Fundamenta Informaticae XX (2017) 1–21
DOI 10.3233/FI-2012-0000
IOS Press

TripICS - a Web Service Composition System for Planning Trips and Travels

Artur Niewiadomski, Piotr Switalski, Marcin Kowalczyk
Siedlce University, Faculty of Science, Institute of Computer Science,
3-Maja 54, 08-110 Siedlce, Poland
artur.niewiadomski@uph.edu.pl; piotr.switalski@uph.edu.pl; kontakt@marcinkowalczyk.pl

Wojciech Penczek*
Institute of Computer Science, Polish Academy of Sciences
Jana Kazimierza 5, 01-248 Warsaw, Poland
penczek@ipipan.waw.pl

Abstract. We present the web service composition system TripICS, which allows for an easy and user-friendly planning of visits to interesting cities and places around the world in combination with travels, arranged in the way satisfying the user's requirements. TripICS is a specialization of the concrete planning of PlanICS viewed as a constrained optimization problem to the ontology containing services provided by hotels, airlines, railways, museums etc. The system finds an optimal plan by applying a modification of the most efficient concrete planner of PlanICS based on a combination of an SMT-solver with the algorithm GEO. The modification has been designed in order to solve quickly multiple equality constraints. The efficiency of the new planning algorithm is proved by experimental results.

Keywords: Web Service Composition, Concrete Planning, PlanICS, TripICS, SMT, Hybrid Algorithm, GEO

Address for correspondence: Institute of Computer Science, Siedlce University of Natural Sciences and Humanities, 3-Maja 54, 08-110 Siedlce, Poland
*Also affiliated with: Institute of Computer Science, Siedlce University of Natural Sciences and Humanities

1. Introduction

Automatic composition of Web services is a very active area of research which has provided a lot of important results [1, 8, 13, 24] as well as many implemented approaches [3, 5, 17, 19, 21]. Unfortunately, the problem of finding a composition of Web services satisfying user requirements is hard [21], so its solution requires efficient heuristic algorithms in order to be of practical applications.

In this paper we follow the approach of our system PlanICS [10, 11, 21, 22] which has been inspired by [1]. The main goal of PlanICS is to find a plan, i.e., a composition of services that satisfies a user query, in the process divided into three main phases: abstract planning, offer collecting, and concrete planning. Abstract planning (AP) consists in finding service types that are expected to be useful in achieving the user goal. The result of this phase is called an abstract plan. Offer collecting (OC), which interacts with service types found in abstract planning, aims at collecting data necessary for the next stage of service composition. Concrete planning (CP) selects concrete services (provided by OC) of types specified by an abstract plan such that the user requirements are met and the quality conditions are maximized.

This paper is an improved and extended version of our CS&P'16 paper [20]. The main contribution consists in offering the system TripICS - a real-life application of our Web service composition system PlanICS to planning trips and travels around the world. While there are systems offering some support for planning excursions and travels [6, 7], our system uses advanced automated concrete planning methods [21, 23, 26]. TripICS is a specialization of the concrete planning viewed as a constrained optimization problem to the ontology containing services provided by hotels, airlines, railways, museums etc. The system finds an optimal plan (solution) satisfying the requirements of the user by applying the most efficient concrete planner of PlanICS based on a combination of an SMT-solver [9] with the nature inspired algorithm GEO [23, 26]. However, the algorithm has been significantly improved in order to solve more efficiently multiple equality constraints. These constraints result from a new kind of user queries supported by the current version of our system, i.e., those where the order of city visits is not specified. The above improvement required to modify the neighbourhood operator as well as the fitness function. The efficiency of the new planning algorithm is proved by results of our extensive experimental results shown in Section 5.2.

Contrary to PlanICS, the first phase of planning, called abstract planning, is realized by TripICS in a semi-automatic way by giving the user a possibility to choose the elements of an abstract plan using a Graphical User Interface (GUI). In what follows we give also a new distributed architecture of TripICS based on a Web application and REST services.

In the remainder of this paper we present: related systems (Section 1.1), the description of the system TripICS (Section 2), the theory behind it (Sections 3 and 4), and the implementation (Section 5) followed by some experimental results (Section 5.2) and conclusions (Section 6).

1.1. Related Systems

To the best of our knowledge there are no systems that allow for combining Web services related to travels, hotels, and entertainments such that the user can specify requirements corresponding to the way these services fit one to another. The best example of limitations of the most advanced existing systems is Opodo, which new functionality allows for finding a flight and a hotel (for whole or part of a trip), but no entertainments can be combined with these. Opodo does not allow to make plans including many flights and many hotel stays. The function 'Multi-stop' cannot be combined with finding hotels or entertainments. Moreover, 'Multi-stop' cannot be used for finding optimal trips between two cities via other cities without specifying the order of the cities to be visited together

with exact dates. TripICS allows for finding plans in which multiple travels are combined with hotels and an optimal order of visits to cities is found by the system. Other systems can be used for a more sophisticated specification of single services like hotels (Booking.com, Trivago) or flights (Bravofly, Flyhacks), separately flights or hotels (Skyscanner, Flysiesta, Mytrip) or have functionality similar to Opodo (e.g., Kayak). In fact, Kayak is one of a few systems which allows for finding a flight or a hotel, or a flight + hotel, or a car for renting, or an entertainment, but no requirements on combining these functions can be specified before selecting one of them.

Another interesting solution for a frequent traveller could be also the TripIt application. Although TripIt does not offer planning capabilities, but it allows for easy managing of reservations and bookings, and during the trip provides additional useful information.

The InspiRock application is also very helpful by planning travels. It allows to interactively search for transportation, accommodation and tourist attractions, however, contrary to TripICS, the offers are not downloaded automatically, but the user is redirected to several external web sites. Recently, there appeared also the eCOMPASS application [14] supporting city visits. It allows for setting some user preferences and provides detailed daily plans, however the functionality is limited to visiting one city, and there is no way to plan a trip to several distant places.

2. TripICS Description

Our system is to allow the user for an easy and user-friendly planning of visits to interesting cities and places around the world in combination with travels in and out, arranged in the way satisfying the user's requirements. The general assumption is that the user would like to receive an optimal plan of a travel starting and ending in given locations and offering a possibility of visiting some specified cities within some specified dates. A plan is optimal if its quality value is the highest according to the given criteria. Below, we make the above description much more precise by giving three lists of requirements: 1) the user has to set (obligatory requirements), 2) the user can optionally set (optional requirements), and 3) the predefined quality requirements.

2.1. Obligatory Requirements

1. Trip starts and ends in two given locations (cities),
2. Trip starts from a given date (or a period of time) and lasts for a given number of days (optionally can be shorter or longer by a specified number of days),
3. Trip involves visiting given cities, each city within a specified minimal/maximal number of days,
4. In each city to be visited, the attractions specified in the optional rules, are available within the period of stay.

2.2. Optional Requirements

1. In each city, attractions (museums, exhibitions, matches, concerts, restaurants etc.) to attend are specified,
2. Quality of hotels is specified by giving a minimal number of stars (0 - 5) and a minimal score (0 - 10),
3. Travels do not last longer than a given number of hours.

4. Cities are to be visited in the order specified by the user or their optimal order is to be found by the planner.

Clearly, a plan is expected to be optimal in the sense that the travels should conveniently fit to the stays and the prices should be as low as possible for a required quality of hotels. The aim of TripICS is to return such plans if they exist. Formally, these plans need to satisfy the user requirements as well as the quality requirements specified below.

2.3. Quality Requirements

1. A travel connection between two cities is always direct if it exists,
2. Costs, durations, and the numbers of breaks of the travels are minimized,
3. Costs of the visits are minimized while their standards and durations are maximized.

3. Theory behind TripICS

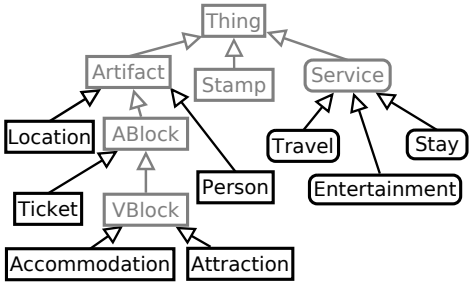
Typically, PlanICS realizes planning in three phases called: abstract planning, offer collecting, and concrete planing, after receiving a user query specifying the requirements. In TripICS we depart from using an abstract planner, which does free the user from formulating a user query in the specification language. Instead, the user is given a possibility to set the obligatory and optional requirements about expected plans using GUI, described briefly in Section 3.2. All the user's choices, as well as the quality requirements, are automatically encoded as a user query and passed to the offer collector and the concrete planners. The available options result from the underlying ontology.

3.1. Ontology

This section discusses the ontology exploited by TripICS. Fig. 1 shows a part of the ontology corresponding to a travel domain. The ontology defines three service types $Travel$, $Stay$, and $Entertainment$ aimed at providing instances of the $Ticket$, $Attraction$, and $Accommodation$ object types (and operating also on objects of type $Person$ and $Location$) which are the trip elements constituting (among others) the abstract plan.

A ticket represents a journey from one location to another, for a certain price. An accommodation corresponds to a stay in some location, for a certain price as well. An attraction represents an admission ticket for an event, a reservation, or a confirmation that the attraction is available at the specified time. All these objects contain attributes describing contexts and details of the particular trip elements. We introduce also two auxiliary object types: $ABlock$ and $VBlock$. This is to avoid duplication of common attributes using the inheritance mechanism.

