bin}(T) := (\bigwedge_{t \in T} (curr(t) = -1) \wedge (\bigwedge_{p \in \mathcal{P} \backslash T} (curr(p) = 0))$$

Prime Number Encoding To read a predicate $p \in \mathcal{P}$ in prime number encoding, it is sufficient to check, if *prime*(p) is a prime factor of $curr(G_{group(p)})$. Thus,

$$read_{prime}(p) := (\exists k : curr(G_{group(p)}) = prime(p) \cdot k)$$

is a sound definition of $read$ with respect to the prime number encoding.

A formula describing the propositional part of an effect must take into account two aspects. First, if p is added by the effect, then the value of $curr(G_{group(p)})$ is to be multiplied with $prime(p_j)$ only if prime(p_j) is not already a prime factor of $curr(G_{group(p)})$. Similarly, if p is deleted by the effect, then the value of $curr(G_{group(p)})$ is to be divided by $prime(p_j)$ only if $prime(p_j)$ is a prime factor of $curr(G_{group(p)})$. Second, it must update each group variable according to all updates that affect some member of the group. If the formula simply was a conjunction of subformulae, where each subformula describes the value of $succ(G_{group(p)})$ for exactly one $p \in \mathcal{P}$ that is updated by the effect, then the formula might be contradictious when a group is affected by more than one update.

We first build formulae, which characterize updates to single predicates, ignoring what happens to all group variables other than the one affected by the update. For a predicate $p \in \mathcal{P}$ and two variables h_0 and h_1, we define

$\mathit{add}(p, h_0, h_1) := (((\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \land h_1 = h_0) \lor (\neg(\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \land h_1 = \mathit{prime}(p) \cdot h_0))$

$\mathit{del}(p, h_0, h_1) := (((\exists k : \mathit{curr}(G_{group(p)}) = \mathit{prime}(p) \cdot k) \land \mathit{prime}(p) \cdot h_1 = h_0) \lor (\neg(\exists k : \mathit{curr}(G_{\mathit{group}(p)}) = \mathit{prime}(p) \cdot k) \land h_1 = h_0))$.

The formula $\mathit{add}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ is *true* for a transition $t = (s_1, s_2)$ if either p is *true* in s_1 and the variable representing the group p belongs to remains unchanged so that p is also *true* in s_2, or p is $false$ in s_1 and the variable representing the group p belongs to is multiplied with $\mathit{prime}(p)$, such that p is *true* in s_2. Thus, $\mathit{add}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ correctly characterizes the adding of p ignoring all other group variables. An analogous argumentation shows that $\mathit{del}(p, \mathit{curr}(G_{\mathit{group}(p)}), \mathit{succ}(G_{\mathit{group}(p)}))$ correctly characterizes the deletion of p ignoring all other group variables.

The definitions of *add* and *del* cover the first aspect. The parameters h_0 and h_1 enable a formula that represents the whole effect to cover the second aspect, too. In order to avoid contradictions, an update to p does not write its result to $\mathit{succ}(G_{\mathit{group}(p)})$, but into an own auxiliary variable, which becomes the input for the next update. The first auxiliary variable is identical to $\mathit{curr}(G_{\mathit{group}(p)})$ and becomes the input for the first update, the last auxiliary variable is the output of the last update and is identical to $\mathit{succ}(G_{\mathit{group}(p)})$. By existentially quantifying the auxiliary variables away, we obtain the formula we were looking for. For $i = 1, \dots, l$, be $G_i \cap D = \{d_{i,1}, \dots, d_{i,l_i^D}\}$ and $G_i \cap A = \{a_{i,1}, \dots, a_{i,l_i^A}\}$. Now we define

$$\begin{aligned}
\mathit{write}_{\mathit{prime}}(D, A) := (&\textstyle\bigwedge_{1 \le k \le l} \exists h_0 : \dots \exists h_{|G_k|} : \\
&(h_0 = \mathit{curr}(G_k) \land \mathit{succ}(G_k) = h_{l_k^D + l_k^A} \\
&\land \textstyle\bigwedge_{1 \le j \le l_k^D} \mathit{del}(d_{k,j}, h_{j-1}, h_j) \\
&\land \textstyle\bigwedge_{1 \le j \le l_k^A} \mathit{add}(a_{k,j}, h_{l_k^D + j - 1}, h_{l_k^D + j})))
\end{aligned}$$

$\mathit{init}_{\mathit{prime}}(T) := (\bigwedge_{1 \le k \le l} \mathit{curr}(G_i) = \prod_{t \in T \cap G_k} \mathit{prime}(t))$.

Mutex Group Encoding To check, if a predicate p is *true* in a mutex group encoding it suffices to compare the appropriate group variable to $\mathit{num}(p)$:

$\mathit{read}_{\mathit{mutex}}(p) := (\mathit{curr}(G_{\mathit{group}(p)}) = \mathit{num}(p))$.

If an effect adds $p \in \mathcal{P}$, $\mathit{num}(p)$ will be the correct new value for $\mathit{curr}(G_{\mathit{group}(p)})$. Now we assume that no member of G_i is affected by an add. If $p \in G_i$ is deleted by the effect while p is currently *true*, then we must choose $\mathit{succ}(G_i) = 0$, otherwise $\mathit{succ}(G_i) = \mathit{curr}(G_i)$ is appropriate. The formula $\mathit{delset}(G, D)$ correctly updates the variable representing a group G, where no $p \in G$ is affected by an add, and D is the set of all deletes affecting some member of G. Now this subsection can be concluded with the definition of $\mathit{write}_{\mathit{mutex}}$ and $\mathit{init}_{\mathit{mutex}}$.

$\mathit{delset}(G, D) := (((\bigvee_{d \in D} \mathit{curr}(G) = \mathit{num}(d)) \land \mathit{succ}(G) = 0) \lor ((\bigwedge_{d \in D} \mathit{curr}(G) \neq \mathit{num}(d)) \land \mathit{succ}(G) = \mathit{curr}(G)))$

$\mathit{write}_{\mathit{mutex}}(D, A) := ((\bigwedge_{G \in \{G_1, \dots, G_l\} : A \cap G = \emptyset} \mathit{delset}(G, G \cap D)) \land (\bigwedge_{G \in \{G_1, \dots, G_l\} : A \cap G = \{p\}} \mathit{curr}(G) = \mathit{num}(p)))$

$\mathit{init}_{\mathit{mutex}}(T) := ((\bigwedge_{G \in \{G_1, \dots, G_l\} : T \cap G = \{p\}} \mathit{curr}(G) = \mathit{num}(p)) \land (\bigwedge_{G \in \{G_1, \dots, G_l\} : T \cap G = \emptyset} \mathit{curr}(G) = 0))$.

3.4 Derived Predicates

A grounded derived predicate is of the form `(:derived` $(p_j)\,\Phi$`)`, where p_j is a predicate and Φ is a goal description. Φ may contain other predicates, which may be derived themselves. For two derived predicates p and q be $p \prec q$ if, and only if, q depends on p, that is, there is a sequence $z_0, \dots, z_l$ of derived predicates, such that $z_0 = p$, for each z_k, $1 \leq k \leq l$ there is a `:derived` definition, which derives z_k from a goal description containing z_{k-1}, and $z_l = q$. $\prec$ is reflexive and transitive. As the set of all `:derived` definitions of a domain must not contain cyclic definitions, $\prec$ is also antisymmetric. Thus, $\prec$ is a partial order.

Now it is clear that a `:derived` definition can be seen as a macro for its goal description with the predicate it derives as its name. Our aim is to remove all derived predicates from the domain by substituting each occurence of a derived predicate q by a goal description, which is equivalent to q. In the domain, we first replace all `:derived` definitions $d_1, \ldots, d_z$ deriving the same predicate q by one new `:derived` definition deriving q from the disjunction of the goal descriptions of $d_1, \ldots, d_z$. Afterwards we topologically sort all derived predicates according to $\prec$ and obtain a linear order $q_1 \prec \ldots \prec q_c$. Predicate q_1 is one of the most independent derived predicates, i.e., q_1 is independent from all other derived predicates. For $k = 1, \ldots, c$, all other derived predicates predicate q_k depends on lay within $\{q_1, \ldots, q_{k-1}\}$. For $k = 1, \ldots, c$, in the goal description of the `:derived` definition belonging to q_k we substitute each occurrence of a derived predicate q by the goal description of the `:derived` definition belonging to q (which is already free of derived predicates). When this has been done, then each occurrence of a derived predicate q in a precondition of an action or in the goal specification of the problem is substituted by the goal description of the `:derived` definition belonging to q.

3.5 Goal Descriptions, Effects and Initializations

We extend our translation function *trans* to goal descriptions. For a predicate $p \in \mathcal{P}$ and goal descriptions $d_1, \ldots, d_z$ the following definitions are given:

- $\mathit{trans}(p) := \mathit{read}(p)$
- $\mathit{trans}(\texttt{(not }d_1\texttt{)}) := (\neg \mathit{trans}(d_1))$
- $\mathit{trans}(\texttt{(and }d_1 \ldots d_z\texttt{)}) := (\bigwedge_{1 \leq k \leq z} \mathit{trans}(d_k))$
- $\mathit{trans}(\texttt{(or }d_1 \ldots d_z\texttt{)}) := (\bigvee_{1 \leq k \leq z} \mathit{trans}(d_k))$
- $\mathit{trans}(\texttt{(imply }d_1\, d_2\texttt{)}) := (\neg \mathit{trans}(d_1) \vee \mathit{trans}(d_2))$

Goal descriptions only read current state variables.

A PDDL2.2 effect is a conjunction of numerical effects, where literals are either positive or negative. For an arbitrary effect e, let $\mathit{pos}(e)$, $\mathit{neg}(e)$ and $\mathit{fluent}(e)$ denote the set of predicates occurring in a positive literal, the set of predicates occurring in a negative literal and the set of numerical effects contained in e, respectively. Let $\mathit{pres}(e)$ be the set of all functions $f \in \mathcal{F}$ that are not affected by some $a \in \mathit{fluent}(e)$. The translation of e is

- $\mathit{trans}(e) := (\mathit{write}(\mathit{neg}(e) \backslash \mathit{pos}(e), \mathit{pos}(e)) \wedge (\bigwedge_{a \in \mathit{fluent}(e)} \mathit{trans}(a)) \wedge (\bigwedge_{f \in \mathit{pres}(e)} \mathit{succ}(f) = \mathit{curr}(f)))$

An initialization specified by a problem can be seen is a conjunction of literals and equations. Literals not mentioned in the `:init` section are assumed to be *false* (closed world assumption). Here, functions that are not explicitly initialized are initialized with 0. For an arbitrary `:init` section *start*, let $\mathit{pos}(\mathit{start})$, $\mathit{equat}(\mathit{start})$ and $\mathit{uinit}(\mathit{start})$ denote the set of predicates occurring in a positive literal, the set of equations and the set of not explicitly initialized functions $f \in \mathcal{F}$, respectively. Then $\mathit{trans}(\mathit{start})$ can be defined as

- $\mathit{trans}(\mathit{start}) := (\mathit{init}(\mathit{pos}(\mathit{start})) \wedge (\bigwedge_{u \in \mathit{equat}(\mathit{start})} \mathit{trans}(u)) \wedge (\bigwedge_{v \in \mathit{uinit}(\mathit{start})} \mathit{curr}(v) = 0))$

3.6 Domains and Problems

For an action *act* with a precondition *pre* and an effect *post*, the translation is defined through

- $\mathit{trans}(\mathit{act}) := (\mathit{trans}(\mathit{pre}) \wedge \mathit{trans}(\mathit{post}))$

The transition relation defined by $\mathit{trans}(\mathit{act})$ contains a transition $t = (s_1, s_2)$ if, and only if, the predecessor state s_1 fulfills the precondition, and the successor state s_2 is the result of applying *post* to s_1.

The translation of a problem and its domain consists of the following steps:

1. Substitution of derived predicates in all conditions, preconditions and in the goal specification of the problem
2. Selection of a logical state encoding
3. Translation of the actions
4. Translation of the `:init` section

How a `:metric` specification of a problem is translated, will be explained in the planning section.

3.7 Building Automata for the Formulae

The next step is to create a minimized deterministic finite state automaton (DFA)

$$A_\varphi = (Q_\varphi, \Sigma_{trans}, \delta_\varphi, init_\varphi, F_\varphi)$$

for each formula φ, using the algorithm described in [20], with Q_φ being the state set, Σ_{trans} being the alphabet, $\delta_\varphi : Q \times \Sigma_{trans} \rightarrow Q$ being the transition relation, $init_\varphi \in Q_\varphi$ being the initial state and $F_\varphi \subseteq Q$ being the set of accepting states. A_φ recognizes the set of all solutions to φ. The numerical state space is represented by n current state variables and n successor state variables. Let l be the number of pairs consisting of a current state variable and a successor state variable, which represent the logical state space. The bitvectors of the alphabet $\Sigma_{trans} = \{0,1\}^{2(n+l)}$ have one component for each variable. We assign all current state variables to the components with the indices $1, 3, \ldots, 2(n+l)-1$, whereas all successor state variables are assigned to the components with the indices $2, 4, \ldots, 2(n+l)$. For the initial state and the goal state set, automata are constructed, too. Since these automata recognize sets of planning states rather than sets of transitions from one planning state to another, they use the alphabet $\Sigma_{state} := \{0,1\}^{n+l}$.

The entire representation of the domain and the problem is denoted by $\mathcal{R}$, with $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, where $\mathcal{A}$ is the set of all automata constructed for the actions, $\mathcal{I}$ is the automaton constructed for the initial state, and $\mathcal{G}$ is the automaton constructed for the goal state set.

4 Planning

For deterministic finite automata A_1 and A_2, $A_1 \cap A_2$ ($A_1 \cup A_2$) denote the minimized deterministic finite automata recognizing the intersection (union) of the languages $L(A_1)$ and $L(A_2)$ recognized by A_1 and A_2; $A_1 \backslash A_2$ be the minimal DFA recognizing $L(A_1) \backslash L(A_2)$. $A_1 = A_2$ hold if, and only if, $L(A_1) = L(A_2)$. For a DFA $A = (Q, \Sigma_{trans}, \delta, init, F)$ the *projection* $proj_{curr}(A)$ is defined as the result of determinizing and minimizing

$$A' := (Q, \Sigma_{state}, \delta', init, F)$$

with $\delta'(q, (a_1, \ldots, a_{n+l})) := \{r \in Q \mid \exists b_1, \ldots, b_{n+l} : \delta(q, (a_1, b_1, \ldots, a_n, b_n)) = r\}$, whereas the *projection* $proj_{succ}(A)$ is defined as the result of determinizing and minimizing

$$A' := (Q, \Sigma_{state}, \delta', init, F)$$

with $\delta'(q, (b_1, \ldots, b_{n+l})) := \{r \in Q \mid \exists a_1, \ldots, a_{n+l} : \delta(q, (a_1, b_1, \ldots, a_n, b_n)) = r\}$. In both cases, before determinization and minimization is done, A' is modified by another algorithm to ensure that all representations of a solution are accepted (see ...). The alphabet can be restored using the *expansions* exp_{curr} and exp_{succ}. The set $exp_{curr}(A)$ is obtained by replacing the transition relation $\delta'' : Q \times \Sigma_{state} \rightarrow Q$ of A by $\delta''' : Q \times \Sigma_{trans} \rightarrow Q$, defined through $\delta'''(q, (a_1, b_1, \ldots, a_{n+l}, b_{n+l})) := \delta''(q, (a_1, \ldots, a_{n+l}))$. The set exp_{succ} is defined analogously with $\delta'''(q, (a_1, b_1, \ldots, a_{n+l}, b_{n+l})) := \delta''(q, (b_1, \ldots, b_{n+l}))$. The minimal DFA A with $L(A) = \emptyset$ using the alphabet Σ_{state} is denoted by $\perp$ (the complement of $\perp$ is $\top$).

4.1 Breadth First Search

Given a representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, we are looking for a shortest sequential plan, which transforms the initial planning state recognized by $\mathcal{I}$ into a goal state $G \in L(\mathcal{G})$. The planning graph induced by $\mathcal{R}$ is

Algorithm 1 Breadth-First-Search

```
Input: Representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$
Output: Shortest sequential plan or 'no plan'
  $stack := \emptyset$; $layer := reached := \mathcal{I}$; $stack{-}{>}push(\mathcal{I})$
  while $layer \cap \mathcal{G} = \perp$ do
    $layer' := \perp$
    for all $A \in \mathcal{A}$ do
      $trans := exp_{curr}(layer) \cap A$
      $layer' := layer' \cup proj_{succ}(trans)$
    $layer := layer' \setminus reached$
    if $layer = \perp$ then
      return 'no plan'
    else
      $reached := reached \cup layer$
      $stack{-}{>}push(layer)$
  $stack{-}{>}pop()$
  $stack{-}{>}push(layer \cap \mathcal{G})$
  return $Extract\text{-}Plan(\mathcal{R}, stack)$
```

infinite, directed and unweighted, but the outdegree of each node is finite; it cannot exceed $|\mathcal{A}|$. A shortest sequential plan corresponds to a shortest path from $L(\mathcal{I})$ to a $G \in L(\mathcal{G})$. In the following, we identify automata A with their languages $L(A)$.

Obviously, a breadth first search that starts from $\mathcal{I}$ is capable of finding a shortest path (cf. Algorithm 1). For each action whose precondition is *true* in the initial state, we compute the successor state that results from applying the action to the initial state. The set of all these successor states, without the initial state, forms the next layer. Because DFAs are closed under union, it is possible to represent the layer by one single DFA. Be $L_0 := \mathcal{I}$. When the exploration has completed layer $L_i (i \geq 0)$, layer L_{i+1} is obtained by doing the following for all $A \in \mathcal{A}$. First, $B := exp_{curr}(L_i) \cap A$ is computed. L_i is not compatible to A since L_i is a set of planning states, whereas A is a set of transitions. $exp_{curr}(L_i)$ converts L_i to the set of transitions from any planning state in L_i to any planning state. B covers exactly those transitions (s_1, s_2) with s_1 in L_i, s_1 fulfilling the precondition of the action the automaton A corresponds to, and s_2 being the successor state that results from executing the action in s_1. The projection $C := proj_{succ}(B)$ only recognizes the successor states. L_{i+1} becomes the union of all C computed for the $A \in \mathcal{A}$ minus the set of all states reached by the exploration so far. By intersecting $L_i (i = 0, \ldots)$ with $\mathcal{G}$ before computing L_{i+1}, we check whether or not we have already found a goal state. All L_i are stored, as they are needed for the reconstruction of a plan. If $L_{i+1} = \perp$, then that part of the state space, which is reachable from the initial state has been completely explored without finding a goal state. In this situation, we have proved that there is no solution. Exploration does not terminate if infinite many planning states are reachable from the initial state while there is no solution.

4.2 Reconstruction of a Solution

In situations where algorithm 1 reaches a nonempty subset $G \subseteq \mathcal{G}$, we will mainly be interested in a plan, which transforms the initial state into one of the members of G. If the length z of a shortest plan is at least 1, then algorithm 1 creates a stack with L_0 on its bottom and $G = L_z \cap \mathcal{G}$ on its top (otherwise the stack will only contain $L_0 \cap \mathcal{G}$, and plan extraction becomes trivial). The stack may induce various plans since it results from executing in each $s \in L_i$ all actions, which are executable in s. Plan extraction cannot be done in forward direction like breadth first search, because the stack does not provide any information about, which action is selected next in order to reach a goal. For this reason, plan extraction works in backward direction, starting from G. The algorithm searches for an action *act* which transforms a subset $S \subseteq L_{z-1}$ into a subset of G. Action *act* will be the last action executed by our plan. Afterwards, if $z \geq 2$, then an action to be executed just before *act* is determined by finding an action that transforms a subset $S' \subseteq L_{z-2}$ into a subset of S. More precisely, plan extraction considers pairs (L_{i-1}, L_i) of layers and pairs (S_{i-1}, S_i) with $S_{i-1} \subseteq L_{i-1}$ and $S_i \subseteq L_i$ $(i = z, \ldots, 1)$. We choose $S_z := G$. S_{i-1} is obtained from S_i by first finding an automaton $A_i \in \mathcal{A}$ with $\emptyset \neq B_{i-1} := \mathit{exp}_{\mathit{curr}}(L_{i-1}) \cap \mathit{exp}_{\mathit{succ}}(S_i) \cap A_i$. $\mathit{exp}_{\mathit{curr}}(L_{i-1})$ is the set of all transitions $t = (s_1, s_2)$ with $s_1 \in L_{i-1}$ and s_2 being an arbitrary planning state. $\mathit{exp}_{\mathit{succ}}(S_i)$ is the set of all transitions $t = (s_1, s_2)$ with s_1 being an arbitrary planning state and $s_2 \in S_i$. Thus, B_{i-1} is the set of all transitions $t = (s_1, s_2)$ with $s_1 \in L_{i-1}$, $s_2 \in S_i$ and $t \in A_i$. In $S_{i-1} := \mathit{proj}_{\mathit{curr}}(B_{i-1})$ only the current states remain. S_{i-1} is the set of all predecessors of S_i under $\mathit{action}(A_i)$, the action that corresponds to A_i. $\mathit{action}(A_i)$ will be the i-th action executed by the plan. Algorithm 2 summarizes the ideas described. The validity of the plan extracted by the algorithm results from $S_0 = \mathcal{I}$, $S_z = \mathcal{G}$ and the fact that, for $i = 1, \ldots, z$, $\mathit{action}(A_i)$ transforms a state $s_{i-1} \in S_{i-1}$ into a state $s_i \in S_i$.

