Jarosław Skaruz, Artur Niewiadomski
Wojciech Penczek

Towards Automated Abstract Planning Based on a Genetic Algorithm

Nr 1026

Warsaw, December 2012
Abstract

The paper presents a new approach based on nature inspired algorithms to an automated abstract planning problem, which is a part of the web service composition problem. An abstract plan is defined as an equivalence class of sequences of service types that satisfy a user query. Intuitively, two sequences are equivalent if they are composed of the same service types, but not necessarily occurring in the same order. The objective of our genetic algorithm (GA) is to return representatives of abstract plans without generating all the equivalent sequences. The paper presents experimental results, which show that GA finds solutions for very large sets of service types in a reasonable time.

Keywords: genetic algorithms, web service composition, abstract planning

Streszczenie

Zastosowanie algorytmu genetycznego do problemu planowania abstrakcyjnego

Raport przedstawia nowe podejście do problemu planowania abstrakcyjnego za pomocą algorytmów genetycznych (AG). Problem planowania abstrakcyjnego polega na takiej kompozycji usług sieciowych, która spełnia zapytanie użytkownika. W raporcie pokazano sposób zastosowania AG do rozwiązywania problemu planowania abstrakcyjnego oraz zaprezentowano wyniki eksperymentalne.

Słowa kluczowe: algorytmy genetyczne, kompozycja usług sieciowych, planowanie abstrakcyjne
1 Introduction

The number of web services available in the Internet has recently increased tremendously [25]. The users may want to achieve some goals taking advantage of these services, but they also demand more sophisticated functionality from computer systems. Frequently, a simple web service does not realize the user objective, so a composition of them need to be executed to this aim. The problem of finding such a composition is hard and well known as the Web Service Composition Problem (WSCP) [3, 1, 22]. There is a number of various approaches to solve WSCP [5, 4, 7, 8], some of them we discuss in the next section. In this paper, we follow the approach of our system PlanICS [14, 12, 13], which has been inspired by [1, 2]. The main assumption is that all the web services in the domain of interest as well as the objects which are processed by the services, can be strictly classified in a hierarchy of classes, organised in an ontology. Another key idea is to divide planning into several stages. The first phase of the planning process works with types (classes), while the second one - in the space of concrete services (instances of classes). The first stage produces an abstract plan, which becomes a concrete plan in the second phase. Such an approach enables to reduce dramatically the number of concrete services which are taken into account. This paper focuses on the abstract planning problem only.

The current approaches to the abstract planning (see Section 2) behave nicely for small and medium size ontologies. However, for ontologies containing a large number of service types, the computational time could be very long or even prohibitive. In this paper we propose a new approach based on an application of genetic algorithms. Despite of the fact that genetic algorithms have been widely used to solving the concrete planning problem its application to the abstract planning is much more sophisticated. The main challenge in this work is related to the fact that the search space could contain only a few or just one feasible solution, which need to be found by the algorithm. On the other hand, in the concrete planning problem, initial population of individuals in
genetic algorithms represent only feasible potential solutions, which is not the case here. The abstract planning approach based on a genetic algorithm (GA) is much more involved and this is our first contribution. Our second contribution consists in the new representation of the abstract plans (by multisets of service types), which allows for pruning the state space from all the sequences that correspond to the generated abstract plans so far.

An individual of GA represents a multiset of service types and all the operators of GA are performed on this multiset. This feature of GA constitutes a great improvement in comparison to the linear form of service types. The main advantage of this approach is that the algorithm does not need to care about the correct order of the service types represented. It means that in comparison to a linear representation of the individual, the offspring created through genetic operators do not have to contain service types in the correct order. Next, a linear form of an abstract plan is created using the heuristic procedure before the fitness function evaluation has been done. An abstract plan is defined by a multiset of service types such that its linear form satisfies a user query. If GA finds a new abstract plan, then it is stored. All the individuals in the subsequent iterations are then 'punished' by decreasing their fitness value if they are similar to the abstract plan found. The individual fitness value is lowered proportionally to the similarity to the abstract plans stored. To the best of our knowledge, the above approach is novel, and as our experiments show is also very promising.

The rest of the paper is organized as follows. Related work is discussed in the next section. Section 3 deals with the abstract planning problem. In Section 4 it is shown how GA is applied to finding abstract plans. Section 5 presents the ontology generator, used for generating the service types and the user queries for GA. Section 6 discusses experimental results obtained from our algorithm. The last section summarizes and discusses the results.
2 Related Work

Some approaches to the abstract planning are shortly discussed below. However, to the best of our knowledge neither of the existing algorithms uses genetic algorithms. Peer [21] illustrates a plan-space based algorithm which improves the plan search with a feedback gained from a plan execution for the automatic Web Service Composition. A new framework for incorporating QoS in a dynamic workflow system is presented in [9]. This algorithm is actually a depth-first traversal of all service types with an intermediate pruning. The selection of the best workflow is done by evaluating the QoS constraints of each candidate. In [17] a dynamic service composition framework with two layers is presented. Semantics of the components and the user query is modeled and then in the second layer an execution path is discovered based on the query and the semantics of the components. In [24], the authors present a logic based planner for DAML-S services which is the predecessor of OWL-S. Eduardo et al. define in [15, 16] a framework called Dynamic Composition of Service (DynamiCoS) that aims at supporting service composition on demand at a runtime.