For example, all the mentioned trip elements are described by the attributes $begin$, end, $price$, and $type$, and therefore they are inherited from the object type $ABlock$. The $type$ attribute defines the transportation type (bus, train, plane, ship, etc.), the accommodation type (hotel, guest house, hostel, apartment, etc.), or the attraction type (museum, exhibition, match, concert, restaurant etc.), when used in the $Ticket$, $Accommodation$, or $Attraction$ object, respectively. On the other hand, the number of breaks is specific to travels only, and thus the attribute $breaks$ is introduced in the object type $Ticket$. Each accommodation (and attraction) has been assigned a number stored in the attribute $reviews$ corresponding to the average score given by people who stayed there (or enjoyed the attraction) before. Similarly, since a fixed location is a common feature of accommodations and



Object type	Attributes' names and types
$Location$	$continent$, $country$, $city$: $String$
$Person$	$name$: $String$; loc: $Location$
$ABlock$	$begin$, end: $Date$; $price$: $Real$; $type$: $String$
$Ticket$	$from, to$: $Location$; $breaks$: int
$VBlock$	$reviews$: $Real$; loc: $Location$
$Accommodation$	$stars$: Int; $freeCancel$: $Bool$
$Attraction$	$reserved$: $Bool$

Figure 1. The TripICS ontology. The rectangles stand for object types while the rounded rectangles correspond to the service types. The types irrelevant for the working example are marked grey. The table describes the object types and their attributes.

attractions, the attribute loc of type $Location$ is introduced in the class $VBlock$ and inherited by $Attraction$ and $Accommodation$ object types. The attributes are summarized in Fig. 1. Note that their meanings follow intuitively from their names.

3.2. Specifying requirements

Using an intuitive web application GUI, the user inputs information about stages of the trip, which are added to the list at the right hand side and, simultaneously, the cities to be visited are shown on the map (see Fig. 2). The user provides the names of the start city and a city to be visited. Next, he specifies the transportation type (e.g., plane, train or any type), the starting date range, and other constraints like duration, price, and numbers of breaks ranges. If the accommodation option is chosen, then the form allowing to define additional parameters of the stay is shown (see Fig. 3, left). The last step of specifying every stage of the trip is (optionally) adding some attractions to enjoy in a selected city (see Fig. 3, right).

Finally, the user starts the planning process using the dedicated button, and optionally sets some planner options, such as a planning algorithm (SMT, GA, IPH, SCGEO, SCSA[1] [26, 23]), timeout, maximal number of offers etc., as well as some parameters specific to the particular planning method, e.g., a population size of GA and IPH. However, since SCGEO proved to be the most efficient planning method for the travel domain [23], the further parts of this paper are devoted mainly to the SCGEO algorithm and its improvements.

3.3. Collecting Offers and Planning

Basing on the city list and other data provided in the previous step, an abstract plan, i.e., a sequence of service types, is built. Next, this abstract plan is used by the *offer collector* (OC), i.e., the tool which in cooperation with the service registry queries real-world services. The service registry keeps an evidence of web services, registered accordingly to the service type system. Usually, each service

[1]SMT - the SMT-based planner, GA - the GA-based planner, IPH - the initial population hybrid planner (SMT + GA), SCGEO - the SMT combined with GEO planner, SCSA - the SMT combined with SA planner, where SMT (Satisfiability Modulo Theories), GA (Genetic Algorithm), SA (Simulated Annealing), GEO (Generalised Extremal Optimization)

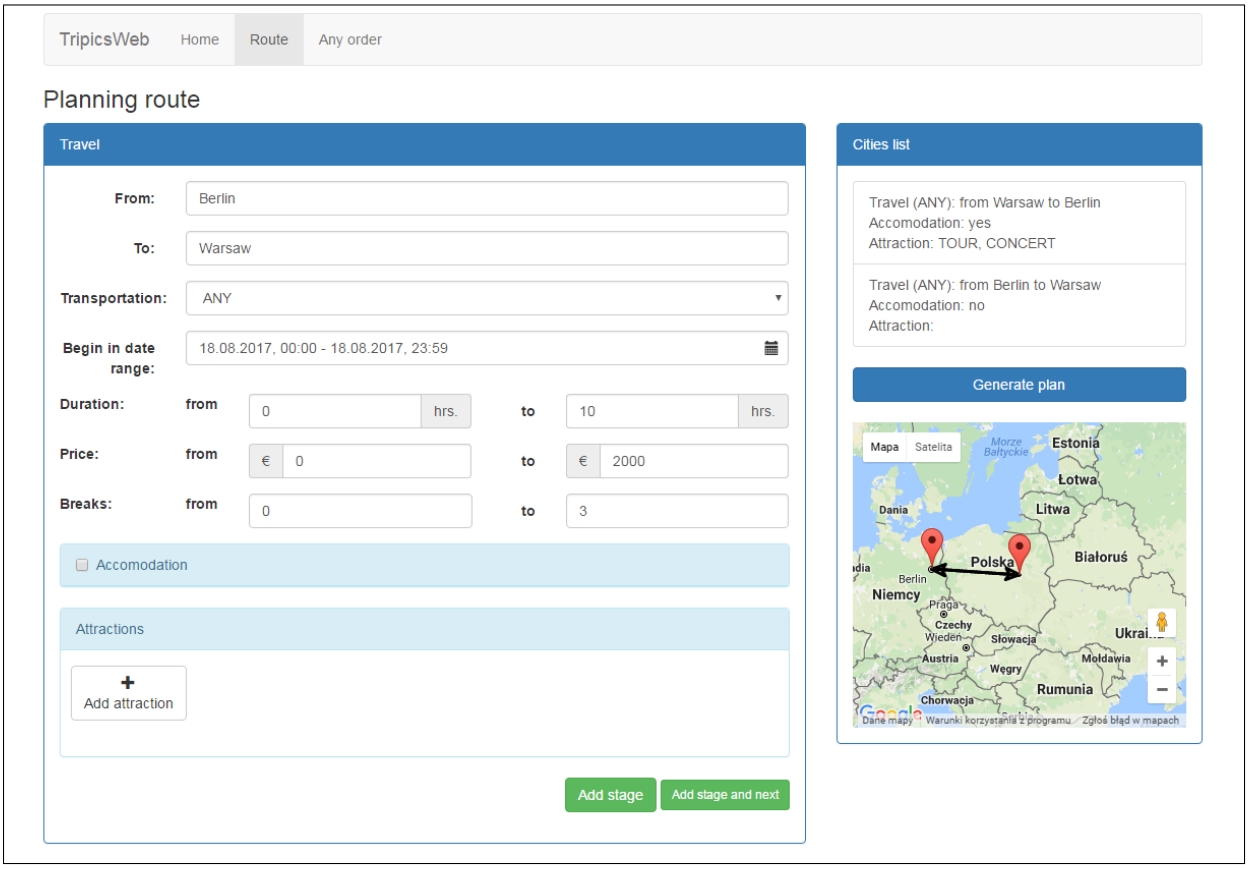


Figure 2. TripICS web application GUI

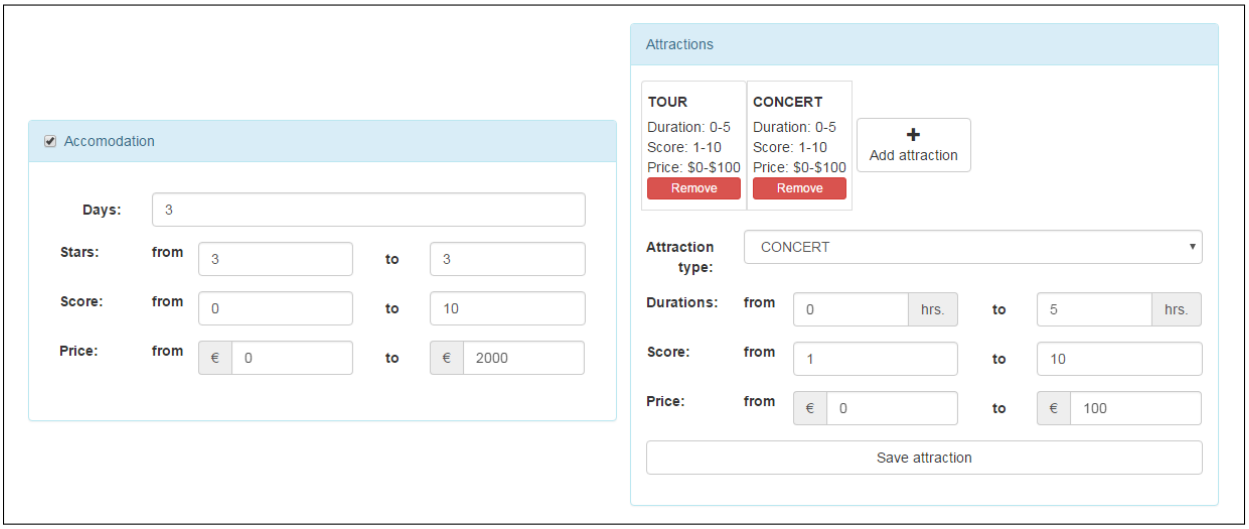


Figure 3. TripICS web application GUI: the accommodation and attractions form

type of the ontology represents a set of real-world services of similar functionality. For example, using the service type *Stay* one could register *Booking.com* as well as *Hilton* service.

OC queries web services of types present in the abstract plan and retrieves data called *offers*. An offer is a tuple of values representing a possible realization of one service type of the plan. Each tuple element (called an *offer attribute*) corresponds to an attribute of some object processed by the service type. We define the function $attr : Attributes \mapsto \mathbb{N}$, where $Attributes$ stand for the set of all attributes of the object types defined in the ontology. This function assigns a natural number (referred to as an *attribute index*) to every object attribute. The mapping of the attributes to numbers designates the positions of particular values in the tuple, called *offer values*. The offers collected for a single service type of the plan constitute so called *offer sets*.