Algorithm 2 Extract-Plan

```
Input: Representation $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G})$, a stack stack
Output: Sequential plan of length $|stack| - 1$
  $plan := []; post := stack\text{->}pop()$
  while $stack \neq \emptyset$ do
    $pre := stack\text{->}pop(); post' := \perp$
    while $post' = \perp$ do
      $a := \mathbf{next} A \in \mathcal{A}$
      $post' := exp_{curr}(pre) \cap exp_{succ}(post) \cap A$
      $post' := proj_{curr}(post')$
    $post := post'$
    $plan := [action(a)].plan$
  return plan
```

4.3 Solving Metric Problems

An extension of algorithm 1 is also able to solve metric problems (cf. Algorithm 3). For metric problems, the representation $\mathcal{R}$ includes an additional component $\mathcal{M}$ which represents the metric, i.e., $\mathcal{R}$ now is a 4-tuple $(\mathcal{A}, \mathcal{I}, \mathcal{G}, \mathcal{M})$. $\mathcal{M}$ is a linear expression $a_1 z_1 + \cdots + a_d z_d$ with pairwise different variables $z_i \in \{x_1, \ldots, x_n\}$, $a_1, \ldots, a_d \in \mathbb{Z} \backslash \{0\}$ and $d \geq 1$. Since we permit the use of rational numbers and divisions in metrics, the original metric must be scaled by a suitable factor. If the original metric is to be maximized, we turn the maximization problem into a minimization problem by multiplying the metric with -1. If a summand $c \cdot \texttt{total-time}$ occurs in the simplified scaled metric, then this summand will be removed, and we keep c in memory for later use (setting $c := 0$ if no such summand occurs). Minimizing $\mathcal{M}$ is equivalent to optimizing the original metric, but $\mathcal{M}$ is compatible to Presburger arithmetic. Given a value $\mathcal{M}$ takes, it is easy to calculate the corresponding value of the original metric. For $\mathit{upperValue} \in \mathbb{Z}$ let $\mathit{boundAut}(\mathcal{M}, \mathit{upperValue})$ be the minimal DFA for $\mathcal{M} \leq \mathit{upperValue}$, using the alphabet Σ_{state}. Define $\mathit{boundAut}(\mathcal{M}, \infty) := \top$.

Again, the planning graph is explored using breadth first search. The `while` loop now uses true as its guard because hitting the goal set does not exclude that further hits in following layers would lead to a smaller value for the metric. The algorithm maintains the value *bound* the metric $\mathcal{M}$ takes for the best goal state found so far, with initially $\mathit{bound} := \infty$.

As long as the exploration has not found a goal state yet, each layer L_i is intersected with $\mathcal{G}$. If the intersection B is not empty, then B may contain various goal states, where each goal state makes the metric take a certain value. We are interested in the smallest value the metric takes for some $s \in B$ (a smallest value exists since B is finite). For a finite set $S \neq \emptyset$ of planning states and a metric $\mathcal{M}$, let $\mathit{minimalValue}(\mathcal{M}, S)$ be the smallest value $\mathcal{M}$ takes for some state in S. To compute $\mathit{minimalValue}(\mathcal{M}, S)$, first an upper bound u is found by evaluating the metric for an arbitrary $s \in S$ and then intersecting S with $\mathit{boundAut}(\mathcal{M}, u - e \cdot 2^i)$ for $i = 0, 1, 2, 3, \ldots$ and some integer $e \geq 1$, until the intersection is empty. The last $u - e \cdot 2^i$ is a lower bound for $\mathit{minimalValue}(\mathcal{M}, S)$, and the exact value can be determined using a binary search approach. We update bound with $\mathit{minimalValue}(\mathcal{M}, B)$ and copy the stack, since solution reconstruction will need the stack as it is now, if exploration terminates without finding a better goal state.

After the goal state has been hit at least once, whenever a layer L_i is completed, the algorithm computes $B := L_i \cap \mathcal{G} \cap \mathit{boundAut}(\mathcal{M}, \mathit{bound} - \mathit{timeterm} - 1)$. We subtract 1 because we would like to know if L_i contains a goal state which reaches a metric value that is smaller than the one we are already able to realize. But this alone would not be correct, as the metric actually contains the summand $c \cdot \texttt{total-time}$, which has increased by $c \cdot (i - j)$ since the last update to *bound*, if the last update was done when examining layer L_j. The algorithm guarantees that, at this point of computation, *timeterm* is equal to $c \cdot (i - j)$. Thus, $\mathit{bound} - \mathit{timeterm}$ is the value $\mathcal{M}$ must take now in order to let the original metric take the same value as before. If B is not empty, the algorithm proceeds as described above. Exploration terminates if, and only if, $L_i = \emptyset$ for some i. At any time, it is possible to stop the algorithm and extract a plan from the stack saved when the latest update to *bound* was performed.

Algorithm 3 Metric-Breadth-First-Search

```
Input: $\mathcal{R} = (\mathcal{A}, \mathcal{I}, \mathcal{G}, \mathcal{M})$, coefficient $c$ of total-time
Output: Shortest seq. plan minimizing $\mathcal{M}$ or 'no plan'
  $stack := stack' := \emptyset$; $layer := reached := \mathcal{I}$
  $bound := \infty$; $timeterm := 0$; $stack{-}{>}push(\mathcal{I})$
  while true do
    $inter := layer \cap \mathcal{G} \cap boundAut(\mathcal{M}, bound - timeterm - 1)$
    if $inter \neq \bot$ then
      $bound := minimalValue(\mathcal{M}, inter)$; $timeterm := 0$
      $stack' := stack$; $stack'{-}{>}pop()$
      $stack'{-}{>}push(inter \cap boundAut(\mathcal{M}, bound))$
    $timeterm := timeterm + c$
    for all $A \in \mathcal{A}$ do
      $trans := exp_{curr}(layer) \cap A$
      $layer' := layer' \cup proj_{succ}(trans)$
    $layer := layer' \setminus reached$
    if $layer = \bot$ then
      if $bound = \infty$ then
        return 'no plan'
      else
        return Extract-Plan$((\mathcal{A}, \mathcal{I}, \mathcal{G}), stack')$
    else
      $reached := reached \cup layer'$
      $stack{-}{>}push(layer)$
```

5 The Triple-A Planning System (TTAPS)

We have implemented our approach in *Java*, adapting an already existing automata library. Our tool TTAPS (for *The Triple-A Planning System*) has full PDDL2.2 functionality up to the restrictions $\mathbf{R1} - \mathbf{R7}$, but its performance has not yet left the status of a prototype.

TTAPS comes as an executable *Java* archive, including sample PDDL-files Automata currently use bitvector alphabets as described above. For each pair (q_1, q_2) of automaton states q_1 and q_2 with at least one transition from q_1 to q_2, there is exactly one object that represents all transitions from q_1 to q_2. A BDD characterizes the set of labels that belong to one of those transitions. Our BDDs do not share structures, so there is space for optimization, too. But another, very promising approach is to make $\{0, 1\}$ the alphabet of all automata, and let automata read each vector serially instead of parallelism. This approach can also be combined with another encoding of integers leading to asymptotic smaller automata in certain cases [1].

In the diploma thesis of Björn Borowski, example domains and problems are provided that have designed as test instances the current planner prototype is able to solve.

6 Conclusion

The paper proposes an optimal planner in Presburger arithmetic for metric problems. It contributes to the area of planning via model checking [12]. The expressiveness covers a large part of current PDDL. The memory problems that are inherent to representing large sets of states are addressed by using a symbolic representation based on minimized (and therefore unique) automata. As a feature, the approach can deal with infinite state sets.

The paper is the first report on optimal plan finding in infinite state numerical domains. It implements a planner on top of the existing library AAA[2]. All other related planners are non-optimal and provide no guarantee on optimal plan finding. Previous work on metric action planning is based on representing the states explicitly. For example, Metric-FF [15] is a forward-state heuristic search planner that uses an involved Graphplan inspired heuristics to guide the search process. LPG [11] is a local search planner that gradually improves an initial invalid plan resolving conflicts by inserting and deleting actions, until it eventually becomes sound. SGPlan [6] also uses local search and partitions the overall problem into tightly connected subproblems. Until a plan is found, the planner first resolves local constraint violations and then global constraint violations.

There is recent work on SAT encodings of state-space planning in numeric domains [17]. The approach proposed in this paper first infers a finite state encoding using an approximate fix-point analysis, and then uses the finite-state variables to infer the domain. The approach is fast, provides optimality guarantees, and applies well to current benchmark domains. However, as it relies on finite domain encodings of numbers, it is less general than the approach presented here.

In the diploma thesis of Björn Borowski example cases, correctness proofs and experimental results were provided as well as an extension to Dijkstra search.

In future work we will work on performance improvements in the exploration and might try tackling real numbers using the results of [4].

References

1. C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. *Int. J. Found. Comput. Sci.*, 14(4):605–624, 2003.
2. P. Bertoli, A. Cimatti, and M. Roveri. Heuristic search symbolic model checking = efficient conformant planning. In *IJCAI*, pages 467–472, 2001.
3. A. Blum and M. L. Furst. Fast planning through planning graph analysis. In *IJCAI*, pages 1636–1642, 1995.
4. B. Boigelot, S. Jodogne, and P. Wolper. An effective decision procedure for linear arithmetic over the integers and reals. *ACM Transaction on Computational Logic*, 6:614–633, 2005.
5. R. E. Bryant. Symbolic manipulation of boolean functions using a graphical representation. In *DAC*, pages 688–694, 1985.
6. Y. Chen and B. W. Wah. Subgoal partitioning and resolution in planning. In *Proceedings of the International Planning Competition*, 2004.
7. S. Edelkamp. Cost-optimal symbolic planning with state trajectory and preference constraints. In *ECAI*, 2006.
8. M. Fischer and M. Rabin. Super-exponential complexity of Presburger arithmetic. In *SIAM-AMS*, 1974.
9. M. Fox and D. Long. PDDL2.1: An extension to PDDL for expressing temporal planning domains. *Journal of Artificial Intelligence Research*, 20:61–124, 2003.
10. A. Gerevini and D. Long. Plan constraints and preferences in PDDL3. Technical report, Department of Electronics for Automation, University of Brescia, 2005.
11. A. Gerevini and I. Serina. Fast plan adaptation through planning graphs: Local and systematic search techniques. In *AIPS*, pages 112–121, 2000.
12. F. Giunchiglia and P. Traverso. Planning as model checking. In *ECP*, pages 1–19, 1999.
13. M. Helmert. Decidability and undecidability results for planning with numerical state variables. In *AIPS*, pages 303–312, 2002.
14. M. Helmert. A planning heuristic based on causal graph analysis. In *ICAPS*, pages 161–170, 2004.
15. J. Hoffmann. The Metric FF planning system: Translating "Ignoring the delete list" to numerical state variables. *Journal of Artificial Intelligence Research*, 20:291–341, 2003.

[2] texttthttp://sourceforge.net/projects/triple-a

16. J. Hoffmann and S. Edelkamp. The deterministic part of IPC-4: An overview. *Journal of Artificial Intelligence Research*, 24:519–579, 2005.
17. J. Hoffmann, C. P. Gomes, B. Selman, and H. A. Kautz. Sat encodings of state-space reachability problems in numeric domains. In *IJCAI*, pages 1918–1923, 2007.
18. H. Kautz and B. Selman. Pushing the envelope: Planning propositional logic, and stochastic search. In *ECAI*, pages 1194–1201, 1996.
19. F. Klaedtke. On the automata size for Presburger arithmetic. In *LICS*, pages 110–119, 2004.
20. P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *TACAS*, pages 1–19, 2000.
21. R. Zhou and E. Hansen. Breadth-first heuristic search. In *ICAPS*, pages 92–100, 2004.

Externalizing the Multiple Sequence Alignment Problem with Affine Gap Costs

Stefan Edelkamp and Peter Kissmann

Computer Science Department
Otto-Hahn-Str. 14
University of Dortmund

Abstract. Multiple sequence alignment (MSA) is a problem in computational biology with the goal to discover similarities between DNA or protein sequences. One problem in larger MSA instances is that the search exhausts main memory. This paper applies disk-based heuristic search and further compression schemes to solve MSA benchmarks. We extend iterative-deepening dynamic programming, a hybrid of dynamic programming and IDA*, for which optimal alignments with respect to similarity metrics and affine gap cost are computed. We achieve considerable savings of main memory with an acceptable time overhead. By scaling buffer sizes, the space-time trade-off can be adapted to existing resources.

1 Introduction

Computational biology or *bioinformatics* is dedicated to the discovery and implementation of algorithms that facilitate the understanding of biological processes [6]. The field encompasses different areas such as building evolutionary trees and operating on molecular sequence data. We observe a tight analogy between biological and computational processes. For example generating test sequences for a computational system relates to generating experiments for a biological system. On the other and many biochemical phenomena reduce to the interaction between defined sequences.

We selected one problem in which heuristics have been applied for increasing the efficiency of the exploration. The sequence alignment problem that originates in computational biology plays a rising role as a testbet for search algorithms in AI. It has been denoted as the *Holy Grail* of algorithms on string, trees and sequences [6]. DNA (or protein) sequences are compared and made as similar as possible by introducing gaps. The problem is of interest, as it allows to detect evolutionary developments and relationships, and to determine the function of certain parts of the DNA. For example, during the transition from living in the water to living on land, the DNA part that is responsible for breathing should have been inserted.

For DNA sequence alignment we have the alphabet $\Sigma = \{\mathtt{A}, \mathtt{G}, \mathtt{C}, \mathtt{T}\}$ for the 4 standard nucleotides adenine, guanine, cytosine and thymine. For the case of protein sequences, Σ consists of 20 amino acids. When comparing sequences at

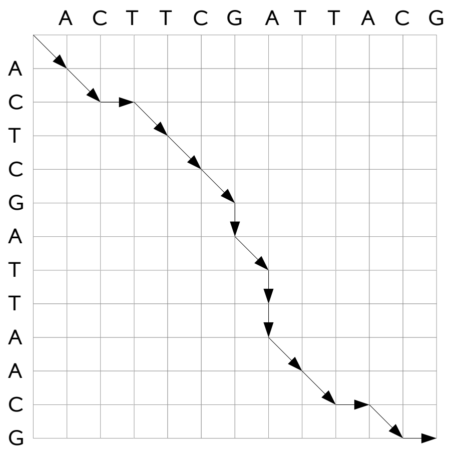


Fig. 1. Suboptimal alignment of two sequences.

a given index we observe either a match, mismatch, or a gap in one (or several) of the sequences. Gaps can be interpreted as a loss/gain of abilities. A mismatch can link to a mutation in the evolution. The global alignment often results in the biologically most plausible one.

The standard representation of the MSA problem is a grid with an origin in the upper left corner node. Along each axis one sequence is written, prefixed by the gap character. A solution of the MSA problem corresponds to a path from the origin to the lower right corner node. A step along the axis corresponds to introducing a gap, a step diagonal to two sequences a match or mismatch. This yields an interpretation of the grid as an acyclic graph with edges that correspond to one of the above steps. An example graph for the two sequences `ACTTCGATTACG` and `ACTCGATTAACG` and an alignment path for the alignment `ACTTCG_A_TTACG` and `AC_TCGATTAAC_G` is shown in Fig 1. Storing the graph explicitly causes memory problems, as the number and the lengths of the sequences are supposed to be large. An implicit graph is generated on-the-fly. Only after a successor v of v' is generated the edge (v, v') is stored in main memory.

Despite their limitation to moderate number of sequences, however, the research into *exact* algorithms is still going on, trying to push the practical boundaries further. They still form the building block of heuristic techniques, and incorporating them into existing tools could improve them. For example, an algorithm iteratively aligning two groups of sequences at a time could do this with three or more, to better avoid local minima. Moreover, it is theoretically important to have the "gold standard" available for evaluation and comparison, even if not for all problems.

Since MSA can be cast as a minimum-cost path finding problem, it is amenable to heuristic search algorithms developed in the AI community; these are actually among the currently best approaches. Therefore, while many researchers in this area have often used puzzles and games in the past to study heuristic search algorithms, recently there has been a rising interest in MSA as a test-bed with practical relevance. A number of exact algorithms have been developed previously that can compute alignments of a moderate number of sequences. Some of them are mostly constrained by available memory, some by the required computation time, and some on both. It is helpful to roughly group them into two categories: those based on the dynamic programming paradigm, which proceed primarily in breadth-first fashion; and best-first search, utilizing lower and/or upper bounds to prune the search space. Some recent research, including the one we refer to, attempts to beneficially combine these two approaches.

Even with on-the-fly generation of the weighted problem graph, state-of-the-art heuristics and advanced memory reduction techniques, the amount of RAM used by the search algorithms becomes crucial in large alignment problems. This paper considers strategies to overcome the problem, applying external algorithms that limit the RAM usage and control the access to the hard disk.

The structure of the paper is as follows. First, we introduce known MSA algorithms including IDDP. Next, we present two external variants of IDDP with increasing complexity. We study mathematical aspects of the programs, including the correctness, optimality and run-time complexities. Finally, we provide experimental evidence for the impact of the externalization.

2 MSA

The sequence alignment problem is a generalization of the problem of computing the *edit distance*, that aims at changing a string into another by using the three main edit operations of modifying, inserting, or deleting a letter. Each edit operation is charged, and the minimum-cost operations sequence is sought. For instance, *spell checkers* have to determine the lexicon word whose edit distance from a (possibly misspelled) word typed by the user is minimal. The same task arises in *version control systems*.

The state space of the multiple sequence alignment problem consists of all possible alignments of prefixes of the input sequences $m_1, \ldots, m_k$. If the prefix lengths serve as vector components we can encode the problem as a set of vertices $x = (x_1, \ldots, x_k)$, $x_i \in \{0, \ldots, |m_i|\}$ with associated cost vector v_x. A state x' is a (potenital) successor of x if $x'_i - x_i \in \{0, 1\}$ for all i.

When designing a cost function, computational efficiency and biological meaning have to be taken into account. The most widely-used definition is the *sum-of-pairs* cost function. First, we are given a symmetric matrix containing penalties (scores) for substituting a letter with another one (or a gap). In the simplest case, this could be one for a mismatch and zero for a match, but more biologically relevant scores have been developed.

An optimal alignment corresponds to a path with minimal costs. A simple example of a cost function assigns cost 0 to a match, 1 to a mismatch, and 2 to a gap. Therefore, the solution in Figure 1 has costs 16, 12 for all gaps and 4 for the mismatches. The optimal alignment `ACTTCGATTA_CG` and `AC_TCGATTAACG` has cost 4.

A substitution cost matrix correspond to a model of molecular evolution and estimate the exchange probabilities of amino acids for different amounts of evolutionary divergence. Based on such a substitution matrix, the sum-of-pairs cost of an alignment is defined as the sum of penalties between all letter pairs in corresponding column positions.

A major issue in MSA algorithms is their ability to handle gaps. Gap penalties can be made dependent on the neighbor letters. Moreover, it has been found that assigning a fixed score for each indel sometimes does not produce the biologically most plausible alignment. Since the insertion of a sequence of x letters is more likely than x separate insertions of a single letter, gap cost functions have been introduced that depend on the length of a gap.

Altschul [1] argues that this model is too simple and that at least a character-pair scoring matrix and affine gap costs have to be included in the cost. Affine gap costs induce a linear function $a + bx$, where x is the size of the gap, a is the cost for gap opening and b is the cost for extending the gap. Use of an affine gap cost in multiple sequence alignment is a challenge because identifying the opening of a gap is challenging. In terms of [1] the gap costs we consider are *quasi-natural*. It is the cost model used in practice by biologists and their alignment programs [21].

In order to deal with the biologically more realistic affine gap costs, we can no longer identify nodes in the search graph with lattice vertices, because the cost associated with an edge depends on the preceding edge in the path. Similarly as in route planning with turn restrictions, in this case, it is more suitable to store lattice edges in the priority queue, and let the transition costs for $u \to v, v \to w$ be the sum-of-pairs substitution costs for using one character from each sequence or a gap, plus the incurred gap penalties for $v \to w$ followed by $u \to v$. Note that the state space in this representation grows by a factor of 2^k.