The problem of concrete planning has been recently also extensively studied in the literature. Besides various kinds of metaheuristics [10], there is a large number of papers concerning an application of non-deterministic algorithms, namely evolutionary algorithms. In [23] a simple genetic algorithm was used to obtain a good quality concrete plan. The problem is also tackled with a multiobjective optimization genetic algorithm to find a set of optimal Pareto solutions from which a user can choose the most interesting tradeoff [11]. In [6] the authors applied genetic algorithms to concrete planning problem based on a delivered abstract plan. The number of genes is the same as the number of the abstract services in the abstract plan and each gene corresponds to an offer of a given abstract service from the abstract plan. In the experimental study the authors used 25 abstract services and up to 25 offers for each service. Their approach allows to find the optimum in 500
iterations of the genetic algorithm.

In [19] the authors used a genetic algorithm to one phase planning, which combines an abstract and a concrete planning. They studied a QoS-aware semantic web service composition and showed how to effectively compute optimal compositions of QoS-aware web services by considering their semantic links. In the fitness function they maximize semantic quality attributes, while minimizing the QoS attributes. The experiments were conducted using 500 offers for each of 500 abstract services.

Parejo et. al present in [20] an application of a combination of two algorithms, namely Tabu Search (TS) and a genetic algorithm (GA). In this approach the idea of incorporating TS as a local procedure of GA was to escape GA from local minimum. Their experimental results were compared to the results obtained a from standalone TS and GA. They show that a hybrid algorithm outperforms the other two approaches.

A combination of two different algorithms was also defined in [18]. The authors transformed the problem of a concrete planning into a selection of the optimal path in the weighted directed acyclic graph. Unfortunately, they used only 10 abstract services and 35 offers belonging to each of them. The proposed algorithm works better than a simple GA.

3 Abstract Planning Problem

This section introduces the Abstract Planning Phase (APP) as the first stage of WSCP in the Planics framework. APP makes intensive use of the service types and the object types defined in the ontology. A service type represents a set of web services with similar capabilities, while object types are used to represent data processed by the service types. The set of all object types is denoted by $T$. The attributes are components of the object types, while a single attribute consists of a name and a type. The ontology defines the inheritance relation, such that a subtype of some base object type retains all the attributes of a
Abstract Planning Problem

base type, and optionally introduces some new attributes. The objects are instances of the object types, distinguishable by unique identifiers. The set of all the objects is denoted by $\mathcal{O}$. The values of the attributes of an object determine its state. A set of the objects in a certain state is called a world. One of the crucial concepts of PlanICS is a world transformation, described in details in the next subsection.

The abstract values are another important idea in APP. Since for APP there is no need to know the exact states of the objects, it is enough to know only whether an attribute does have some value or it does not. Thus, we introduce special functions $isSet$ and $isNull$ defining abstract values of the attributes of the objects, to be used in specifications of the user queries and the service types.

3.1 User queries and service types

The main aim of PlanICS is to find a composition of web services, which allows to achieve a user goal. The user requirements are specified in a form of a user query. A user query specification, as well as a service type specification, consists of three sets of objects: $in$, $inout$, and $out$, and two Boolean formulas over the attributes of the objects from these sets, namely preCondition and postCondition ($pre$ and $post$, for short). More precisely, $pre$ is defined over the attributes of the objects from $in$ and $inout$, while $post$ can involve also the objects from $out$. Before APP the $pre$ and $post$ formulas are reduced to their DNF forms, and the values of the attributes are mapped to the abstract values only. That is, in a general case, a reduced formula is a disjunction of conjunctions of literals without negations, where each literal is of the form $isSet(o.a)$ or $isNull(o.a)$ with $o$ denoting an object and $a$ its attribute. Thus, during APP we deal with sets of objects, and such reduced formulas only. The interpretation of a single disjunct of a reduced formula and of the respective set of objects leads to obtaining an abstract world, i.e., a set of objects which attributes have abstract values: set or null. We simply say that an attribute is set or is null. Thus, an interpretation of a whole reduced formula constitutes a set of abstract worlds. As this
paper deals with APP only, in what follows we use the notions *worlds* and *values* instead of *abstract worlds* and *abstract values*, resp., when it is clear from the context.

Now, we are in a position to define the service types and the user queries. A service type \( s \) is a pair of world sets \((W^s_{\text{pre}}, W^s_{\text{post}})\), called the *input* and the *output worlds*, respectively. That is, a service type is an interpretation of the respective service type specification, such that the input worlds are defined by \( \text{in}, \text{inout}, \) and \( \text{pre} \), while the output worlds are determined by \( \text{in}, \text{inout}, \text{out}, \) and \( \text{post} \). Moreover, let \( \mathbb{S} \) denote a set of all the service types defined in the ontology. The definition of a user query is similar: a user query \( q \) is a pair of world sets \((W^q_{\text{init}}, W^q_{\text{exp}})\), called the *initial* and the *expected worlds*, respectively. That is, a user query is an interpretation of the respective user query specification, such that the initial worlds are defined by \( \text{in}, \text{inout}, \) and \( \text{pre} \), while the expected worlds are determined by \( \text{in}, \text{inout}, \text{out}, \) and \( \text{post} \).