The offers are searched by a *concrete planner* in order to find the best solution satisfying all constraints and maximizing the quality function. Thus, the concrete planning problem (CPP) can be formulated as a constrained optimization problem. Its solution consists in selecting one offer from each offer set such that all constraints are satisfied and the value of the quality function is maximized.

Definition 3.1. (CPP)
Let n be the length of the abstract plan and let $\mathbb{O} = (O^1, \ldots, O^n)$ be a vector of offer sets collected by OC such that k_i denotes the number of offers of the i-th offer set, for each $i = 1, \ldots, n$. Let P^i_j stand for the j-th offer of O^i. Let $\mathbb{P}$ be the set of all possible sequences $(P^1_{j_1}, \ldots, P^n_{j_n})$, such that $j_i \in \{1, \ldots, k_i\}$ and $i \in \{1, \ldots, n\}$, and let $\mathbb{C}(S) = \{C_j(S) \mid j = 1, \ldots, c \text{ for } c \in \mathbb{N}\}$, where $S \in \mathbb{P}$, be a set of the constraints to be satisfied. The Concrete Planning Problem is defined as follows:

$$max\{Q(S) \mid S \in \mathbb{P}\} \text{ subject to } \bigwedge_{j=1}^{c} C_j(S), \tag{1}$$

where $Q : \mathbb{P} \mapsto \mathbb{R}$ is an objective function defined as the sum of all quality constraints.

That is, a solution of CPP consists in selecting one offer from each offer set such that all constraints are satisfied and the value of the objective function is maximized. More details and the proof of NP-hardness of the concrete planning problem can be found in [21]. The constraints and the quality function result from the user requirements and preferences what is shown in the next subsection.

3.4. Constraints and Quality Function

The constraints and the quality function play a crucial role in the planning process. In this section, using a simple example, we show how the user requirements and preferences in combination with several general rules (described in Sec. 2) result in a set of constraints and a quality function.

Example 3.2. Assume that the user wants to make a trip on the 15th of August from Warsaw (W) to Berlin (B) and then back in a few days. In Berlin, he prefers to stay in a 3-star hotel for 3 days and during the visit he plans to take a city tour and attend a concert. The specified requirements result in an abstract plan consisting of the following 5 service types: ($Travel$, $Stay$, $Entertainment$, $Entertainment$, $Travel$). Then, OC searches for the matching offers, and retrieves the following example five offer sets ($O^1, \ldots, O^5$). In the tables below we show only the offer attributes relevant to the planner, i.e., those occurring in constraints or in the quality function.

$O^1(Travel)$

id	begin / end	price	fr. / to	br.
1	15.08, 10:40 / 15.08, 12:05	565	W / B	0
2	15.08, 06:20 / 15.08, 07:45	565	W / B	0
3	15.08, 07:20 / 15.08, 12:05	533	W / B	1
4	15.08, 14:05 / 15.08, 19:18	170	W / B	0
5	15.08, 18:05 / 15.08, 22:58	276	W / B	0

$O^2(Stay)$

id	begin / end	price	sc. / stars	loc
1	15.08, 14 / 18.08, 11	1044	9.1 / 3	B
2	15.08, 15 / 18.08, 11	1211	9.1 / 3	B
3	15.08, 15 / 18.08, 12	1729	9.0 / 3	B
4	15.08, 15 / 18.08, 11	1032	7.0 / 3	B
5	15.08, 15 / 18.08, 12	1259	8.9 / 3	B

$O^3(Entertainment)$

id	begin / end	price	score	loc
1	15.08, 18 / 15.08, 21	84	8.5	B
2	16.08, 12 / 16.08, 15	84	8.5	B
3	16.08, 15 / 16.08, 18	84	8.5	B
4	17.08, 12 / 17.08, 15	79	7.2	B
5	17.08, 15 / 17.08, 18	79	7.2	B

$O^4(Entertainment)$

id	begin / end	price	score	loc
1	16.08, 20 / 16.08, 23	280	7.2	B
2	16.08, 21 / 17.08, 1	130	8.1	B
3	17.08, 21 / 18.08, 1	110	3.0	B
4	17.08, 18 / 17.08, 22	580	9.3	B
5	16.08, 20 / 16.08, 23	164	7.9	B

$O^5(Travel)$

id	begin / end	price	fr. / to	br.
1	18.08, 11:50 / 18.08, 16:30	429	B / W	1
2	18.08, 15:10 / 18.08, 19:30	524	B / W	1
3	18.08, 08:50 / 18.08, 10:10	561	B / W	0
4	18.08, 09:37 / 18.08, 15:19	170	B / W	0
5	18.08, 14:37 / 18.08, 20:36	276	B / W	0

This example deals with a plan of length 5 where every service of the plan has 5 possible realizations. Thus, the search space is of size $5^5 = 3125$ as there is so many possible offer combinations. However, the number of plans (solutions) is much lower if we take constraints into account.

For example, assume that the user wants to synchronise travels and hotel in such a way that the time between arrival and hotel check-in is not longer than 3 hours. Similarly, the return travel should be not later than 3 hours after the hotel check-out time. After adding these two constraints the number of the possible solutions decreases to 2200. Another constraint could be to have at least a three-hour break between attractions. When this constraint is taken into account, there are only 1408 possible solutions. The underlying constraints are encoded by the following expressions: $(o_2.begin - o_1.end \leq 3)$, $(o_5.begin - o_2.end \leq 3)$, $(o_4.begin - o_3.end \geq 3)$, where o_i represents an offer from the i-th offer set.

As to the quality function, the user can choose between several schemes, but he can also enable/disable some of the function components. For example, if the user prefers only to minimize the total price, the quality function is expressed by $W_{price} * \sum_{i=1..5} o_i.price$, and the optimal solution is $(4, 4, 5, 3, 4)$ with the price 1561, where W_{price} is some negative constant (a weight). The numbers in the sequence correspond to the indices of the offers in the corresponding offer sets. That is, both the travels are for the price of 170 each, the stay is in the cheapest hotel, and the cheapest tour and concert are chosen. However, if the user also wants to maximize the reviews of the stay and attractions, the quality function is then as follows: $W_{price} * \sum_{i=1..5} o_i.price + W_{score} * \sum_{i=2..4} o_i.score$. Assuming $W_{price} = -1$ and $W_{score} = 10$, we obtain the optimal solution $(4, 1, 3, 2, 4)$ where the accommodation and attractions with a low price and a high score are chosen. Fig. 4 presents the resulting plan.

4. Extensions of Concrete Planners

Exploration of the travelling domain with our concrete planning methods, as well as a further development of TripICS, revealed several new challenges. One of them is solving multiple equality constraints by the planners. These constraints result from a new kind of user queries supported by the current version of our system, i.e., those where the order of city visits is not specified. As an

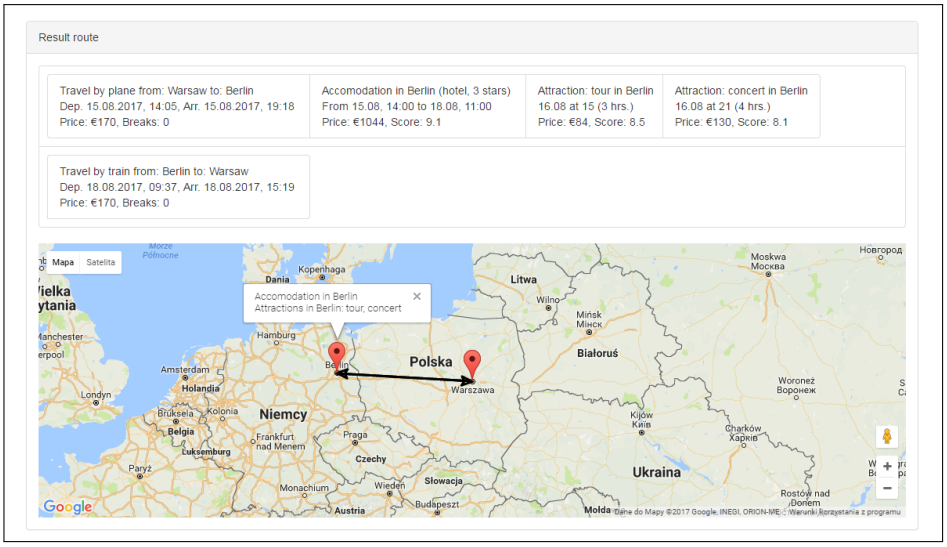


Figure 4. Visualisation of the last plan of Example 3.2

example, consider two consecutive offer sets O^i and O^{i+1} collected from services of types $Travel$ and $Stay$. They are coupled by the constraint $o_i.to = o_{i+1}.loc$ in order to ensure that the user arrives to the city where he stays. Moreover, these equality constraints often form sequences involving several decision variables. For example, if the next offer set represents the possible attractions to enjoy during the visit, we have another constraint $o_{i+1}.loc = o_{i+2}.loc$ enforcing that the accommodation and the event are located in the same city. This type of constraints significantly limits the capabilities of our SA- and GEO-based planners to find near-optimal solutions. In the following subsections we explain this problem in detail and show how to solve it on the example of the SCGEO algorithm.

4.1. Algorithm SCGEO

We start with recalling the algorithm combining SMT with GEO (i.e., SCGEO) aimed at solving the concrete planning problem, introduced in [23]. Algorithm 1 presents its pseudo-code. The coloured (and boxed) lines indicate the fragments subject to changes by our modifications, introduced in the next subsections.

SCGEO starts with generating an initial solution by the SMT-based procedure in such a way that all constraints are satisfied, but usually the quality of the first solution is far from the optimum. A solution is represented by a sequence of n decision variables of the values corresponding to offer indices chosen from the consecutive offer sets. Thus, n denotes the length of the plan.

