Warsaw University of Technology
Faculty of Mathematics
and Information Science

MAGISTERS'S THESIS
MATHEMATICS AND INFORMATION SCIENCE

TRAVEL SUPPORT SYSTEM

SYSTEM WSPOMAGANIA PODRÓŻNYCH

Author:
Mariusz Marek Mesjasz

Supervisor:
prof. dr hab. Marcin Paprzycki

Warsaw, April 2013

....

Supervisor's signature Author's signature

Streszczenie

Obecnie najpopularniejszym źródłem informacji jest Internet. Codziennie, miliony ludzi wyraża swoje opinie na blogach, pisze recenzje w sklepach internetowych lub prowadzi serwisy tematyczne. Ze względu na wykładniczy przyrost tych danych, problemem nie jest już bark informacji, ale jej nadmiar. Rozwiązaniem tego problemu mogą być systemy skupione na dostarczeniu informacji przystosowanej do użytkownika. Budowa takiego systemu jest poważnym problemem badawczym. Z tego powodu, rozwiązanie prezentowane w ramach tej pracy ogranicza się do informacji związanych z podróżą.

Obecnie istnieje wiele popularnych systemów wspierających osobę podróżującą. Niestety, już wstępna analiza pokazuje ich największe wady – monotematyczność, mieszanie różnego typów informacji, zbyt mała personalizacja lub uzyskiwanie zbyt wielu poufnych informacji (najprawdopodobniej w celach marketingowych). System zaproponowanych w ramach tej pracy stara się przeciwdziałać tym ograniczeniom jednoczenie dostarczając użytkownikowi informacje które są dostosowane do jego preferencji. Pomysły przedstawione przy definiowaniu założeń systemu zostały oparte o dokładną analizę poprzedniego systemu wspomagania podróżnych zaprezentowanego w serii publikacji z lat 2000 – 2008.

Ze względu swój charakter, system został opracowany z użyciem agentów programowych. Z najważniejszych czynników wpływających na tę decyzję należy wymienić, że agent podróżny jest popularną metaforą agenta systemowego. Ponad to, agentowy system wspomagania podróżnych jest klasycznym problemem dla systemów agentowych. Cechy agentów programowych – proaktywne zachowanie, umiejętność komunikacji, praca na korzyść zleceniodawcy – naturalnie pasują do tego zagadnienia.

W celu znajdywanie spersonalizowanych wyników, system używa algorytmu Rhee-Ganzha, który mierzy siłę relacji między dwoma obiektami. By prawidłowo przeprowadzić obliczenia, algorytm używa ontologii zapisanej w formacie OWL lub RDF jako struktury danych. Z tego powodu, system jest w stanie korzystać z zasobów umieszczanych w ramach Semantic Web.

Częścią tej pracy jest również implementacja systemu na urządzeniach mobilnych. Ze względu na coraz większe zainteresowanie i rozwój tych urządzeń, stanowią one idealną platformę docelową dla aplikacji podobnych do travel support system. Praca magisterska zawiera dokładny opis implementacji oraz analizę wszystkich napotkanych problemów.

Abstract

Currently, the most popular source of information is the Internet. Every day, millions of people express their opinions on blogs, write reviews at online stores or run thematic websites. Due to the exponential growth of data, the problem is no longer a lack of information, but an excess of information. The solution to this problem can be systems that focus on providing personalized information. The construction of such systems is a major research problem. For this reason, the solution presented in this work is limited to information related to travel.

There are many popular systems that support travel. However, a preliminary analysis shows their greatest drawbacks - monothematicity, mixing different types of information, too little personalized content or getting too much confidential information (most likely for marketing purposes). The system proposed in this work tries to counteract these limitations, and at the same time provide the user with information that is personalized to his preferences. The proposed ideas and goals of the system are based on careful analysis of the previous travel support system presented in a series of publications in 2000 – 2008.

Due to its nature, the system was developed using software agents. Among the most important factors influencing the decision it should be mentioned that the travel agent is a common metaphor for the agent system. Moreover, an agent-based system that supports travellers is a classic problem in the field of software agents. Features of software agents - proactive behaviour, communication skills, work for the benefit of the customer - naturally fit into this issue.

In order to find personalized results, the system utilizes the Rhee-Ganzha algorithm, which measures the strength of the relationship between two objects. To properly perform calculations, the algorithm uses an ontology stored in RDF or OWL format as a data structure. For this reason, the system is able to use the resources contained in the Semantic Web.

Part of this thesis is the implementation of the system for mobile devices. Due to the growing interest in and development of these devices, they are an ideal target platform for applications similar to the travel support system. This master thesis contains a detailed description of the implementation and analysis of all related problems.

Contents

Chapter 1

Introduction

With the development of technology, people are able to solve a variety of problems. One of the interesting areas is the "world of travel". Here, the fundamental questions concern destination and means of transport:

- Where do I want to go?
- How do I get there?

Depending on the situation, the answers to these questions can be very difficult to find. For example, from the perspective of a person living in a small town who goes to a near-by store, the answers are trivial. However, a hungry tourist, who is in a foreign city, may have a big problem with finding the "correct" answers. First of all, he is only able to give a very abstract description of his destination – a place where he can order something to eat. He knows neither the exact address nor the name of the place. Thus, the answer to the first question is inaccurate / incomplete. Furthermore, no one knows his food preferences. A place recommended to him by encountered people may not suit him well (sometimes at all). Second, he is able to select a variety of means of transport (car, subway, go on foot); i.e. there is no single answer to the second question. Therefore he has to find out what would be the most appropriate means of transport.

Nowadays, he can find answers to these questions through the use of modern technology. For example, he can use his smart-phone's built-in 3G modems (not to mention the emergence of 4G infrastructures) to find suitable restaurants in the nearby area. Then, based on the ranking published on a website, he can choose the one that is the most recommended by other users and matches his food preferences. Similarly, he can use other web-based services to find the most appropriate means of transport to the selected restaurant.

Unfortunately, this approach is not without flaws. First of all, it forces him to visit several websites to gather the necessary information. Second, he has to spend time comparing the results and choosing the best one. Due to the amount of necessary work, this approach may be counterproductive. Namely, a tourist may choose a simpler but inferior (or even contradictory to his actual preferences) result, instead of looking for the most suitable one. For example, a hungry tourist may decide to choose a well-known and widely-accessible fast food chain (even though he does not like to eat there) over restaurants that serve his favourite type of food, because he does not know where to find one. The problem remains unsolved also in the scenarios where a tourist is in his room with a computer that has internet access and has spare time to plan everything. He should enjoy his vacation instead of spending time

working on the computer. This shows the need for a travel support system that will be able to perform some of the repetitive work for the user. For example, a tourist should be able to choose the best possible result from a limited set of candidates (containing results that match his preferences, while gathering and filtering should be done by the system).

This thesis focuses on modern applications that support travellers. The following section will carefully analyse the existing solutions to find their limitations. Next, a new idea of the Travel Support System will be presented, which will be able to overcome these limitations. Details of the implementation and results of tests will be also presented.

1.1. Existing travel-related systems

There are several examples of services or applications, which provide a variety of travel-related information to their users. The list contains also two no longer developed systems — Travel Support System (predecessor of the new Travel Support System proposed in this thesis), and Chefmoz (ontology-based collection of restaurants). Despite the fact that they are no longer active they provide a useful source of information and reflection.

Travel Support System (TSS; 2000–2008) The idea of the Travel Support System was introduced in the MS Thesis from 2000 [42]. The TSS was an agent-based system responsible for collecting information available on the internet, storing them in the form of an RDF database, and creating highly personalized answers to users' queries. The initial idea was explored in a series of agent-related publications that appeared between 2001 and 2008 (available at [31]). The working system was released in 2006 at the SourceForge [32].

Interestingly, the need to redesign the system became evident immediately after its initial implementation. Most important lessons learned in the process of developing and evaluating the original system were summarized in [40]. The publication pointed out several design flaws concerning the utilization of agents in the original system.

However, none of these flaws have been resolved. Due to broken software dependencies, the system has not been in development since 2008. Therefore, the new implementation of the system proposed in this thesis has to carry out a detailed analysis of system requirements, taking into account the previously acquired knowledge and new opportunities associated with the development of new technologies.

Chefmoz (2000 – 2011) Chefmoz was a branch of the Open Dictionary project [22] and contained an online directory of restaurants and reviews. The collection covered more than 300,000 restaurants from 142 countries. Due to the persistent technical difficulties and lack of funds, the Chefmoz project was announced "dead" in 2011. Similarly to the Open Dictionary Project, the Chefmoz provided its data as a hierarchical ontology scheme. In 2009, ChefMoz had become the largest global directory of restaurants available on the internet.

Booking.com Booking.com [11] is a web portal that focuses on hotels. It allows its users to post their opinion, browse hotel deals and book rooms. Website has a number of important features such as a large database of hotels (over 250,000 entries), unbiased opinions (posted by real hotel guests), and worldwide accessibility (consumer support in more than 40 languages). Here, it represents a broad class of hotel-focused services (information aggregators) such as: hotels.com, venere.com, etc.

Despite the fact that Booking.com is a great web service, it is also highly specific (it is limited only to hotel information). Due to this, Booking.com can not answer more complex queries (for example, it can not create a list of hotels which are located near museums or art galleries). Moreover, Booking.com does not provide personalized recommendations. For example, it can be conjectured that a user who frequently uses this service would be pleasantly surprised by receiving offers adjusted to her preferences, instead of a regular newsletter containing only "statistically good deals" (for example, the cheapest prices).

Social Services Many social services may also be a source of travel-related information and may affect users decisions of where they want to spend their time. The two most popular social networks are Facebook [13] and Twitter [34]. Twitter is a service that allows its user to send short text messages that are visible to the public. In addition to the text, its users can also add a picture or indicate a place on the map. Because of this, some of the message can influence our travel preferences. For example, if we follow the accounts of our friends who frequently visit one place, publish a lot of cool pictures and indicate the position of this places on the map, it is more likely that we also decide to go there in the future.

Facebook, like Twitter, provides tools to communicate with other people. Its users can post comments, upload photos, share information with their friends or press the "like" button to show that they like something. As with the frequent messages on Twitter, the information provided by our friends may influence our decision to visit a specific place. Moreover, many places and events have their own Facebook pages, which makes Facebook a kind of a travel-related service. Observe that TripAdvisor [33], a web page similar to Booking.com, uses this idea and integrates some of their services with Facebook. Customers may connect to their Facebook accounts and see offers from places visited by their friends.