Fig. 1 presents an example of a service type specification. *BookSelling* is a simple service type, which does not take any “read-only” object as an input (its \( \text{in} \) set is empty), modifies an object of type *Book*, and produces an object of type *Invoice* with all the attributes set. Note that in the absence of alternatives in \( \text{pre} \) and \( \text{post} \) formulas, *BookSeling* defines exactly one input and one output world.

### 3.2 World transformations

A fundamental concept of PlanICS is a world transformation by a service of a given type. However, before we get to the details, we need to introduce a notion of the object states and the worlds compatibility. Assume we are given two objects \( o_1 \) and \( o_2 \) of some worlds, thus we know the (abstract) valuation of their attributes. We say that the state of the object \( o_1 \) is *compatible* with the state of the object \( o_2 \), if \( o_1 \) contains all the attributes of \( o_2 \) (thus both objects are of the same type, or \( o_1 \) is a subtype of \( o_2 \) type), and their valuations are not contradictory. This means that if \( o_1 \) is compatible to \( o_2 \), then every attribute of \( o_2 \) with the set \((\text{null})\) value, corresponds the same attribute of \( o_1 \), which is also
BookSelling = {
    in = Ø,  inout = { (Book, b) },  out = { (Invoice, i) },
    pre = isSet(b.title) and isNull(b.owner),
    post = isSet(b.title) and isSet(b.owner) and
           isSet(i.price) and isSet(i.address) and isSet(i.name)
}

Figure 1: A simple service type specification and its interpretation as a pair of worlds

set (null, resp.). Moreover, we say that a world \( w_1 \) is compatible to a world \( w_2 \), if both of them contain the same number of objects, and there exists a one-to-one mapping of the objects from \( w_1 \) to \( w_2 \), such that every object from \( w_2 \) corresponds to a compatible object from \( w_1 \).

Finally, by a sub-world of a world \( w \) we mean a restriction of \( w \) to some subset of objects from \( w \), and by the size of \( w \) we mean the number of the objects in \( w \), denoted by \(|w|\). Thus, a service of type \( s \) transforms a world \( w \) into \( w' \), denoted by \( w \xrightarrow{s} w' \), if all of the following conditions hold:

- \( w \) contains a sub-world \( IN \) compatible with a sub-world of some input world \( w_I \) of \( s \), restricted to the objects from \( in \),

- none of the objects from \( IN \) does change its state during the transformation,

- \( w \) contains a sub-world \( IO \) compatible with a sub-world of \( w_I \), restricted to the objects from \( inout \),
• $w'$ contains a sub-world $IO$ compatible with a sub-world of some output world $w_O$ of $s$, restricted to the objects from $inout$,

• $w'$ contains a sub-world $OU$ compatible with a sub-world of $w_O$, restricted to the objects from $out$,

• the sets of objects from $IN$, $IO$, $OU$ are mutually disjoint, and $w$ does not contain any of the objects from $OU$,

• $|w'| = |w| + |OU|$.

Intuitively, a service type $s$ transforms $w$ into $w'$ by matching some sub-worlds of $w$ (denoted by $IN$ and $IO$) to one of its input worlds, “copying” all the objects from $w$ to $w'$, changing the states of the objects from $IO$ according to the $post$ formula, and creating new objects according to the set $out$ and setting their states to be consistent with $post$ (i.e., the sub-world $OU$). We refer to a world transformation by a service type $s$ also as an execution of a service type $s$ and by $\text{transform}(w, s)$ denote the world $w'$ if $w \xrightarrow{s} w'$.

**Transformation sequences.** Let $seq = (s_1, \ldots, s_k)$ be a sequence of service types of length $k$, and let $w_0$ and $w_k$ be worlds, for some $k \in \mathbb{N}$. We say that the sequence $seq$ transforms the world $w_0$ into $w_k$, denoted by $w_0 \xrightarrow{seq} w_k$, if there exist worlds $w_1, \ldots, w_{k-1}$, such that $w_{i-1} \xrightarrow{s_i} w_i$, for every $i = 1, \ldots, k$.

A sequence $seq$ of service types is called a transformation sequence, if there are worlds $w, w'$, such that $w \xrightarrow{seq} w'$. The world $w'$, i.e., the world obtained after the transformation of $w$ by the transformation sequence $seq$, is called a final world of $seq$. The set of all the transformation sequences is denoted by $\mathbb{S}^*$ while by $M_{seq}$ we denote the multiset of the service types $[s_1, \ldots, s_k]$ of the transformation sequence $seq$. A transformation sequence $seq$ that transforms a given world $w$ is called a transformation sequence for $w$, and the process of transformation of $w$ by $seq$ is called the execution of $seq$ in $w$. 
Quasi-transformation sequences. Let \( seq = (s_1, \ldots, s_k) \) be a sequence of service types of length \( k \), and \( w \) be a world. We say that \( seq \) is a quasi-transformation sequence for \( w \), if there exists \( 1 \leq j < k \) and \( (s_1, \ldots, s_j) \) is a transformation sequence for \( w \). Such a maximal \( j \) is called the q-length of the quasi-transformation sequence for \( w \), and the prefix of \( seq \) of length \( j \) is called the executable prefix of \( seq \). The final world of the quasi-transformation sequence for \( w \) of q-length is the world obtained by transformation of \( w \) by the executable prefix of \( seq \). Intuitively, if \( seq \) is not a transformation sequence for \( w \), but some non-empty prefix of \( seq \) is so, then \( seq \) is a quasi-transformation sequence for \( w \).