2.1 Optimal Alignment Algorithms

There is a host of algorithms that has been applied to solve the MSA problem. In contrast to its name, dynamic programming is a static traversal scheme, traversing the problem graph in a fixed order. The storage requirements are considerable, all reachable nodes are visited. Given k sequences of maximal length n this accumulates to $O(n^k)$ nodes and $O(2^k \cdot n^k)$ edge visits.

In order to save memory Hirschberg [8] proposes a strategy that stores only the search frontier and reconstructs the solution path in divide-and-conquer manner. Hirschberg noticed that when we are only interested in determining the cost of an optimal alignment, it is not necessary to store the whole matrix; instead, when proceeding e.g. by rows, it suffices to keep track of only k of them at a time, deleting each row as soon as the next one is completed. This reduces the space requirement by one dimension, a considerable improvement for long sequences. Unfortunately, this method doesn't provide the actual solution path; In order to recover it after termination of the search, re-computation of the lost cell values is needed. The solution is to apply the algorithm twice to half the grid each, once in forward direction, and once in backward direction, meeting at a some intermediate relay layer. By adding the corresponding forward and backward distances, the cell lying on an optimal path can be recovered.

The underlying problem graph structure is directed and acyclic and follows a k-dimensional *lattice* or *hypercube*. It allows to apply traditional search algorithms. Dijkstra's algorithm [4] generates the graph with increasing cost of reaching a node. It has the advantage that it can stop traversing the graph when the goal has been reached, which reduces the number of generated states. A* [7] behaves similar to Dijkstra's algorithm, but explores the graph according to the cost function $f(v) = g(v)+h(v)$, where $h(v)$ is a heuristic function that estimates the cost of reaching the goal from v. The effect is that the goal is reached faster. Given that the heuristic is a lower bound, the first solution found is still optimal. IDA* [10] is an iterative-deepening variant of A*. In each iteration IDA* expands all states below a cost threshold that is successively increased (by the minimum possible) in case no goal is found.

The reduction of the search frontier has inspired most of the upcoming algorithms. Frontier search [12] combines A* with Hirschberg's approach to omit already expanded states from the search. It is motivated by the attempt of generalizing the considerable space reduction for the *Closed* list achieved by Hirschberg's algorithm to general best-first search. It mainly applies to problem graphs that are directed or acyclic but has been extended to more general graph classes. It is especially effective if the ratio of *Closed* to *Open* list sizes is large.

Sparse-memory graph search [18] stores some of the already expanded states to speed-up the computation. It is based on a compressed representation of the *Closed* list that allows the removal of many, but not all nodes. A node is not deleted until all its neighbors have been expanded. Compared to frontier search it describes an alternative scheme of dealing with back leaks.

Sweep-A* [19] is the MSA adaption of breadth-first heuristic search [20]. It traverses the cost-bounded graph g- (instead of f-)wise as the breadth-first search frontier is expected to be smaller than the best-first search frontier.

2.2 IDDP

Iterative-deepening dynamic programming [15], IDDP for short, is a hybrid of dynamic programming and IDA*. A difference to the above approaches is that not the nodes but the edges are expanded; due to the fact that the algorithm is designed to work with affine gap costs. The algorithm distinguishes between the level of an edge (v, v'), i.e., the search tree depth of v', and the g-value of an edge, i.e., the distance to the start edge s_e. Analogously, the h-value is defined as the estimated distance to the target edge t_e.

IDDPs fixed search order matches the one in dynamic programming and has several advantages over pure best-first selection. Since *Closed* nodes can never be reached more than once during the search, it is safe to delete useless ones (those that are not part of any shortest path to the current *Open* nodes) and to apply path compression schemes, such as the Hirschberg algorithm. No sophisticated schemes for avoiding 'back leaks' are required. Besides the size of the *Closed* list, the memory requirement of the *Open* list is determined by the *maximum* number of nodes that are open *simultaneously at any time* while the algorithm is running. As opposed to that, if dynamic programming proceeds along levels of anti-diagonals or rows, at any iteration at most k levels have to be maintained at the same time, and hence the size of the *Open* list can be controlled more effectively. By arranging the exploration order such that edges with the same head node (or more generally, those sharing a common coordinate prefix) are dealt with one after the other, much of the computation can be cached, and edge generation can be sped up significantly.

The remaining issue of a static exploration scheme consists in adequately bounding the search space using the h-values. A* is known to be minimal in terms of the number of node expansions. If we knew the cost of a cheapest solution path beforehand, we could simply proceed level by level of the grid, however only immediately prune generated edges e whenever $f(e) > f^*$. This would ensure that we only generate those edges that would have been generated by algorithm A*, as well.

IDDP applies a search scheme that carries out a series of searches with successively larger thresholds, until a solution is found. The use of such an upper bound parallels that in IDA*. Similar to A*, an estimate for the path from s_e to t_e via the current edge e is given by $f(e) = g(e) + h(e)$. As said, IDDP combines the advantages of dynamic programming and IDA*. The traversal order remains fixed, so that each node is expanded at most once. The order is along an increasing level, such that all states that do not lie on a shortest path can be removed. Additionally, path compression algorithms can be applied to store less states in main memory. By using lower bounds the search space is additionally reduced, such that no node expanded by A* is expanded by IDDP (up to tie breaking). By the same argument as in IDA*, IDDP will find an optimal solution. An example is provided in Figure 2.

IDDP itself shares similarities with iterative-deepening bounded dynamic programming (IDBDP) as introduced by [12]. IDBDP also performs a series of iterations with gradually increasing upper bounds, starting with a lower bound on the alignment cost, until a solution is found. The work refers to [16] for approximate string matching, but has not been adapted to affine gap costs.

The use of a global upper bound U additionally saves memory as it prunes generated edges whose f-value is outside the current threshold. If a node is the only (remaining) successor of its parent node, then it can cause further deletions.

IDDP main search routine is simple. A sequence of sub-searches with increasing cost thresholds is invoked until the goal is found. The threshold Θ_i plays the same role as in IDA*. It is initialized with the lower bound L of the problem, and Θ_i is determined by the smallest possible cost value of generated nodes with cost larger than Θ_{i-1}.

The estimate is a pattern database [3]. It is determined by the alignment of $m < n$ sequences, bootstrapping the alignment algorithm on all possible subsets of size m. The lower bound is the sum of the cost of these subset alignments,

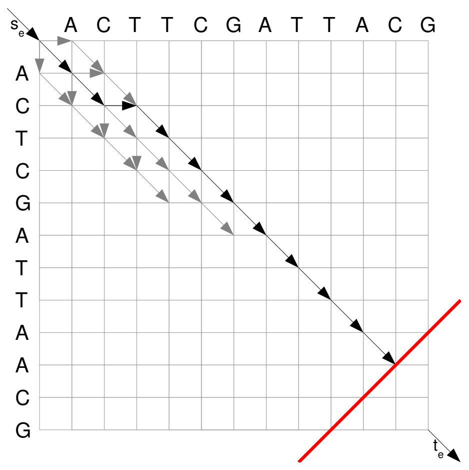


Fig. 2. Computing the optimal alignment with IDDP. Bold arrows denote edges on the optimal path, shallow arrows denote generated edges not pruned by the current cost threshold, the anti-diagonal line denotes the active layer.

divided by $\binom{n-2}{m-2}$ [9]. For the *triple heuristic* that we use we have $m = 3$. If an upper bound U is known, then the pattern database construction is truncated, very much in the sense of limiting external memory pattern database construction in the work of [22].

3 Externalizing IDDP

Externalization considers maintaining data structures on (one or several) hard disks by the application program. All MSA approaches we discuss maintain layers on disk.

In the first implementation (see Figure 3) only the nodes are flushed. The main reason is that edges were heavily linked due to the successor relationship, to the origin and target nodes, and the heap. When writing the nodes to disk memory addresses for the edges can be flushed, too, as they do exist in main memory. Whenever a node is needed, it has to be read from disk; a random access that may cause one I/O. Internally, the grid coordinates of a node are maintained in a trie data structure, in which common prefixes of the coordinate vector are shared. For this simple externalization, however, we store the full vectors according to an inorder traversal of the trie on disk. The next step is to substitute the file pointers that are used in the edges by condensed information about the filename and byte offset. Moreover, each layer is stored in one file.

The above external IDDP algorithm calls for many random accesses. In a buffered extension of the externalization, all data requested from disk is read into an internal read buffer, and all data that has to be flushed is written into

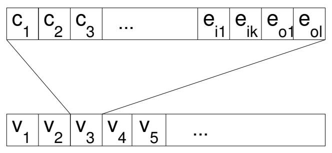


Fig. 3. File structure for simple external IDDP without edges flushed. A file consists of all nodes $v_1, v_2, \ldots$ in one layer. Each node consists of its coordinates $c_1, c_2, \ldots$, and the addresses of the first and last edge of the ingoing and outgoing adjacency lists $(e_{i,1}, e_{i,k}, e_{o,1}, e_{o,l})$.

an internal write buffer. If the buffers become empty (or full) they are refreshed (or flushed) via block-wise file accesses.

Nodes are to be read, if an outgoing edge is expanded. On the other hand, the coordinates of the nodes are sufficient to determine if the edge has introduced a gap. For the affine gap costs we further have to check, whether a gap is either continued or opened.

Moreover, the buffered variant does no longer support the internal compression of the list of already expanded nodes through deletion of states, as the buffers already reduce the memory for each layer considerably, and further backup data is flushed to the disk. This implies that each coordinate between two nodes v and v' can differ by either 0 or 1, such that the coordinate difference between v and v' can be encoded by the integer $\mathit{diff}(v, v') =$

$$(v'_1 - v_1)2^{k-1} + (v'_2 - v_2)2^{k-2} + \ldots + (v'_k - v_k)2^0.$$

Using the reversed encoding given v' and $\mathit{diff}(v, v')$ we can reconstruct v. Therefore, the node that is referred to as the origin of an edge does not have to reside in memory (see Figure 4).

Moreover, using layers an internal heap for maintaining the expansion order is no loner needed. We can simply take the order that is present in the file for the active layer[1].

How many buffers are needed? For the expanded layer, we need one read buffer for the nodes with edges that have not yet been expanded, and one write buffer for the nodes with edges that have been expanded. Additionally, we need n write buffers for placing successors in one of the next n layers. Nodes are sorted with respect to their coordinates.

External sorting follows the mergesort paradigm using as many file-pointers as there are peaks with respect to the sorting criteria. We do not provide extra buffers to each of the file pointers, as we assume that these disk read operations

[1] The important aspect is that the algorithm works on expansion sets so that any order will do. For external algorithms this implies that we can sort the file. For a parallel extension this implies that we can distribute the workload among different processors.

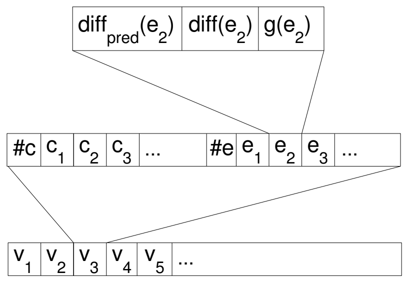


Fig. 4. File structure for external IDDP with edges flushed, including trie compression. A file consists of all nodes $v_1, v_2, \ldots$ in one layer. Each node itself consists of its coordinates $c_1, c_2, \ldots, c_{\#c}$ and its adjacency list $e_1, \ldots, e_{\#e}$; each edge consists of its *diff*-value, its g-value, the *diff*-value of its predecessor.

are buffered already on the hardware. If there are enough file pointers available, each state is looked at exactly once, so that the sorting matches the scanning complexity. Unfortunately, the number of file pointers is limited on most operation systems. If the number exceeds 1024, we invoke early merges to unify the first 1024 peaks into one, an idea that also saves disk space [11].

If duplicates are detected during the expansion, their adjacency lists are merged. An advantage for the buffered version, denoted as external IDDP, is that the memory requirements can be adapted to match the existing hardware. Another advantage is that the internal trie structure for the coordinates can be used to compress the data on disk. Only the coordinates are stored that do not match the prefix of the previous one.

4 Mathematical Aspects

External algorithms are measured in the number of file accesses (I/Os) and the number of times they sort or scan N data items, denoted as $scan(N)$ and $sort(N)$.

We distinguish between the total number of nodes $|V| = O(n^k)$ and the number of expanded nodes $|V_{Exp}| = |\{n' \in V \mid e = (n, n') \wedge f(e) \leq f^*\}|$ as well as between the total number of edges $|E| = O(n^k \cdot 2^k)$ and the number of expanded edges $|E_{Exp}| = |\{e \in E \mid f(e) \leq f^*\}|$ where f^* is the optimal solution path cost.

Let L be the initial lower and U be the global upper bound. The optimality of IDDP is inherited from IDA* and dynamic programming, provided that U is a correct upper bound. It is also simple to see that the last iteration is actually the largest, since each iteration contains at least one edge more than the previous one.

Theorem 1. *The number of iterations is polynomial in n, k and the maximal edge cost C.*

Proof. Any alignment is bounded by nk, since otherwise there is a gap at one index in all sequences. Using the sum-of-pairs, the maximal cost of a path is less than $k^2C \cdot nk = O(k^3nC)$.

We first analyze the externalization without edge storage and buffering.

Theorem 2. *Simple external IDDP requires at most $O((U-L) \cdot |E_{Exp}|)$ I/Os.*

Proof. The last iteration of the unbuffered externalization applies $O(|E_{Exp}|)$ I/Os in the worst case, since every read access to a node may require one I/O, and for each expanded edge we may issue a read request. Therefore, the overall time complexity is $O((U-L) \cdot |E_{Exp}|)$ I/Os.

Now we analyze the complexity of the buffered externalization with full node and edge storage.

Theorem 3. *External IDDP requires at most $O((U-L) \cdot (\mathrm{sort}(|E_{Exp}|) + \mathrm{scan}(|V_{Exp}|)))$ I/Os.*

Proof. The last iteration of the buffered externalization applies $O(sort(|E_{Exp}|) + scan(|V_{Exp}|))$ I/Os. To perform delayed duplicate detection we sort the layers with respect to the nodes. The number of nodes in the next layer is bounded by the number of edges from the nodes in the previous layer. Therefore, the cumulated sorting efforts for removing duplicates in the individual layers are less than the sorting efforts for the entire set E_{Exp}. For reading a layer and for solution reconstruction at most $scan(|V_{Exp}|)$ I/Os are needed. Therefore, given that there are at most $U-L+1$ iterations, the overall run time is bounded by $O((U-L) \cdot (sort(|E_{Exp}|) + scan(|V_{Exp}|)))$ I/Os.

It has to be said that for both cases the factor $U-L$ can be avoided by using a strategy called refined threshold determination [17]. It adjusts the threshold increase between every two iterations in such a way that at least twice as many elements are considered in the next iteration. With refined threshold determination the efforts for the last iteration are larger than the joint efforts for all other iterations. As the jumps in the thresholds are bigger than the minimum possible, one has to expand more nodes in the last iteration to guarantee optimality, which in turn affects the exploration efficiency.

Theorem 4. *The triple heuristic can be computed in polynomial time wrt. k and n.*

Proof. We have $\binom{k}{3} = O(k^3)$ triples and the maximum number of expanded nodes for each triple is $O(n^3)$. Therefore, the total internal time complexity for the heuristic is $O(n^3 \cdot k^3)$.

Theorem 5. *For storing all sequences of length 1 the uncompressed representation requires $O(k2^k)$ space, while trie compression requires $O(2^k)$ space.*

	k	Cost	$h(s_e)$	Time	RAM
1lcf	6	134,097	133,540	5:30:22	219,824
1rthA	5	70,387	70,243	0:00:20	11,216
1taq	5	119,552	119,160	12:02:26	678,020
1ac5	4	39,675	39,515	0:03:13	44,352
1bgl	4	80,552	80,406	0:03:45	42,380
1dlc	4	49,276	49,141	0:02:48	29,724
1eft	4	33,151	33,053	0:00:39	12,060
1gowA	4	40,727	40,577	0:03:09	22,348
2ack	5	69,608	69,060	3:39:14	419,944
arp	5	58,300	57,865	1:12:22	91,636
glg	5	66,606	66,408	0:06:22	44,644
1ajsA	4	34,501	34,277	0:09:57	66,964
1cpt	4	36,612	36,414	0:02:54	38,784
1lvl	4	39,849	39,602	0:20:18	194,432
1ped	3	16,333	16,170	0:00:05	11,244
2myr	4	41,281	40,935	5:48:18	937,612
gal4	5	-	56,632	-	>1,048,576

Table 1. IDDP (time in *hh:mm:ss* and space in kilobytes).

Proof. We take the alphabet $\Sigma = \{0, 1\}$ to denote whether or not a gap is introduced. We observe that for full storage of all sequences of length 1, all bitstrings for the values from 0 to 2^k in binary are generated. This accumulates to space requirements in the order of $O(k2^k)$. The trie representation will store only the changes from the bitstring of value l to the bitstring of value $l - 1$, added for all $l \in \{0, \ldots, 2^k\}$. By the amortized time complexity analysis of a binary counter [2], it is well-known that these bit-flips induce a time complexity in the order of $O(2^k)$.

5 Experimental Results

We experimented on an Opteron 2.2GHz Linux machine with 1 gigabyte memory and a total runtime limit of 480 hours.

In Table 1 we display the cost-optimal solutions obtained with internal IDDP on long sequences of the BAliBASE[2]. We denote the number of sequences to be aligned, the initial and optimal cost, as well as the resource consumption of the exploration. IDDP was invoked with tree collapsing.

The results for the first externalization are shown in Figure 2. Unfortunately, this approach failed, as additional to the loss in computation time due to external writing and reading, it does not save RAM wrt. internal IDDP. The core reason is that the memory data gain for externalizing the nodes is fully consumed by the file pointers that are used to substitute them.

[2] Some long sequence problems were solved too quickly to collect accurate memory data and thus omitted from this presentation.

	Time	RAM	DISK
1lcf	65:06:04	293,396	978,885
1rthA	0:00:47	11,684	7,065
1taq	>480:00:00	-	-
1ac5	0:50:37	58,340	21,314
1bgl	0:49:51	65,088	42,398
1dlc	0:18:11	43,120	25,779
1eft	0:01:39	14,400	6,901
1gowA	0:16:37	50,732	13,718
2ack	114:31:16	630,808	1,000,490
arp	22:18:53	109,856	523,375
glg	0:21:56	63,744	11,167
1ajsA	1:35:28	70,232	84,077
1cpt	0:39:52	49,396	28,271
1lvl	9:54:40	337,068	183,147
1ped	0:07:05	9,708	15,279
2myr	>480:00:00	-	-
gal4	>480:00:00	-	-

Table 2. Simple external IDDP.

The results of the second externalization (including buffering and edge flushing) are shown in Table 3. We see that there are considerable savings in the crucial resource of main memory especially for the larger problems, while the increase in time remains moderate. As an example for 1taq we have a 4.4-fold decrease in space and a 5-fold increase in time. The discrepancy of disk space and RAM usage can be large (in 1taq with a factor of about 23.2). The external version could newly solve gal4 with optimal cost 57,286.

Up to the storage structures for the estimate, the RAM requirements remain constants. An edge requires 72, a node 32, and the coordinates 40 bytes. External IDDP used 40 bytes for an edge, 56 bytes for a node, and 40 bytes for the coordinates of a node. We also conducted experiments on a 32-bit system. For IDDP an edge now requires 40, a node 20, and a coordinate 20 bytes. In the external version an edge consumes 24, a node 28, and node coordinates 20 bytes. For example, on a 32-bit system, IDDP consumed 140,392 kilobytes to solve 1lcf, while in the external version only 79,988 kilobytes were used.

There is another time-space trade-off. As the heuristic is computed in bootstrapping manner – calling the IDDP algorithm for a smaller set of sequences (3 in case of the triple heuristic) – it is possible to externalize the heuristic calculations, too. This results in a constant RAM usage. In the experiments, we obtained memory requirements of 670,249 kilobytes for solving 1lcf externally. The time increased from 6:40:23 for external IDDP with internal heuristic calculations to 18:18:54 for external IDDP with the external triple heuristic.

Last but not least, we measured the effect of trie compression. Compared to efforts in literature [5] the compression does not loose accuracy. To our own surprise, the savings were only moderate. We uncompressed the compressed

	Time	RAM	DISK
1lcf	6:40:23	106,772	437,888
1rthA	0:00:43	13,684	1,365
1taq	60:03:41	129,356	3,012,294
1ac5	0:03:34	40,276	4,787
1bgl	0:03:31	48,888	8,860
1dlc	0:03:29	36,992	2,044
1eft	0:00:44	11,184	1,262
1gowA	0:03:41	30,896	1,246
2ack	4:59:43	264,240	363,705
arp	1:26:30	69,300	182,314
glg	0:07:01	51,940	3,332
1ajsA	0:10:28	60,148	19,148
1cpt	0:03:10	42,220	6,094
1lvl	0:20:11	179,744	42,567
1ped	0:00:22	255	3,311
2myr	12:05:19	722,324	533,749
gal4	182:55:51	580,008	7,354,524

Table 3. External IDDP.

files for gal4, consisting of 7,531,032,328 bytes. The files grew to a total size of 7,734,791,156 bytes. We applied Lempel-Ziv compression to obtain a reduction of the files to 2,991,879,250 and 3,086,344,138 bytes, respectively. Therefore, Lempel-Ziv obtains a compression factor of about 60%, independently of the fact, whether or not the files were already compacted by trie compression.