Although this is an interesting use of social services, this approach has many drawbacks (from the point of view of our main ideas). First, these services allow its users to share all kinds of information (not only information related to travel). For example, the user can post on his Facebook wall either information about a restaurant which, in his opinion, serves the best pepperoni pizza (travel information about a certain place) or a comment about how he feels about the current political discussion concerning taxes (information not related to a specific place which can be recommended). Moreover, one of his recent posts can contain information about his new girlfriend and a club where they first met. In this case, the system is only interested in a part of the information (specifically, the location of the club). Thus, all the information has to be filter by an external application. Secondly, in case of the social services, the term "recommendation" is not appropriate (for example, the number of similar messages may encourage a user to go to a specific location or be involved in a particular event, but none of them was a "direct" recommendation). Finally, the external system does not utilize the personal profile, but makes recommendations based on people's opinion from the user's list of friends. Therefore, recommendations may not match the current / actual user preferences.

Yelp! Yelp! [37] is a web catalogue of local businesses. Yelp! operates as a social network service – its content is filled by advertisers (local businesses) and ordinary users (reviewers and potential customers). The drawback of such a solution is that the system does not provide personalized or context-driven recommendations. For this reason, the users receive only regularly published newsletter which is barely connected to the user

profile (for example, the name of the city where the user declared to live). As the result, the user receives a limited number of local offers that may or may not match his preferences.

Yelp! is also a medium for friend-based recommendations. Unfortunately, this approach is not as good as automated recommendations. First of all, such recommendation has to be done manually, which takes time that the user may not be willing to spend. Second, it is very dependent on the person who sends the recommendation, not on the actual user preferences. Therefore, the system should relay on automatic recommendation made be the system (the user-based recommendations can be utilized as a supplementary function).

Google Now Google Now [15] is an intelligent personal assistant provided by Google for mobile devices capable of running the Android operating system [7] in version 4.1 or newer. This application has been released on July 9, 2012 as an extension of Android's native Google Search application. The most important feature of Google Now is the ability to display information obtained in the interaction with the Google services. An example of such information may be today's weather, traffic information or recommended dishes in a restaurant. All the information provided by the application are selected based on Google data such as Web History and location services (for instance, the GPS).

Although Google Now seems to be similar to the Travel Support System, there are some significant differences. First of all, the Travel Support System has been proposed as a system focused entirely on supporting travellers. In the case of Google Now, the primary function is to help users to use the Google services. Because of this, the understanding of user's needs is limited by the functionality of a number of independent service. For example, Google Now is able to provide some local information such as nearby attractions and events. According to the description provided at the Google Now website, such information is shown if location services are turned on (there is no reference to the user preferences and search history). In comparison to the Travel Support System (which always takes the user preferences into account), the utilized Google service does not understand that some of the proposed places may be opposed to those favoured by the user (and therefore Google Now inherits this limitation). Second, Google Now forces its users to store all their personal information on Google's servers. Despite the fact that this is a useful feature (for example, if the device was stolen), the users have no choice (otherwise it would be difficult to share one Google profile with the multiple remote and independent services). In the case of an application such as the Travel Support System, the number of needed information is limited by the specific use of the system (only the travel-related information). Thus, the user may feel safer if all the information is not stored in one place (not mentioning the fact that the personal profile can be stored locally).

1.2. Proposed System

This section describes the main features of the Travel Support System. Many of them correspond to the well-known functions from travel support systems presented in the previous section, but several are also new. Before the list of requested features will be presented in detail, let us concentrate on those that will make the Travel Support System unique. Our considerations start from three use case scenarios, which emphasize the difference between the currently existing applications and the desired solution.

Scenario 1 Peter is an employee in an IT company. Every Friday after work, he goes to the city to take a break from his computer. Out of habit, he stores such events in his Google calendar (for example, labelled as "going to club"). At first, he was very happy with the club that he had managed to find. However, after a few visits he began to be bored. Fortunately, the next Friday morning, the Travel Support System installed on his smart-phone will prepare for him a new recommendation based on the entries in his Google Calendar (system will recognize the word "club", and match it with the repeating entries). It will recommend him a new club which he did not know before. This club plays his favourite type of music and is on the way between his home and work.

Scenario 2 Jack is a student. He recently met a girl named Jill, and began to chat with her via SMS. After Jack sends a text message to Jill asking for a date, his instance of the Travel Support System will start negotiations with the one running on Jill's phone. Since Jack is in her contacts, her Travel Support System will be able to share some information about her preferences (learned through her previous interactions with the system). As a result, Jack will receive a notification from his Travel Support System with the recommendation of a Chinese restaurant. Moreover, Jack will also be informed that Jill will most likely be satisfied with this choice since she likes Chinese food (this information will be confirmed by her instance of the Travel Support System).

Scenario 3 David is a pensioner and has two hobbies – golf and travelling. He used to start preparations for his holidays by creating a list of hotels. Then he devoted his time to comparing this list with the list of available golf courses. Fortunately, he uses now the Travel Support System, which is aware that David likes golf (based on his search history). The system will automatically prepare a list of hotels prioritizing those, which offer special medical / spa treatment and have easy access to golf courses.

From the presented use case scenarios, one can derive several noteworthy observations. First of all, each scenario presents one unique feature of the system.

- In the first scenario, the system shows its pro-activity. The system tries to propose Peter something new before he starts to get bored and falls into a routine.
- The second scenario presents the concept of negotiations between instances of the Travel Support System. Jack does not know Jill, so the system works in his favor by negotiating the best place for a date.
- The last scenario focuses on cross-referencing. Typically, similar systems use only one specific source of information (for example, Booking.com provides its services only based on the hotels database, and nothing else). Due to this, David does not have to waste his time browsing several web portals.

Furthermore, comparing the second and third scenario, we can observe that the system does not dependent on the distance (a local restaurant or a foreign hotel) and amount of preparation (now or in a few weeks). The system does not separate a spontaneous decision (Jack spontaneously invites Jill for a date) from a deliberate action (David carefully plans his holidays a few week ahead of time). Depends on a situation, the system can either produce immediate results or take some time to prepare an in-depth list of potentially interesting places.

Let's summarize our assumptions toward the TSS as use case diagram. The diagram 1.1 shows three entities located outside the system: the *GPS*, the *User* and the Data Source. They interact with the system by asking queries (*User*), providing information (GPS, data

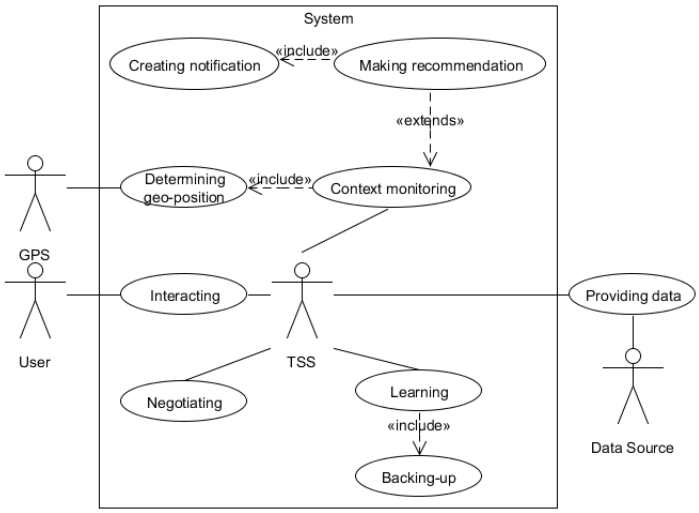

Figure 1.1: Use case of the Travel Support System

source) and negotiating. The main actor of the system is the User. The TSS is involved in five functions: *Interacting* with the *User*, *Context monitoring*, *Learning*, *Negotiating* and *Cross Referencing*.

The *Interacting* function involves preparing a list of results that match user preferences and correspond to a given query. It is also connected to *Cross-referencing* and *Learning*. *Cross-referencing* is used to obtain the results from one or many *Data Sources*. By the nature of the system, each action taken by the *User* triggers the *Learning* function. During the learning phase, the system adjusts user preferences stored in the hierarchical structure called the user profile. The *Learning* function is also connected to the *Backing-up* function that stores the current user profile on a remote server. Due to this, the system is able to obtaining only useful information from the *Data Sources* (the information that match the user preferences). The *Context monitoring* involves dealing with all contextual information available to the system. The most important contextual information is the geo-position provided by the *GPS*, but the *Context monitoring* may also be related to many other types of information (for example, text messages, calendar entries and remaining battery power in case of mobile devices). *Context monitoring* can trigger the *Making recommendation* function. The *Making recommendation* function involves proactive searching for information based on the current user context. It is also connected to the *Cross-referencing* and *Creating Notification* functions. The *Creating notification* function involves displaying a notification about a new set of recommendations made by the system. The *Negotiating function*, that connects two instances of the TSS, involves exchanging messages in order to achieve an agreement. In case of the TSS, the agreement may relate to a meeting place that will be satisfactory for both users. The *Negotiating* function is also connected to *Checking permissions*. *Checking permissions* ensures the confidentiality of shared information (for example, the local TSS will not start negotiations with a remote TSS if its owner is not on our contact list).

From the above description we can formulate the following functional requirements:

Context-awareness In order to provide the best possible response to user queries and be proactive, the system has to monitor the user's contextual information. Since the proposed system is related to widely understood travel, the most useful information is

the user's current geo-postion (GPS feed). Such contextual information is used by the system almost all the time (except situations where the user decides otherwise).

In scenarios 1 and 2, two additional proposals were presented; entries in the calendar, and text messages. Such information is to be used by the system to be proactive. It does not directly alter user queries, but they may trigger some autonomous actions within the application. For instance, in the first scenario, the entries from the calendar are not included in the user data (the system does not understand what "going to" means), but it tries to use them to produce a useful recommendation (it can query the word "club" and check the results against the user profile).

Personalized Searching The result of performing a search process has to be personalized to individual user's requirements. The requirements may be specified directly (user explicitly adds them to the query), or indirectly (the system is aware of user's previous choices and tendencies). The system has to intelligently reject data contrary to the direct, or indirect, requirements and favor those which overlap with user's preferences, and may more likely fulfil the user's needs. Due to this, only the personalized results are presented as an answer to the specified query. In the third scenario, the system is aware of David's preferences. Therefore, the system dynamically combines results and the user's preferences and goes beyond mere presentation of answers to queries. Acting pro-actively, the system is able to suggest possible places and events that may prove useful to the user. Therefore the system favours hotels with easy access to golf courses over those that do not have it or where such access is very difficult.

However, the situation in the first scenario is a little bit different. The system tries to be proactive and proposes the new club to Peter. So the system should take changes of user preferences and desires for something new into account. Thus, a user who sticks to his/her old habits may develop new preferences (for example, going to the same club over and over again). From time to time the system should also return a result which is neutral or even contradictory to the stored preferences. The average time between such propositions may be measured by analysing contradictory user's queries. The contradictory result should always be somehow separated from others and marked by the program as a suggestion. However, the system should be aware that some preferences may be nearly unchangeable.