Equivalent transformation sequences. Let \( seq = (s_1, \ldots, s_k) \) and \( seq' = (s'_1, \ldots, s'_k) \) be two transformation sequences of length \( k \). Let us define a reflexive, transitive, and symmetric relation \( \equiv \subseteq S^* \times S^* \) such that \( (seq, seq') \in \equiv \), denoted by \( seq \equiv seq' \), if \( M_{seq} = M_{seq'} \).

User query solutions. Let \( seq = (s_1, \ldots, s_k) \) be a transformation sequence of length \( k \), and \( q = (W^q_{init}, W^q_{exp}) \) be a user query. We say that the transformation sequence \( seq \) is a solution of a user query \( q \), if there are worlds \( w, w' \), such that \( w \overset{seq}{\Rightarrow} w' \), \( w \in W^q_{init} \), and \( w' \in W^q_{exp} \). The set of all solutions of the user query \( q \) is denoted by \( QS_q \).

Intuitively, a solution of the user query \( q \) is every transformation sequence that transforms some initial world into some expected world, defined by the user query \( q \).

Abstract plans. Let \( seq \in QS_q \) be a solution of some user query \( q \). An abstract plan is a set of all solutions being equivalent to \( seq \), i.e., it is the equivalence class \([seq]_\equiv\). An abstract plan \([seq]_\equiv\) is represented by the multiset of the service types \( M_{seq} \) for \( q \).

Example 1 Assume that Selling (S), Transport (T), Assembly (A) are service types, while Boards, Nails and Doghouse are object types.
extending an object type Ware. A service type Selling is able to provide
any Ware, Transport can deliver any Ware to the requested destina-
tion, and capability of Assembly is to build a doghouse using nails and
boards. If the user wants to obtain a doghouse, there are several possi-
bilities to achieve this goal.

The shortest solution is the sequence (S,T), and this is the only
solution of the abstract plan represented by the multiset [S,T]. Another
possibility is (S,T,S,T,A), where the first pair (S,T) provides and trans-
ports boards and the second pair provides and delivers nails, which are
finally assembled by A providing a doghouse. This solution constitutes
another abstract plan represented by [A,S,S,T,T]. Note, that there ex-
ists another equivalent solution, that is the sequence (S,S,T,T,A).

The next possible plan is represented by [A,S,S,T,T,T], when the
requested doghouse is assembled elsewhere than at the client, and it has
to be finally transported.

4 Application of GA to abstract planning

The objective of GA is to find abstract plans for a user query q While
GA maintains a population of individuals representing a multiset M of
service types, it is essential to check whether M represents an abstract
plan. To this aim, for M a sequence of service types seqM is constructed
according to the procedure seqGen (see Sec. 4.2). If seqM is a solution
to the user query q, then M represents a new abstract plan. In the next
four subsections we describe in detail how GA works.

4.1 An abstract plan coding scheme

An individual is used for modelling an abstract plan we would like to
find. A gene of an individual models a service type. So, the number of
the genes of an individual is equal to the number of service types in an
abstract plan. Let n denote the number of service types defined in the
ontology, i.e., \( n = |S| \), and let Num = \{0, 1, \ldots, n-1\}. We define a one-
to-one function $stype : S \rightarrow Num$, which for every service type assigns a natural number between 0 and $n - 1$. Finally, in our implementation an individual is a multiset over $Num$.

All the individuals in the initial population of GA are generated randomly. This means that at the beginning of the algorithm the whole population contains multisets of service types, which do not necessarily represent abstract plans. One of the advantages of our approach is that while an individual is a multiset of service types, we do not need to care about the order of the service types within the individual. This non-standard form of a GA individual allows for performing genetic operations in such a way that we do not have to receive offspring containing service types in the correct order. However, before the fitness function evaluation, a sequence of service types should be generated from an actual multiset. Since we do not generate all the sequences, the state space searched is dramatically reduced. This feature of GA allows us to obtain user query solutions in search spaces of sizes exceeding even $2^{100}$ (see Sec. 6).

4.2 Generating a sequence from a multiset

Although, an individual is a multiset of service types, at some points of our algorithm (like, for example, computation of a fitness value) we need to consider a transformation sequence built over the elements of the multiset. Obviously, we search for user query solutions, and therefore the sequences able to transform an initial world are of our particular interest.

The procedure $seqGen$ allowing to obtain such sequences from a multiset is given in Alg. 1. In the successive iterations we build a resulting sequence by removing from the multiset a service type $s$, which is able to transform\(^1\) a current world $w$, starting from some initial world $w_0$, randomly selected at the start of GA from $W_{init}^q$, of the user query $q$.