Compared to the literature, the cost function used in [12] does neither use similarity measures nor affine gap costs. As in our initial example, it charges no penalty for a match, one unit for a mismatch, and a two units for a gap in either string. Moreover, [12] took the 1pamA example from reference 3 (instead from reference 1) from the BALiBASE.

Niewiadomski [13] showed very good results in solving BAliBASE alignment problems with internal parallel frontier search. The solution process for 1pamA with delayed duplicate detection on 16 processors using MPI communication took about a day, but the peak RAM requirement for solving 1pamA was 55.8 gigabytes; much more than the 1 gigabyte that we have provided. As a general rule, the more RAM available, the larger the buckets and the faster the algorithm. [13] used similarity matrices but fixed gap cost.

K-group A* [21] extends Sweep-A* [19] and uses quasi-natural gap costs. In difference to our approach that takes Jones,Taylor and Thorton's PET-91 matrix, the authors applied Dayhoff's PAM-250 matrix with gap opening cost of 20 and gap-extension cost of 8 (where we used gap opening costs of 8 and extension costs of 9 as in [15]). Therefore, the optimal costs of IDDP variants with K-group A* and the OMA program by [14] that applies the same cost function in Table 4 do not match. Moreover, the algorithms were run on 300 and 400 MHz systems. Nonetheless, the comparison shows that IDDP does challenge

	OMA		5-Group A*		IDDP		Ext-IDDP	
1aboA	10,674	973	10,674	199	10,665	8	10,665	15
1aho	9,807	6	9,807	0	8,828	0	8,828	0
1hfh	19,208	23	19,208	3	17,628	2	17,628	9
1idy	9,542	3	9,508	45	8,637	3	8,637	10
1krn	11,409	3	11,409	0	10,302	0	10,302	1
1pfc	17,708	19	17,708	3	15,843	0	15,774	5
1plc	14,205	4	14,195	0	12,745	0	12,745	3
2mhr	16,687	4	16,687	0	14,765	0	14,765	3
451c	13,364	200	13,364	74	12,171	1	12,171	8

Table 4. Comparison with other quasi-natural gap cost sequence alignment solvers (costs in units, time in seconds).

the state-of-the-art for quasi-natural gap costs and the externalization does not loose much of its efficiency.

6 Conclusion and Discussion

We have presented two externalizations of IDDP. As expected, the brute-force externalization is less time and space efficient. As there are more edges than nodes, outsourcing the nodes only does not pay off. For the buffered externalization in which edges are also externalized, we established a memory gain even for small instances. As the heap is no more needed, an additional space advantage is obtained.

Out of the 82 BAliBASE (Reference 1) entries, only 1pamA remained unsolved within the slot of two weeks time. The process terminated with a hard disk consumption of 41 gigabytes. We have to resume the exploration given that all information is saved in the current threshold and the generated layers on disk. In order to keep the RAM requirements small for less complex problems, we have used small buffers. As a consequence for solving 1pamA the buffer size has to be enlarged in further experiments.

The question for the user is, whether time or space is his crucial resource. If there is not much time and considerable RAM he should first use internal IDDP. If the system still starts swapping, a buffered version with large buffers should be tested. If there is much time and less space, smaller buffers can be used.

As for the MSA problem both time and space are critical resources, the next step will be to distribute external IDDP among different processors. As the order of expansion in one layer is arbitrary, by the virtue of delayed duplicate detection, we expect almost linear speed-ups. We will also study file compression schemes to save space.

Besides parallelization, in the future we will look closer at variants that show anytime behavior. An anytime divide-and-conquer heuristic search algorithm that splits the sequences has been applied together with several other acceleration techniques by [14]. The method is adaptive in that depending on the time one wants to spend on the alignment, a better (up to an optimal alignment) can be obtained.

References

1. S. Altschul. Gap costs for multiple sequence alignment. *Journal of Theoretical Biology*, 138:297–309, 1989.
2. T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms.* The MIT Press, 1990.
3. J. C. Culberson and J. Schaeffer. Pattern databases. *Computational Intelligence*, 14(4):318–334, 1998.
4. E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.
5. A. Felner, R. Meshulam, R. C. Holte, and R. E. Korf. Compressing pattern databases. In *AAAI*, pages 638–643, 2004.
6. D. Gusfield. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press, 1997.
7. N. Hart, J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on System Science and Cybernetics*, 4(2):100–107, 1968.
8. D. S. Hirschberg. A linear space algorithm for computing common subsequences. *Communications of the ACM*, 18(6):341–343, 1975.
9. H. Kobayashi and H. Imai. Improvement of the A* algorithm for multiple sequence alignment. *Genome Informatics*, pages 120–130, 1998.
10. R. E. Korf. Depth-first iterative-deepening: An optimal admissible tree search. *Artificial Intelligence*, 27(1):97–109, 1985.
11. R. E. Korf and T. Schultze. Large-scale parallel breadth-first search. In *AAAI*, pages 1380–1385, 2005.
12. R. E. Korf, W. Zhang, I. Thayer, and H. Hohwald. Frontier search. *Journal of the ACM*, 52(5):715–748, 2005.
13. R. Niewiadomski, J. N. Amaral, and R. C. Holte. Sequential and parallel algorithms for frontier A* with delayed duplicate detection. In *AAAI*, 2006.
14. K. Reinert, J. Stoye, and T. Will. An iterative method for faster sum-of-pairs multiple sequence alignment. *Bioinformatics*, 16(9):808–814, 2000.
15. S. Schroedl. An improved search algorithm for optimal multiple sequence alignment. *Journal of Artificial Intelligence Research*, 23:587–623, 2005.
16. E. Ukkonen. Algorithms for approximate string matching. *Info. Contr.*, 64:100–118, 1995.
17. B. W. Wah and Y. Shang. A comparison of a class of IDA* search algorithms. *International Journal of Tools with Artificial Intelligence*, 3(4):493–523, 1995.
18. R. Zhou and E. Hansen. Sparse-memory graph search. In *IJCAI*, pages 1259–1268, 2003.
19. R. Zhou and E. Hansen. Sweep A*: Space-efficient heuristic search in partially-ordered graphs. In *ICTAI*, pages 427–434, 2003.
20. R. Zhou and E. Hansen. Breadth-first heuristic search. In *ICAPS*, pages 92–100, 2004.
21. R. Zhou and E. Hansen. K-Group A* for multiple sequence alignment with quasi-natural gap costs. In *ICTAI*, pages 688–695, 2004.
22. R. Zhou and E. Hansen. External-memory pattern databases using structured duplicate detection. In *AAAI*, pages 1398–1405, 2005.

A Region-based Direction Heuristic for Solving Underspecified Spatio-temporal Planning Problems

Inessa Seifert

Department of Mathematics and Informatics,
SFB/TR8 Spatial Cognition
University of Bremen, Germany
seifert@informatik.uni-bremen.de

Abstract. The paper addresses an underspecified spatio-temporal planning problem, which encompasses weakly specified constraints such as different kinds of activities together with corresponding spatial assignments such as locations and regions. Alternative temporal orders of planed activities together with underspecified spatial assignments available at different levels of granularity lead to a high combinatorial complexity of the given tour planning problem. The proposed Region-based Direction Heuristic resembles the human spatial problem strategies such as regionalization as well as operation on different levels of abstraction. The paper introduces a cognitively motivated approach for efficient generation of alternative solutions. The produced alternative solutions meet user's anticipation, i.e., cognitive optimization criteria such as correspondence to abstract high-level plans.

Motivation

Constraint satisfaction is a widely accepted technology for dealing with complex combinatorial problems that gained much attention in such scientific fields like scheduling, planning, and configuration. Constraint satisfaction is based on the logical programming paradigm: "*the user states the problem, the computer solves it*" (Freuder, 1997). Therefore, the problem solving process is usually hidden from the user. Yet, there exists a variety of real-world problems, where an exact definition of constraint satisfaction criteria is difficult due to the missing knowledge of a user about the problem domain. An example of such type of problems is an underspecified spatio-temporal planning task like planning of an individual journey to a foreign country.

For the illustration we consider planning of an individual journey to Crete, a famous holyday island in Greece. Let the journey be constrained in time – say, 14 days, and involve different activity types, such as hiking, water sports and sightseeing. Our traveler visits Crete for the first time in her life and wants to rent a car and to enjoy swimming at beautiful sea coasts of Crete, hiking in natural parks and visiting different sightseeing attractions. Additional constraints involve activities such as hiking in widely known Samaria Gorge and visiting the famous sightseeing attraction the ancient Venetian harbor in Chania. To define her activities the traveler has to deal with a variety of alternative locations that can be visited during the trip together with different options regarding the activity types. The goal of the illustrated spatio-temporal planning task is to select an appropriate set of activities which take place at different locations and put them into a feasible temporal order under consideration of constraints regarding specific activity types, temporal scope of a journey and partially defined locations.

The exemplified spatio-temporal planning problem can be represented as a set of partially constrained activities. Each activity involves an activity type, duration and a location. In other words, our traveler may have certain idea regarding the activity types she wants to pursuer or locations she wants to visit. Yet, especially in the early tour planning stage, most of the activity types are not assigned to specific locations. An assisting constraint satisfaction system should complete the underspecified data by finding alternative values for activity types and locations and an appropriate order of activities which fit into the temporal scope of a journey.

Weakly specified constraints contribute to a high combinatorial complexity of the problem and a large number of alternative solutions. Observation of all possible feasible solutions is a cognitively demanding task (Knauff et al., 2002). Therefore, when dealing with the addressed underspecified planning problems we have to consider the following two aspects: a) computational complexity when searching for all feasible solutions exhaustively, b) a huge solution space.

The paper introduces a cognitively motivated approach for dealing with underspecified spatio-temporal planning problems. The Region-based Direction Heuristic is proposed, which allows for efficient generation of alternative solutions. The proposed heuristic resembles the human spatial problem strategies such as regionalization (clustering) as well as operation on different levels of abstraction (Hayes-Roth & Hayes-Roth, 1979; Wiener & Mallot, 2003; Seifert, to appear) and allows for pruning of the significant parts of the problem space. The solutions generated in this vain meet the cognitive optimization criteria, which are derived from the psychological findings as well as empirical results discussed in this paper.

Definition of the underspecified spatio-temporal planning problem

Let V be the set of locations (vertices), where activities can take place and E be the set of edges between the locations in V. Then G = {V, E} is a complete graph. Each location *i* in V is associated with an activity type *at*, such as for example sightseeing, hiking or swimming.

Each edge in E has symmetric, nonnegative cost c_{ij} associated with it, where c_{ij} is the distance between location *i* and location *j*, or c_{ij} is the cost of traveling between the two locations.

The spatio-temporal planning problem involves finding alternative orders of activities, further defined as tours, which are specified by a set of activity constraints AC = $\{ac_1, \ldots, ac_n\}$. An activity constraint is represented as $ac_k = (dc_k, atc_k, sa_k)$, $0 < k \leq n$, where the number of activity constraints corresponds to the number of the resulting activities, dc_k is an activity duration constraint, atc_k is an activity type constraint and sa_k is a spatial assignment. An activity type constraint involves a set of optional activity types, for example hiking or swimming. A spatial assignment can be defined as a specific location, or as a location from a set of selected locations. The duration constraints of the corresponding activities as well as the start, the end location and the temporal bound T_{max} have to be defined by a user.

The resulting tours consist of alternative orders of activities. Each activity is represented as $a_k = (d_k, at_k, l_k)$, $0 < k \leq n$, where n is total number of activities, d_k a duration, at_k is a type, and l_k is a location of an activity. The locations in each resulting tour should be all different from each other and the total time taken to visit locations cannot exceed a specified limit T_{max}.

Related problems

In the past forty years much scientific effort has been put into the research of the related complex combinatorial problems such as the Traveling Salesman Problem (TSP), Generalized TSP, Orienteering Problem (OP), as well as different variants of such optimization problems (e.g. the one-period bus touring problem, Deitch & Ladany, 2000).

The classical Traveling Salesman Problem involves finding an optimal route between specified destinations, where each of the destinations can be visited only once. Since in our case the destinations are specified partially by specific activity types, the addressed spatio-temporal planning problem has a closer relation to the Orienteering Problem. The OP considers vertices, which have nonnegative rewards and aims at finding a set of destinations, which bring the greatest reward requiring the total path cost not exceeding a given bound. Due to the high combinatorial complexity of the OP many heuristic methods have been proposed in the area of Operations Research (e.g. Tsiligirides, 1984; Chao et al., 1996; Tasgetiren, 2002). Recently, the algorithms for solving OP have been utilized for solving Over-Subscribed Planning problems, such as planning of scientific experiments performed for example by an autonomous system such as a Mars rover. Due to certain limitations such systems can achieve only a subset of possible goals. The OSP problems aim at finding an optimal plan for an unmanned Mars rover, which brings the greatest reward from a set of possible goals (e.g., Smith, 2004).

The addressed underspecified spatio-temporal planning problem has a close relation to the OP and OSP problems, since in both cases specific limitations like a predefined temporal bound have to be taken into consideration. However, the algorithms providing solutions to the OP and OSP problems aim at finding an optimal set of goals, where the goal rewards are independent from each other. In our case, the assistance system has to provide alternative tours, which fulfill logical constraints on specific activity types and spatial assignments, which means that a set of goals are not independent, but have to fulfill certain logical constraints. Such dependency cannot be easily transformed into weighted rewards, which are used for optimization in the described OP and OSP problems.

Most of the algorithms for the optimization problems generate a large number of solutions to select the best among them as a result. The first idea that comes to mind is to take the solutions produced so far and test if they fulfill the logical constraints.

Yet, since during the search procedure only those locations are selected, which contribute to the improvement of the current optimal state, this approach would lead to gaps in the solution space. Such unexpected gaps in the solution space can make a user insecure in terms of system's reliability.

The state of the art planning algorithms deal with problems that have clear quantitative optimization criteria, whereas the requirements on the solutions produced by the addressed spatio-temporal assistant system are different. First, the solutions have to be generated efficiently, to avoid long response times of the system. Second, the generated set of feasible solutions should be acceptable for the user, which means the solution space should be transparent to the user and should preferably not contain any unexpected gaps. In the following, we are going to grasp the usability requirements on the assistance system by proposing a cognitively motivated approach.

Cognitively motivated approach

When provided with a map and information about possible activities people are able to solve weakly specified planning problems without any help of an artificial system. However, depending on the size of a map and information available, people need a significant amount of time to produce a single or a limited number of alternative solutions. An important requirement on the assistance system is to respond to a weakly specified user input by proposing a set of acceptable solutions, which can be easily examined and understood by a user. The main idea of the proposed cognitively motivated approach is to utilize cognitive principles used by humans to generate solutions efficiently that fulfill the requirements described above.

The following figure (Fig. 1) illustrates the relation between the entire solution space, the solution space produced by humans, and the solution space generated by the assistance system.

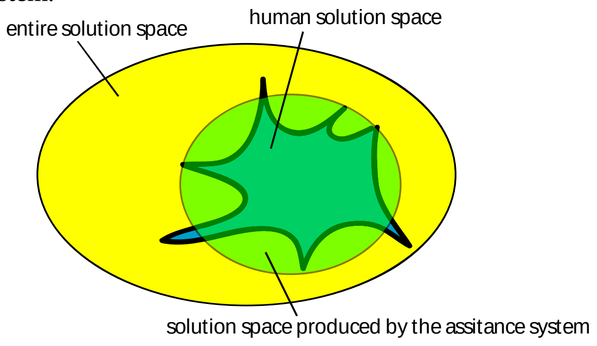


Fig. 1. Solution space: human and assistance system

The entire solution space contains all possible solutions to a given underspecified planning problem. The proposed heuristics uses optimization criteria, which aim at minimizing the difference between the human-like solution space and the solution space produced by the assistance system.

In the next section we introduce empirical findings regarding human problem solving strategies, in order to derive the cognitive principles and optimization criteria, which are going to be formalized and applied to the generation procedure.

Human spatial problem solving strategies

Human cognitive processing of information is considered to be model-based. A mental model is a symbolic representation, which integrates information from all senses and from general knowledge and simulates a possible situation in the world (Johnson-Laird, 1983). Such models consist of a finite number of interrelated items, or informational units, and the basic processes that operate on them. Knauff et al. (1995) have shown that when reasoning about underspecified temporal relations, where many alternative solutions were possible, only few were mentally constructed and utilized for subsequent reasoning. The subset of solutions constructed by the humans has been called Preferred Mental Models (PMM).

Various psychological findings and experimental studies regarding mental spatial knowledge representation provide evidence that humans operate on a hierarchically organized, topologically interrelated, loosely coupled knowledge fragments. In the literature spatial mental knowledge representation are denoted as cognitive maps (e.g, Hirtle, 1998), cognitive collages and spatial mental models (Tversky, 1993).

Wiener & Mallot (2003) showed that hierarchically structured, i.e., regionalized, large-scale environments facilitate navigation and route-planning tasks. In regionalized environments subjects' knowledge representations contained different levels of granularity, such as places and regions, as well as so called *connectivity* relations between regions. Such representations helped to reduce the memory load and contributed to a better performance in route-planning and navigation tasks. The addressed study revealed an interesting behavioral phenomenon. In the scope of the study the participants had to visit multiple targets situated in different regions. The tracks of the resulting routes have shown that people navigated from one region to another, visiting the targets within a region and then moving to the next one, and avoided returning back to already visited regions.

Based on the assumptions, that (1) mental knowledge is hierarchical, (2) regions help to solve spatial problems more efficiently, and (3) humans avoid visiting the same region twice, we conducted an exploratory study to identify the cognitive principles utilized for planning of activities in advance using a geographical map. The first results of the study have been introduced in (Seifert, to appear). The next section provides a brief overview of the study and its results.

Tour planning study

The participants of the study were provided with a map of Crete, which included topographical information together with symbols for different activity types annotated to the corresponding locations. The participants of the study were asked to produce a plan of a journey for two imaginary friends, draw the solution on the maps and describe their decision steps on an additional sheet of paper.

The study revealed that the participants utilized hierarchical spatial problem solving strategies operating on different levels of granularity to solve the spatial planning task efficiently. The identified strategies involved structuring of a map into regions and imagining a circle or an ellipse that helped to put regions as well as locations to be visited in a particular order (Seifert, to appear).

Region-based Direction Heuristic

In our previous works we introduced the Region-based Representation Structure, which allows for definition of spatial constraints at different levels of granularity, for example as super-ordinate regions, activity regions or locations (Seifert et al, 2007).

Locations are associated with specific activity types and represent nodes of the graph, which are connected with each other via edges carrying distance costs. Activity regions contain locations, which share specific properties, like the user's requirements on activity types, which can be accomplished in that region. Super-ordinate regions divide a given environment into several parts. The structuring principles for super-ordinate regions are based on the empirical findings regarding mental processing of spatial information (e.g., Lyi et al., 2005; Tversky, 1993; Hirtle, 1998). The RRS includes topological relations: how different locations are connected with each other. Containment relations between locations, activity regions, activity regions and super-ordinate regions are represented as *part-of* relations. Such spatial partnomies (Bittner & Stell, 2002) allow for specifying spatial constraints and reasoning about spatial relations at different levels of granularity. The RRS includes neighboring relations between corresponding super-ordinate regions, which resemble the region connectivity identified by Wiener & Mallot (2003). The neighboring relations between the super-ordinate regions as well as edges between locations are supplemented with the 8 cardinal direction relations (East, South, North, West, North-East, North-West, South-East, South-West), in order to apply the proposed Region-based Direction Heuristic. In the following section we demonstrate our approach using a simple tour planning example.

Tour planning example

The original constraints involve an overall scope of the journey of 14 days. Our traveler begins and ends her journey at the international airport in Chania. The traveler wants to spend 2 days in Chania, to enjoy the sight of the Venetian harbor, and spend 3 days for hiking in the Samaria Gorge. She plans 2 days for visiting several sightseeing attractions in the Northern Coast, and plans 3 days for hiking in some other Cretan natural park. The rest 4 days of her journey are left unspecified (see Fig. 2).

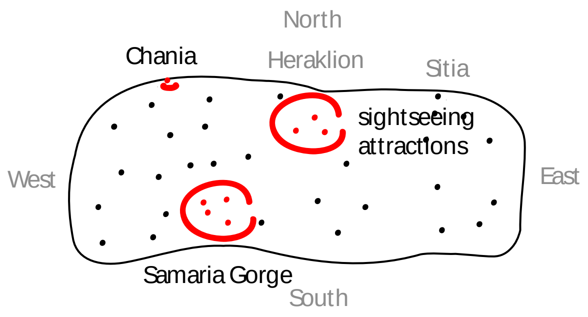


Fig. 2. Spatial constraints of an individual journey to Crete.