Collecting important user information In order to allow intelligent searching, the system must be able to collect important information about a user to process it in a collection of preferences and requirements (user profile). This collection is to be dynamically developed over time based on user interaction with the system (user's queries /activities and decisions).

Negotiations Topics of negotiations may be different depending on the implementation of the system. For example, the Travel Support System proposed in 2000 [42] assumed the possibility of negotiations between two instances of the system – a travel-related service (seller) and a tourist (buyer). That system was to have the ability to negotiate special offers and promotions based on user preferences. For instance, a restaurant's recommendation might also include a free bottle of wine liked by the user.

The new version of the system extends this concept. Negotiations are also present in communications between two or more instances of the system on the user's side. Applications can come to an agreement if their goals are common. In the second scenario, Jack and Jill want to go on a date, but none of them knows the best common place to

go to. Therefore, the two systems are able to agree on a place and inform Jack, so he can make a good impression on Jill.

Cross-referencing Many modern information services provide only data pertinent to a single subject. For example, Booking.com provides information concerning hotels, but does not provide information about nearby restaurants (not counting hotel restaurants). Similarly to the example presented in the third scenario, the users have to cross-reference multiple sources of information on their own. Therefore, to achieve a reliable system that is able to advise a user in any situation, the system has to have the possibility to combine different sources of information in such a way that the cross-referencing data will be allowed to respond to the more sophisticated queries.

Online data storage and accessibility of the user profile from multiple devices Due to the fact that our system is targeted mainly at mobile devices (see, scenario 1 and 2), we may encounter several problems to deal with. Namely,

- growing risk of data loss caused by accidents while traveling (for example, losing or damaging the device),
- need of copying the user profile from one device to another.

In the first case, an extremely important factor is the time that the system spends learning user preferences (creating a user profile). Total loss of the user profile indicates the need to re-adjust the system to individual needs. Through this, the system loses its functionality to present the user relevant and personalized results and suggestions for the time of relearning the profile.

The second problem is not so important and can be easily solved by connecting the device with an old user profile to one with the newest version and simply copy the profile files. However, if the user forgets to copy his profile from one device to another then the system may not give the most accurate results with respect to the current user's preferences.

The solution to these two problems is the possibility of online storage and synchronization. The user should be able to enable backing-up of his current profile, which will be stored on a remote server (e.g. in a cloud).

Moreover, the system should fulfil the number of non-functional requirements:

Accessibility All three scenarios show that the system can be used by people of all ages (students, young workers and pensioners). Due to this, the system should present results in a readable and accessible form such as, for example, a list. The results should be sorted according to individual preferences from the most to the least satisfying the user's requirements. Each result should include a brief description of the key information. After selecting a response relevant to the user's query, the system should display more detailed information.

Security Because the system can negotiate, a security mechanism should be used to decide what information should be shared. First of all, not everyone should be able to start negotiations. For example, people who are not listed contact user are most likely strangers and should not be allowed to participate in the negotiations. Secondly, the shared information should be limited to the subject of negotiations. For example, if the system starts negotiations on the place where the user likes to eat, the system should not provide information about the user's other preferences.

Software and hardware independence Due to a variety of devices which may be used when traveling, the system should be as software and hardware independent as possible. The system should be able to run on smart-phones, tablets, notebooks and desktop PCs. Thanks to this, the system will be more flexible and will give the user an opportunity to choose the most comfortable device to be used at a particular time (in the hotel room, the most preferable device could be a laptop, when on a crowded street it may be a PDA or a smart-phone). The system should take limitations of each device into account and provide a suitable display mode for the amount of available resources.

Chapter 2

Software agents in the Travel Support System

The system proposed in this thesis is an attempt to expand upon the initial idea of the old Travel Support System. One of the most important features introduced in the old proposal was the utilization of software agents [43]. However, the resulting system was subjected to a thorough analysis in [40], which pointed out several serious design flaws concerning the usage of agents. The list below summarizes the most important remarks:

- Software agents should not be used as a middle-ware solution, unless it is beneficial.
- Software agents should not replace currently existing technologies, unless there is a specific need for such an approach.
- Designing an agent-based system requires finding balance between the autonomy of an agent and its need to communicate.

Unfortunately, none of the above flaws were fixed, because the old Travel Support System became obsolete due to software dependencies. Keeping in mind the time that has passed since the last publication and update in 2008, it is necessary to re-evaluate the decision to utilize software agents. Let us start our considerations by defining what a software agent is.

2.1. Software Agents

There is no one generally accepted definition of a software agent. On the basis of a wide variety of publications and books on the subject, one can specify a range of features which the authors attribute to agents and agent systems. Based on these characteristics and well-known examples of agents (such as an insurance agent or travel agent), it is possible to intuitively define what a software agent is.

A software agent, like a human agent, can work in an organized structure called a multi-agent system (MAS) [41] in which each agent is assigned its own agenda. In the MAS, each agent has tools relevant to its duties. For example, in a MAS acting as an office suite, a calendar agent, whose task is to optimally organize meetings, has access to the calendar, but it cannot read e-mails. Similarly, an e-mail agent can deal with viewing the e-mails and filtering spam, but it cannot read entries in the user's calendar. The presented example shows another

important feature of the agent system – agents must be able to cooperate with other agents. If the calendar agent wants to pre-book time on the calendar, taking e-mails into account, it will have to work with the e-mail agent. On the other hand, the interaction between agents may change in a competition. For example, if several customer agents want to buy the same item, they will have to compete with each other (for example, by organizing an auction). As a result, the MAS can model a very complex system containing agents with common and conflicting goals. In addition, the agent has to decide whether to cooperate to achieve the common good or to compete in order to achieve its own goal.

Agents can interact indirectly (by acting on the environment, in which they live) and directly (through communication and negotiation). Communication and negotiation are particularly important in the context of cooperation and competition. The ability to communicate is also connected with another important feature of the agent system – it is assumed that a software agent is trustful. This means that an agent cannot deliberately provide false information. For example, an agent, which takes part in an auction, cannot offer higher rates than allowed by its client. Otherwise, this agent will prevent other agents from reaching their goals — the vendor agent cannot sell its product if the client is unable to pay and the other buyer agents are unable to buy the goods. In addition, the agent can expose his client to various troubles and therefore it will lose its reliability.

Agents are also mobile. This means that they can migrate in search of the resources needed to achieve their objective. For example, an agent performing calculations can go from one machine to another to gain access to a better processor for more computing power (and thus to do the job faster). This is a very interesting feature, because it often requires the agent to decide what is more cost-efficient migration to another platform or communication with other agents.

The ability to migrate between devices and the possibility of indirect interaction between agents (impact on the environment) indicates that the software agent must be autonomous, adaptive and pro-active. The agent must be able to properly react to changes in the environment. Otherwise, the agent can get to a state where it will not be able to achieve its goal. For example, an agent working on a device with limited battery capacity must manage resources wisely. In case of low battery power, the agent may decide to either migrate to another platform to continue its work or to turn off less important functions. Pro-activity is one of the biggest advantages of software agents because it is not just a simple response to changes in the environment. Adapting to the environment and user requirements indicates that the software agent must also be intelligent and able to learn. For example, the mail agent should learn how to classify incoming messages and learn the patterns considered as spam.

2.2. Software agents and the new Travel Support System

The Travel Support System shares a lot of features with the software agent. The system lives in a changing environment and has to properly react to these changes. For example, some places can be closed before the user will be able to get there. The system can conclude this based on the current time, distance between the user and the destination and opening hours. Adapting to a changing environment is one of the software agent's features. Software agents are also autonomous and pro-active. For example, if the user does not manage to be on time before a place is closed (due to a traffic jam), then the agent can recommend a similar one

which is sill open. A similar situation takes place in the first use case scenario. The system acts by itself and makes a new recommendation to Peter.

In the second use case scenario, the two instances of the Travel Support System negotiate the best place for the meeting. Typically, it would take a lot of time and effort to implement a suitable interface for communication and negotiations. Fortunately, software agents provide such an interface since it is a required feature of the MAS. Moreover, the MAS can have many different agents with common or conflicting goals. Thus, such negotiations can be organized with more than two instances of the Travel Support System. For example, a manager can invite all his co-worker to a business meeting in a restaurant and all the negotiations (time, date, place) will be completed in the background (by several instances of the Travel Support System).

The Travel Support System, like a software agent, has to be intelligent and able to learn. In the third scenario, the system learned about the user's preferences and was able to prioritize hotels near golf courses over others. This scenario also mentioned cross-referencing the list of the hotels and the golf courses. It can be a very difficult task to implement one application to connect to many different services (using different interfaces). Such a complex problem can be model as the MAS. For example, each data source can have an appropriate software agent assigned. The cross-referencing process can be executed in the form of communication between one main agent (the agent, which will prepare the list of results) and several data source agents.

The above considerations prove that utilization of software agents can be beneficial for the Travel Support System. For this purpose, the decision was made to use the JADE agent platform. A detailed description of the platform is presented in the next section.

2.3. JADE agent platform

The JADE [19] is the leading open-source framework for agent systems written entirely in the JAVA programming language. The copyright holder of the JADE is Telecom Italia. The JADE significantly simplifies development and deployment of any multi-agent system (MAS) through middle-ware that complies with the Foundation for Intelligent Physical Agents standards [14].

It also provides a set of graphical tools that support the debugging and deployment phases.

The following list describes the most important functions of the JADE agent platform:

JADE is written in the JAVA programming language An extremely important feature of the Java programming language is the ability to easily move code between different operating systems. Due to the fact that the Travel Support System supports travellers, this is a very useful property. The traveller will benefit not only from a computer in a hotel room or a laptop in a cafeteria, but will also receive help during a walk around the city when the most convenient device is a smart-phone. The JADE ensures that once written agent will be able to run on different operating systems (with support for the Java programming language) without changing a single line of code.

JADE is popular JADE has a community of developers who create add-ons for it. In addition, on forums and mailing lists, you can find a lot of useful information in case of problems.

JADE can be run on mobile devices With the increasing popularity of mobile devices with high speed Internet connection, software agents obtain new ways to support their users. An agent that lives only on a desktop computers (even migrating between different computers) cannot always interact with its user. It would be a huge restriction that would probably significantly limit the use of agents in general. For example, an agent that supports travellers would force its user to either plan everything in advance or frequently return to the hotel to use a computer. Moreover, this approach would be very sensitive to all random events. For instance, the selected restaurant can be leased to a private party, and the user may not be able to enter without an invitation. In this situation, the user has to either go back to the hotel or (more likely) be angry and choose the next available restaurant, which may not match his preferences. The JADE developers were aware of the benefits of running agents on mobile devices and therefore have provided several add-ons to the JADE agent platform to make this possible. The first add-on (for J2ME [20]) is called JADE LEAP and is capable of running an agent container in the split-execution mode (this approach requires a persistent connection to a remote machine running an agent main container). The second add-on (for the Android operating system) is called JADEAndroid and it is a part of JADE LEAP. The most significant difference in comparison to JADE LEAP is the capability of running a stand-alone JADE platform (without the need to be non-stop connected to the internet; see, also, next section).