The subsequent iterations of SCGEO consist of the following steps. First, the value of each decision variable is temporarily changed by applying the neighbourhood operator (which simply sets a random value to the given decision variable), and the obtained candidate solution is evaluated and stored in order to build the ranking list. Next, a ranking list is built, according to the differences between the fitness value of the current solution and the changed ones. The lower value, the higher position of the solution in the ranking. If a changed solution violates some constraints of $\mathbb{C}(S)$, then it is assigned a very high fitness value in order to push it at the bottom of the ranking.

Then, one of the candidate solutions of the ranking list is selected randomly. The probability of its

Algorithm 1: Pseudocode of the SCGEO algorithm for the concrete planning

SCGEO(n, K, τ)
Input: the number of decision variables: n, the number of iterations: K, a parameter: τ
Result: a solution with the highest fitness value

1: $I_{cur} \leftarrow SmtGen(n)$; // generate an initial solution by SMT
2: $Q_{best} \leftarrow$ Q(I_{cur}) ; // calculate and store the best fitness value found so far
3: $I_{best} \leftarrow I_{cur}$; // remember the best solution found so far
4: **for** $(i \leftarrow 1..K)$ **do**
5: **for** $(j \leftarrow 1..n)$ **do**
6: $I_{tmp,j} \leftarrow$ neighbourhood(I_{cur}, j) ; // j-th decision variable changed
7: **if** $(I_{tmp,j}$ satisfies all constraints$)$ **then**
8: $Q_j \leftarrow$ Q$(I_{tmp,j})$; // calculate the fitness value of $I_{tmp,j}$
9: **else**
10: $Q_j \leftarrow \infty$; // solutions violating constraints fall low in the ranking
11: $\Delta_j \leftarrow Q_j -$ Q(I_{cur}); // relative change of fitness resulting from change
12: $Rank \leftarrow sort((I_{tmp,1}, \Delta_1), \ldots, (I_{tmp,n}, \Delta_n)), ascending)$; // build the ranking by sorting the candidate solutions according to increasing Δ_j values
13: $changed \leftarrow false$;
14: **while** $(\neg changed)$ **do**
15: $j \leftarrow random(1..n)$; // randomly choose a candidate solution to be changed
16: $k \leftarrow Rank.find(j)$; // the position of the j-th solution in the ranking
17: $p \leftarrow k^{-\tau}$; // the acceptance probability of the j-th solution
18: $x \leftarrow random([0.0, 1.0])$; // a random value from the range [0.0, 1.0]
19: **if** $(p > x)$ **then**
20: $I_{cur} \leftarrow I_{tmp,j}$; // a new solution becomes the current one
21: $changed \leftarrow true$;
22: **if** $((\infty > Q_j > Q_{best}))$ **then**
23: $Q_{best} \leftarrow Q_j$; // update the best fitness value found so far
24: $I_{best} \leftarrow I_{cur}$; // update the best solution found so far
25: **return** I_{best}; // return the best solution

acceptance as a new solution is calculated as $p_j = k_j^{-\tau}$, where j represents the changed decision variable, k_j is the ranking position of the corresponding candidate solution, and τ is one of the algorithm parameters. If the j-th solution is not accepted, then another one of the ranking is chosen with a uniform probability. The algorithm repeats the attempts to accept a new solution until the current solution is replaced. All the steps described above are repeated until some stop criterion is satisfied. In our case we run the algorithm for a fixed number of steps, in order to fairly compare the efficiency of different versions of the algorithm, introduced in the next subsections.

The analysis of the SCGEO behaviour in the presence of equality constraint sequences revealed the following problem - the returned solutions are of relatively low quality. Using several small datasets, for which only a few plans exist, we have observed that SCGEO was unable to find any other solution than the initial one returned by the SMT-based procedure. The reason behind this was the fact that, in order to transform a current solution into a new valid plan, one has to change more

than one decision variable in a single step. Otherwise, after changing the value of only one variable by the neighborhood operator some equality constraints were violated and the obtained candidate solution was placed at a very low position in the ranking. Thus, in the next subsections, we suggest two modifications of SCGEO fixing the problem. The experimental evaluation of the improvements is the subject of Section 5.2.

4.2. Improvement of Neighbourhood Operator

We present the EQN algorithm to compute the new neighbourhood operator specialised to deal with a set of equality constraints. Let us start from giving formal definitions of constraints that EQN deals with.

Definition 4.1. (Simple equality constraint (SEC))
By a SEC we mean an expression of the form $o_i.a = o_j.b$, where $i, j \in \{1, \ldots, n\}$, $i \neq j$, and a, b are some offer attributes of the i-th and j-th offer set, respectively.

It is worth noting that in the above definition, if $i = j$, then both the compared values belong to the same offer. This kind of constraints can be easily checked and the offers violating them filtered out before passing data to the planner, similarly to comparisons with constants.

Let $osi : EQ \mapsto \mathbb{N} \times \mathbb{N}$ be a function that every SEC assigns the indices of the offer sets involved in the constraint, i.e., $osi(o_i.a = o_j.b) = (i, j)$, for $i, j \in \{1, \ldots, n\}$. Moreover, let $\overline{(i, j)}$ denote a transformation of the ordered pair into a set (i.e., $\overline{(i, j)} = \{i, j\}$).

Definition 4.2. (Set of related simple equality constraints (RSEC))
Let $EQ = \{c_1, \ldots, c_e\} \subseteq \mathbb{C}(S)$, where $e \leq |\mathbb{C}(S)|$, be a set of all SECs passed to a concrete planner. Let $G = (EQ, E)$ be an undirected graph, where every SEC is a node, and $E \subseteq EQ \times EQ$ is the set of graph edges, such that $E = \{(c, c') \mid (\overline{osi(c)} \cap \overline{osi(c')} \neq \emptyset)\}$. By a set of the related simple equality constraints (RSEC in short) we mean any subset of EQ that is the set of the nodes of some strongly connected component of the graph G.

Intuitively, the edges of G connect the SECs which share an offer set, and by RSEC we call such a subset of EQ that is either a singleton $\{c\}$ if c does not share any offer set with any other SEC, or such a maximal set of constraints of EQ that every SEC of RSEC shares an offer set with another constraint of the same RSEC, in such a way that there exists a path in the graph G between every pair of constraints of the RSEC.

Having SECs and RSECs formally defined, let us continue with the algorithm. Since EQN makes use of several auxiliary functions and precomputed data sets, we describe them briefly below.

First, we analyse the constraints in order to choose all SECs which constitute the EQ set passed to the algorithm. Note that in the TripICS' context, all constraints of EQ form a single RSEC, because all elements of the plan concerning a single city visit are bound by SECs using their loc attributes, and every consecutive ticket is bound by SECs, with the previous, and the next stay, using attributes $from$ and to, respectively. Thus, for simplicity, the given version of the algorithm considers only one RSEC consisting of all constraints of EQ, but it can be easily generalized to deal with a set of disjoint RSECs. Moreover, since the planners work on finite data sets, we represent the values of offer attributes occurring in EQ by integers.

We define also the auxiliary function $ati : EQ \mapsto \mathbb{N} \times \mathbb{N}$ that every constraint of EQ assigns a pair of numbers representing the attribute indices which occur in the constraint. That is, $ati(o_i.a = o_j.b) = (attr(a), attr(b))$, for $i, j \in \{1, \ldots, n\}$.

The next step consists in selecting all offer attributes occurring in the considered RSEC. They are represented by a set $A \subset \mathbb{N} \times 2^{\mathbb{N}}$, i.e., a set containing pairs where the first value is the index of an offer set, and the second element is a set of natural numbers representing attributes of the given offer set used to build the SECs.

Using the obtained set A, we transform the offers into a data structure representing the function $part : \mathbb{N} \times \mathbb{N} \times \mathbb{Z} \mapsto 2^{\mathbb{N}}$ which given the offer set index, the attribute index, and the attribute value assigns a set offer ids, for which the given attribute has the given value. This is explained in the following example.

Example 4.3. (Function $part$)
Consider the first offer set of Example 3.2. Assume, that $attr(to) = 5$, i.e., the attribute to of the object type $Ticket$ is represented by the number 5, the number 0 stands for the value $Warsaw$, and 1 corresponds to $Berlin$. Then, we have $part(1, 5, 1) = \{1, 2, 3, 4, 5\}$ and $part(1, 5, 0) = \emptyset$.

Another function used by the EQN algorithm is $pgv : \mathbb{N} \mapsto 2^{\mathbb{N}}$ that a decision variable index assigns a set of *possible good values*, i.e., a set of all offer ids restricted to only those, which allow to set the coupled offer attributes occurring in EQ constraints to the same value. The pgv function is also represented by a precomputed data structure and passed to the EQN algorithm. This notion is explained below.

Example 4.4. (Possible good values)
Consider the offers of Example 3.2. Assume that we add an offer of $id = 6$ to the first offer set which represents a ticket from $Warsaw$ to $London$. Consider the constraint $o_1.to = o_2.loc$. It is clear that no value of $o_2.loc$ is equal to $London$. Thus, we have $pgv(1) = \{1, 2, 3, 4, 5\}$, since the sixth offer (of the first offer set) has no counterpart (of the second set), satisfying the constraint.

Before we present the EQN algorithm in detail, we need to define another helper function which provides access to the offers data. Let $oValue : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \mapsto \mathbb{Z}$ be the function which for a given offer set index, offer id, and attribute index returns the corresponding offer value.

A pseudocode of EQN is presented as Algorithm 2. The procedure takes as input a current solution and the index of a decision variable from which we start the changes (the source variable). For simplicity, we assume that all constant and precomputed values described above are available globally, and we do not have to pass them explicitly as parameters.