The set of the partially specified activity constraints can be defined as follows: AC={ac_1(2, ?, Chania), ac_2(3, ‘hiking’, Samaria Gorge), ac_3(2, ‘sightseeing’, set of locations at the Northern Coast), ac_4(3, ‘hiking’, ?), ac_5(4, ?, ?)}. The symbol ‘?’ stands for an unspecified value.

To solve spatial planning problem efficiently people generate an abstract plan, which involves various types of activities, priorities and a representation of an environment at different levels of granularity (Hayes-Roth, Hayes-Roth, 1979; Wiener & Mallot, 2003). Accordingly, the generation procedure for alternative solutions involves two levels of abstraction.

The first step of the proposed cognitively motivated approach is to generate a set of abstract plans, consisting of various orders of neighboring super-ordinate regions. Each of the produced orders of the neighboring super-ordinate regions has to fulfill the specified constraints regarding the activity types and locations to be visited.

The second step should be performed for each valid abstract plan. The Region-based Direction Heuristic utilizes the direction relations between the neighboring super-ordinate regions of an abstract plan. For example, since the journey starts in the super-ordinate region R1 (Chania), one of the abstract plans contains tours that go through R2, R3, R4, R5, R6 returning to R1 (see Fig. 3).

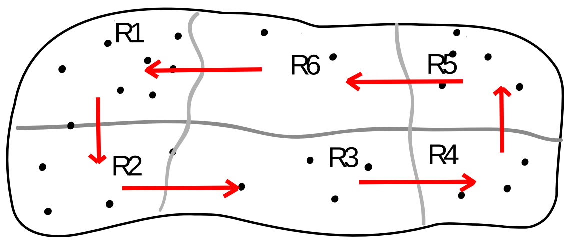


Fig. 3. Example abstract plan: order of super-ordinate regions.

The following figure (Fig. 4) illustrates cardinal direction relations between the neighboring super-ordinate regions. The edges between different locations, which represent nodes of the graph, are also supplemented with cardinal direction information between the corresponding nodes.

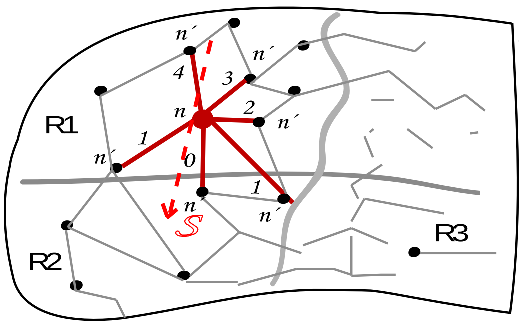


Fig. 4. Direction deviation cost depends on the abstract plan: direction relations between the super-ordinate regions.

In the following, we describe the Region-based Direction Heuristic for the A* forward search algorithm. A* employs an additive evaluation function *f(n) = g(n) + h(n)*, where *g(n)* is the cost of the currently evaluated path from the starting node *s* to the current node *n* and *h* is a heuristic estimate of the cost of the path remaining between *n* and some goal node. In our case, the heuristic estimate depends on an abstract plan. Returning to the illustrated tour planning example, the direction relation between the neighboring super-ordinate region *R1* and *R2* is for example *South*, correspondingly the direction relation between the super-ordinate regions R2 and R3 is *East*. The heuristic estimate *h* is a deviation cost of the direction relation between the current node *n* and *n´*, from the direction relation between the corresponding super-ordinate regions. In other words, each deviation from the main course of a tour has a greater value. For example, if the main direction between *R2* and *R1* is *South*, the direction deviation costs according to the direction relation between *n* and a successor *n'* would be *South*=0, *South-East*=1, *South-West*=1, *East*=2, *West*=2, *North-East*=3, *North-West*=3, and finally, if *n´* lies in the opposite direction North, the heuristic estimate receives the greatest "penalty" value *North*=4 (see Fig. 4).

The classical A* forward search procedure keeps expanded nodes in a list of paths, further called Routes. Currently optimal path is kept in $\text{Path}_{\text{min.}}$

1. Put the start node s as a first path on a list of Routes with 0 as a solution cost.
2. Select from the Routes a path with the minimal solution cost, further Path_{min}.
3. If Path_{min} reached the final destination and all activity constraints are fulfilled, return the result, if not - fail.
4. For all nodes *n´* sharing an edge with the last node *n* of the Path_{min} repeat the following procedure:
    a. Nodes n and *n´* belong to the same super-ordinate region, or *n´* belongs to the next super-ordinate region of the plan.
    b. If *n´* belongs to the next super-ordinate region of the plan, check if the constraints regarding the executed part of the abstract plan are fulfilled. For example, all spatial assignments which belong to the visited super-ordinate regions have to be included in the current path.
    c. Calculate the heuristic estimate from the abstract plan and current edge direction relation and add it the solution cost of the path.

d. If *n´* is already on the path, add additional loop cost to the solution cost.
e. Add the distance cost from n to *n´* to a path cost.
f. Check if a path cost doesn't validate the temporal scope T_{max}.
g. Add the node n with the corresponding solution cost to a list of Routes.

5. Go to the step 2.

The accumulated value of the evaluation function represents a solution cost measure, which indicates, how good a tour fits into a given abstract plan, i.e., meets the cognitive optimization criteria.

The proposed heuristic prunes not only significant parts of the problem space, but also generates tours that avoid detours to already visited super-ordinate regions. Solutions generated in this vain are inline with the empirical findings presented in the section, which describes the human problem solving strategies.

The algorithm has been implemented in the logical programming language PROLOG. To improve the computational performance of the generation procedure we limited the number of Routes, i.e., the size of the tree with the expanded nodes and the number of possible loops. This paper reports the early results on implementation of the proposed Region-based Directions heuristic. In the future, we are going to assess the quality of the produced solutions according to the abstract plans given to the system.

Conclusion

In the scope of the paper we introduced a Region-based Direction Heuristic, which prunes significant parts of the problem space, when generating solutions for underspecified spatio-temporal planning problems. The proposed heuristic resembles human spatial planning strategies and searches for a cognitive optimum, i.e. correspondences to abstract plans, consisting of alternative orders of super-ordinate regions. The assistance system produces a set of solutions which avoid detours between super-ordinate regions. Due to the hierarchical structure of the solution space, the produced tours can be easily communicated to a user, since it suits the hierarchical principles of mental processing of spatial information.

In our previous work (Seifert, 2006) we introduced a collaborative assistance approach, which provides a user with operations to modify solutions by relaxation and specification of constraints. Herewith, the proposed generation procedure serves as a starting point for a further collaborative search for an improved solution. The operations for modification of produced tours complement user's reasoning capabilities and allow for moving from human-like solutions towards unexplored parts of a solution space.

References

Bittner, T., & Stell, J.G. (2002). "Vagueness and Rough Location". Geoinformatica, Vol. 6. Pp.: 99-121.

Chao, I.-M.; Golden, B. L.; and Wasil, E. A. 1996. A fast and effective heuristic for the orienteering problem. European Journal of Operational Research Vol. 88. Pp.: 475–489. Elsevier.

Deitch R.. Ladany S.P. (2000). “The one-period bus touring problem: Solved by an effective heuristic for the orienteering tour problem and improvement algorithm.” European Journal of Operational Research, Vol. 127, (1), Pp.: 69-77. Elsevier.

Freuder, E., C. “In Pursuit of the Holy Grail”. (1997) Constraints: An International Journal, 2, Pp.:57–61. Kluwer Academic Publishers. The Netherlands.

Hayes-Roth, B. & Hayes-Roth, F..(1979) “A Cognitive Model of Planning“, Cognitive Science 3, Pp.: 275-310.

Hirtle, S. C. (1998). “The cognitive atlas: using GIS as a metaphor for memory”. In M. Egenhofer & R. Golledge (Eds.). Spatial and temporal reasoning in geographic information systems. Pp.: 267-276. Oxford University Press.

Johnson-Laird, P. N. (1983). “Mental models”. Cambridge, MA: Harvard University Press.

Knauff, M., Rauh, R., & Schlieder, C. (1995). “Preferred mental models in qualitative spatial reasoning: A cognitive assessment of Allen's calculus”. Proceedings of the Seventeenth Annual Conference of the Cognitive Science Society (pp. 200-205). Mahwah, NJ: Lawrence Erlbaum.

Knauff, M., Schlieder, C., & Freksa, C. (2002). Spatial Cognition: “From Rat-Research to Multifunctional Spatial Assistance Systems”. Künstliche Intelligenz, Heft 4/02, arendtap Verlag, Bremen.

Lyi Y., Wang, X., Jin, X., Wu, L. (2005) “On Internal Cardinal Direction Relations”. Pp.: 283-299, in Proceeding of Spatial Information Theory 2005, edited by Cohn, A., G., and Mark, D., M., LNCS, Springer.

Seifert, I., Barkowsky, T., Freksa, C. “Region-Based Representation for Assistance with Spatio-Temporal Planning in Unfamiliar Environments”. In Gartner, G., Cartwright, W., Peterson, M., P. (Eds.), Location Based Services and TeleCartography, Lecture Notes in Geoinformation and Cartography. Springer-Verlag Heidelberg.

Seifert, I. (to appear). “Region-based Model of Tour Planning Applied to Interactive Tour Generation”. In Proceedings of International Conference for Human-Computer Interaction (HCII-2007).

Seifert, I. (2006). “Collaborative Assistance with Spatio-Temporal Planning Problems”. In Spatial Cognition 2006, Lecture Notes in Computer Science. Springer, Berlin.

Smith, D. (2004). Choosing objectives in over-subscription planning. In Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling, pages 393–401.

Tasgetiren, M. F. (2002). “A genetic algorithm with an adaptive penalty function for the orienteering problem”. Journal of Economic and Social Research Vol. 4 ,(2). Pp: 1–26.

Tsiligirides, T. 1984. Heuristic methods applied to orienteering. Journal of the Operational Research Society. Vol. 35. (9). Pp.:797–809.

Tversky, B. (1993). “Cognitive maps, cognitive collages, and spatial mental models.” In A. Frank & I. Campari (Eds.), Spatial information theory. Pp. 14-24. Berlin: Springer.

Wiener, J. M., & Mallot, H. A. (2003). “‘Fine-to-coarse’ route planning and navigation in regionalized environments”. Spatial cognition and computation., Vol. 3 (4). Pp.: 331-358.

Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung

Wolfgang Runte

Technologie-Zentrum Informatik, Universität Bremen
Postfach 33 04 40, D-28334 Bremen
Am Fallturm 1, D-28359 Bremen
woru@tzi.de
http://www.tzi.de

Zusammenfassung In wissensbasierten Konfigurierungssystemen werden Komponenten zur Constraint-Verarbeitung für die Verwaltung von Abhängigkeiten während des Konfigurierungsprozesses eingesetzt. Die Effizienz von Constraint-Lösungsverfahren ist allerdings stark problemabhängig. Zudem kann es aufgrund einer Vielzahl unterschiedlicher Domänen und der Vielfältigkeit der Anwendungsszenarien notwendig werden, an die jeweilige Domäne angepasste Constraint-Lösungsverfahren einzusetzen. Zur Behandlung von unterschiedlichen Constraint-Domänen innerhalb des strukturbasierten Konfigurierungswerkzeugs EngCon sind Constraint-Solver sowohl für finite als auch infinite Domänen erforderlich. Die Steuerung muss durch eine Komponente geleistet werden, mit der sich unterschiedliche Constraint-Solver je nach Bedarf und domänenspezifisch einsetzen lassen. Die Modularität ist dabei entscheidend für die Austauschbarkeit einzelner Komponenten. Es stellt daher einen sinnvollen Ansatz dar, ein modulares Constraint-Framework einzusetzen, in das je nach Bedarf, und dem in der jeweiligen Wissensbasis definiertem Problem, unterschiedliche Constraint-Solver mit jeweils für das spezifische Problem effizienten Lösungsverfahren eingesetzt werden können. Das Framework wird durch einen strategiebasierten Ansatz unterstützt, der eine flexible Kooperation unterschiedlicher Constraint-Lösungsalgorithmen erlaubt.

Key words: Wissensbasierte Konfigurierung, Konfiguration, Constraint Satisfaction Problem, Constraint-Solver, Java-Framework, finite, infinite, diskrete, kontinuierliche Domänen, reellwertige Intervalle

1 Motivation

Für die Konfigurierung variantenreiche Produkte lassen sich von Konfigurierungswerkzeugen mit unterschiedlichen wissensbasierten Methoden aus einzelnen Komponenten komplexe Aggregate erstellen. Constraints[1] sind ein Mittel

[1] *constraint* (engl.): Einschränkung, Beschränkung, Restriktion

zur Repräsentation von Abhängigkeiten zwischen den Komponenten einer Konfiguration (vgl. [11], [23], [25]).

Wissensbasierte Konfigurierungswerkzeuge verwenden Constraints zur Beschreibung von Abhängigkeiten zwischen Konzepten der Wissensbasis [19]. Die Auflösung der Abhängigkeiten wird von integrierten Constraint-Systemen verwaltet [26]. Die eingesetzten Constraint-Solver hingegen werden üblicherweise nicht innerhalb des eigenen Systems implementiert, sondern sondern sind von Fremdherstellern eingebundene externe Komponenten. Diese weisen häufig Eigenschaften auf, die den Einsatz innerhalb eines wissensbasierten Konfigurierungssystems einschränken. Zudem ist die Anbindung an das Konfigurierungssystem in der Regel auf einen einzigen Constraint-Solver beschränkt und damit sehr unflexibel.

Benötigte Constraint-Solver müssen arithmetische Funktionen zur Berechnung von intensional in Form von algebraischen Ausdrücken formulierten Constraints bieten. Neben klassischen Constraint-Solvern zur Behandlung von finiten Domänen sind für die Constraint-Verarbeitung in wissensbasierten Konfigurierungswerkzeugen Constraint-Lösungsmethoden für infinite, d. h. reellwertige Intervalldomänen erforderlich. Entsprechende Constraint-Solver müssen eine hohe Präzision durch Intervallarithmetik aufweisen (z. B. für Anwendungen im Maschinenbau) sowie unabhängig von der Constraint-Domäne ein inkrementell anwachsendes Constraint-Netz propagieren können.

Darüber hinaus ergeben sich Anforderungen an den Constraint-Lösungsmechanismus häufig in Abhängigkeit von der Aufgabenstellung des jeweiligen Konfigurierungsproblems. Neben stabilen Constraint-Lösungsverfahren, die eine hohe Effizienz für möglichst viele Problemstellungen bieten, ist es deshalb erforderlich, problemabhängig unterschiedliche Verfahren nutzen zu können. Benötigt wird eine Komponente, an der sich flexibel unterschiedliche Constraint-Solver mit verschiedenen Eigenschaften, sowohl bezogen auf die Lösungsverfahren als auch auf die zu verarbeitenden Wertedomänen, einbinden lassen.

Die vorliegende Ausarbeitung gliedert sich in die folgenden Anschnitte: Nach einer kurzen Einführung in die wissensbasierte Konfigurierung mit EngCon in Abschnitt 2 werden in Abschnitt 3 die benötigen Methoden zur Constraint-Verarbeitung vorgestellt. In Abschnitt 4 wird das Konzept für ein hybrides Constraint-Framework entwickelt. Anschließend wird in Abschnitt 5 eine Übersicht über verwandte Arbeiten gegeben. Die Ausarbeitung endet in Abschnitt 6 mit einer Zusammenfassung und einem Ausblick.

2 Wissensbasierte Konfigurierung mit EngCon

Wissensbasierte Konfigurierungssysteme nutzen deklarativ und explizit repräsentiertes Wissen, um komplexe Konfigurierungsaufgaben zu lösen oder Experten bei der Lösung einer Konfigurierungsaufgabe zu unterstützen. Sowohl EngCon [19] als auch dessen Vorläufer PlaKon [5] und KonWerk [10] sind in erster Linie strukturbasierte Konfigurierungswerkzeuge. Der Schwerpunkt dieser Systeme liegt auf einem begriffshierarchie-orientierten Kontrollmechanismus. Zusätzlich kommen weitere Inferenzmechanismen zum Einsatz, wie z. B. ein ausgereiftes Constraint-System.

Der Schwerpunkt von EngCon liegt im Gegensatz zu klassischen Systemen nicht auf Inferenzen, die aufgrund von Expertenregeln gebildet werden („regelhafte Expertise“), sondern auf Inferenzmechanismen, die aufgrund der wissensbasierten Architektur der Domäne angewendet werden. Zur Repräsentation des Objektwissens der Domäne wird in EngCon eine framebasierte Repräsentation verwendet, die die zusammengefasste Spezifikation der Objektstruktur, der Eigenschaften und möglichen Belegungen in einer sog. *Begriffshierarchie* innerhalb der Wissensbasis ermöglicht. In einer Ontologie stehen hier die Konzepte über *is-a*- und *has-parts*-Relationen in taxonomischen und partonomischen Hierarchien zueinander in Beziehung.

Die Art des Vorgehens bei der Lösung des Konfigurierungsproblems wird durch den strukturbasierten Konfigurierungsansatz anhand der Struktur der Konfigurierungsobjekte innerhalb dieses Domänenmodells definiert. Zur Erstellung einer Konfiguration werden die Konfigurierungsschritte *Zerlegung* (Instantiierung der Bestandteile eines Aggregates), *Spezialisierung* („Verfeinerung“ einer Instanz zu einer spezielleren Instanz) und *Parametrierung* (Setzen bzw. Einschränken der Wertebereiche von Objekteigenschaften) eingesetzt [19].

Die Wissensbasis von EngCon ist eine flexibel austauschbare, deklarative Beschreibung, d. h. für jede Anwendung kann spezielles Wissen definiert und eingebunden werden. EngCon ist dadurch domänenunabhängig und kann durch einen Austausch der Wissensbasis beliebige variantenreiche Komponenten konfigurieren.

Eine Besonderheit von EngCon ist der inkrementell und interaktiv verlaufende Konfigurierungsprozess. Am Ende einer strukturbasierten Konfigurierung steht immer genau eine Lösung. Im Gegensatz zu anderen Konfigurierungswerkzeugen, die häufig eine Breitensuche vornehmen und abschließend alle gefundenen Lösungen präsentieren, findet während der Konfigurierung mit EngCon eine benutzergesteuerte Tiefensuche statt, mit dem Ziel, interaktiv eine für den Benutzer geeignete Lösung zu finden.

3 Constraints

Die Verwendung von Constraints ist eine vielfach eingesetzte Methode zur Repräsentation und Auswertung von Abhängigkeiten. Durch Constraints werden Relationen zwischen (Constraint-)Variablen definiert. Neben der Definition solcher „Beschränkungen“ werden Constraints eingesetzt, um die Werte von Variablen dynamisch den Anforderungen der Constraints entsprechend anzupassen. Constraints sind in diesem Sinne Randbedingungen, welche die Konsistenz der Variablenwerte in einem System sicherstellen [26].

3.1 Finite-Domain-Constraints

Das allgemeine *Constraint Satisfaction Problem* (CSP) bezeichnet eine Klasse von kombinatorischen Problemen, die mittels einer Menge von Randbedingungen bzw. Constraints über eine Menge von Variablen formuliert werden. Die Aufgabe besteht darin, eine wohlgeformte Belegung für eine endliche Menge von Variablen zu finden. Wohlgeformt bedeutet in diesem Fall eine konsistente Belegung der Variablen mit Werten, so dass alle Randbedingungen erfüllt werden [7].

Definition 1. (Constraint Satisfaction Problem, CSP)
Ein Constraint Satisfaction Problem wird durch ein Tripel (V, D, C) *beschrieben, wobei* $V = \{v_1, \ldots, v_n\}$ *eine endliche Menge von Variablen mit assoziierten Wertebereichen* $D = \{D_1, \ldots, D_n\}$ *mit* $\{v_1 : D_1, \ldots, v_n : D_n\}$ *ist.* C *ist eine endliche Menge von Constraints* $C_j(V_j)$, $j \in \{1, \ldots, m\}$, *wobei jedes Constraint* $C_j(V_j)$ *eine Teilmenge* $V_j = \{v_{j_1}, \ldots, v_{j_k}\} \subseteq V$ *der Variablen zueinander in Relation setzt und deren gültige Wertekombinationen auf eine Teilmenge von* $D_{j_1} \times \cdots \times D_{j_k}$ *beschränkt.*

Klassischerweise bezieht sich die Literatur bei der Definition von CSPs häufig auf eine strengere Form, in der die Domänen der Variablen aus diskreten, endlichen Mengen (engl. *finite domains*, FD) bestehen (vgl. [28], [17]).