JADE provides useful tools (WADE) WADE [36] is a software platform based on JADE. It provides support for execution of tasks described as work-flows. WADE comes with a development environment called WOLF. WOLF is an Eclipse [12] plug-in. WADE will be utilized within the system due to its robust functionality and capability of being integrated with an agent on an Android device (requires the JADE Android add-on). Moreover, it will also provide a set of developer tools during the implementation phase (for example graphical representation of processes defined as workflows).

2.4. JADE on mobile devices

The JADE LEAP add-on was developed by the JADE team to support running the JADE agent platform in split-container execution mode on devices with JAVA Micro Edition (formally known as J2ME). Split-container execution mode requires a persistent connection to a remote machine running a JADE main container. Execution of the agent container is split between the device (front-end) and the remote machine (back-end). The drawback of this approach is the need to maintain the connection. If the connection is broken, then the agent container (and all its agents) is suspended until the connection with the remote machine is re-established. In the case of some rare devices (with the less popular version of J2ME), it was possible to run the JADE agent platform in stand-alone container mode. The stand-alone container mode, unlike the split-execution mode, allowed to run the full JADE agent platform. However, devices utilizing J2ME quickly passed away, making room for the first smart-phones.

Due to the growing interest in more powerful mobile devices, such as smart-phones and tablets, the Travel Support System will use the JADEAndroid add-on. The JADEAndroid is an add-on for JADE which allows to run a multi-agent system on Android. It is also a part of the JADE LEAP add-on. Since Android was developed for smart-phones and tablets (more powerful than the J2ME devices), the JADEAndroid supports split execution and stand-alone

modes. Due to the nature of the system, it is a very important feature. Namely, a traveler may not be able to maintain the network connection with the remote host in all situations or the cost of the internet connection may be too high (due to the cost of roaming). In such situations, data transfer should be limited only to receiving the most important data. Running the JADE platform in split execution mode can be inefficient. The Travel Support System should be autonomous and, therefore, it will run in the stand-alone execution mode.

This section covers a careful analysis of the JADEAndroid add-on. The section starts with a short summary of the most important information about the Android Operating System. Then we discuss some problems concerning running the JADE agent platform with JADE-Android. The section ends with a proposition of solutions to address these problems and a presentation of their implementation.

2.4.1. Android Operating System

Android [7] is an open source, Linux-based operating system [21] developed by Google for touch-screen mobile devices such as smart-phones and tablets. Nowadays, Android is the most widely used mobile operating system. The keys to its success are the increasing number of new applications that extend the initial functionality of the system and a large community of developers. The source code of the system is provided by the Google under the Apache License [9]. Due to this, the system can be freely modified and distributed by manufacturers, wireless carriers and enthusiast developers (for example, OUYA [23], an open source game console, runs a modified version of Android 4.1). Moreover, Google provides a set of developer tools called Android SDK which aids developers in providing their application to the users. The Android applications are distributed either for free or at a cost by a digital application distribution platform called Google Play [16] (previously known as Android Market). Users can customize the functionality of their devices by installing new applications. Furthermore, a lot of popular PC applications have their versions for Android (for example, FireFox, Opera, Chrome, Avast Anti-virus, and so on).

Android Application

Android applications are written in the Java programming language. Since the Android operating system is a multi-user Linux system, each application is a different user identified by a unique Linux user ID. A Linux user ID is unknown to its application and is used only by the system.

An Android application is composed of one or more application components — activities, services, context providers and broadcast receivers – which are packed into a single .apk file. All four types of application components can be described as below.

Activity An activity [1] is an action that can be performed by the application on a single screen with a user interface (for example, in an e-mail application, one activity can show a list of new e-mails and another one can compose an e-mail). Despite the fact that they work together, each activity is independent of the others. By the nature of the Android operating system, activities can be called between applications (for example, a camera application can start an activity from an email application – if the email application allows it – in order to send a picture).

Service A service [6] is an application component that performs long-running operations in the background (for example, playing music or monitoring data). It does not provide a user interface. On Android, Services can be accessed by local application components (in the same VM) and by other processes via the AIDL. There are two types of services with respect to their lifespan – started and bound. We can say that a service is started if it an application component (for example, an activity) starts it by calling the *startService* method. A started service can run in the background indefinitely until it is directly stopped by calling the *stopService* method. On the other hand, a bound service is created when an application component binds to it by calling the *bindService* method. There can be multiple application components, which are bound to a single service at the same time. Clients interact with a bound service via a provided interface; if the binding was successful. A bound service runs as long as at least one application component is bounded to it. If all its clients call the unbindService method, the service is immediately removed. A service can be started and bound at the same time. For instance, a service that has to be continuously running, but also provides the inter-process communication (IPC). In such a situation, both conditions must be satisfied before the service is destroyed (an application component calls the stopService method and all its clients are unbound).

Content provider A content provider [10] is a standard interface that manages access to a structured set of data for applications running in a different process. Through the content provider, the other applications can query and modify the data (if the content provider allows it). For example, the Android operating system provides a content provider for accessing the user's contact information. If an application has the proper permissions, it can read and write information about a particular person. A content provider is only required if data has to be shared by multiple applications. However, some developers can also find it useful for reading and writing private information.

Broadcast listener The Android system provides system-wide broadcast announcements as a form of communication. A broadcast listener [10] is an application component responsible for listening for and responding to them. For example, an application can be informed when the screen is turned off or when the battery capacity reaches its minimum. Applications can also initiate broadcasts. For example, an application can inform others that some data has been downloaded. Broadcast listeners do not provide a graphical user interface. Typically, they use simple listeners which initiate other application components and, in general, do a very minimal amount of work.

Since the Android system is an operating system for mobile devices, such as smart-phones and tablets, it is important to understand how application components work together and how to properly utilize them. First of all, the system tries to reduce the amount of resources used and preserve battery power by killing unused application components. A detailed description of how the system marks an application component as unused (or as killable) is presented in the next section. Second, activities, services and broadcast listeners are executed in the main thread of their application. Therefore, it is necessary to carefully plan the execution of long-running operations to avoid Application Not Responding (ANR) errors. Fortunately, the Android system allows to run some application components in separate processes, if it is explicitly written in the application configuration file.

Life-cycle of Android Application

As mentioned above, an application is a set of application components. To properly understand how an application works, it is necessary to understand when its application components are created and when the system destroys them. Thus, this section focuses on the life-cycles of activities and services.

In the Android system, an activity can be in one of three states: Resumed, Paused, Stopped. In the Resumed state, the activity is running in the foreground of the screen (it is visible to the user). This state implies that the activity cannot be killed. In the Paused state, the activity is partially covered by another activity. Thus, the system can kill such activity when it is in the "extremely low memory" situations. In the Stopped state, the activity is in the background, so it is not visible to the user. In such a situation, the system can kill such an activity when memory of the activity is needed elsewhere.

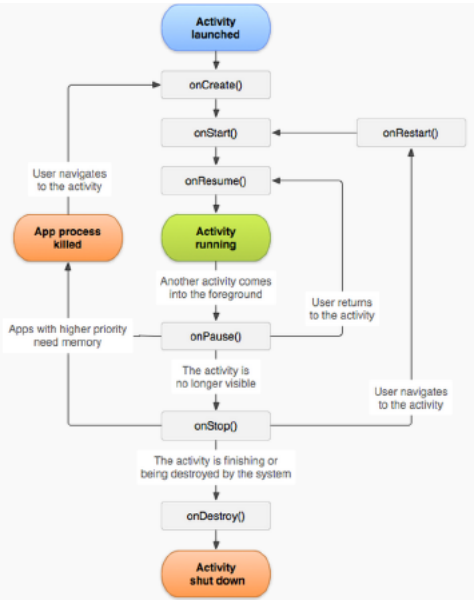

Figure 2.1: The activity life-cycle

Figure 2.1 presents the detailed visualization of the activity life-cycle. The states of the activity are represented by its methods and the transitions between them are shown as arrows:

onCreate – the activity is created for the first time. In this state, it should set everything up – bind or start services, create user interface and so on. The activity is not Killable.

onStart – the activity is visible to the user. This state means either the activity was created (onCreate) or it was stopped (onResume). The activity is not Killable.

onResume – the activity is at the top of the activity stack. This means that the activity receives user input. This state is also called after the activity was paused (another activity partially covered the screen). The activity is not Killable.

onPause – the activity is partially covered by another activity. The activity is Killable if an application with higher priority requires additional memory.

onStop – the activity is no longer visible to the user. The activity is Killable if an application with higher priority requires additional memory.

onResume – the activity was invoked by the user and brought back to the foreground. The activity is not Killable.

onDestroy – the activity is finishing or being destroyed by the system. The activity is Killable if an application with higher priority requires additional memory.

There are four additional states: Activity launched, Activity running, Application process killed and Activity shut down. The initial state is the Activity launched. This state indicates that the activity has been started by the user, or another application. The Activity running state represents the time, in which the activity is in the foreground (it is completely visible to the user) and performs some work. Two states indicate that the activity was killed: 1) Application process killed, and 2) Activity shut down.

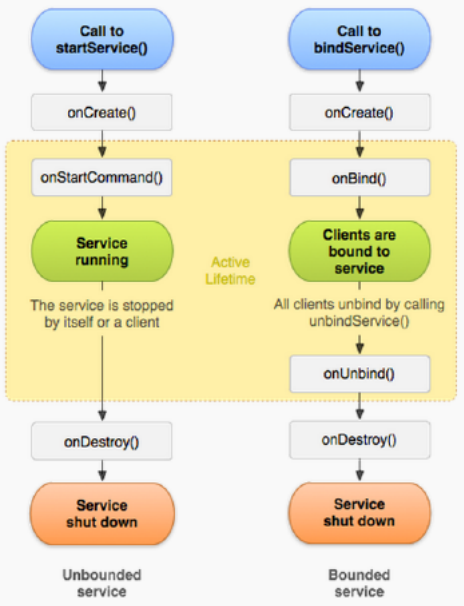

Figure 2.2: The service lifecycle. The diagram on the left shows the lifecycle when the service is created with the startService method and the diagram on the right shows the lifecycle when the service is created with the bindService method

The service, unlike the activity, does not depend on visibility on the screen. There are two types of services – started and bound. Each type works a little bit different than the other. Figure 2.2 shows a detail diagram representing the life-cycles of the started (on the left) and bound (on the right) services. The states (represented by the methods of services) can be described as follows:

onCreate – the service is created for the first time. The service is not Killable.

onStartCommand – the activity is visible to the user. This state means either the activity was created (*onCreate*) or it was stopped (*onResume*). The activity is not Killable.

onBind – this state is only available to the bound service. A new client is bound to the service.

onUnbind – this state is only available to the bound service. One of the clients becomes unbound. If there are no more bound clients, the service is immediately destroyed (*onDestroy*).

onDestroy – the service is either stopped by one of its clients (started service) or it has no bound clients (bound service).