Algorithm 2: Pseudocode of the EQN algorithm

EQN(I_{cur}, i)
Input: a current solution to be modified: I_{cur}, an index of the source decision variable: i
Result: a new candidate solution
if $(i \notin V)$ **then return** $neighbourhood(I_{cur}, i)$; `// the standard neighbourhood operator`
;
else
  $I_{cur}(i) \leftarrow random(pgv(i))$; `// set the i-th value to a randomly chosen of` $pgv(i)$
  if $(all\ constraints\ from\ EQ\ are\ satisfied)$ **then**
    return I_{cur}; `// if one change is enough to satisfy EQ ctrs.`
  else return propagate(I_{cur}, $\{i\}$); `// adjust other variables to the` i`-th one`
  ;

The algorithm starts with checking if the source variable is involved in some SEC. If this is not the case, the standard neighbourhood operator is used to modify the current solution, i.e., the value of the i-th decision variable is set to a random offer id. Otherwise, the value is randomly chosen of a (possibly) smaller set $pgv(i)$. Next, after the i-th value has been changed, the equality constraints are checked. If some of them is still unsatisfied, the algorithm calls the recursive procedure $propagate$ presented as Algorithm 3. Note that in order to change a variable at most once, we remember the indices of the modified variables as a set passed to the $propagate$ procedure.

The overall goal of the procedure is to find a candidate solution which satisfies all equality constraints. To this aim, the algorithm attempts to sequentially change the values of all decision variables involved in equality constraints but the fixed ones having values already set. After every change, the obtained solution is tested against the equality constraints and either all are satisfied and the algorithm ends, or the set of fixed variables is extended by the recently changed one and the next propagation step begins.

Algorithm 3: Pseudocode of the **propagate** procedure

propagate(I, FV)
Input: a solution to be modified: I, a set of indices of fixed decision variables: FV,
Result: a candidate solution
if $(|FV| = |V|)$ **then return** I ; `// all considered decision variables already changed`
;
$U \leftarrow getUnsatCtrs(EQ, I)$; `// the set of equality constraints violated by` I
$c \leftarrow random(U)$; `// randomly choose one of the violated constraints`
$(v_1, v_2) \leftarrow osi(c)$; `// indices of decision vars. involved in the constraint`
if $(v_1 \in FV \wedge v_2 \in FV)$ **then return** I ; `// dead-end propagation: both v. already changed`
;
$(a_1, a_2) \leftarrow ati(c)$; `// indices of attributes involved in the constraint`
if $(v_1 \in FV)$ **then**
  $(f, tc) \leftarrow (v_1, v_2)$; `//` v_1 `fixed,` v_2 `to be changed`
  $(a_f, a_{tc}) \leftarrow (a_1, a_2)$; `// value of` a_1 `will be propagated, or`
else
  $(f, tc) \leftarrow (v_2, v_1)$; `//` v_2 `fixed,` v_1 `to be changed`
  $(a_f, a_{tc}) \leftarrow (a_2, a_1)$; `// value of` a_2 `will be propagated`
$val \leftarrow oValue(f, I(f), a_f)$; `// the value to be propagated`
$I(tc) \leftarrow random(part(tc, a_{tc}, val))$; `// set the` tc`-th var a value satisfying` c
if $(all\ constraints\ from\ EQ\ are\ satisfied\ by\ I)$ **then return** I;
;
else return propagate(I, $FV \cup \{f\}$); `// further recursive propagation`
;

Going into details, the $propagate$ procedure takes a solution and a set of fixed variables' indices as input, but it makes use also of several precomputed sets and functions, like the previously defined functions osi, ati, and $oValue$, as well as V - the set of indices of all decision variables occurring in all SECs. The procedure starts with checking the recursion stopping condition, that is, if there are still decision variables to change. If so, then one of the violated equality constraints c is randomly selected, and the decision variables, denoted by v_1 and v_2, occurring in c are extracted. Next, we check if both variables v_1 and v_2 have already been modified by the algorithm. If so, then we deal with an unsuccessful propagation caused by some value randomly chosen during one of the previous steps. In this case we abort the procedure by returning the current solution.

Otherwise, if at least one of the two variables can be modified, we proceed with applying the actual change. To this aim, we extract the attribute indices of c, and determine which variable should be treated as the fixed one (the f-th variable and the a_f-th attribute), and which one is to be changed (the tc-th variable, the a_{tc}-th attribute). Finally, we read the value of the fixed attribute and set the value of the tc-th decision variable to some id randomly chosen of the set provided by the function $part$. This operation either ends the propagation if all EQ constraints are satisfied, or the recursive propagation continues.

4.3. Improvement of Fitness Function

The standard version of the SCGEO algorithm [23] makes use of the fitness function which punishes the solutions for violating constraints using some high constant value. We observed that this disrupts creating proper GEO rankings, because the punished solutions are indistinguishable w.r.t. the number of violated constraints. The improvement introduced to the fitness function consists in returning a value depending on the number of the violated constraints. This is calculated as follows: $Q'(I) = Q(I) - pun^{BC(I)}$, where $Q(I)$ is the quality of the candidate solution I, pun is the punishment factor[2], and $BC(I)$ is the number of the constraints violated by the candidate solution I. This modification makes the fitness values to gradually decrease when the number of violated constraints increases. The structures of the rankings computed according to the new method are more adequate to the nature of the GEO algorithm.

5. Implementation and Experiments

TripICS has been implemented as a distributed application according to the SOA (Service Oriented Architecture) [2] paradigm. The main modules of the system include the *web application* front-end, *Trip Agent*, *Offer Collector*, and the *Planner* services. The TripICS architecture and interactions between the components are presented in Fig. 5.

The web application module transforms the user's requirements ①, specified according to the description of Sec. 3.2, into a query encoded in the JSON [4, 12] format, and sends it to Trip Agent (TA) ②. Notice that due to providing a RESTful API [25] endpoint, TA is ready to consume JSON queries from any compatible client, like, e.g., a mobile application. Thus, the main role of TA is to provide a clearly defined interface enabling an access to the system modules. Moreover, it manages tasks for OC and the Planner services. Therefore, TA passes the query ③ to an OC instance which, via sub-collectors specialized for particular service types and dedicated adapters, queries in parallel the collaborating services ④ ⑤ ⑥. Using collected data and the query, OC prepares offers and constraints, and finishes its task by passing them back as a response ⑦. The next step of TA consists in calling the planer service ⑧ which converts the offers from JSON to the required numerical form and runs the requested planning method. The planner response ⑨ containing either a plan found or a message, if no plan exists, is passed back to the client application ⑩, and visualised by GUI.

The front-end application has been implemented using JavaScript, HTML, and CSS in combination with the AngularJS [15] and Bootstrap [18] frameworks. It exploits also Google Maps JavaScript API to handle the map component. The back-end modules have been implemented in Java and make use of the Spring Framework [16].

[2]The pun value should be relatively high in comparison to the maximal possible value of the quality function. For example, in the presented experiments, we set pun to 100000.

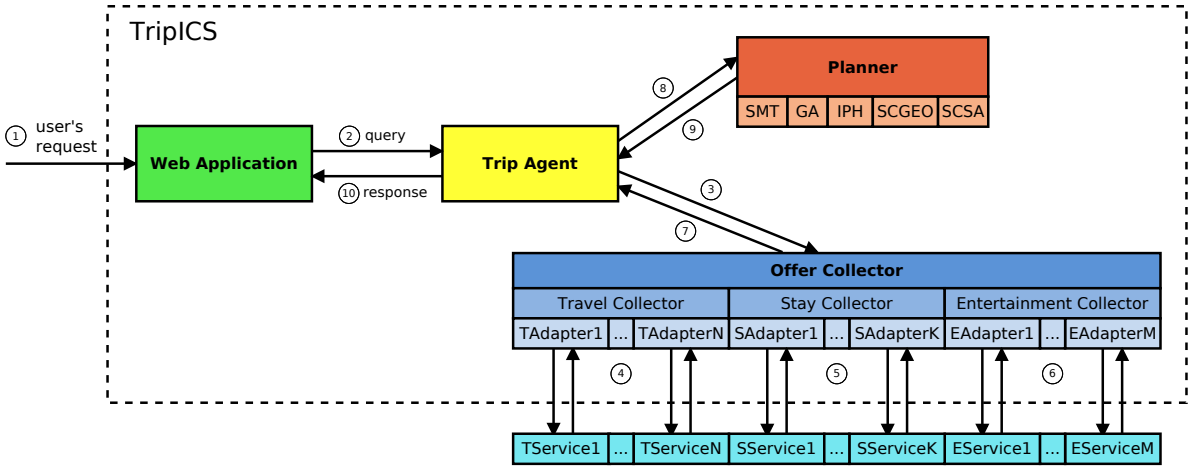


Figure 5. The TripICS architecture and data flow

The application is still being extended and improved. For example, for several reasons, our Offer Collector supports only a limited number of services. First of all, the vast majority of the existing travel-related services requires a fee for access to data, and the available demo endpoints often limit the query volume to only several requests. Secondly, since APIs from different providers are heterogeneous, one requires a significant effort to implement a dedicated adapter communicating with the particular service and mapping gathered data to the common representation.

Despite the difficulties in gathering massive data sets to the experiments, we have evaluated the efficiency of TripICS focusing on the planning modules. To this aim, we have used several benchmarks generated by our software Travel Benchmark Generator (TBG), described in the next subsection.

5.1. Travel Benchmark Generator

Our software TBG is able to generate massive offer sets and constraints emulating results gathered by the OC module in response to a user query which does not specify the order of city visits. TBG accepts a lot of arguments corresponding to data provided by users via GUI. The most important parameters, i.e., those used to scale the size of benchmarks, are as follows:

- c - the number of cities to visit during the journey,
- a - the number of attractions to enjoy in every city, and
- N - the number of offers in each offer set.