Definition 2. (Finite Constraint Satisfaction Problem, FCSP)
Sei $P = (V, D, C)$ *ein CSP. Wenn die Domäne* D_i *jeder Variablen* $v_i \in V$ *diskret und endlich ist, wird* P *ein Finite Constraint Satisfaction Problem (FCSP) genannt.*

Weil bei einem FCSP die Wertebereiche der Variablen endlich sind, ist auch der Lösungsraum endlich. Die Anzahl der möglichen Lösungen ergibt sich wiederum aus dem kartesischen Produkt aller Wertebereiche $D_1 \times \cdots \times D_n$. Die Kardinalität dieser Menge, und entsprechend auch der Aufwand zur Berechnung dieser möglichen Lösungen, wächst exponentiell mit der Anzahl der Variablen.

3.2 Intervall-Constraints

Neben klassischen CSP über finite Domänen stellt sich für die Constraint-Verarbeitung in wissensbasierten Konfigurierungssystemen das Problem der Behandlung von reellwertigen algebraischen Constraints. Die Wertebereiche der Constraint-Variablen werden hier als Ober- und Untergrenzen von reellwertigen Intervallen definiert, zwischen denen sich unendlich viele, nicht abzählbare Elemente befinden [2]. Diese Wertebereiche werden deshalb auch *kontinuierlich* genannt.

Intervall-Constraints werden z. B. eingesetzt, wenn unscharfe Informationen behandelt werden müssen, d. h. wenn das Wissen über Parameter nur in Form eines Werteintervalls bekannt ist, oder wenn die Aufzählung aller Lösungen nicht möglich ist, da unendlich viele existieren.

Im Gegensatz zu diskreten, endlichen Wertebereichen, für die Konsistenz- und Lösungsalgorithmen die einzelnen Werte in den Domänen der Constraint-Variablen mittels kombinatorischer Methoden aufzählen und auf Zugehörigkeit zu den Constraint-Relationen überprüfen können, lässt sich im Fall von reellwertigen Intervallen nicht für jeden einzelnen Wert bestimmen, ob er als konsistente

Belegung geeignet ist. Während CSPs über endliche Domänen zur Klasse der NP-vollständigen Probleme zählen, sind CSPs über unendliche Domänen unentscheidbar. Daher werden hier durch Konsistenz- und Suchverfahren lediglich die Ober- und Untergrenzen der Intervalle überprüft bzw. angepasst, so dass inkonsistente Werte ausgeschlossen werden (vgl. [4], [6], [14], [16], [3]).

Definition 3. (Intervall Constraint Satisfaction Problem, ICSP)
Ein Intervall Constraint Satisfaction Problem wird durch ein Tripel (V, I, C) *beschrieben. Neben den die Menge der Constraints* C *beschränkenden Variablen* $V = \{v_1, \ldots, v_n\}$ *wird eine Menge von Intervallen* $I = \{I_1, \ldots, I_n\}$ *als Wertebereiche der Variablen mit* $\{v_1 : I_1, \ldots, v_n : I_n\}$ *definiert.* C *ist die endliche Menge von Constraints* $C_j(V_j)$, $j \in \{1, \ldots, m\}$. *Jedes Constraint* $C_j(V_j)$ *setzt eine Teilmenge* $V_j = \{v_{j_1}, \ldots, v_{j_k}\} \subseteq V$ *zueinander in Relation, und beschränkt den Lösungsraum der involvierten Variablen auf eine Teilmenge des kartesischen Produkts* $I_{j_1} \times \cdots \times I_{j_k}$ *von* k *Subintervallen.*

Das kartesische Produkt mehrerer Intervalle wird aus geometrischen Gründen auch kurz *Box* genannt. Ziel ist es, eine Menge n-stelliger, möglichst *kanonischer* Boxen zu isolieren, die den Lösungsraum des CSP approximieren, ohne gültige Lösungen zu verlieren. Jede n-stellige Box approximiert jeweils eine mögliche Lösung des CSP. Eine Box wird „kanonisch“ genannt, wenn sie aus Intervallen besteht, deren Grenzen entweder jeweils dieselben oder direkt aufeinander folgende Zahlen sind, d. h. wenn sie möglichst punktgenaue Lösungen darstellen. Um dies zu erreichen wird mittels Such- und Konsistenztechniken sowie mathematischer Verfahren durch den Raum navigiert, der durch das kartesische Produkt $I_1 \times \cdots \times I_n$ aufgespannt wird [2].

4 Das YACS-Framework

Zur Behandlung unterschiedlicher Constraint-Domänen innerhalb eines wissensbasierten Konfigurierungswerkzeugs ist eine Kooperation von mehreren Constraint-Solvern erforderlich. Diese Kooperation muss durch eine Komponente geleistet werden, mit der sich unterschiedliche Constraint-Solver je nach Bedarf und domänenspezifisch einsetzen lassen. Die Modularität des Systems ist dabei entscheidend für die Austauschbarkeit einzelner Komponenten. Die Constraint-Komponente muss im Detail die folgenden Anforderungen erfüllen:

– Das System muss eine modulare Architektur und einheitliche Schnittstellen für unterschiedliche Lösungsverfahren aufweisen.

– Constraint-Lösungsverfahren müssen sich flexibel einbinden und problemabhängig austauschen lassen.

– Das System muss *hybrid* sein, d. h. neben finiten Domänen müssen infinite Domänen in Form von reellwertigen Intervallen unterstützt werden.

– Der inkrementelle Aufbau des Constraint-Netzes innerhalb einer interaktiv durchgeführten, wissensbasierten Konfigurierung muss unterstützt werden.

Der im folgenden Vorgestellte Ansatz ermöglicht die flexible Kooperation und Kombination von Lösungsalgorithmen zu Constraint-Solvern mit neuen oder erweiterten Eigenschaften. Der Name YACS steht dabei für *Yet Another Constraint Solver*. YACS ist allerdings mehr als nur ein einzelner Constraint-Solver. Es bezeichnet vielmehr ein hybrides System für den flexiblen Einsatz von Constraint-Lösungsverfahren für finite und infinite Domänen. Sie sind eingebettet innerhalb einer modularen Framework-Architektur.

4.1 Der Framework-Ansatz

Durch das Aufkommen von objektorientierten Sprachen und objektorientierter Programmierung (OOP) entstanden Ansätze, welche verstärkt die Steigerung der Wiederverwendbarkeit von einmal entwickelten Software-Komponenten zum Ziel hatten [15]. Im Besonderen sind dies objektorientierte Software-Frameworks, in denen ein Rahmenwerk und eine Architekturhilfe für die Bewältigung eines bestimmten Aufgabenspektrums bereitgestellt wird [8]. Objektorientierte Constraint-Frameworks im Speziellen dienen dazu, OOP-Sprachen und Techniken zur Constraint-Verarbeitung zu verbinden und in unterschiedlichen Szenarien nutzbar zu machen. Ein Constraint-Framework ist eine Möglichkeit, Constraints als Inferenz-Mechanismus unabhängig von einer konkreten Domäne nutzbar für unterschiedliche Anwendungen zu machen. Ein Framework bietet hierfür allgemeine Mechanismen, die zur Nutzung durch eine bestimmte Anwendung an die jeweils spezielle Problemstellung angepasst werden können [21].

Durch ein Constraint-Framework wird ein allgemeiner Kontrollzyklus vorgegeben, in den unterschiedliche Lösungsverfahren je nach Bedarf eingebunden werden können. Allgemeine Verfahren zur Constraint-Verarbeitung sind innerhalb eines Frameworks in einer erweiterbaren Bibliothek bereits enthalten. Die Architektur eines Frameworks sollte dabei eine einfache Nutzung garantieren, und in diesem Fall die komplexen Mechanismen des CSP-Formalismus vor dem Benutzer weitestgehend verbergen [21].

4.2 Constraint-Lösungsstrategien

Flexibilität hinsichtlich der einzusetzenden Lösungsverfahren kann durch ein Konzept von modularen und austauschbaren Constraint-Lösungsstrategien erreicht werden. Zur Strukturierung des Constraint-Lösungsvorgangs wird dieser Prozess in drei Phasen eingeteilt: *Preprozessing*, *Konsistenzherstellung* und *Lösungssuche*. Diese Phasen spiegeln sich innerhalb von Constraint-Lösungsstrategien wieder (vgl. Abbildung 1). In der ersten Phase wird ein Preprozessing des Constraint-Problems vorgenommen. Dies kann sich z. B. auf die Binärisierung eines Constraint-Netzes oder die Zerlegung von Constraints in primitive Constraints beziehen, um anschließend darauf aufbauende Lösungsalgorithmen anwenden zu können. In der zweiten Phase werden Filter- bzw. Konsistenzalgorithmen zur Einschränkung der Domänen der Constraint-Variablen angewendet. Da dies allein i. A. nicht zu einer Lösung des Constraint-Problems führt, können in einer dritten Phase Suchverfahren zum Auffinden von Lösungen in den reduzierten Wertebereichen eingesetzt werden.

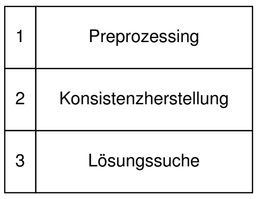


Abbildung 1. Aufbau einer Constraint-Lösungsstrategie

Zu beachten ist, dass in jeder Phase mehrere Einträge innerhalb einer Lösungsstrategie existieren können. So ist es z. B. möglich, mehrere Preprozessing-Schritte auf ein Problem anzuwenden, bevor Verfahren aus der nächsten Phase zum Einsatz kommen. Dies gilt ebenso für Konsistenz- und Suchverfahren.

Ebenso ist es möglich, dass für eine Phase innerhalb einer Strategie keine Einträge existieren. Nicht für alle Konsistenz- und Suchverfahren ist ein Preprozessing erforderlich. Gleichfalls kann ein Suchverfahren auch ohne vorherigen Filteralgorithmus angewendet werden, insbesondere wenn das Suchverfahren bereits Filtermechanismen enthält. Sind für eine Anwendung keine exakten Lösungen sondern nur eingeschränkte Wertebereiche erforderlich, kann auf ein Suchverfahren in der dritten Phase verzichtet werden. Mehrere Beispiele für mögliche Constraint-Lösungsstrategien sind in Abbildung 2 zu sehen.

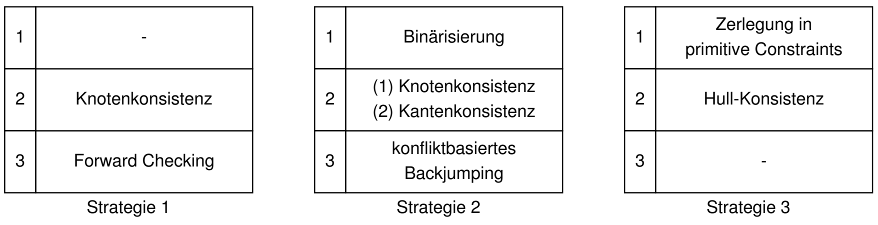


Abbildung 2. Beispiele für Constraint-Lösungsstrategien

Die Verwaltung derartiger Constraint-Lösungsstrategien muss von einer Komponente vorgenommen werden, die in der Lage ist, aufgrund einer klaren Spezifikation die Zuordnung der jeweiligen Strategien zu einzelnen Teilproblemen des gesamten Constraint-Problems vornehmen zu können.

4.3 Ein hybrides Constraint-System

Ein hybrides Constraint-System zeichnet sich dadurch aus, dass es in der Lage ist, ein hybrides Constraint Satisfaction Problem zu verarbeiten:

Definition 4. (Hybrides Constraint Satisfaction Problem)
Ein System zur Verarbeitung eines hybriden Constraint Satisfaction Problems H wird durch die Angabe von sieben Komponenten

$$H = (C, S, \delta, V_{fd}, D_{fd}, V_{int}, D_{int})$$

beschrieben. Dabei ist $C = \{C_1, \ldots, C_m\}$ *eine endliche Menge von Constraints und* $S = \{S_1, \ldots, S_n\}$ *eine endliche Menge von Constraint-Lösungsstrategien. Die Funktion* δ *ordnet jedem Constraint* C_i, $i \in \{1, \ldots, m\}$, *eine eindeutige Strategie* S_j, $j \in \{1, \ldots, n\}$, *zu:*

$$\delta : C_i \rightarrow S_j.$$

Der Bezeichner V_{fd} *steht für eine endliche Menge von FD-Variablen* $\{v_1, \ldots, v_k\}$, *mit denen die Wertebereiche* $D_{fd} = \{D_1, \ldots, D_k\}$ *mit* $\{v_1 : D_1, \ldots, v_k : D_k\}$ *assoziiert sind. Ebenso sind die Intervallvariablen* $V_{int} = \{v_1, \ldots, v_l\}$ *mit den Wertebereichen* $D_{int} = \{D_1, \ldots, D_l\}$ *mit* $\{v_1 : D_1, \ldots, v_l : D_l\}$ *assoziiert. Jedes Constraint* C_i *setzt eine Teilmenge der Variablen aus* V_{fd} *und* V_{int} *zueinander in Relation und beschränkt deren gültige Wertekombinationen auf eine Teilmenge des kartesischen Produkts ihrer Wertebereiche.*

Ein hybrides CSP vereinigt somit Constraints über Variablen mit finiten und infiniten Domänen. Jedem Constraint ist eine Lösungsstrategie zu dessen Verarbeitung zugeordnet. Dies führt zu einer Aufteilung des ursprünglichen Constraint-Problems in unterschiedliche Teilprobleme, welche durch die jeweils zuständige Constraint-Lösungsstrategie definiert werden (vgl. Abbildung 3).[2]

Ausführungsmodell Der „YACS Constraint-Manager" (YCM) ist die Komponente an der Schnittstelle zwischen einer vorhandenen Anwendung und dem YACS-Framework. Dem YCM obligt die Verwaltung der neu hinzukommenden unterschiedlichen Constraint-Netze, der Constraint-Lösungsstrategien, der Constraint-Solver und letztendlich der Steuerung des Lösungsprozesses.

Von dem aufrufenden System erhält der Constraint-Manager YCM die Informationen, welche Constraints mit welcher Strategie aufzulösen sind. Der Constraint-Manager aktiviert die entsprechenden Constraint-Lösungskomponenten und übergibt in der jeweiligen Bearbeitungsphase das entsprechende Constraint-Netz zur Verarbeitung an die dafür vorgesehene Komponente.

Der Prozess des Constraint-Lösens ist analog zum Aufbau der Constraint-Lösungsstrategien in drei Phasen unterteilt. In jeder Phase werden sequentiell jeweils die Constraint-Netze aller Strategien der Reihe nach bearbeitet. Das

[2] Teilprobleme entstehen, indem mehrere Constraints derselben Strategie zugeordnet werden.

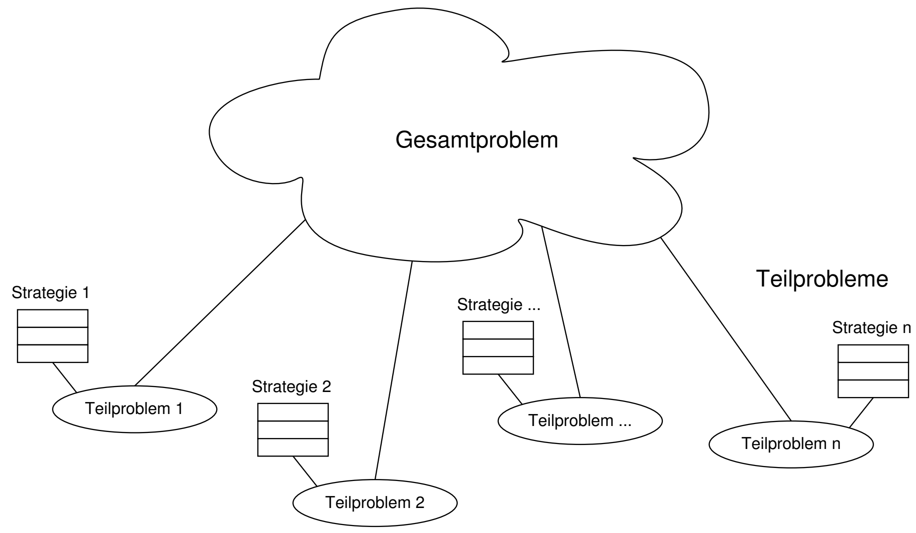


Abbildung 3. Zuständigkeiten unterschiedlicher Strategien für Teilbereiche des Constraint-Problems

heißt die in den Strategien angegebenen Constraint-Verfahren werden in den entsprechenden Phasen auf die zugehörigen Constraint-Netze angewendet.

– *Phase 1 (Preprozessing):* Das jeweilige Constraint-Netz wird wenn möglich vollständig konvertiert oder es wird festgestellt, dass es sich mit den gegebenen Algorithmen nicht umformen lässt.

– *Phase 2 (Konsistenzherstellung):* Es wird solange propagiert, bis keine Änderungen der Wertebereiche mehr eintreten (d. h. Konsistenz hergestellt ist) oder eine Inkonsistenz auftritt.

– *Phase 3 (Lösungssuche):* Die Suchalgorithmen finden die geforderten Lösungen oder stellen fest, dass keine Lösung existiert.

Maximale Flexibilität wird erreicht, wenn in der Wissensbasis für jedes Constraint der Name einer entsprechenden Strategie zu dessen Lösung angegeben werden kann. Der Name der jeweiligen Strategie entspricht dabei einem eigenen Teilbereich des ursprünglichen Constraint-Problems. Jede Strategie ist für die Verarbeitung eines (Sub-)Constraint-Netzes vorgesehen. Bei der Initialisierung wird durch den Constraint-Manager von YACS sichergestellt, dass alle geforderten Strategien existieren und angewendet werden können.

Die Strategien werden separat von der Wissensbasis des Konfigurators definiert. Die Anwendung übergibt die Constraints jeweils mit dem Namen der zugehörigen Strategie an den YACS Constraint-Manager. Dieser generiert daraus die unterschiedlichen Constraint-Netze und wendet die entsprechenden Constraint-Lösungsstrategien an.

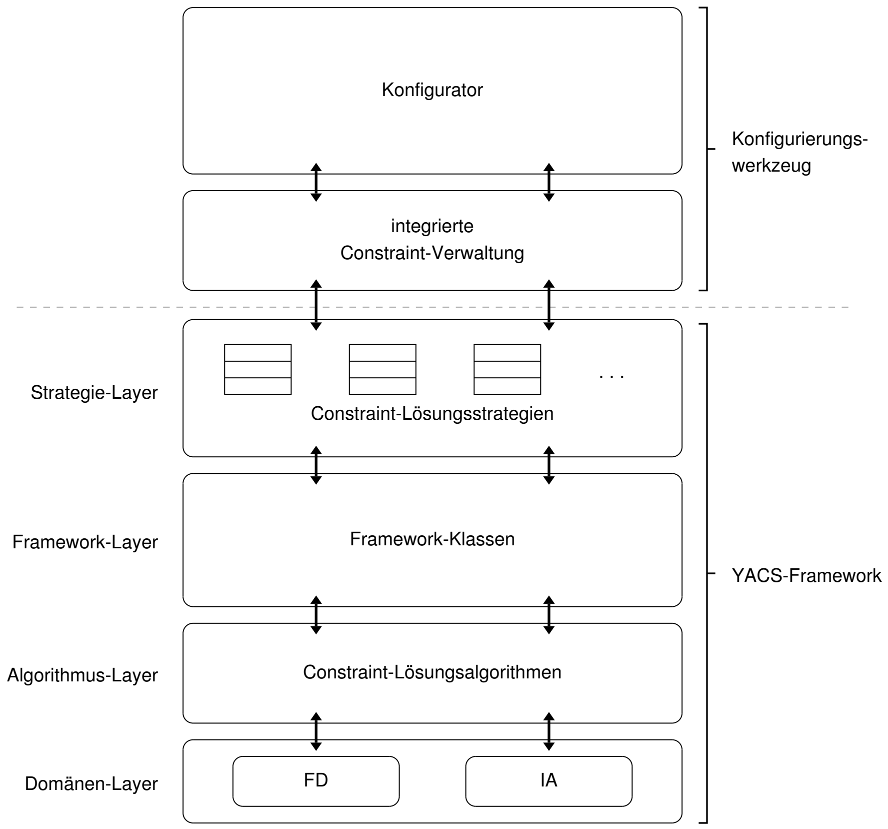


Abbildung 4. Systemarchitektur von YACS

4.4 Architektur des YACS-Frameworks

Das YACS-Framework stellt eine modulare und wiederverwendbare Constraint-Lösungskomponente dar. YACS ist ein hybrides System für den flexiblen Einsatz von Constraint-Lösungsverfahren für finite und infinite Domänen. Die Lösungsverfahren sind eingebettet innerhalb einer strategiebasierten, modularen Framework-Architektur:

– *Constraint-Lösungsstrategien:* Der flexible Einsatz von Constraint-Lösungsverfahren wird über ein Strategiekonzept realisiert. Abstrahiert von den eigentlichen Lösungsalgorithmen können von dem Wissensingenieur problemabhängig bzw. anwendungsspezifisch unterschiedliche Constraint-Lösungsstrategien eingesetzt werden. Diese Lösungsstrategien müssen vorab in Abhängigkeit von den vorhandenen Lösungsverfahren definiert werden.