There are three additional states: Service lunched, Service running and Service destroyed. Each state is a little bit different with respect to the service type. If the service is started, it was created with the *startService* method. Then, it will remain running until one of its clients calls the *stopService* method. If the service is bound, it was created with the *bindService* method and it will remain running until all of its clients call the *unbindService* method.

The term killable means that the system can (but does not have to) kill the application components. In case of an activity, the system should kill it when the activity is in the background and another application with higher priority requests additional memory. It is also possible that the system will kill an activity if it runs out of free memory. In the case of a service, the system will kill it immediately after calling the *stopService* method or when the last client becomes unbound.

Interprocess Communication (IPC [46])

On Android, an application lives in its own security sandbox — the code is isolated from other applications. Each process has its own virtual machine (VM) responsible for executing the code. For this reason, the system provides a very secure environment. On the other hand, this raises the question concerning interprocess communication (how two applications can share data?). Fortunately, the Android system provides several methods to do so. In this section, we describe two methods mentioned in the official documentation.

Android Interface Definition Language (AIDL) AIDL [3] is an Interactive Data Language (IDL) [18]. The bound service can provide AIDL as an interface for the interprocess communication. Clients from different processes can bind to the bound service like to a local one by calling the *bindService* method. It is worth to notice that a service has to be uniquely identified. The drawback of this approach is that the Android system can only send simple types of data such as string, integer, double, list and so on. For more complex objects, the developer has to implement methods to transfer them between different processes.

Two applications can share the same Linux user ID If it is directly specified in the application configuration file, two different applications can share the same Linux user ID. This means that they can access their components (for example, one application can call an activity in its sister application). Moreover, they can share the same VM.

Application's permissions

The Android operating system implements the principle of least privilege [26] (also known as the principle of minimal privilege, or the principle of least authority). This means that an application has access only to the amount of resources required to complete its task. An application can request permissions to access device data (for example, user's contacts, text messages) and services (Blue-tooth, Internet, camera and so on). A user has to confirm all application permissions before installation.

Communication within the Android operating system

Application components communicate via messages called intents [2]. Intents are used in a variety of ways. First of all, they are used to activate application components. For example, an activity, which allows to create an e-mail, can start another activity, which operates the camera, to take a photo as an attachment. Second, intents are used to make system-wide broadcast announcements. Such broadcasts contain a description of something that has happened. For example, the screen was turned off, the battery reached the critical level and so on.

2.4.2. Analysis of JADEAndroid

The system should support a traveller in any situation. Due to this, in the analysis of the JADEAndroid add-on, we focus on the stand-alone mode. However, some problems and solutions can be also applied to the split-execution mode.

The JADEAndroid runs the main agent container as a bound service. Since Android services are more autonomous than other application components and can perform long-running operations, this was an obvious choice. However, a bound service is immediately terminated after all its clients become unbound. Therefore, the life-cycle of the main agent container strongly depends on its clients. Typically, a bound service is bound to activities. From the activity life-cycle, we know that activities have the lowest priority in the Android system. Namely, an activity cannot be killed only if it is completely visible on the screen. When an activity is only partially visible or in the background, the system can decide to destroy such an activity if additional memory is needed. It is important to observe that the official Android documentation does not define *when* the system can make such a decision and the term when additional memory is need can describe multiple different situations. The termination of an activity also means that all its resources (including bindings to services) have to be released and if all activities (clients) bound to a bound service become unbound, then the service is also destroyed. This problem does not apply to all possible agent applications in Android, however, it significantly limits opportunities, in which software agents can be utilized. For example, let us consider an Android application which communicates with sensors placed on an elderly person. The application has to monitor the vital parameters and properly react to critical changes. In this case, it is important to keep the agent platform running as long as possible. However, the developers cannot force the user to keep an activity bound to the JADE service on the screen. Thus, the JADE service has to become independent of the other application components and, therefore, the visibility on the screen.

In the Android system, each application component is uniquely identified. Due to this, it is possible for an application component to call another one from a different application. Services also have this feature. However, the agent main container, as a service, is accessible only locally. This means that only the application components, which belong to the same application that runs the service, can access the agent main container. If an application component tries to run the agent main container when it is already running in the process of another application, the service will be killed and recreated for the calling application component. In the termination process, all existing agents will be killed.

From this analysis, we can clearly identify two major limitations:

1. The expected life-time of the agent platform running on JADEAndroid cannot be estimated.

2. Only one application can access the agent platform.

These two limitations do not apply to all possible usages of software agents on Android devices. However, they restrict some interesting opportunities to utilize the JADE agent platform. In the case of the Travel Support System, the first limitation is the problem that has to be resolved. One of the unique features introduced in the Travel Support System is the proactive behaviour. The agent (or a set of agents) has to monitor the current user context and take proactive actions accordingly. For example, if the user receives a new text message, the system can create a new recommendation based on its content (see the second use case scenario in section 4). Since it is *impossible* to predict when the agent main container is killed to free additional resources, this feature can be restricted by other applications running on the Android system, which contradicts the usage of the software agents as autonomous and independent software components.

The second limitation is not so important from the perspective of the considered system (there are no other applications utilizing software agents). However, a closer look at this problem can provide some benefits for the further development of software agents on Android devices. Thus, the next section presents a detailed description of the solutions to *both* limitations.

2.4.3. JADEAndroid – Proposed Solutions

The first issue can be easily resolved by methods available within the Android system. First of all, the JADEAndroid can replace the *bound* service with a *started* one. Since the started service cannot be stopped by the system (or, at least should be stopped last; only if such radical solution is absolutely necessary), the agent main container can operate as long as the host application needs to. For example, the agent main container can be created when the user starts the application. Then, it can remain running even if the application is in the background. Finally, the main agent container can be destroyed when the user deliberately closes the application. It is also worth mentioning that the JADEAndroid source code contains an error concerning started services. Namely, the override *onStartCommand* method throws an exception which indicates that a service, which implements this method, can be accessed globally. This is not true since the JADEAndroid does not provide an *AIDL* interface, which allows interprocess communication. According to the official Android documentation, the *onStartCommand* method is required to properly manage the life-cycle of the started service. Secondly, the JADE service can be run in the foreground. By setting the foreground flag in the service, the developers notify the system that killing it would be disruptive to the user. Thus, the system increases the priority of the service. The service is required to provide an ongoing notification, which is displayed as long as the service remains in the foreground.

The second limitation is composed of two parts – (1) a service has to be uniquely identified, and (2) the agent main container has to mange multiple agents from different applications. The first part of the problem prevents the JADEAndroid to be used by two or more applications at the same time. To overcome this limitation, the developer can simply override the existing service in such a way that the application will call it as a local application component. Thus, the new service, unlike the service provided by the JADEAndroid, will be uniquely identified, since it belongs to the application name-space. To address the second part of the problem, several different approaches have been tested:

The agent Main Container runs as a stand-alone application By running the agent Main Container as a stand-alone application, the system can easily provide global access

to a single JADE agent platform. In this approach, each agent-based application creates its own agent container connected to the main agent container. Since all the agents are connected to the same agent platform, they can easily interact with each other. However, this approach requires the user to run an additional application, which creates the agent main container and, therefore, an extra amount of work has to be put to develop such an application. Moreover, some developers can find it not user-friendly, since the user has to install the JADEAndroid in the system and then remember to run it before other agent-based applications will be launched. Therefore, this approach more likely fits the PC architecture than the simplified life-cycle of the mobile application.

Each application provides its own agent Main Container In this approach, all agent-based applications implement their own service responsible for creating the agent main container. Each agent main container is identify by the same IP address and a subsequent port number (starting from the default port number). Thus, all agent-based applications are independent of each other. The drawbacks of this approach are (a) the impeded communication between agents belonging to different applications, and (b) the amount of resources required to run several main agent containers.

All the above considerations can be summarized into one conclusion, that there are many ways to improve the JADEAndroid, but the decision which one is the best can differ with respect to the purpose of MAS that has to be designed. For example, if an agent-based application does not need to communicate with other software agents, but it has to work 24/7 without an internet connection (consider an application that monitors vital parameters of an elderly person, communicates only with sensors via a Blue-tooth connection, and its only purpose is to notify an emergency service by sending a text message or calling the emergency number in case when something goes wrong), then the best solution would be to implement its own main agent container. On the other hand, if an application can only work when it is connected to the internet and it uses a lot of battery power for repeated calculations, then the existing split-execution mode can be used. The next section contains the detailed implementation of the JADEAndroid plug-in utilized by the Travel Support System.

2.4.4. Proposed implementation of the JADEAndroid in the Travel Support System

For the purpose of this thesis, the JADEAndroid plug-in has been extended on the basis of the following considerations:

1. The Travel Support System does not communicate with other software agents on the same device. Thus, there is no need to share the agent main container with other applications.

2. An agent (or a set of agents), which operate(s) inside the Travel Support System is continuously monitoring the user's contextual information. Even if the Android system is not connected to the Internet, the Travel Support System should schedule delivery of recommendations and display them to the user as soon as the Internet connection is resumed.

Therefore, the Travel Support System should operate in the stand-alone execution mode with its own main agent container.

The enhanced version of the JADEAndroid plug-in also contains several improvements, which do not concern directly the main subject of our work, but significantly facilitate the use of software agents. Here is the list of the most noticeable changes:

Some methods have been redesigned to be more intuitive RuntimeService, unlike MicroruntimeService used in the split-container execution mode, does not provide simplified versions of some methods.

The service stores references to the object-two-agent (*O2A*) interfaces This change was proposed to address the agent synchronization issue and the activity life-cycle (when an activity is killed, all its resources are released, including the *O2A* interfaces).

The class diagram of the JadeService is presented in Fig. 2.3 . The class is uniquely identified within the application namespace by its canonical name (the package name and the class name), *travel.support.service.JadeService*. In the JADEAndroid, the service is always identified by the same canonical name, *jade.android.RuntimeService*. Thus, if two or more applications request binding to the RuntimeService at the same time, the service is destroyed and recreated for the last calling application. This does not hold for our service, since its canonical name is unique.