\(^1\)If there are more than one such a service type, then one of them is chosen randomly.
Procedure seqGen($M, w_0$)

Input: multiset of service types: $M$, initial world: $w_0$

Result: (quasi) transformation sequence: $seq$, (q-)length of $seq$: $l$, final world of $seq$: $w$

begin

$w \leftarrow w_0$;

$seq \leftarrow \epsilon$; // empty sequence

$l \leftarrow 0$;

while $M$ contains $s$ that can be executed in $w$ do

$seq \leftarrow seq \cdot s$; // append $s$ to $seq$

$l \leftarrow l + 1$;

$M \leftarrow M \setminus \{s\}$; // remove $s$ from $M$

$w \leftarrow transform(w, s)$;

end

while $M$ is not empty do

$M \leftarrow M \setminus \{s\}$; // remove some $s$ from $M$

$seq \leftarrow seq \cdot s$; // append $s$ to $seq$

end

return $(seq, l, w)$

end

Algorithm 1: Proc. $seqGen$ generating a sequence from a multiset
Then, the current world becomes the one obtained from the transformation of $w$ by $s$, and $s$ is appended to the resulting sequence. If none of the service types remaining in the multiset can be executed in the current world, then they are copied in a random order at the end of the sequence. Besides the sequence $seq$, the procedure returns also a natural number $l$, and a world $w$, which are used later to compute the fitness value of the individual. The world $w$ is the final world of the sequence $seq$, while $l$ is the (q-)length of $seq$ if $seq$ is a (quasi-)transformation sequence.

There are several reasons why the procedure does not consider all possible sequences that could be constructed from a given multiset. Firstly, for a given multiset of cardinality $k$ the number of all possible sequences is equal to $k!$. Secondly, we prefer the sequences transforming an initial world of the user query. And finally, if the individual passes to the next generation, still it will be possible to construct another sequence from the same multiset.

### 4.3 Fitness function

To evaluate an individual, its fitness value should be calculated. The fitness function is defined in such a way that it leads to significant improvements of the initially random individuals aiming at obtaining user query solutions.

Before we get to the details of the fitness function, we need to define first the notion of a *good service type*. Assume, we are given a sequence of service types of length $k$ $seq = (s_1, \ldots, s_k)$, and a user query $q$. Let us consider service types $s_i$ and $s_j$, where $i, j \in \{1, \ldots, k\}$, $i \neq j$, and $s_i \neq s_j$. By $in_{s_i}$, $inout_{s_i}$, $out_{s_i}$, $inout_q$, and $out_q$ we denote the sets of objects used, modified, and produced by the services type $s_i$, and requested to be modified, and produced by the user query $q$, respectively. Moreover, let us define the function $T : 2^O \rightarrow 2^T$, which with a set of objects assigns a set of types of these objects. We say that $s_i$ is a *good service type* for the sequence $seq$ and the user query $q$, if $T(in_{s_i} \cup out_{s_i}) \cap T(inout_q \cup out_q) \neq \emptyset$, or there exists $s_j$ in $seq$, such that $s_j$ is a
good service type and $\mathcal{T}(inout_{s_i} \cup outs_i) \cap \mathcal{T}(ins_j \cup inout_{s_j}) \neq \emptyset$.

Intuitively, a service type $s_i$ is good, if it produces objects that can be a part of the expected world, or they can be an input for other good service types. The procedure GST computing a set of the good service types for a transformation sequence and a user query is given in Alg. 2.

Thus, an individual $M$ is transformed to a sequence of service types $seq_M$, using the procedure $seqGen$, described in the previous subsection. Next, the fitness function, taking a triple $(seq_M, l_M, w_M)$ returned by $sepGen$ and an expected world\(^2\) $w_q$ as arguments, is calculated according to Eq. 1:

$$
fitness_M = \frac{f_{w_M} \times \alpha + c_{w_M} \times \beta + l \times \gamma + g_{seq_M} \times \delta}{|w_q| \times \alpha + |w_q| \times \beta + k \times \gamma + k \times \delta} (1)
$$

where:

$f_{w_M} = |w_{sub}|$, where $w_{sub}$ is a maximal sub-world of $w_M$ compatible with a sub-world of $w_q$.

$c_{w_M} = min(cst(w_M), |w_q|)$, where $cst(w_M)$ is the number of the objects from $w_M$, which types are consistent with the types of the objects from $w_q$.

$g_{seq_M}$ is the number of the good service types occurring in $seq_M$.

$k$ is the length of $seq_M$, and

$\alpha, \beta, \gamma, \delta$ are parameters of the fitness function. In all the experiments presented in Sec. 6 we used the following values: $\alpha = 0.1$, $\beta = 0.7$, $\gamma = 0.1$, and $\delta = 0.2$.