The other parameters, such as minimal and maximal prices, travel durations, etc. have been set to some constant values, the same for every benchmark. For example, the parameters $minDate$ and $maxDate$ determining the time span to plan the whole journey have been set to $2017/01/01$ and $2017/01/31$, respectively.

The resulting abstract plans are service type sequences of the form $((T, S, E^a)^c, T)$, where T stands for $Travel$, S denotes $Stay$ type, and E represents $Entertainment$. That is, the plans consist of sequences (T, S, E^a) repeated c times and the returning journey at the end. The notion E^a means

an $Entertainment$ service repeated a times. Next, for the consecutive service types of the plan an offer set of size N is generated.

Moreover, TBG generates also the set of constraint $\mathbb{C}$. Let the following sequence (for better clarity, divided into sub-sequences related to the particular locations) represent an example plan: $((T_1, S_1, E_{1,1}, \ldots, E_{1,a}), (T_2, S_2, E_{2,1}, \ldots, E_{2,a}), \ldots, (T_c, S_c, E_{c,1}, \ldots, E_{c,a}), T_{c+1})$. Then, the set of constraints is described by the formulae:

- $startPlace = T_1.from \wedge endPlace = T_{c+1}.to$ - the journey begins and ends in the locations chosen by the user,
- $T_1.begin \geq minDate \wedge T_{c+1}.end \leq maxDate$ - the journey dates fit to the given time span,
- $\bigwedge_{i=1}^{c}(T_i.to = S_i.loc \wedge \bigwedge_{j=1}^{a} S_i.loc = E_{i,j}.loc)$ - the user arrives to some city, and there he stays and enjoys the attractions,
- $\bigwedge_{i=1}^{c} T_i.end = S_i.begin$ - the user checks in the hotel on the arrival day.
- $\bigwedge_{i=1}^{c} T_{i+1}.begin = S_i.end$ - the user checks out from the hotel on the next departure day,
- $\bigwedge_{i=1}^{c} T_{i+1}.from = S_i.loc$ - the next travel starts from the currently visited place,
- $\bigwedge_{i=1}^{c-1} \bigwedge_{j=i+1}^{c} S_i.loc \neq S_j.loc$ - the user visits each city only once,
- $\bigwedge_{i=1}^{c}(S_i.begin \leq E_{i,1}.begin)$ - the first attraction in each city begins not earlier than the arrival,
- $\bigwedge_{i=1}^{c}(S_i.end \geq E_{i,a}.end)$ - the last attraction in each city ends not later that the end of the stay,
- $\bigwedge_{i=1}^{c} \bigwedge_{j=1}^{a-1}(E_{i,j}.end \leq E_{i,j+1}.begin)$ - each attraction begins not earlier than the previous one finishes.

Overall, we have generated two benchmark series. Both of them use the same offer sets, but differ in sets of constraints. The first series covers all but the last three constraint types described above, while the second series includes all of them. The benchmarks' parameters and features are summarized in Table 1. The meaning of the columns (from left to right) is the following: the benchmark identifier, the number of cities to visit, the number of attractions in every city, the length of the plan, the search space sizes for $512 = 2^9$ and $1024 = 2^{10}$ offers in each offer set, and the numbers of constraints for benchmarks from the first and the second series.

Table 1. The benchmarks' parameters and features

				Search space size		**Num. of constraints**	
Benchmark id	**c**	**a**	**n**	**N=512**	**N=1024**	**Series1**	**Series2**
T1	3	3	16	2^{144}	2^{160}	28	40
T2		4	19	2^{171}	2^{190}	31	46
T3		5	22	2^{198}	2^{220}	45	52
T4	4	3	21	2^{189}	2^{210}	38	54
T5		4	25	2^{225}	2^{250}	42	62
T6		5	29	2^{261}	2^{290}	46	70
T7	5	3	26	2^{234}	2^{260}	49	69
T8		4	31	2^{279}	2^{310}	54	79
T9		5	36	2^{324}	2^{360}	59	89

5.2. Experiments

In this section we show the experimental results examining efficiency of the introduced improvements. The computation time and quality of the solutions found have been used as the criteria to compare the modified SCGEO versions against the original one.

The experiments have been conducted using four variants of our algorithm:

- the standard SCGEO, without improvements (SCGEO),
- SCGEO with Neighbourhood Operator Improved (SCGEO+N),
- SCGEO with Fitness Function Improved (SCGEO+F), and
- SCGEO with both improvements (SCGEO+FN).

The experiments have been performed using the Z3 SMT-solver (version 4.4) running on a standard PC equipped with 4.0 GHz CPU (Intel i7-6700K processor). We have previously tuned the SCGEO algorithm for setting the optimal working conditions using our benchmarks. For this algorithm we use the following parameters: the number of iterations: $K = 2000$, τ=1.0. Each instance of the experiment has been repeated 50 times.

The tested benchmarks are characterised in Table 1, while the detailed results for Series 1 and 2 are given in Tables 2 and 3, respectively. The left part of each table shows the experimental results for benchmarks sets with 512 offers while the right one - for benchmarks sets with 1024 offers.

Table 2. Results of the first experiment series for the sets of 512 offers (left), and 1024 offers (right). The table entries (addressing computation time / quality) are organised as follows. The first row (in bold) contains the average values, the second row shows (in normal font) the result of the best quality among all experiment repetitions, and the third row presents the standard deviation (italic).

	512 offers				1024 offers			
	SCGEO	**SCGEO+N**	**SCGEO+F**	**SCGEO+FN**	**SCGEO**	**SCGEO+N**	**SCGEO+F**	**SCGEO+FN**
T1	**5.3 / 1539.7**	**5.8 / 1662.5**	**3.5 / 2326.0**	**4.0 / 2448.3**	**6.4 / 1561.9**	**6.9 / 1624.4**	**4.1 / 2356.9**	**4.8 / 2411.0**
	5.8 / 1543.7	6.6 / 1914.7	4.3 / 2326.7	4.9 / 2685.5	6.5 / 1592.7	7.2 / 1970.1	4.4 / 2375.7	5.1 / 2728.5
	0.5 / 10.5	*0.5 / 117.9*	*0.2 / 1.6*	*0.2 / 123.5*	*0.1 / 51.0*	*0.1 / 122.6*	*0.1 / 35.1*	*0.2 / 123.1*
T2	**7.5 / 1613.6**	**8.1 / 1689.2**	**4.7 / 2573.9**	**5.4 / 2617.3**	**8.1 / 1727.9**	**8.7 / 1884.9**	**5.6 / 2699.3**	**6.5 / 2871.3**
	7.7 / 1627.7	8.4 / 2042.2	4.9 / 2587.7	5.5 / 2912.9	8.7 / 1777.3	9.5 / 2042.3	5.8 / 2737.5	6.7 / 3053.0
	0.2 / 25.3	*0.2 / 104.2*	*0.1 / 25.2*	*0.1 / 74.4*	*0.3 / 53.7*	*0.3 / 105.0*	*0.1 / 42.8*	*0.1 / 122.6*
T3	**9.4 / 1205.3**	**10.5 / 1657.8**	**6.1 / 2362.5**	**6.9 / 2772.6**	**10.4 / 1544.9**	**11.2 / 1693.1**	**6.8 / 2699.1**	**7.9 / 2874.9**
	10.0 / 1214.4	10.9 / 2189.2	6.4 / 2369.4	7.4 / 3316.6	11.1 / 1548.4	12.1 / 2120.8	7.1 / 2703.3	8.2 / 3274.4
	0.4 / 21.7	*0.3 / 311.0*	*0.1 / 18.4*	*0.2 / 312.0*	*0.4 / 4.8*	*0.4 / 140.2*	*0.1 / 5.9*	*0.1 / 137.3*
T4	**9.3 / 1457.7**	**10.0 / 1778.9**	**5.9 / 2916.5**	**6.8 / 3204.5**	**10.3 / 1917.3**	**11.2 / 2103.2**	**6.9 / 3371.2**	**8.0 / 3553.6**
	9.7 / 1485.5	10.5 / 2154.2	6.0 / 2928.5	6.9 / 3600.9	10.5 / 1950.9	11.4 / 2343.8	7.1 / 3393.9	8.3 / 3720.0
	0.1 / 59.2	*0.2 / 167.5*	*0.1 / 35.2*	*0.1 / 168.9*	*0.1 / 38.4*	*0.1 / 90.3*	*0.1 / 32.1*	*0.1 / 84.3*
T5	**12.7 / 1800.0**	**13.7 / 1946.5**	**8.0 / 3562.7**	**9.3 / 3700.0**	**13.8 / 1514.7**	**15.1 / 1758.4**	**9.2 / 3276.7**	**10.5 / 3562.6**
	12.9 / 1802.8	14.0 / 2408.4	8.3 / 3565.8	10.3 / 4094.0	14.1 / 1549.1	15.8 / 2382.4	10.9 / 3312.5	11.0 / 4020.3
	0.1 / 5.9	*0.1 / 142.6*	*0.1 / 7.6*	*0.3 / 112.9*	*0.1 / 45.9*	*0.2 / 192.8*	*0.4 / 48.0*	*0.2 / 182.4*
T6	**16.7 / 2086.2**	**18.1 / 2102.6**	**10.3 / 4207.3**	**12.3 / 4229.6**	**18.0 / 1983.9**	**19.7 / 2121.9**	**11.8 / 4100.2**	**13.9 / 4204.5**
	17.0 / 2096.1	18.4 / 2172.4	10.7 / 4211.1	12.5 / 4287.4	18.3 / 1998.8	19.9 / 2372.7	12.1 / 4114.5	14.2 / 4449.7
	0.1 / 31.0	*0.1 / 39.4*	*0.2 / 16.1*	*0.1 / 39.8*	*0.2 / 14.5*	*0.2 / 108.3*	*0.1 / 10.9*	*0.2 / 97.7*
T7	**15.2 / 2112.9**	**16.3 / 2270.6**	**9.5 / 4520.8**	**11.0 / 4677.8**	**16.4 / 2108.2**	**17.9 / 2292.4**	**10.7 / 4515.5**	**12.4 / 4689.3**
	15.5 / 2126.8	16.6 / 2599.8	9.7 / 4526.8	11.3 / 4938.4	16.7 / 2133.5	18.2 / 2607.4	11.0 / 4533.3	13.1 / 4881.4
	0.1 / 31.3	*0.2 / 121.5*	*0.1 / 16.2*	*0.1 / 92.8*	*0.1 / 32.3*	*0.2 / 121.9*	*0.1 / 27.0*	*0.2 / 92.0*
T8	**20.9 / 2194.0**	**22.6 / 2417.2**	**12.9 / 5109.7**	**15.2 / 5250.1**	**22.4 / 2079.3**	**24.4 / 2239.4**	**14.4 / 4997.0**	**16.6 / 5152.3**
	21.2 / 2199.5	23.6 / 2847.5	13.3 / 5114.5	15.5 / 5760.8	23.7 / 2089.8	25.0 / 2485.7	14.8 / 5004.5	17.2 / 5496.2
	0.1 / 16.3	*0.2 / 177.2*	*0.1 / 14.2*	*0.1 / 161.7*	*0.3 / 10.0*	*0.2 / 103.7*	*0.2 / 8.7*	*0.3 / 95.3*
T9	**27.8 / 2282.1**	**30.2 / 2656.2**	**17.1 / 5760.0**	**19.4 / 6084.1**	**29.5 / 2201.7**	**30.3 / 2431.9**	**18.5 / 5672.1**	**20.5 / 5938.6**
	28.2 / 2286.6	31.0 / 3067.3	17.4 / 5766.6	20.6 / 6548.4	30.0 / 2232.0	32.7 / 2714.3	19.2 / 5711.0	20.8 / 6225.4
	0.2 / 13.3	*0.3 / 193.2*	*0.2 / 15.2*	*0.7 / 191.6*	*0.2 / 25.3*	*1.3 / 116.7*	*0.3 / 32.9*	*0.1 / 135.1*