– *Framework-Architektur:* Durch die Framework-Architektur wird sichergestellt, dass Lösungsverfahren flexibel ausgetauscht und zudem auf einfache

Weise neue Lösungsalgorithmen implementiert bzw. Fremdsysteme integriert werden können. Das YACS-Framework stellt hierfür einen geeigneten Rahmen mit einheitlichen Schnittstellen bereit.

In Abbildung 4 ist eine Übersicht über die Systemarchitektur von YACS, angebunden an ein wissensbasiertes Konfigurierungswerkzeug, zu sehen. Aufsetzend auf einem *Domänen-Layer*, einer Umgebung zur arithmetischen Verarbeitung von finiten Domänen (FD) und reellwertigen Intervallen (eine Intervallarithmetik, kurz IA), werden die eigentlichen Algorithmen zum Auflösen von Constraint-Problemen implementiert (*Algorithmus-Layer*). Constraint-Verfahren aus Fremdsystemen können an dieser Stelle über Wrapper-Klassen eingebunden werden. Die Algorithmen bzw. die sie umschließenden (Wrapper)-Klassen müssen wiederum den Schnittstellen des *Framework-Layers* von YACS genügen.

Der durch das Framework vereinheitlichte Zugriff auf Constraint-Lösungsverfahren ermöglicht es dem *Strategie-Layer*, dem Anwender bzw. dem übergeordnetem System eine flexible Auswahl an Lösungsverfahren anbieten zu können. Abstrahiert von den Lösungsverfahren können auf dieser Ebene aus einer Reihe vordefinierter Constraint-Lösungsstrategien problemabhängig die für die jeweilige Anwendung geeigneten Strategien ausgewählt werden.

YACS wurde in der Programmiersprache Java implementiert und prototypisch in das Konfigurierungswerkzeug EngCon integriert. Die benötigten Constraint-Lösungsstrategien werden separat mittels XML definiert, eingelesen und angewendet. Die Intervallarithmetik wurde mit Hilfe der Bibliothek *IAMath*[3] realisiert. Der Prototyp von YACS ist im Internet verfügbar.[4]

5 Verwandte Arbeiten

Bekannte Constraint-Bibliotheken mit integrierten Constraint-Solvern sind z. B. die *C-Lib*[5], die *Java Constraint Library* (JCL)[6] [27] und der *IASolver*[7] [12]. Diese Systeme sind allerdings nicht hybrid, d. h. es lassen sich ausschließlich entweder finite oder infinite Domänen verarbeiten. Zudem ist eine inkrementelle Constraint-Verarbeitung nicht vorgesehen und die Erweiterbarkeit ist aufgrund der fehlenden Framework-Architektur nur eingeschränkt möglich.

Constraint-Frameworks im eigentlichen Sinn sind z. B. die Systeme *BackTalk* [20] und *POOC* [24]. Beide sind allerdings ebenfalls auf finite Domänen beschränkt. *POOC* fokussiert zudem den Bereich *Constraint Programming* (CP), d. h. es werden in erster Linie die für die Programmierung mit Constraints benötigten *global constraints* angeboten.

In [13] wird ein allgemeines und formales Schema für die Kombination von Constraint-Systemen und die Kooperation von Constraint-Solvern zur Behand-

[3] `http://interval.sourceforge.net/interval/java/ia_math/`
[4] `http://sourceforge.net/projects/constraints`
[5] `http://ai.uwaterloo.ca/~vanbeek/software/software.html`
[6] `http://liawww.epfl.ch/JCL/`
[7] `http://www.cs.brandeis.edu/~tim/Applets/IAsolver.html`

lung von Constraints mit vermischten Wertedomänen vorgestellt. Strategien werden hier in Form von frei modellierbaren *Kooperationsstrategien* eingesetzt. Mit diesen kann beschrieben werden, welche Constraint-Verfahren wie und in welcher Reihenfolge (sequentiell/parallel) kombiniert werden.

In [18] wird eine Sprache namens *BALI* zur Steuerung von Kooperationen entwickelt. *BALI* erlaubt es, die Kooperation unterschiedlicher Constraint-Solver auf hoher Ebene durch eine eigene Sprache zu beschreiben, und auf diese Weise effizient neue Prototypen von kooperierenden Solvern zu erstellen. Neben unterschiedlichen Kooperationsprimitiven, welche die sequentielle, die unabhängig parallele und die nebenläufige Ausführung von Constraint-Lösungsmechanismen erlauben, bietet *BALI* mehrere Lösungsstrategien an. Darunter befindet auch sich eine inkrementelle Variante [18].

6 Zusammenfassung und Ausblick

Aufgrund einer Vielzahl von Constraint-Lösungsverfahren und möglicher Kombinationen derselben, deren unterschiedlichen Eigenschaften und der problemabhängigen bzw. anwendungsspezifischen Effizienz unterschiedlicher Verfahren, ist zur Unterstützung der wissensbasierten Konfigurierung eine Komponente notwendig, mit der sich flexibel, je nach Problemstellung unterschiedliche Constraint-Lösungsmechanismen einsetzen lassen.

Das Framework-Konzept von YACS bietet eine flexible und nutzerfreundliche Architektur zur Implementierung von Constraint-Lösungsverfahren. Die Framework-Architektur wiederum wird in Bezug auf die Abstraktion von den tatsächlich eingesetzten Verfahren durch ein Strategiekonzept ergänzt.

In den Constraint-Lösungsstrategien werden die tatsächlich einzusetzenden Lösungsverfahren definiert. Dies geschieht problemabhängig und flexibel je nach Anwendung und Einsatzzweck. Der Wissensingenieur kann sich somit bei der Erstellung der Wissensbasis auf vordefinierte und dokumentierte Constraint-Lösungsstrategien stützen, und diese zur Behandlung der im Rahmen der Konfigurierung entstehenden Constraint-Probleme gezielt einsetzen. Sollten Anpassungen notwendig werden, so ist eine einfache Wartung, Pflege und auch Neuimplementierung oder -anbindung von Constraint-Lösungsverfahren durch eine modulare Struktur und die Framework-Architektur gewährleistet.

Die Möglichkeit, auf unkomplizierte und durchschaubare Art und Weise Anpassungen an YACS bzw. an den Constraint-Lösungsstrategien von YACS vornehmen zu können, kann sich positiv auf die Akzeptanz der Lösung bei den zuständigen Wissensingenieuren bei gleichzeitig höchstmöglicher Flexibilität auswirken. Das YACS-Framework wurde zudem für den Anwendungsfall der strukturbasierten Konfigurierung entwickelt und getestet. YACS unterstützt daher die domänenspezifischen Eigenheiten wie ein inkrementell anwachsendes Constraint-Netz und sowohl finite als auch infinite Wertebereiche.

Zur Erhöhung der Flexibilität ist die Anwendung strategiebasierter Ansätze ein gängiges Mittel zur Steuerung der Propagation und Lösungssuche von kooperierenden Constraint-Solvern. Das Konzept für YACS wurde aus der Idee heraus geboren, größtmögliche Flexibilität hinsichtlich der einzusetzenden Constraint-Lösungsmechanismen zu bieten. Weniger im Vordergrund steht der Aspekt der flexiblen *Steuerung* von Kooperationen unterschiedlicher Lösungsverfahren. Weitere Arbeiten werden sich daher damit befassen, das Ausführungsmodell flexibler zu gestalten, so dass z. B. neben einer sequentiellen Abfolge von Constraint-Lösungsalgorithmen eine parallele Verarbeitung gewährleistet werden kann.

Ein weiteres Betätigungsfeld bietet die Untersuchung der „Überlappung“ von Teilproblemen unterschiedlicher Constraint-Lösungsstrategien. Eine Überlappung entsteht, wenn eine Variable in strukturell unterschiedlichen Teilproblemen existiert, d. h. in Constraints, die unterschiedlichen Lösungsstrategien zugeordnet sind. Werden bei Überlappungen infinite Variablen mit intervallwertigen Domänen von diskreten Werten finiter Variablen beschränkt und umgekehrt, besteht die Aufgabe darin, ein *heterogenes Constraint-Problem* zu verarbeiten (vgl. [1], [9]). In jedem Fall ist bei Überlappungen von Teilproblemen ein *Meta-Constraint-Solver* erforderlich, wenn globale Lösungen generiert werden sollen.

Eine ausführliche Fassung dieser Ausarbeitung ist in [22] zu finden.

Literatur

1. Benhamou, Frédéric: Heterogeneous Constraint Solving. In: Hanus, Michael (Hrsg.) ; Rodríguez-Artalejo, Mario (Hrsg.): *Proceedings of the 5th International Conference on Algebraic and Logic Programming (ALP'96), Aachen, 25.–27. September 1996.* Berlin, Heidelberg, New York : Springer Verlag, 1996 (LNCS 1139), S. 62–76. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Ben_ALP96.pdf`. – ISBN 3-540-61735-3
2. Benhamou, Frédéric: Interval Constraints. In: Floudas, Christodoulos A. (Hrsg.) ; Pardalos, Panos M. (Hrsg.): *Encyclopedia of Optimization* Bd. 3. Dordrecht, Netherlands : Kluwer Academic Publishers, August 2001, S. 45–48. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/Ben99_EoO.pdf`. – ISBN 0-7923-6932-7
3. Benhamou, Frédéric ; McAllester, David ; Van Hentenryck, Pascal: CLP(Intervals) Revisited. In: Bruynooghe, Maurice (Hrsg.): *Logic Programming, Proceedings of the 1994 International Symposium (ILPS'94), Ithaca, New York, USA, 13.–17. November 1994.* Cambridge, Massachusetts, USA : The MIT Press, 1994, S. 124–138. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/benhamou/PAPERS/BenMcAlVHen94.pdf`. – ISBN 0-262-52191-1
4. Cleary, John G.: Logical Arithmetic. In: *Future Computing Systems* 2 (1987), Nr. 2, S. 125–149. – ISSN 0266-7207
5. Cunis, Roman (Hrsg.) ; Günter, Andreas (Hrsg.) ; Strecker, Helmut (Hrsg.): *Das PLAKON-Buch, Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen.* Berlin, Heidelberg, New York : Springer Verlag, 1991 (Informatik-Fachberichte, Subreihe Künstliche Intelligenz 266). – 279 S. – ISBN 3-540-53683-3
6. Davis, Ernest: Constraint Propagation with Interval Labels. In: *Artificial Intelligence* 32 (1987), Juli, Nr. 3, S. 281–331. – ISSN 0004-3702
7. Dechter, Rina: *Constraint Processing.* San Francisco, California, USA : Morgan Kaufmann Publishers, 2003 (The Morgan Kaufmann Series in Artificial Intelligence) – xx + 481 S. – ISBN 1-558-60890-7

8. Gamma, Erich ; Helm, Richard ; Johnson, Ralph ; Vlissides, John: *Entwurfsmuster – Elemente wiederverwendbarer objektorientierter Software*. 1. Aufl. München : Addison-Wesley, 1996. – xx + 479 S. – ISBN 3-89319-950-0
9. Gelle, Esther ; Faltings, Boi V.: Solving Mixed and Conditional Constraint Satisfaction Problems. In: *Constraints, An International Journal* 8 (2003), April, Nr. 2, S. 107–141. – URL `http://liawww.epfl.ch/Publications/Archive/Gelle2003.pdf`. – ISSN 1383-7133
10. Günter, Andreas: KONWERK – ein modulares Konfigurierungswerkzeug. In: Richter, Michael M. (Hrsg.) ; Maurer, Frank (Hrsg.): *Expertensysteme 95, Beiträge zur 3. Deutschen Expertensystemtagung (XPS'95), Kaiserslautern, 1.–3. März 1995*. Sankt Augustin : Infix Verlag, 1995, S. 1–18. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/konwerk.pdf`. – ISBN 3-932-79295-5
11. Günter, Andreas ; Kühn, Christian: Knowledge-Based Configuration – Survey and Future Directions. In: Puppe, Frank (Hrsg.): *Knowledge-Based Systems – Survey and Future Directions, Proceedings of the 5th Biannual German Conference on Knowledge-Based Systems (XPS 1999), Würzburg, 3.–5. März 1999*. Berlin, Heidelberg, New York : Springer Verlag, 1999 (LNCS 1570), S. 47–66. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/xps-99.pdf`. – ISBN 3-540-65658-8S
12. Hickey, Timothy J. ; Qiu, Zhe ; Emden, Maarten H. van: Interval Constraint Plotting for Interactive Visual Exploration of Implicitly Defined Relations. In: *Reliable Computing* 6 (2000), Februar, Nr. 1, S. 81–92. – URL `http://csr.uvic.ca/~vanemden/Publications/graphics.pdf`. – ISSN 1385-3139
13. Hofstedt, Petra: Cooperating Constraint Solvers. In: Dechter, Rina (Hrsg.): *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000), Singapore, 18.–21. September 2000*. Berlin, Heidelberg, New York : Springer Verlag, 2000. (LNCS 1894), S. 520–524. – URL `http://uebb.cs.tu-berlin.de/~ph/ph.papers/cp2000.pdf` – ISBN 3-540-41053-8
14. Hyvönen, Eero: Constraint Reasoning Based on Interval Arithmetic: The Tolerance Propagation Approach. In: *Artificial Intelligence* 58 (1992), Dezember, Nr. 1–3, S. 71–112. – Special Volume on Constraint Based Reasoning. – ISSN 0004-3702
15. Johnson, Ralph E.: Components, Frameworks, Patterns. In: Harandi, Medhi (Hrsg.): *Proceedings of the 1997 Symposium on Software Reusability (SSR'97), Boston, Massachusetts, USA, 17.–20. Mai 1997*. New York, NY, USA : ACM Press, 1997, S. 10–17. – URL `ftp://st.cs.uiuc.edu/pub/papers/frameworks/framework97.ps`. – ISBN 0-89791-945-9 – Zugl.: ACM SIGSOFT Software Engineering Notes, 22 (1997), Nr. 3, S. 10–17. – ISSN 0163-5948
16. Lhomme, Olivier: Consistency Techniques for Numeric CSPs. In: Bajcsy, Ruzena (Hrsg.): *Proceedings of the 13th International Joint Conference on Artificial Intelligence (IJCAI'93), Chambéry, France, 28. August – 3. September 1993*. San Mateo, California, USA : Morgan Kaufmann Publishers, 1993, S. 232–238. – ISBN 1-55860-300-X
17. Mackworth, Alan K.: Consistency in Networks of Relations. In: *Artificial Intelligence* 8 (1977), Februar, Nr. 1, S. 99–118. – ISSN 0004-3702
18. Monfroy, Eric: The Constraint Solver Collaboration Language of BALI. In: Gabbay, Dov (Hrsg.) ; Rijke, Maarten de (Hrsg.): *Proceedings of the 2nd International Workshop Frontiers of Combining Systems (FroCoS'98), Amsterdam, The Netherlands, 2.–4. Oktober 1998*. Baldock, Hertfordshire, UK : Research Studies Press, 2000 (Studies in Logic and Computation Vol. 7), S. 211–230. – URL `http://www.sciences.univ-nantes.fr/info/perso/permanents/monfroy/Papers/frocos98.ps.gz`

19. Ranze, Christoph ; Scholz, Thorsten ; Wagner, Thomas ; Günter, Andreas ; Herzog, Otthein ; Hollmann, Oliver ; Schlieder, Christoph ; Arlt, Volker: A Structure-Based Configuration Tool: Drive Solution Designer – DSD. In: Dechter, Rina (Hrsg.) ; Sutton, Rich (Hrsg.) ; Kearns, Michael (Hrsg.) ; Chien, Steve (Hrsg.) ; Riedl, John (Hrsg.): *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI'02) and 14th Conference on Innovative Applications of Artificial Intelligence (IAAI'02), Edmonton, Alberta, Canada, 28. Juli – 1. August 2002.* Menlo Park, California, USA/Cambridge, Massachusetts, USA : AAAI Press/The MIT Press, September 2002, S. 845–852. – URL `http://www.hitec-hh.de/ueberuns/home/aguenter/literatur/IAAI2002.pdf`
20. Roy, Pierre ; Liret, Anne ; Pachet, François: Constraint Satisfaction Problems Framework. In: Fayad, Mohamed E. (Hrsg.) ; Schmidt, Douglas C. (Hrsg.) ; Johnson, Ralph E. (Hrsg.): *Implementing Application Frameworks: Object-Oriented Frameworks at Work.* Chichester, London, New York : John Wiley & Sons, September 1999 (Wiley Computer Publishing), Kap. 17, S. 369–401. – ISBN 0-471-25201-8
21. Roy, Pierre ; Liret, Anne ; Pachet, François: The Framework Approach for Constraint Satisfaction. In: *ACM Computing Surveys (CSUR)* 32 (2000), März, Nr. 1es. – URL `http://doi.acm.org/10.1145/351936.351949`. – CSUR Electronic Symposium on Object-Oriented Application Frameworks – Artikel Nr. 13. – ISSN 0360-0300
22. Runte, Wolfgang: *YACS: Ein hybrides Framework für Constraint-Solver zur Unterstützung wissensbasierter Konfigurierung*, Universität Bremen, Diplomarbeit, 27. Januar 2006. – xx + 405 S. – URL `http://www.informatik.uni-bremen.de/~woru/pub/diplom/runte06diplom.pdf`
23. Sabin, Daniel ; Weigel, Rainer: Product Configuration Frameworks – A Survey. In: *IEEE Intelligent Systems* 13 (1998), Juli/August, Nr. 4, S. 42–49. – ISSN 1094-7167l
24. Schlenker, Hans ; Ringwelski, Georg: POOC: A Platform for Object-Oriented Constraint Programming. In: O'Sullivan, Barry (Hrsg.): *Recent Advances in Constraints, Joint ERCIM/CoLogNET International Workshop on Constraint Solving and Constraint Logic Programming (CSCLP'02), Cork, Ireland, 19.–21. Juni 2002, Selected Papers.* Berlin, Heidelberg, New York : Springer Verlag, 2003 (LNCS 2627), S. 159–170. – URL `http://4c.ucc.ie/web/upload/publications/inProc/ercim02pooc.pdf`. – ISBN 3-540-00986-8
25. Stumptner, Markus: An Overview of Knowledge-Based Configuration. In: *AI Communications (AICOM)* 10 (1997), Juli, Nr. 2, S. 111–125. – ISSN 0921-7126
26. Syska, Ingo ; Cunis, Roman: Constraints in PLAKON. In: Cunis, Roman (Hrsg.) ; Günter, Andreas (Hrsg.) ; Strecker, Helmut (Hrsg.): *Das PLAKON-Buch, Ein Expertensystemkern für Planungs- und Konfigurierungsaufgaben in technischen Domänen.* Berlin, Heidelberg, New York : Springer Verlag, 1991 (Informatik-Fachberichte, Subreihe Künstliche Intelligenz 266), Kap. 6, S. 77–91. – ISBN 3-540-53683-3
27. Torrens, Marc ; Weigel, Rainer ; Faltings, Boi V.: Distributing Problem Solving on the Web Using Constraint Technology. In: *Proceedings of the 10th IEEE International Conference on Tools with Artificial Intelligence (ICTAI'98), Taipei, Taiwan, 10.–12. November 1998.* Los Alamitos, California, USA, : IEEE Computer Society Press, 1998, S. 42–49. – URL `http://liawww.epfl.ch/Publications/Archive/Torrens1998.pdf`. – ISBN 0-7803-5214-9
28. Waltz, David L.: Understanding Line Drawings of Scenes with Shadows. In: Winston, Patric Henry (Hrsg.): *The Psychology of Computer Vision.* New York, NY, USA : McGraw-Hill, 1975, S. 19–91. – ISBN 0-07-071048-1

Modelling and Solving Workforce Scheduling Problems

Jürgen Sauer
Business Engineering, Wirtschaftsinformatik,
Universität Oldenburg, 26111 Oldenburg
juergen.sauer@uni-oldenburg.de

René Schumann
OFFIS, Betriebliches Informationsmanagement,
Escherweg 2, 26121 Oldenburg
rene.schumann@offis.de

Abstract. Workforce scheduling is known as a complex and highly constraint problem. In the last few years the problem even become more complex due to the deregulation of laws limiting working times, flexibility and opening hours for shops at least in Germany. This paper describes an intelligent workforce scheduling component which shall be incorporated in a commercial workforce planning and scheduling system. Main points are the modelling of the workforce scheduling problem and a heuristic to provide a solution that regards the given hard and soft constraints.

Introduction

As cost of personnel has become one of the most significant factors in most service oriented companies it seems clear that one of the main goals when scheduling or improving business processes is to minimize these costs. This also means that one tries to use the minimum necessary personnel to fulfil the tasks to be done but this in the required quality. On the other hand the use of high quality and professional staff brings main advantages in customer satisfaction. Additionally, we have to respect the fact that effective deployment of workforce has not only direct monetary effects, also a higher motivation of the employees can be achieved [Geb04].