After a user query solution is found by GA and stored in the memory, the requirement for GA is to assure that other solutions remaining in the search space will be found. On the other hand, each individual that

\(^2\)Selected randomly from $W^q_{exp}$ at the start of GA
Procedure GST(seq, q)
Input: sequence of service types: seq, user query: q
Result: set of good service types occurring in seq: GS
begin
\[ GS \leftarrow \emptyset; \]
\[ S \leftarrow \text{seq} ;// \text{set of all types occurring in seq} \]
while \( S \neq \emptyset \) do
\[ s \leftarrow x \in S ;// \text{an arbitrary element of } S \]
\[ S \leftarrow S \setminus \{s\} ;// \text{remove } s \text{ from } S \]
if \( T(\text{inout}_s \cup \text{outs}) \cap T(\text{inout}_q \cup \text{outs}) \neq \emptyset \) then
\[ GS \leftarrow GS \cup \{s\} ;// \text{add } s \text{ to } GS \]
end
end
\[ S'' \leftarrow GS; \]
repeat
\[ S' \leftarrow \emptyset; \]
\[ S \leftarrow \text{seq} \setminus GS; \]
while \( S \neq \emptyset \) do
\[ s \leftarrow x \in S ;// \text{an arbitrary element of } S \]
\[ S \leftarrow S \setminus \{s\} ;// \text{remove } s \text{ from } S \]
foreach \( g \in S'' \) do
\[ \text{if } T(\text{inout}_s \cup \text{outs}) \cap T(\text{inout}_g \cup \text{inout}_g) \neq \emptyset \text{ then} \]
\[ S' \leftarrow S' \cup \{s\} ;// \text{add } s \text{ to } S' \]
break;
end
end
\[ GS \leftarrow GS \cup S' ;// \text{add } S' \text{ to } GS \]
\[ S'' \leftarrow S'; \]
until \( S' = \emptyset; \]
return GS
end

Algorithm 2: Proc. GST computing a set of good service types occurring in a sequence
represents a solution equivalent to one of the already known should be eliminated. The latter is the task of the \textit{measure of similarity} between the currently rated individual and the plans found so far. Obviously, the measure of similarity grows with the number of the service types common for the assessed multiset and one of the plans in the memory.

Let $Sol$ denote a non-empty set of plans (in a form of multisets) found at some point of GA. Then, the measure of similarity of a multiset $M$ is computed as follows:

$$sim^S_{Sol} M = \max \left( \left\{ \frac{|M \cap S|}{|M|} \mid S \in Sol \right\} \right)$$

(2)

Note that the similarity measure of a multiset identical to some plan is equal to 1, while 0 is the similarity measure of a multiset built over completely different types than these in the plan.

Finally, when there are solutions found, the fitness value of the individual $M$ is calculated according to Eq. 3:

$$fitness^S_{Sol} M = fitness_M \ast (1.0 - sim^S_{Sol})$$

(3)

The more the individual $M$ is similar to some known plan, the more the value of $sim^S_{Sol}$ decreases the fitness value of $M$.

4.4 Mutation operator

One of our contributions in this paper is a mutation operator specialized for the discussed problem, which takes advantage of the \textit{good service type} concept. Therefore, a gene is mutated only if it does not represent a good service type, and if there exists a \textit{good} service type for the considered sequence, generated by the algorithm $seqGen$. To this aim, one has to compute a set of all service types good for this sequence (see Alg. 3). If this set is not empty, then a randomly selected element of the set replaces the mutated gene. Thus, the mutation operator is not deterministic, and it does not work in a greedy way.
Procedure `mutGST(seq, q)`

**Input:** sequence of service types: `seq`, user query: `q`  
**Result:** set of the good service types for `seq` and `q` to be used by the mutation operator: `GS`

```plaintext
begin
    GS ← GST(seq, q);
    S ← $\mathbb{S} \setminus GS$;
    foreach $s \in S$ do
        if $\mathcal{T}(\text{inout}_s \cup \text{out}_s) \cap \mathcal{T}(\text{inout}_q \cup \text{out}_q) \neq \emptyset$ then
            GS ← GS $\cup \{s\}$ ; // add $s$ to $GS$
            continue;
        end
    endforeach
    foreach $g \in GS$ do
        if $\mathcal{T}((\text{inout}_s \cup \text{out}_s) \cap \mathcal{T}(\text{in} \cup \text{inout}_g)) \neq \emptyset$ then
            GS ← GS $\cup \{s\}$ ; // add $s$ to $GS$
            break;
        end
    endforeach
    return $GS$
end
```

**Algorithm 3:** Proc. `mutGST` computes a set of the good service types for a sequence and a user query to be used by the mutation operator.
5 Generator of Ontologies

We have evaluated our algorithm using the ontologies, the user queries, and the abstract plans generated by our software - Ontology Generator (OG, for short). Each ontology contains an information about the service types and the object types. OG generates the ontologies in a random manner such that semantic rules are met. Moreover, OG provides us with a user query which corresponds to service types and object types contained in the ontology. Each query is also generated randomly in such a way that the number of the existing abstract plans equals to the value of a special parameter of OG. This guarantees that we know a priori whether GA finds all the solutions. Each solution of GA is an abstract plan, however, while writing about the results of GA we use the notion of solution. The remaining parameters of the generator are: the number of various object types, the minimal and maximal number of the object attributes, the number of service types, the minimal and maximal number of objects in the sets \textit{in}, \textit{inout}, and \textit{out}, the number of the objects required by a user in the expected world, and the number of the service types in an abstract plan.