Table 3. Results of the second experiment series. The meaning of the columns is the same as in Table 2.

	512 offers				1024 offers			
	SCGEO	**SCGEO+N**	**SCGEO+F**	**SCGEO+FN**	**SCGEO**	**SCGEO+N**	**SCGEO+F**	**SCGEO+FN**
T1	**7.4 / 1551.1**	**7.9 / 1633.3**	**4.3 / 3149.7**	**4.8 / 3221.2**	**8.3 / 1296.9**	**8.9 / 1316.7**	**5.2 / 2898.2**	**5.8 / 2913.9**
	8.2 / 1555.0	9.0 / 1706.6	5.0 / 3154.0	5.6 / 3319.2	8.5 / 1311.5	9.1 / 1601.5	5.4 / 2910.5	6.1 / 3224.9
	0.2 / 3.9	*0.2 / 51.8*	*0.2 / 5.9*	*0.3 / 56.5*	*0.1 / 15.9*	*0.1 / 61.8*	*0.1 / 13.0*	*0.2 / 68.5*
T2	**9.9 / 1550.9**	**10.4 / 1603.8**	**6.3 / 3662.3**	**6.9 / 3727.1**	**11.0 / 1390.7**	**12.0 / 1558.9**	**7.2 / 3515.2**	**8.1 / 3686.2**
	10.5 / 1557.8	11.1 / 1669.0	6.4 / 3672.8	7.1 / 3958.9	11.6 / 1415.6	12.4 / 1673.4	7.4 / 3531.8	8.3 / 3791.0
	0.4 / 6.7	*0.4 / 46.8*	*0.1 / 7.7*	*0.1 / 53.7*	*0.5 / 26.5*	*0.4 / 82.1*	*0.1 / 17.3*	*0.1 / 73.7*
T3	**13.0 / 1667.0**	**13.6 / 2076.3**	**8.1 / 4376.6**	**8.8 / 4788.8**	**13.8 / 1553.6**	**14.9 / 1704.9**	**9.0 / 4261.2**	**10.1 / 4420.4**
	13.9 / 1687.3	14.8 / 2166.5	8.6 / 4390.3	9.0 / 4871.9	14.4 / 1574.9	15.1 / 2033.3	9.2 / 4284.8	10.4 / 4735.3
	0.5 / 36.4	*0.4 / 85.6*	*0.2 / 27.1*	*0.1 / 96.9*	*0.2 / 22.4*	*0.1 / 105.2*	*0.1 / 20.2*	*0.1 / 96.7*
T4	**12.4 / 1676.1**	**13.0 / 1784.3**	**7.7 / 4591.3**	**8.4 / 4694.9**	**13.4 / 1699.3**	**14.3 / 2021.2**	**8.7 / 4624.4**	**9.8 / 4934.4**
	12.6 / 1680.2	13.3 / 1999.7	7.9 / 4595.2	8.7 / 4913.6	13.7 / 1737.1	14.6 / 2273.4	9.0 / 4652.8	11.8 / 5154.8
	0.1 / 2.7	*0.1 / 105.7*	*0.1 / 3.7*	*0.1 / 107.2*	*0.1 / 47.6*	*0.1 / 123.9*	*0.1 / 37.8*	*0.5 / 118.0*
T5	**17.4 / 1714.3**	**18.4 / 1755.9**	**10.7 / 5560.0**	**11.8 / 5597.4**	**18.8 / 1739.1**	**20.1 / 1943.9**	**12.2 / 5573.9**	**13.6 / 5834.6**
	17.7 / 1739.3	19.0 / 1783.4	12.0 / 5581.5	12.2 / 5628.0	19.2 / 1771.0	20.4 / 2372.0	12.5 / 5613.7	13.9 / 6200.6
	0.1 / 17.2	*0.2 / 22.5*	*0.4 / 10.1*	*0.2 / 27.7*	*0.2 / 30.5*	*0.2 / 186.6*	*0.2 / 37.6*	*0.2 / 202.8*
T6	**23.6 / 1960.0**	**25.0 / 2054.4**	**14.4 / 6859.8**	**15.9 / 6948.0**	**25.0 / 2007.8**	**26.8 / 2021.8**	**15.9 / 6896.4**	**17.7 / 6928.3**
	24.2 / 1969.2	25.4 / 2272.7	14.6 / 6867.2	16.3 / 7113.9	25.2 / 2048.1	27.9 / 2216.7	16.2 / 6942.0	18.2 / 7130.2
	0.2 / 16.5	*0.2 / 64.7*	*0.1 / 15.9*	*0.2 / 68.1*	*0.2 / 38.0*	*0.2 / 63.3*	*0.2 / 39.5*	*0.2 / 66.8*
T7	**20.2 / 1969.2**	**21.3 / 2184.6**	**12.3 / 6730.3**	**13.6 / 6895.3**	**21.5 / 2204.8**	**22.9 / 2342.4**	**13.8 / 6965.8**	**15.2 / 7129.7**
	21.5 / 1978.8	21.6 / 2365.1	12.5 / 6738.5	13.9 / 7127.4	21.9 / 2223.0	23.2 / 2472.3	14.0 / 6983.0	15.7 / 7231.5
	0.3 / 10.4	*0.1 / 124.3*	*0.1 / 7.6*	*0.1 / 99.2*	*0.2 / 22.0*	*0.2 / 93.0*	*0.1 / 25.4*	*0.2 / 83.7*
T8	**28.7 / 2372.1**	**28.7 / 2392.7**	**17.2 / 8616.0**	**18.7 / 8626.1**	**29.0 / 2211.5**	**30.1 / 2525.3**	**18.5 / 8466.6**	**19.7 / 8745.8**
	29.1 / 2385.5	29.9 / 2476.2	17.7 / 8625.5	19.8 / 8728.7	30.8 / 2286.6	30.5 / 2794.6	19.6 / 8525.4	20.2 / 8893.1
	0.2 / 33.1	*0.3 / 27.6*	*0.2 / 21.9*	*0.6 / 32.5*	*1.3 / 73.0*	*0.2 / 98.5*	*0.6 / 55.7*	*0.2 / 100.4*
T9	**36.7 / 2314.0**	**38.2 / 2395.4**	**22.0 / 10233.0**	**23.1 / 10311.0**	**38.0 / 2328.5**	**45.0 / 2427.6**	**22.6 / 10249.7**	**26.7 / 10324.0**
	38.1 / 2326.6	40.3 / 2522.5	22.7 / 10246.6	24.4 / 10462.6	38.5 / 2373.9	49.5 / 2688.3	23.0 / 10293.9	34.6 / 10568.0
	0.3 / 16.8	*0.8 / 59.8*	*0.2 / 16.9*	*0.5 / 75.4*	*0.5 / 53.3*	*2.7 / 98.0*	*0.2 / 54.2*	*3.9 / 77.3*

In general, the experimental results prove the efficiency of the introduced SCGEO modifications. The average quality of solutions increases after applying the neighbourhood operator by about 8% in the first experiment series, and by about 5% in the second series (see Fig. 6, right). However, comparing *the best* solutions found, we observed that enabling the specialized neighbourhood operator results in the quality improvement of 22% and 11% for the first and the second experiment series, respectively. It is worth noting, that in some cases the EQN algorithm is able to find a solution of the quality higher even by 80%, comparing to the standard SCGEO.