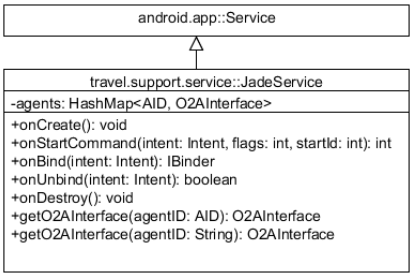

Figure 2.3: The class diagram of travel.support.service.JadeService

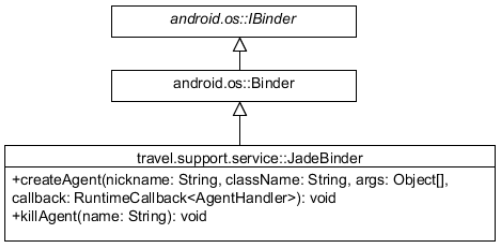

Figure 2.4: The class diagram of travel.support.service.JadeBinder

The JadeService class provides a field called agents, which is a type of *HashMap¡AID, O2AInterface¿*. This field stores all *O2A* interfaces (interfaces to communicate with agents) listed by agent IDs (*AID*). This change especially addresses the life-cycle of activities. Each time, an activity is destroyed and recreated, it has to obtain all needed *O2A* interfaces. Since, all the agents in the Travel Support System monitor the user's contextual information, it can be a common situation that they will live longer than the activities which utilize them. Unfortunately, in the stand-alone execution mode, there are no such methods which can be used to determine if an agent is alive and to obtain its O2A interface. Therefore, the JadeService class provides such functionality by monitoring events concerning the JADE agent life-cycle.

The JadeService class implements all the previously described methods required by a started service:

onCreate – JadeService is created. The service initiates the agent main container and registers all listeners. In this method, the service is also started in the foreground. This means that an ongoing notification is registered in the Android system.

onStartCommand – Unlike RuntimeService available in the JADEAndroid plug-in, JadeService implements the onStartCommand method. Since the service is started, this method is required to properly manage its life-cycle.

onBind and *onUnbind* – All activities in the Travel Support System can bind to JadeService. The *onBind* method provides an instance of the IBinder interface, which is used to manage the agent main container.

onDestroy – JadeService is destroyed and the agent main container is terminated.

The class diagram of the IBinder interface provided by the JadeService is available in Fig. 2.4. JadeBinder extends the Binder class, which provides a basic implementation of the IBinder interface. Since the Travel Support System does not require running multiple agent containers, the JadeBinder provides only two simple methods:

- *createAgent* – create an agent in a "default" agent container. The agent is also activated within this process.
- *killAgent* – an agent is terminated and all its resources are released.

It is assumed that the JadeService creates only one agent container as a "default" container, so there is no need to specify this in the *createAgent* method.

Chapter 3

The Recommender System

The central component of the Travel Support System is the recommender system [48]. This is a good decision since the recommender system proved to be useful in the item-to-user recommendations. In the case of the Travel Support System, a real word location can be seen as an item with a set of properties, which the user may or may not like.

The detailed description of the recommender system used in the new implementation of the Travel Support System is presented in the following sections.

3.1. Introduction to the Recommender Systems

Recommender systems are a subclass of information filtering systems, dedicated to predicting user preferences and determining items/places they may be interested in. Predictions are calculated, based on the history of user activity in the system such as browsing, searches, purchases, and so on.

The idea of the recommender system came directly from the assumptions of the information filtering: user has relatively stable needs, but content (the number of possible choices) is increasing over time. A new content is passed through the user profile (used as a filter) to obtain the outcome.

However, the old recommender systems did not work efficiently when the total size of the content was large (for example, the number of articles on the internet concerning a particular subject). Due to this limitation, researchers proposed manual collaborative filtering (users recommend items to each other), followed by automated collaborative filtering (the system recommends items to users). This extended the initial idea (keywords and topics) by more complex specifications: quality and taste.

The first automated collaborative filtering system was proposed in 1994 by the GroupLensProject [17] for the Usernet News. The tests made on the GroupLens system showed that the automated collaborative filtering worked and the participants found it very useful. Some of them kept using the test software after the end of the test.

Nowadays, recommender systems are widely used in filtering interfaces (email filters), recommendation interfaces (suggestion lists, promotions and offers) and prediction interfaces (predicted ratings). There are several different types of recommender systems:

Purely Editorial Recommender Recommendations are made by users to other users. Algorithms are not involved in the process.

Content Filtering Recommender Recommendations are made by the system based on user profile.

Collaborative Filtering Recommender Recommendations are made based on several "nearest" neighbours of the current user profile.

Hybrid Recommender Recommendations are made by utilizing different technologies in one system.

3.2. Common Pitfalls concerning Recommender System

Despite the vast research done on the filed of recommender systems, some people can still make common mistakes. The list of such pitfalls is presented next:

Collecting user opinions It is necessary to determine a set of user actions, which will be taken into account. The most simple and direct approach is to provide a rating system within the user interface. However, it was empirically proved that the indirect actions such as browsing, searching and purchasing are also a valid source of information since users frequently do not rate all the items they were interested in. Moreover, most of users do not re-validate the "once rated" items after some time (for example, when their preferences have changed). Due to this, some user ratings can become obsolete after some time.

Making the best recommendation Recommender systems should produce "the best possible" recommendation. However, they are supposed to avoid the most obvious recommendations (for example, recommending an item, which is on the top of user's list of favourite items and will be bought regardless of the recommendation). A perfect recommender system should recommend an item, which is interesting to a user and (without a recommendation) would be missed.

3.3. The algorithm proposed for the Travel Support System

The Travel Support System utilizes the Rhee-Ganzha algorithm [47] for measuring semantic closeness of ontologically demarcated resources (or simply the Rhee-Ganzha algorithm). The decision to use this particular algorithm was made based on a number of factors:

- Nowadays, the recommender systems favour algorithms based on very large user-to-product or user-to-user matrices. The idea of finding k-nearest elements in an ontology is not so distinct from the current trend, but also provides something new. Thus, it is more interesting to take the risk and evaluate this new algorithm rather than utilize an old one.
- Due to the fact that the Travel Support System is a multi-agent system, the algorithm will easily integrate with the ontological representation of knowledge inside the system.
- Despite the fact that the algorithm was proposed in 2009, its performance was never evaluated on a realistic data. The implementation of the algorithm used within the"Agents

in Personalized Information Provisioning" (APIP) [30] project is not consistent with its description.

General description of the algorithm

The algorithm was proposed based on the observation that, in any knowledge space, information resources do not exist in isolation from each other, but are connected through multiple relations. The relations can be explicit or inferred. Thus, the existence, the strength and the number of relations between two resources can determine a degree to which they are semantically similar (or semantically close).

In context of our previous discussion concerning recommender systems, semantically close resources can be used to create personal recommendations. Namely, if the system knows that the user likes a resource A, then it can ask the algorithm for a resource B which is the closest with respect to the semantic closeness. Moreover, the proposed algorithm can determine the level of semantic closeness between two entities, which have completely different nature. For example, a restaurant and a club can have smoking areas and free wifi connections. The system can use such information to create an alternative recommendation, which does not match all the user preferences, but is not completely contradictory to them.

Assumptions concerning the relevance calculation

The relevance calculation is based on the following three basic assumptions:

1. Having more relations from one object to another means that they are closer (more relevant).
2. Each relation has different importance depending on the type of the relation.
3. Even if the relations are of the same type, the weight of the connection can vary between individual objects (instances).

Since these assumptions are often referenced in the remaining parts of the thesis, it is important to understand them. The first assumption is very intuitive. For example, let us consider three restaurants – A, B and C. All three of them are located in the same city. However, restaurants A and B, unlike restaurant C, do not allow one to smoke (or do not provide a designated smoking area outside). Therefore, it is intuitive that restaurants A and B are semantically closer to each other than to the restaurant C (which has tables outside, where smoking is allowed).

The second assumption means that relations are weighted according to their semantic importance. In the previous example, restaurants can have a relation indicating if a customer is permitted to smoke in the restaurant. Let us also consider another relation indicating a type of cuisine. In general, we can assume that the type of cuisine is more important than the presence of a smoking area. For example, a smoking tourist will probably choose a restaurant serving his favourite dish, but without a smoking area, rather than a restaurant, which serves the food that he does not like, but permits to smoke. Of course, the tourist can do the opposite and go the second restaurant where he is permitted to smoke (e.g. when he is a hungry chain smoker), but this illustrative example shows clearly that some relations are more important than other. Thus, the system should have means to determine the strength of a relation.

The third assumption states that two individual objects of the same type can have different weights. This problem can be illustrated by the following observation: a famous Chinese restaurant and a low-quality Chinese fast-food serve meals demarcated as "Chinese cuisine." However, the quality of the meals in these restaurants differs significantly. Note that the cuisine relation has the same importance for all restaurants. Thus, the system requires an additional weight, which describes user preferences with respect to an individual information object (e.g. one tourist may not care what is the "status" of the restaurant, while another may not want to eat in a Chinese-fast-food).

General description of the relevance calculation process

Following the proposal described in [47], the core component of the algorithm is a matching engine, which is responsible for finding semantic closeness represented as a single number. In order to use the matching engine, a matching criterion has to be specified. The matching criterion is defined as an ordered quadruple $< x, q, a, g >$, where:

x is a source object

q is a query which defines a subset of objects that are considered potentially relevant; these objects will be matched against the source object x

$a \geq 0$ specifies the threshold of closeness between objects to be judged actually relevant to each other

g is a sub-query which is used to optimize the matching process by reducing the number of considered nodes

The relevance calculation process is composed of two stages (1) Graph Generation, and (2) Relevance Calculation. In the first stage, an ontology is used to create a graph, which is used by the algorithm to find all direct and indirect relations between information objects. The second stage requires the graph corresponding to the ontology, the source object and a set of target objects. In this stage, the algorithm calculates relevance values between the source and each target object. Let us discus each stage in details.

Graph Generation A graph generated in the first stage is called a Relevance Graph. A Relevance Graph is a directed label graph $G = (V, E)$, where

- V is a set of nodes (a set of individual information objects)
- E is a set of edges (a set of relations)

It is worth to notice that the structure of the graph does not restrict the number of edges between two adjacent nodes, so if two information objects are connected by multiple relations, the two corresponding nodes in the relevance graph will also be connected by the same number of edges. Moreover, the graph can also contain cycles.

To avoid high computational costs, the graph generation process has to be carefully planned and optimized. Let us consider the node and edge generation methods used within the Travel Support System.

A naive approach for node generation is when all individuals in an ontology become relevance graph nodes. However, the naive approach requires a huge amount of resources to process large ontologies. Moreover, this will also affect further calculations concerning semantic closeness because the complexity of the calculations depends on the number

of nodes and edges. However, not all nodes are relevant in every context. For example, a relevance graph can only include restaurants, which are within a certain distance form the user.

To optimize the node generation method, in the Travel Support System, a relevance graph is built only from the information objects, which are relevant with respect to the contextual information. This means that the algorithm processes only a small part of the ontology. Specifically, in the current approach, the algorithm does not need to include information objects exceeding a certain distance from the user. For example, a maximum distance can be set to 2 km for walking and 50 km for a car drive.