Thanks to many different settings of OG, one can receive such data which are helpful for checking how well GA scales for finding solutions. The scalability can be examined by fixing different sizes of service types in the ontology and the number of service types in the abstract plans.

6 Experimental Results

The experimental study was partitioned into two stages. First, we conducted some number of experiments to estimate the best values of all GA parameters. In result, we applied the following parameter values in the main part of the experiments: the number of the individuals equals to 1000, the probability of mutation equals to 5\%, and the probability of the two-point crossover operator equals to 95\%. Moreover, the roulette selection operator was used in all the experiments, and every experiment
### Experimental Results

**Table 1: Experimental results**

<table>
<thead>
<tr>
<th>Exp.</th>
<th>k</th>
<th>sol.</th>
<th>service types</th>
<th>prob.</th>
<th>max sol.</th>
<th>avg sol.</th>
<th>time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>exp1</td>
<td>6</td>
<td>1</td>
<td>64</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>6</td>
</tr>
<tr>
<td>exp2</td>
<td>6</td>
<td>1</td>
<td>128</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>7.5</td>
</tr>
<tr>
<td>exp3</td>
<td>6</td>
<td>10</td>
<td>64</td>
<td>100%</td>
<td>6</td>
<td>4.1</td>
<td>10</td>
</tr>
<tr>
<td>exp4</td>
<td>6</td>
<td>10</td>
<td>128</td>
<td>100%</td>
<td>6</td>
<td>3.7</td>
<td>6</td>
</tr>
<tr>
<td>exp5</td>
<td>6</td>
<td>10</td>
<td>256</td>
<td>100%</td>
<td>4</td>
<td>2.6</td>
<td>11</td>
</tr>
<tr>
<td>exp6</td>
<td>6</td>
<td>1</td>
<td>64</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>11</td>
</tr>
<tr>
<td>exp7</td>
<td>9</td>
<td>1</td>
<td>64</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>exp8</td>
<td>9</td>
<td>1</td>
<td>128</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>15</td>
</tr>
<tr>
<td>exp9</td>
<td>9</td>
<td>1</td>
<td>256</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>exp10</td>
<td>9</td>
<td>10</td>
<td>64</td>
<td>80%</td>
<td>2</td>
<td>1.0</td>
<td>12</td>
</tr>
<tr>
<td>exp11</td>
<td>9</td>
<td>10</td>
<td>128</td>
<td>60%</td>
<td>2</td>
<td>0.8</td>
<td>15</td>
</tr>
<tr>
<td>exp12</td>
<td>9</td>
<td>10</td>
<td>256</td>
<td>50%</td>
<td>1</td>
<td>0.5</td>
<td>22</td>
</tr>
<tr>
<td>exp13</td>
<td>12</td>
<td>1</td>
<td>64</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>16</td>
</tr>
<tr>
<td>exp14</td>
<td>12</td>
<td>1</td>
<td>128</td>
<td>100%</td>
<td>1</td>
<td>1</td>
<td>22</td>
</tr>
<tr>
<td>exp15</td>
<td>12</td>
<td>1</td>
<td>256</td>
<td>90%</td>
<td>1</td>
<td>0.9</td>
<td>34</td>
</tr>
<tr>
<td>exp16</td>
<td>15</td>
<td>1</td>
<td>64</td>
<td>80%</td>
<td>1</td>
<td>0.8</td>
<td>28</td>
</tr>
<tr>
<td>exp17</td>
<td>15</td>
<td>1</td>
<td>128</td>
<td>20%</td>
<td>1</td>
<td>0.2</td>
<td>35</td>
</tr>
<tr>
<td>exp18</td>
<td>15</td>
<td>1</td>
<td>256</td>
<td>20%</td>
<td>1</td>
<td>0.2</td>
<td>49</td>
</tr>
</tbody>
</table>

has been repeated 10 times.

In each of all the 18 experiments (see Tab. 1) we have examined a probability of finding a solution, the number of abstract plans found, and the total runtime of GA. The experiments have been run on a standard PC computer with 2.8GHz CPU and 8GB RAM, using ontologies and queries generated by our ontology generator.

Tab. 1 presents an overview of all 18 experiments. The columns from one to four of the table contain the following data: (Exp.) - the labels of the experiments, (k) - the lengths of the abstract plans, (sol.) -
the numbers of the existing solutions, and (service types) - the ontology sizes, i.e., the number of the service types in each ontology used. The columns from five to eight of the table display the summary results of the experiments as follows: (prob.) - the probabilities of finding solutions, (max sol.) - the maximal numbers of the solutions found, (avg sol.) - the average numbers of the solutions found, and (time) - the GA runtimes.

The general observations following from the experiments can be summarised as follows. The probability of finding a solution, as well as the number of solutions found, are inversely proportional to a solution length and the ontology size. Surprisingly, it is harder to find a solution when there is more than one abstract plan.

![Figure 2: GA performance in exp1, exp2, and exp3 - each abstract plan of length 6 and one solution in the search space](image)

The experiments exp1, exp2, and exp3 (see Fig. 2) have been performed using the ontologies and the queries for which there is only one abstract plan of length 6. The solutions have been found quickly in all three cases. Notice that every run of GA yields a solution, regardless of the size of the ontology. However, this behaviour of GA is quite obvious
because of a large number of the individuals in the population. Decreasing the number of the individuals would shorten the running time of the algorithm, but GA would need a few more iterations in order to find a solution.