However, the fitness function improvement brings even more radical gain in the solutions quality. It enables to find the solutions of average qualities higher by about 100% in the first series, and by about 200% in the second experiment series. These factors stay similar if we consider the best quality values, as well. It is worth mentioning that for the longest plans of the second experiment series the solution quality improvement reaches as much as 340%.

To summarize the discussion on quality, one can conclude that adding the inequality constraints in the second experiment series degrades the EQN algorithm performance, because even if the equality constraints are satisfied, other conditions may be still violated. On the other hand, the improved fitness function excels at both kinds of constraints. This proves that SCGEO works much better when candidate solutions violating some constraints are permitted to take also higher positions in the ranking lists.

Concerning the computation time, one can observe that the additional computations performed by the modified neighbourhood operator generate from 5% to 10% overhead. In average, the SCGEO+N algorithm consumes 9.25% and 6.5% more time than the standard one, during the first and the second experiment series, respectively (see Fig. 6, left). On the other hand, application of the improved fitness function reduces the run time of SCGEO+F and SCGEO+FN by about 30% comparing to

the time consumed by SCGEO and SCGEO+N. This unexpected benefit also results from the better structure of the ranking list, because a new candidate solution is chosen much easier, with much less of the random generator calls (see lines 13-21 of the Algorithm 1).

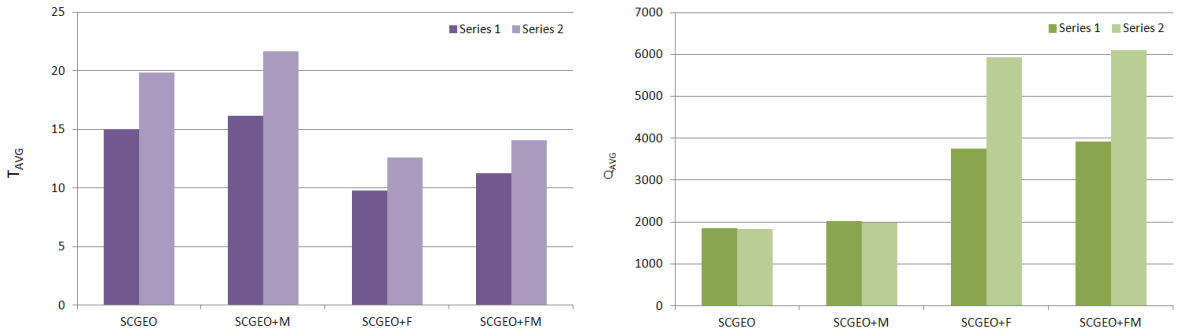


Figure 6. Comparison of the average computational time (left) and quality values (right) obtained using the considered SCGEO versions.

In Fig. 7 (left) we show a comparison of the standard deviation for the average quality values obtained with all considered SCGEO versions. It is easy to observe, that enabling the modified neighbourhood operator results in a higher dispersion of the quality values. We believe that the EQN algorithm has still a potential for further improvements.

The general summary of the results is presented in Fig. 7 (right). This is a chart showing the average quality of the solutions divided by the average computation times of all the experiments. The conclusion of this juxtaposition is that the most efficient version of the algorithm is SCGEO+F, because it yields solutions of very high quality, but consumes the least time amongst the compared methods.

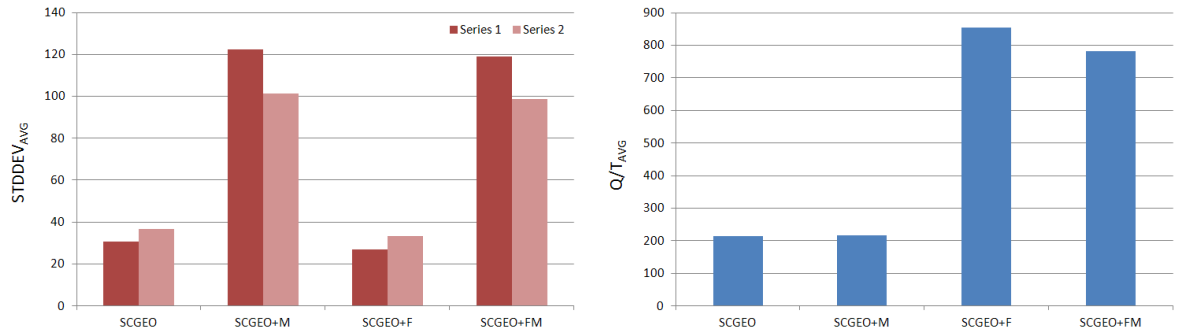


Figure 7. Standard deviation for average quality values (left) and the overall efficiency comparison (right) of different SCGEO versions.

6. Conclusions

We have presented a new version of our system TripICS, which can be used for planning trips and travels around the world. Its efficiency is obtained due to several modifications of the most efficient

concrete planner of PlanICS based on a combination of an SMT-solver with the algorithm GEO. The modifications have been designed in order to improve algorithms solving multiple equality constraints. We plan to introduce similar modifications also to our other concrete planning methods.

Our main motivation for developing this system was twofold. Firstly, we wanted to show that web service composition can be successfully used in practice for real-life applications, and secondly our aim was to offer a new useful tool, which could be publicly used. Therefore, we keep working on introducing TripICS to the Internet market.

References

[1] S. Ambroszkiewicz. Entish: A language for describing data processing in open distributed systems. *Fundam. Inform.*, 60(1-4):41–66, 2004.

[2] M. Bell. *Introduction to Service-Oriented Modeling (SOA): Service Analysis, Design, and Architecture*. John Wiley & Sons, 2008.

[3] D. Berardi, F. Cheikh, G. De Giacomo, and F. Patrizi. Automatic service composition via simulation. *Int. J. Found. Comput. Sci.*, 19(2):429–451, 2008.

[4] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159, 2014.

[5] D. Calvanese, G. De Giacomo, M. Lenzerini, M. Mecella, and F Patrizi. Automatic service composition and synthesis: the roman model. *IEEE Data Eng. Bull.*, 31(3):18–22, 2008.

[6] D. Damljanovic and V. Devedzic. Applying semantic web to e-tourism. In *In Ma, Z.,Wang, H. (Eds.), The Semantic Web for Knowledge and Data Management: Technologies and Practices*, pages 243–265. IGI Global, New York, 2008.

[7] D. Damljanovic and V. Devedzic. Applying semantic web to e-tourism. In *In Mehdi Khosrow-Pour (Ed.), Encyclopedia of Information Science and Technology*, pages 3426–3432. IGI Global, Hershey, 2009.

[8] G. De Giacomo, M. Mecella, and F. Patrizi. Automated service composition based on behaviors: The roman model. In Athman Bouguettaya, Quan Z. Sheng, and Florian Daniel, editors, *Web Services Foundations*, pages 189–214. Springer, 2014.

[9] L. De Moura and N. Bjorner. Satisfiability modulo theories: Introduction and applications. *Commun. ACM*, 54(9):69–77, September 2011.

[10] D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Półrola, and J. Skaruz. HarmonICS - a tool for composing medical services. In *ZEUS*, pages 25–33, 2012.

[11] D. Doliwa, W. Horzelski, M. Jarocki, A. Niewiadomski, W. Penczek, A. Pólrola, M. Szreter, and A. Zbrzezny. Planics - a web service composition toolset. *Fundam. Inform.*, 112(1):47–71, 2011.

[12] ECMA. The JSON Data Interchange Format. Technical Report Standard ECMA-404, 2013.

[13] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2005.

[14] D. Gavalas, V. Kasapakis, C. Konstantopoulos, G. Pantziou, N. Vathis, and C. Zaroliagis. The eCOMPASS multimodal tourist tour planner. *Expert systems with Applications*, 42(21):7303–7316, 2015.

[15] Google. AngularJS – Superheroic JavaScript MVW Framework, 2010.

[16] R. Johnson et al. Spring Framework Reference Documentation, 2011.

[17] Z. Li, L. O'Brien, J. Keung, and X. Xu. Effort-oriented classification matrix of web service composition. In *Proc. of the Fifth International Conference on Internet and Web Applications and Services*, pages 357–362, 2010.

[18] M. Mark, J. Thornton, et al. Bootstrap. The world's most popular mobile-first and responsive front-end framework., 2011.

[19] W. Nam, H. Kil, and D. Lee. Type-aware web service composition using boolean satisfiability solver. In *Proc. of CEC'08 and EEE'08*, pages 331–334, 2008.

[20] A. Niewiadomski and W. Penczek. Tripics - a web service composition system for planning trips and travels (extended abstract). In *Proc. of CS&P'16*, 2016.

[21] A. Niewiadomski, W. Penczek, J. Skaruz, M. Szreter, and M. Jarocki. SMT versus Genetic and OpenOpt Algorithms: Concrete Planning in the PlanICS Framework. *Fundamenta Informaticae*, 135(4):451–466, 2014.

[22] A. Niewiadomski, W. Penczek, J. Skaruz, M. Szreter, and A. Półrola. Combining ontology reductions with new approaches to automated abstract planning of PlanICS. *Applied Soft Computing*, 53:352–379, 2017.

[23] A. Niewiadomski, J. Skaruz, P. Switalski, and W. Penczek. Concrete Planning in PlanICS Framework by Combining SMT with GEO and Simulated Annealing. *Fundam. Inform.*, 147:289–313, 2016.

[24] J. Rao and X. Su. A survey of automated web service composition methods. In *Proc. of SWSWPC'04*, volume 3387 of *LNCS*, pages 43–54. Springer, 2005.

[25] L. Richardson, M. Amundsen, and S. Ruby. *RESTful Web APIs*. O'Reilly Media, Inc., 2013.

[26] J. Skaruz, A. Niewiadomski, and W. Penczek. Hybrid planning by combining SMT and simulated annealing. In *Proc. of CS&P'15*, pages 173–176, 2015.