Edges in the relevance graph represent relations between nodes and are generated from relations between information objects from an ontology. Each edge can be defined as $e \in E = (x, y, distance, weight)$, where

- $x \in V$ is the tail node of the edge e
- $y \in V$ is the head node of the edge e
- $distance \in N$ is the conceptual distance value (the second assumption)
- $0 \leq weight \leq 1 \in R$ is the individual level weight value (the third assumption)

The distance between two adjacent nodes can be expressed via a formula

$$Distance = \frac{1}{Relevance} \tag{3.1}$$

and, in practice, all distance values can be initialized to a single value (for example, distance = 1). For non-adjacent nodes, the distance value has to be calculated by an equation presented later in this section. Since the description of the algorithm does not provide an automated method to update these values, they have to be manually modified by the ontology developer or a domain expert. On the other hand, weights of particular information objects are determined by interactions between the system and the user. A more detailed description of how they are calculated is presented in the section concerning the user profile.

Relevance Calculations The relevance calculations start from optimizing edges in the relevance graph. Here, we distinguish two important operations: edge scaling and edge merging. Edge scaling individualizes each edge's distance by multiplying its relevance value by its weight. This operation is a consequence of the third assumption. Equation 3.2 presents a formula to calculate a scaled distance for an edge $e = (x, y, Distance, weight) \in E$.

$$\begin{aligned} NewRelevance &= Relevance \times weight \\ NewDistance &= \frac{1}{NewRelevance} \end{aligned} \tag{3.2}$$

Since the structure of the relevance graph is not restricted, there can be multiple edges between two adjacent nodes. To simplify the relevance calculation and preserve the cumulative strength of connections, the algorithm applies the edge merging operation. For two adjacent nodes $x, y \in V$, and edges $e_1(x, y, Distance_1), \ldots, e_n(x, y, Distance_n) \in E$, the operation creates a merged edge $e' = (x, y, NewDistance)$ (it is assumed that,

in this step, all the edges are scaled by the edge scaling operation, thus the weights are omitted). The formula is defined as follows:

$$NewDistance = (\sum_{i=1}^{n} \frac{1}{Distance_i})^{-1} \tag{3.3}$$

Note that the calculated distance represents semantic closeness of adjacent nodes.

Let us now consider the relevance between non-adjacent nodes. First of all, the algorithm tries to find all valid paths between two non-adjacent nodes (the source node and one of the target nodes). A path is valid if and only if it is a simple path (namely, it does not contains repeating nodes). If there is no valid path between two non-adjacent nodes, it means that they are not related, hence their relevance is equal to 0 (the distance is equal to infinity). If there exists one or many valid paths, first, the algorithm calculates distances for each of them. In graph theory, the path weight in a weighted graph is the sum of the weights of edges in the path. Equation 3.4 shows the formula for a simple path, where denotes the distance between two adjacent nodes and

$$Distance_P = \sum_{i=1}^{n-1} Distance_{a_i a_{i+1}}. \tag{3.4}$$

However, by applying equation 3.4, the algorithm can produce undesirably close results for two non-adjacent nodes with multiple connections. For example, consider nodes x, y and z, where x and y are connected via medium-strength direct relations, and x and z are connected via multiple long paths. Thus, the algorithm should return y. Unfortunately, based on equation 3.4, it can return z if there exists a large number of very long paths between x and z. From the example, we can derive two important adjustments which have to be applied to the formula: 1) direct relations should be stronger than indirect relations and 2) the length of a path makes the corresponding indirect relation weaker. Therefore, equation 3.4 can be redefined as follows

$$Distance_P = \sum_{i=1}^{n-1} i \times Distance_{a_i a_{i+1}} \tag{3.5}$$

By applying equation 3.5 in the calculation, distances of long paths are weaker, but it is still possible to overcome direct relations. Thus the algorithm becomes more intuitive.

In the last step, the algorithm calculates the cumulative influence of all paths between the source and the target node. This operation is very similar to edge merging. For n paths $P_1, P_2, \ldots, P_n$ from $x \in V$ to $y \in V$ the relevance value $Relevance_{xy}$ is defined as follows

$$Relevance_x y = \sum_{i=1}^{k} \frac{1}{Distance_{P_k}} \tag{3.6}$$

3.3.1. Additional algorithms and data structures

The critical part of the algorithm is to determine all simple paths between two nodes. Unfortunately, the description of the algorithm does not suggest how this task can be accomplished. Therefore, the decision we made to implement a new implementation of Yen's algorithm [49] for finding k-shortest simple paths.

Fibonacci heap

The Fibonacci heap used in the Matching Engine is implemented according to [39]. A Fibonacci heap is a collection of item-disjoint heap-ordered trees. A heap-ordered tree is a rooted tree containing a set of items arranged in heap order. Namely, if x is a node, then its key is no less than the key of the parent (provided that x has a parent). Thus, the root contains an item with the smallest key. There is no explicit constraint on the number or structure of the trees. However, the number of children of a node represents its rank. The rank of a node with n descendants is $O(\log n)$. The heap-ordered trees can perform an operation called linking. The linking operation combines two item-disjoint trees into one.

The heap is accessed by a pointer to its root, which contains an item with the smallest key. We can also call this root the minimum node. If the minimum node is undefined, the heap is empty. Each node contains a pointer to its parent, another pointer to one of its children and its rank. The children of each node are doubly linked in a circular list. This helps the heap maintain the low cost of its operations. For example, the double linking between a root and its children makes removing elements possible in O(1). Similarly, the circular linking between children makes concatenation of two such lists possible in O(1).

Since a Fibonacci heap will be used by Yen's [49] and Dijkstra's [38] algorithms as a priority queue, the most important operations are the *delete_min* operation and the *decrease_key* operation. The *decrease_key* operation starts by subtracting a given number from the key of an item i in a heap h. Secondly, the algorithm finds a node x containing i and cuts the edge joining x to its parent (it also decreases the rank of the parent). As a result, the algorithm creates a new sub-tree rooted at x. If the new key of i is smaller than the key of the minimum node then the algorithm redefines the minimum node to be x. The complexity of the decrease_key operation is O(1) (this operation assumes that the position of i in h is known).

The *delete* operation is similar to the *decrease_key* operation. First, the algorithm finds the node x containing i and cuts the edge with its parent node. Second, it concatenates the list of children of x with the list of roots and destroys x. The delete operation complexity is O(1) (this operation assumes that the position of i in h is known).

The *delete_min* operation requires finding pairs of tree roots of the same rank to link. This is achieved by using an array indexed by ranks. After deleting the minimum node and forming a list of new tree roots, the algorithm inserts the roots one by one into the array. If the root is inserted into an array position which is occupied, then the algorithm performs the linking operation on the roots in conflict. After successful linking, the resulting tree is inserted to the next higher position. The *delete_min* operation ends when all the roots are stored in the array and its amortized time is equal to $O(\log n)$.

Dijkstra's algorithm

Dijkstra's algorithm [38] is a graph search algorithm that solves the single-source shortest path problem for a graph with non-negative edge path weights. It is also used to produce a shortest path tree. For a given source node in the graph, the algorithm finds a path with the lowest cost (the shortest path).

To optimize Dijkstra's algorithm, the implementation uses a Fibonacci heap as a priority queue. This improves the time complexity of the algorithm to $O(|E| + |V| \log |V|)$, where E is a set of edges,V is a set of vertices and |.| denotes the size of a set.

The algorithm proceeds as follows:

Djikstra's algorithm

Constructs an array of nodes which can be used to reconstruct the shortest path

Input: a graph G, the source and the target

Output: an array of nodes which can be used to reconstruct the shortest path

for each vertex v in the graph G **do**
distance from the source to v (denoted by $dist[v]$) is equal to infinity
the previous node in the optimal path from the source to v (denoted by $previous[v]$) is undefined
end for
$dist[source]$ is equal to 0
initialize a queue Q, which contains all the nodes
while Q is not empty **do**
let u be a vertex in Q with the smallest distance from the source
remove u from Q
if $dist[u]$ is equal to infinity **then**
break
end if
for each neighbour v of u **do**
calculate an alternative distance $alt = dist[u] + distance(u, v)$
if alt is smaller than $dist[v]$ **then**
$dist[v] = alt$
$previous[v] = u$
$reordervinQ$
end if
end for
end while
return $previous$

The shortest path can be reconstructed by executing the following algorithm:

Reconstruct

Reconstructs the shortest path from an array of nodes

Input: the previous array of nodes constructed by Dijkstra's algorithm, the target node

Output: a sequence containing the shortest path

let S be an empty sequence
u is equal to the target

while *previous*[u] is defined **do**
  insert u at the beginning of S
  $u = previous[u]$
end while
return S

Yen's algorithm

Yen's algorithm [49] is a deviation algorithm that finds K single-source, shortest, loopless paths in a graph with non-negative edge cost. Let us consider the k-th shortest path $p_k = \{{v_1}^k, {v_2}^k, \ldots, {v_n}^k\}$. In order to find the $(k+1)$-th shortest path, the algorithm analyses every node ${v_i}^k$ in p_k and computes the shortest loopless path p which deviates from p_k at this node. Loopless path p_k is said to be a parent of p and ${v_i}^k$ is its deviation node. To avoid loops during the calculation of a new path, the algorithm should remove all sub-paths between the source node and a deviation node ${v_i}^k$. Therefore, all the nodes are temporarily removed and the algorithm calculates a new path in the resulting graph.

For this thesis, the algorithm utilizes a new implement of Yen's algorithm described in [44]. The new implementation of Yen's algorithm was proved to be generally be more efficient than other implementations (the publication covers the straightforward implementation and the implementation proposed by Perko). Specifically, the new implementation replaces node deletion with its reinsertion. This allows to solve the problem faster by labelling and correcting labels of some nodes. In general, the number of corrected nodes is assumed to be smaller than the total number of nodes in a graph. The tests performed by the authors of the article seem to prove this hypothesis. However, in the worst case scenario, the algorithm performs its computation in $O(Kn(m+n\log n))$, where K is the number of paths, n is the number of nodes and m is the number of edges. This time complexity is also the worst-case time complexity of the straightforward implementation (assuming that Dijkstra's algorithm is used for finding the shortest path).

Let us start describing the new implementation by defining basic concepts. A shortest path tree rooted at a vertex x is a spanning tree T_x of a graph G. If x is the terminal node, T_x represents the tree of the shortest paths from every node to x. Otherwise, T_x represents the tree of the shortest paths from x to every node. A loopless path from $v \in V$ to x in T_x is denoted by $T_x(v)$. The cost of $T_x(v)$ is denoted by π_v and is also used as a label of v. To generate a shortest path tree and to find the shortest path in a graph G, Yen's algorithm uses Dijkstra's algorithm.