Therefore workforce scheduling has become increasingly important to provide the firms with schedules that present the right number (which means minimal) of the right persons (which means with the necessary qualification) at the right time. For the customers the goals are a little different: they demand a high number of highly qualified service personnel in order not to have to wait to long for being served. And last but not least, a schedule that respects the wishes and personal plans of the employees will lead to more satisfaction on their side.

But several other constraints have to be regarded as well when trying to schedule the staff for a given time period. These reach from guaranteeing a minimal number of staff to all legal restrictions e.g. those for breaks.

Thus we start with a description of a workforce scheduling problem. This is likely to be found in several stores, retail or other service companies. The main goal of workforce scheduling is to find a schedule which is basically an assignment of persons to time intervals. With the schedule the demand of staff shall be fulfilled and all the other constraints have to be regarded and most often a cost function shall be optimized.

Among the requirements and constraints are:

- the number of personnel must fit the demand,
- the qualification of the scheduled staff must fit the needs,
- the different types of contracts have to be regarded,
- the legal regulations have to be obeyed, e.g.
  - staff should not work longer than specified in contract
  - staff should not work more than one shift per day
  - holidays have to be regarded,
- different shift models may occur (one, two, three shifts etc.),
- breaks have to be regarded.

An important sub-problem when scheduling the workforce is the break placement problem, which has to be done for each assignment of each employee. While an employee has a break he can not cover the workforce requirements. So breaks reduce the number of available employees. As a consequence break placement is an aspect which has to be coped with simultaneous to the staff assignment. In the worst case a bad break placement increases existing periods of understaffing, in the best case one can reduce overstaffing. But break placement has to respect a number of rules based on laws and other restrictions as well.

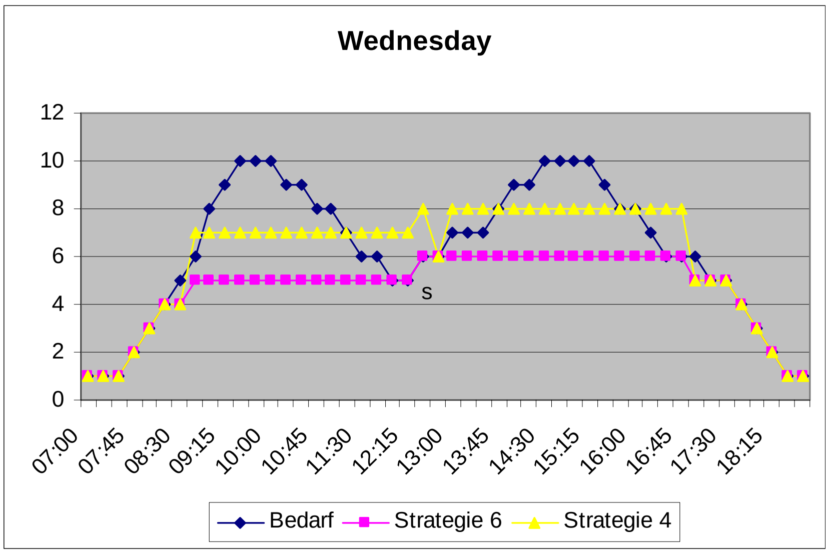


Figure 1. Workforce scheduling: demand and possible solutions

The goals the schedule shall reach can be formulated in terms of

- overtime: try to have no overtime, which means that there are time periods with more persons than needed
- undersupply: try to have no undersupply, which means that there are time periods with less persons than needed
- minimal cost schedule: try to have a schedule where the least costs erase.

Most often a combination of these goals seems adequate. The demand is given by typical numbers e.g. calculated from forecasts. Figure 1 shows such a demand for one day (Bedarf) and two possible schedules (Strategie 4, Strategie 6) that do not meet the demands. The schedule here is presented cumulative not by the list of persons which is another kind of presentation.

The rest of the paper describes an intelligent workforce scheduling component. First a model on the basis of other scheduling problems is presented. Then the heuristic strategy for constructing a weekly schedule is described.

Modelling the Workforce Scheduling Problem

According to our modelling approach of scheduling problems we will also use the 7-tuple (O, R, P, HC, SC, G, E) used for several scheduling problems [Sau06] to describe the workforce scheduling problem. For the workforce scheduling example the sets mean:

- **O**rders
  are the demand of personnel, i.e. the given workload profile
- **R**esources
  are the staff with their individual quality profiles and their individual contracts
- **P**roducts
  are the services the staff can provide
- **H**ard **C**onstraints
  are legal and other restrictions, e.g. maximum working hours, maximum numbers of shifts, not to work in two following shifts
- **S**oft **C**onstraints
  represent the constraints we want to achieve, e.g. not to have overtimes, not to have undersupply, have a fair distribution of the workload to the personnel, sometimes overtime may be allowed, sometimes undersupply may be possible, etc.
- **G**oal functions
  Goal functions can give an impression of the schedule quality. They may be time oriented, e.g. sum of overtimes, sum of overtimes and sum of undersupply, etc.
- **E**vents
  events have to be formulated for the reactive scheduling part of the workforce scheduling problem, e.g. ill persons, late persons, persons missing

for other reasons, changes in demand, etc. In our first version the reactive workforce scheduling problem is not regarded.

Compared to other problem representation based on this modelling approach, e.g. transportation scheduling [Sto04], here the resources have to be described in more detail. The schedule to be created shows the temporal assignment of the personnel.

Search Space for the Workforce Scheduling Problem

According to the modeling of the workforce scheduling problem the problem space can be presented as an And/Or tree. The tree represents the problem structure described above and already includes some of the constraints. One of the main differences to the other scheduling problems is that that combination of persons working together is not known in advance. This means that on the level of personnel group a subset of the personnel is to be found, i.e. a node represents a subset of the personnel. The solution is a subtree of the And/Or-tree for which holds:

- it contains the root node,
- for every And-node all successors are in the solution
- for every Or-node one successor is in the solution.

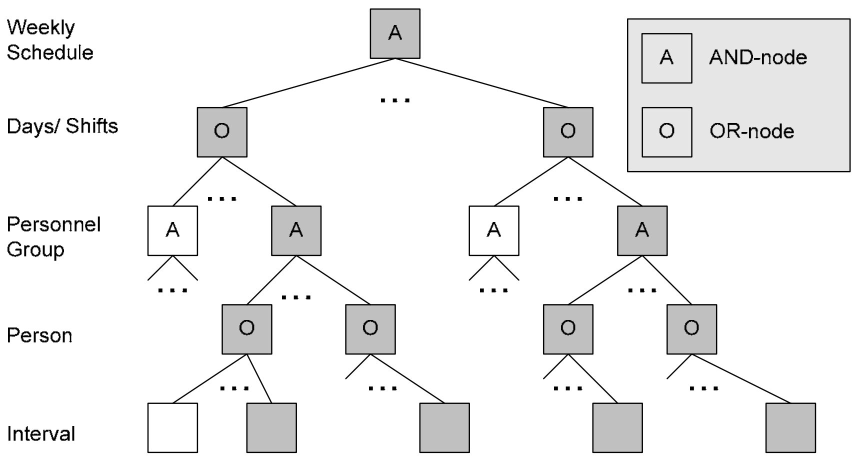


Figure 2. AND-/OR- tree for workforce scheduling

In figure 2 a solution is depicted by the darkened nodes. The complexity of the solution space [Sau04] for such presentation is

$L = (I^P * PG)^D$, with

L	number of solutions
D	number of days/ shifts
PG	number of staff groups, in the worst case this may be 2^P

- P number of persons
- I number of intervals per person.

This means that even for small problem sizes the complexity of the search space is very high. Additionally, it is not possible to estimate the impact of the constraints on the size of the solution space.

Heuristic for solving the WSP

The general phases of the workforce scheduling process as presented by Ernst et. al. [EJKS04] are:

- demand modelling
- days off scheduling
- shift scheduling
- line of work scheduling
- task assignment
- staff assignment

Within the project only two of the problems have to be solved, because the other tasks are already exist. In the retail trade business one can state a flexible demand model, which means that forecasting methods have to be used to determine future demand. The workforce requirements forecasting is done by an already existing module and gives a demand for a fixed period, in this case a week, showing the employees needed for the time periods of the days.

```
//** Basic workforce scheduling strategy
WHILE days to schedule {
       select most difficult day to schedule
       calculate maximum interval
       select staff group for the day
       WHILE demand {
              IF conflict
                 THEN solve_conflict
                 ELSE
                      select best suited person
                      select best suited interval
                      plan person
                      reduce demand
       }
       remove scheduled day
}
```

Figure 3. Basic scheduling strategy

Main input-data is the set of employees, each of them with potential different restrictions for assignment and break placement. This set of employees consists of two subsets. First is the primary pool, containing all employees working mainly for the organizational entity for which the schedule is computed. The second subset, called the secondary pool consists of employees that may be scheduled additionally but mainly are working in other entities.

In a first step several heuristics were checked that could be useful for solving the workforce scheduling problem [GT04]. Alternatively a solution on the basis of constraint programming packages was evaluated. But in most cases several constraints especially individual preferences could not be realized by the existing approaches. Thus a constructive heuristic was developed that incorporates the scheduling behaviour of human planners as well as it tries to respect the individual constraints of the employees.

The main scheduling scheme looks as presented in figure 3. The basic approach is to construct the schedule in several steps, starting with a schedule neglecting breaks and in a second phase adding the breaks. Another general heuristic used is that of problem decomposition, i.e. the schedule is constructed day by day. The main scheduling heuristic then tries to find the right person for a given time period.

In the heuristic scheduling knowledge is incorporated in the “select” statements and in the conflict resolution, e.g.

- Select most difficult day: a difficult day is a day with high demands in different times
- Select staff group: the staff group is composed of those persons that can be scheduled principally for that day and then reduced to the persons in the schedule
- Select best suited person: this combines a number of rules, e.g. persons with most remaining working time, persons that exactly fit an interval, persons with longest remaining working time interval, persons with preferences on that interval
- Select best suited interval: within the demanded interval a personal interval is chosen that best fits the demand and the working time constraints of the person.

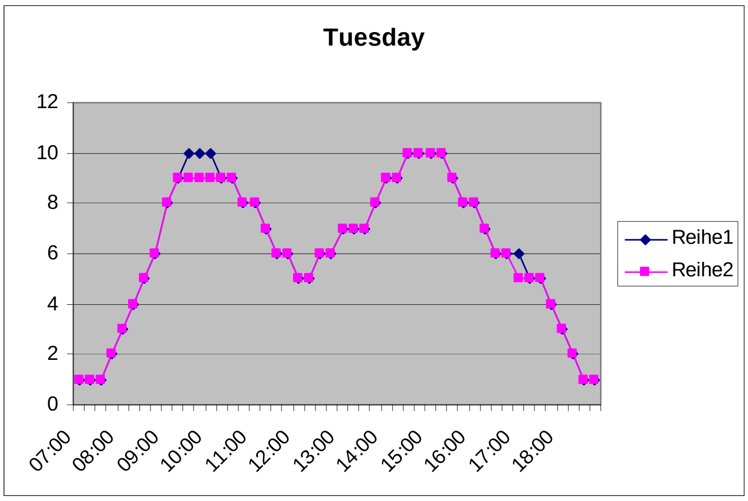


Figure 4. Matching curve

In a second phase the necessary breaks are added to the schedule, which sometimes lead to additional requirements in personnel, which then also is added to the schedule.

The result is a daily workforce schedule, that shows who is working in which shift and time interval. Figure 4 shows a matching curve for a schedule showing an “optimal” result.

Related Work

Often the retail trade workforce scheduling problem is treated as a scenario comparable to the workforce scheduling problem in call centers. And in fact there are a number of similarities [EJKS04]:

- workforce requirements are dynamic fluctuating and therefore change from day to day
- shifts can change in their length and starting points
- over- and understaffing are possible and shall be minimized and can therefore be treated as a soft constraint
- the number of possible shifts can become intractable high.

Generally, several systems for staff scheduling, crew scheduling and nurse rostering have been proposed and some systems are available at the market. But in nearly all cases the solution is restricted to a dedicated problem and thus only concepts and ideas can be used for other similar problem scenarios.

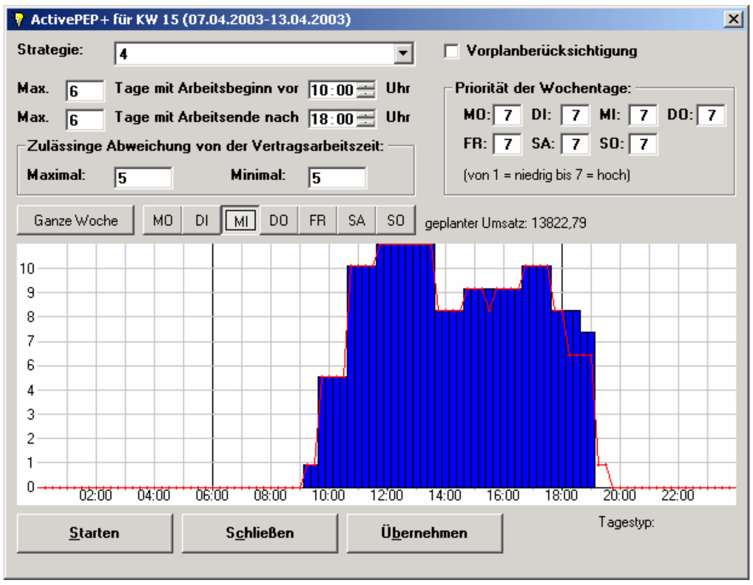
Most commercial systems have only simple scheduling capabilities and leave the scheduling tasks to the user of the system. But surprisingly there has not been done much research done in the area [EJK+04]. Most systems base on the constraint based approach, i.e. they see workforce scheduling as a constraint satisfaction problem and all constraints used are interpreted as hard constraints. Examples of such systems are systems developed with ILOG tools [Har07, ILOG07] or standalone systems like OC:Planner [Tol07]. Other work tries to solve the staff scheduling problem with integer programming tools but here only a few of the constraints, especially of the soft constraints, are regarded [BBD03].

Summary and Future Work

Within the project a first prototype of a heuristic staff scheduler was developed which is now part of a commercial product in the staff planning area. The system combines general heuristic strategies with specific heuristic rules from the human schedulers and presents a solution that is visualized in an interactive staff planning system. Figures 5 and 6 give an impression of the user interface. Figure 5 shows the cumulative view of the demand and its fulfilment, figure 6 shows the detailed information for the staff.

In next steps it has to be evaluated how flexible and extensible the approach is, i.e. how complex it is to integrate new requirements or constraints.

Further research can be done on the integration of heuristic based and constraint-based approaches to solve the workforce scheduling problems.


ActivePEP+ für KW 15 (07.04.2003-13.04.2003)

Strategie: 4

Vorplanberücksichtigung

Max. 6 Tage mit Arbeitsbeginn vor 10:00 Uhr

Max. 6 Tage mit Arbeitsende nach 18:00 Uhr

Zulässinge Abweichung von der Vertragsarbeitszeit:

Maximal: 5 Minimal: 5

Priorität der Wochentage:

MO: 7 DI: 7 MI: 7 DO: 7

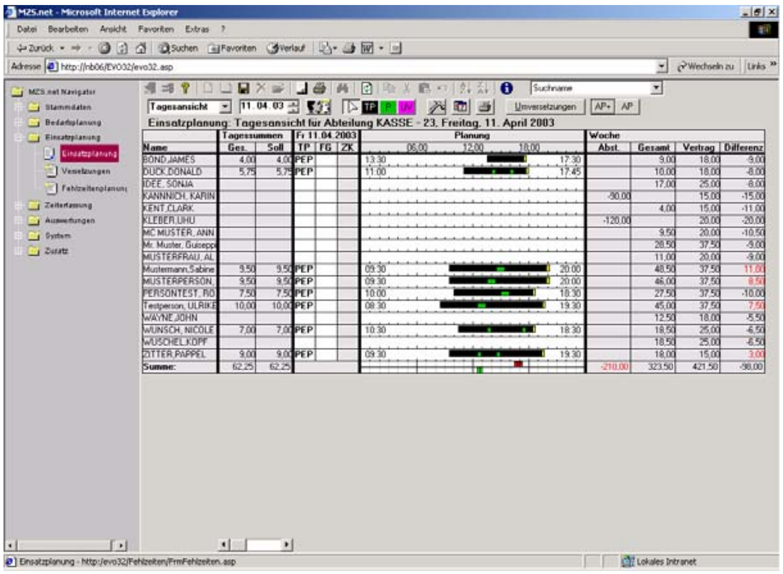
FR: 7 SA: 7 SO: 7

(von 1 = niedrig bis 7 = hoch)

geplanter Umsatz: 13822,79

Tagestyp:

Figure 5. Matching curve within the goal system


MZS.net - Microsoft Internet Explorer

Adresse http://nb06/EVO32/evo32.asp

Tagesansicht 11.04.03

Einsatzplanung: Tagesansicht für Abteilung KASSE - 23. Freitag, 11. April 2003

	Tagessummen		Fr 11.04.2003			Planung	Woche			
Name	Ges.	Soll	TP	FG	ZK	06:00 12:00 18:00	Abst.	Gesamt	Vertrag	Differenz
BOND,JAMES	4,00	4,00	PEP			13:30 17:30		9,00	18,00	-9,00
DUCK,DONALD	5,75	5,75	PEP			11:00 17:45		10,00	18,00	-8,00
IDEE, SONJA								17,00	25,00	-8,00
KANNNICH, KARIN							-90,00		15,00	-15,00
KENT,CLARK								4,00	15,00	-11,00
KLEBER,UHU							-120,00		20,00	-20,00
MC MUSTER, ANN								9,50	20,00	-10,50
Mr. Muster, Guisepp								28,50	37,50	-9,00
MUSTERFRAU, AL								11,00	20,00	-9,00
Mustermann,Sabine	9,50	9,50	PEP			09:30 20:00		48,50	37,50	11,00
MUSTERPERSON,	9,50	9,50	PEP			09:30 20:00		46,00	37,50	8,50
PERSONTEST, RO	7,50	7,50	PEP			10:00 18:30		27,50	37,50	-10,00
Testperson, ULRIKE	10,00	10,00	PEP			08:30 19:30		45,00	37,50	7,50
WAYNE,JOHN								12,50	18,00	-5,50
WUNSCH, NICOLE	7,00	7,00	PEP			10:30 18:30		18,50	25,00	-6,50
WUSCHEL,KOPF								18,50	25,00	-6,50
ZITTER,PAPPEL	9,00	9,00	PEP			09:30 19:30		18,00	15,00	3,00
Summe:	62,25	62,25					-210,00	323,50	421,50	-98,00

Einsatzplanung - http://evo32/Fehlzeiten/FrmFehlzeiten.asp

Figure 6. Example of detailed staff schedule

References

[BBD03] Bard, J. F., Binici, C. und deSilva, A. H.: Staff scheduling at the United States postal service. *Computers and Operations Research*, 30 (5), 745 - 771, 2003

[EJK+04] Ernst, A., Jiang, H., Krishnamoorthy, M., Owens, B. und Sier, D.: An annotated bibliography of personnel scheduling and rostering. *Annals of Operations Research*, 127, 21-144, 2004.

[EJKS04] Ernst, A., Jiang, H., Krishnamoorthy, M. und Sier, D.: Staff scheduling and rostering: A review of applications, methods and models. *European Journal of Operational Research*, 153 3 – 27, 2004.

[Geb04] Gebhardt, B.: *Optimierungsmoeglichkeiten der Personaleinsatzplanung am Beispiel des deutschen Einzelhandels*. University of Cologne, Köln, 2004.

[GT04] Goodale, J. C. und Thompson, G. M.: A comparison of heuristics for assigning individual employees to labor tour schedules. *Annals of Operations Research*, 128, 47-63, 2004.

[Har07] Hare, D. R.: *Staff Scheduling with ILOG Solver*, Okanagan University College, 2007.

[ILOG07] ILOG: *Staff scheduling*. http://www.ilog.com/industries/transportation/staffscheduling.cfm, 2007.

[Sau04] Sauer, J.: *Intelligente Ablaufplanung in lokalen und verteilten Anwendungsszenarien*. Teubner, 2004.

[Sau06] Sauer, J.: *Modeling and solving multi-site scheduling problems*, In: W. v. Wezel, R. Jorna und A. Meystel (ed.):*Planning in Intelligent Systems: Aspects, Motivations and Methods*, 281-299. Wiley, 2006.

[Sto04] Stoerk, J.: Planning transport in the supply chain, *Intl. Conference on Artificial Intelligence and Applications, IASTED*, Inssbruck. ACTA Press, 2004.

[Tol07] Tolzmann, E.: OC:Planner, Automatische Dienstplanung mittels Constraintpropagierung. *KI*, 1/07, 53-54, 2007.