![Graph showing GA performance in exp4, exp5, and exp6](image)

Figure 3: GA performance in exp4, exp5, and exp6 - each abstract plan of length 6 and 10 solutions in the search space

In the experiments exp4, exp5, and exp6, ontologies and user queries have been generated for which there are 10 abstract plans. In Fig. 3 one can observe that in all three cases several solutions were found by GA in the first 20 iterations. For the ontologies with 64, 128, and 256 service types, we have obtained at most 6, 4, and 3 abstract plans, respectively, in a single run of GA. Each time, after a solution has been found, the fitness value of the best individual decreased rapidly, because of the similarity factor. There are some number of exceptions to this behaviour, when two solutions were found just one after the other. This may happen only occasionally, when in the same generation are individuals very similar to different solutions. After a plan has been found, the fitness of the best individual decreased, but another individual, obtained through
crossover and mutation, became a solution in the next iteration. Thus, we can observe that the similarity measure works nicely as demonstrated by the graph of the best individual fitness, which proves that our algorithm works as expected.

![Graph showing GA performance](image)

Figure 4: GA performance in exp7, exp8, and exp9 - each abstract plan of length 9 and one solution in the search space

The experiments exp7, exp8, and exp9 have been performed to examine how well GA works when each abstract plan is of length 9 and there is only one solution in the search space. While each abstract plan is composed of 9 service types selected from 64, 128, and 256 possible service types, respectively, the search space contains $64^9 = 2^{54}$, $128^9 = 2^{63}$, and $256^9 = 2^{72}$ sequences, accordingly. Despite the fact that the search space is quite huge, GA finds the solutions in the 9th and 11th iteration (see Fig. 4). In the next experiment, the number of solutions available in the search space has been increased to 10. In Fig. 5 one can observe that starting from a random population, GA finds solutions between the 15th and the 26th iteration. After a solution has been found the algorithm is looking for another one, which is reflected by the shape of the best
6 Experimental Results

Figure 5: GA performance in exp10, exp11, and exp12 - each abstract plan of length 9 and ten solutions in the search space.

individual fitness function. However, due to much larger search spaces, than in the previous experiments, and the limited number of iterations, GA is unable to find more solutions in a single run.

In the next experiments the search space contains only one solution. Fig. 6 presents the results of searching solutions of length 12. One can observe that the solutions were found before the 30th iteration. The more service types an ontology contains, the more iterations of GA is required to obtain the solution. Similarly to the previous experiments, the algorithm is looking for another solution after it has found the first one.

The experiments exp16, exp17, and exp18 have been performed in order to examine the behaviour of our algorithm while searching for solutions of length 15. In Fig. 7 one can observe that the solutions were found in the 21th and the 22nd iteration of GA. The shape of the fitness function of the best individual remains similar to these presented in the former figures. Despite the size of the search space in exp18 equal to
Figure 6: GA performance in exp13, exp14, and exp15 - each abstract plan of length 12 and one solution in the search space $2^{120}$, GA is still able to find the solution.

7 Conclusions

In the paper we presented a new approach to the abstract planning problem with use of a genetic algorithm. This has been a non-trivial task for GA as an optimal individual had to be found. The initial population of GA contains randomly generated individuals, which may represent unfeasible solutions. Our main idea consists in defining such an evolution of individuals maintained by GA that leads to obtaining solutions that represent abstract plans. A high efficiency of GA in finding solutions has been obtained by using individuals representing multisets of service types and by transforming them to sequences before computing values of the fitness function.

In the experimental study we have shown that our approach allows
Figure 7: GA performance in $exp_{16}$, $exp_{17}$, and $exp_{18}$ - each abstract plan of length 15 and one solution in the search space

for finding abstract plans of length up to 15. The probability of finding a solution, as well as the number of solutions found, are inversely proportional to a solution length and the ontology size. Surprisingly, it is harder to find a solution when there is more than one abstract plan. Since our application of GA to the problem of the abstract planning is not very time consuming, this solution seems to be very promising.

8 Acknowledgments

The research described in this paper has been supported by the National Science Centre under the grant No. 2011/01/B/ST6/01477.
References


REFERENCES


Pracę zgłosił Antoni Mazurkiewicz

Adresy autorów: Jarosław Skaruz, Artur Niewiadomski
Instytut Informatyki UPH
ul. 3 Maja 54
08-110 Siedlce, Polska

Wojciech Penczek
Instytut Podstaw Informatyki PAN
ul. Jana Kazimierza 5
01-248 Warszawa, Polska

E-mail: jaroslaw.skaruz@uph.edu.pl
        artur@ii.uph.edu.pl
        penczek@ipipan.waw.pl

Symbol klasyfikacji rzeczowej (MSC 2010): II.C.2.4

Na prawach rękopisu
Printed as manuscript

---

Nakład 90 egzemplarzy. Papier kserograficzny klasy III. Oddano do druku
w grudniu 2012 r. Wydawnictwo IPI PAN. ISSN 0138-0648.