Yen's Algorithm

Input: a graph G with the source and the target nodes, an integer k

Output: k the shortest paths
  p = shortest path from the source to the target nodes in the graph G
  $d(p) = s$
  $X = \{p\}$
  $k = 0$
  while $X \neq \emptyset$ and $k < K$ **do**
    $k = k + 1$
    p_k = shortest, loopless path in G
    $X = X \setminus \{p_k\}$

$\pi_v = \infty$, for any $v \in V$
remove loopless path p_k, except target node t, from the graph G
remove edges $(d(p_k), v), v \in V$ of $p_1, p_2, ..., p_k$
T_t = shortest tree rooted at t
for $v_i^k \in \{v_{l_k}^k, ..., d(p_k)\}$ **do**
restore node v_i^k in the graph
calculate $\pi_{v_i^k}$ (label) using forward star form
if $\pi_{v_i^k}$ is defined **then**
correct labels of v_i^k successors using backward star form
$p = sub(s, v_i^k) \cdot T_t(v_i^k)$
$d(p) = v_i^k$
$X = X \cup \{p\}$
end if
restore (v_i^k,v_{i+1}^k) in the graph
if $\pi_{v_i^k} > \pi_{v_{i+1}^k} + cost_{v_i^k, v_{i+1}^k}$ **then**
$\pi_{v_i^k} = \pi_{v_{i+1}^k} + cost_{v_i^k, v_{i+1}^k}$
correct labels of v_i^k successors using backward star form
end if
end for
Restore the graph G
end while

Forward star form

Correct labels of x successors

Input: a graph G
for edge $e = (i, j) \in E$ **do**
if $\pi_i > \pi_j + cost_{i,j}$ **then**
$\pi_i = \pi_j + cost_{i,j}$
end if
end for

Backward star form

Correct labels of x successors

Input: a graph G and a vertex
$list = \{x\}$
repeat
i = element of $list$
$list = list \setminus \{i\}$
for edge $e = (i, j) \in E$ **do**
if $\pi_i > \pi_j + cost_{i,j}$ **then**
$\pi_i = \pi_j + cost_{i,j}$
$list = list \cap \{j\}$
end if
end for
until $list \neq \emptyset$

3.4. Implementation of the Rhee-Ganzha algorithm

This section summarizes and concludes the above discussion concerning the use of the Rhee-Ganzha algorithm in the Travel Support System.

3.4.1. RDF Ontology

The Semantic Web [28] (also referred to as a "web of data") is a collaborative movement led by the World Wide Web Consortium (W3C) [35], The main idea is to provide a common way to store data on the Internet by using data formats such as RDF [27].

The data stored inside the Semantic Web can be easily processed directly or indirectly by the machines. Since the Semantic Web is beneficial for usage of agent-related technologies (agents' autonomy, accessibility to the information, unified way of communication with the environment), the Travel Support System provides data compliant with the RDF specifications.

An ontology, in computer science or information science, is a formal description of knowledge as a set of concepts within a domain [45]. Two concepts are connected to each other by one or many relations. In an agent system, an ontology is used to model a domain and represents things that can exist for the agents. In the case of this thesis, the ontology describes concepts within the "Travel" domain (such as Hotel, Room, Restaurant) and relations between them (for example "Hotel has Room"). Ontologies used within the Travel Support System are stored in the RDF format.

The Resource Description Framework (RDF) is a family of World Wide Web Consortium (W3C) specifications. RDF is used to represent information about resources in the form of a graph. In text form, RDF is represented by a set of semantic triples. Each triple contains a subject, a predicate and an object.

Resources in RDF are defined by the Uniform Resource Identifier (URI) or the Internationalized Resource Identifier (IRI). URI is a superset of the URL. In comparison to the URL, the URI can identify any resource, not only network documents. The IRI is an extension of the URI that allows to use international (Unicode) characters. Typically, the URI or the IRI may look like http://www.example.com/ontology/ontology.rdf# individual.

3.4.2. Implementation of the Matching Engine

The Figure 3.1 presents the component diagram of the Matching Engine. The Matching Engine is composed of 6 components: Graph Generator, Filter, Matching Engine, Yen's algorithm, Dijkstra's algorithm, and Fibonacci heap. The Graph Generator takes an ontology as an input. Then, the ontology is filtered by the Filter. The Filter selects the information objects, which satisfy the Matching Criteria. For example, it will return all hotels within a 10 km range. The selected nodes are returned to the Graph Generator and are used to create a relevance graph. The relevance graph is required by the Matching Engine. The Matching Engine, based on the graph and the Matching Criteria, creates an appropriate request for the Yen's algorithm. The Yen's algorithm is responsible for finding all simple paths within the graph between the source node and all target nodes. To do so, the Yen's algorithm interacts with two other components — the Dijkstra's algorithm and the Fibonacci heap. The Dijkstra's algorithm performs operations listed in the description of Yen's algorithm (see, Section

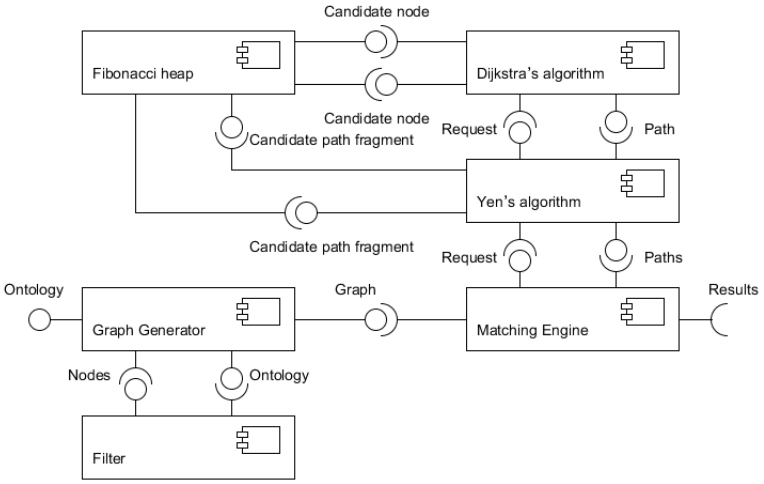

Figure 3.1: The component diagram of the Matching Engine

3.3.1). Typically, the algorithm returns a simple path (or its fragment), which can be used to create another simple path in the next iteration. Both Yen's and Dijkstra's algorithms use the Fibonacci heap as a priority queue for nodes or path fragments respectively. In the final stage, all simple paths are sent to the Matching Engine for merging and selection of the closest information object with respect to the source (if, after merging, the distance from the source is smaller than the current one, then this information object becomes the closest).

Chapter 4

Implementation of the Travel Support System

The Travel Support System is implemented using the following additional software artifacts:

JENA Apache Jena [8] is a JAVA framework for building Semantic Web [28] applications. The Jena framework is capable of:

- reading, processing and writing RDF
- storing RDF triples efficiently (storage file or database)
- SPARQL queries
- publishing RDF data

Jena will be used to provide semantic data into the system.

Android SDK 4.2 The Android SDK [5] provides the API library and developer tools necessary to build, test and debug applications for Android OS.

Java Virtual Machine (Java SE 1.7 update 7) The Java Virtual Machine is a platform used to run all Java-based programs. Since JADE is written in the JAVA programming language, it is necessary to use it during the implementation and deployment of the system. Moreover, the system takes the advantage of the portability of JAVA.

PLAY framework 2.0.4 The Play framework [25] is a lightweight, stateless Web framework written in JAVA and Scala. The Play framework will be responsible for maintaining communication between the recommender system (one or a set of agents available as a web service) and personal agents.

4.1. User profile

A user profile is an organized structure that collects all user-related information. Due to the use of the Rhee-Ganzha algorithm, the user profile contains weights of the information objects which are used during the relevance computation.

The user profile is stored outside the system as a file. The file has an ordered structure, which allows the system to load it into memory and create an interface which is used to read and modify the represented user profile. Since the file is not a part of the system, it can be moved from one device to another. This also allows to make backups and synchronize the user profile with a remote server.

The user profile stores pairs of keywords and corresponding weights. Each keyword represents one information object from an ontology. For example, a pair

(`http://www.example.com/restaurant/restaurant.rdf#FastFood`, 0.4)

indicates that the information object identified by this URI has a weight equal to 0.4. However, not all information objects are stored in the user profile. The system based on interaction with the user chooses the most frequent objects and modifies their weights. If the weight of an object is modified then it is stored in the user profile. If the system requires a weight that is not present in the user profile, the default value is returned.

To model changes in user's behaviour, weights decrease over time. The decreasing factor decreases a weight to the default value. If the default value is reached, then the weight and the corresponding key are removed (the system will return the default value for non–existent entries in the user profile). However, not all user preferences should have the same value of the decreasing factor. For example, if the user likes fast food, but he is overweight and his doctor recommended him a diet, then the system should relatively quickly adjust his preferences according to his new habits (the decreasing factor is high). On the other hand, if the user loves action genre films, but, from time to time, he also sees a romantic comedy, then the system should remember that his favourite genre is action films even if he saw several romantic comedies in a row (the decreasing factor is low). Therefore, in the system, there are three stages which are modelled based on human types of memory: 1) sensor memory (very short), 2) short–term memory, and 3) long–term memory. Let us discuss how the decreasing factor changes in the system based on the proposed memory model.

Initially, information objects are assigned to sensor memory. In this stage, the decreasing factor is the largest, but information objects are also faster promoted to short–term memory. Sensor memory works like a buffer for occasional selections which does not indicate habits or patterns in the user behaviour. Short–term memory has an average decreasing factor. This stage covers most of the information objects. It is also a place where the system can find multiple alternatives for recommendations. For example, short-term memory can have several types of food and each type is a "good" recommendation. The final stage corresponds to long–term memory. Here, the decreasing factor is very low, but it contains only the most frequent information objects. In comparison to the other memory types, long–term memory stores user preferences which are almost unchangeable. For example, sensor memory can have information that the user went to a club with electronic music. Unfortunately, he did not like it, so the system will quickly forget about it. Otherwise, this information will be promoted to short–term memory and it will become one of many alternatives (for example, there are other types of music like rock and pop). However, in long term memory, the system can store information that the user does not smoke, so he always chooses places where smoking is not permitted.

Weights can also represent negative feelings (the user does not like something), but the system is not allowed to use negative numbers since all the algorithms require graphs with non-negative costs. This problem can be addressed by using an appropriate scale. Specifically, the default value for a weight can be set to 0.5 . This divides the interval (0;1) into two subintervals – (0;0.5) and (0.5;1). The system can use the first interval to express negative

feelings (weight equal to 0 means that the user strongly dislikes something). Similarly, the second interval can be used for "good" recommendations (weight equal to 1 means that the user strongly like something). The default value (here, a sample value 0.5) represents neutral feelings (the system does not know if the user likes something or not). To these two intervals, the system can apply the presented memory model. The weights from the interval (0;0.5) are obtained by analysing the structure of an ontology. For example, "smoking area" and "non–smoking area" are contradictory. Thus, the system can decrease the weight of "smoking area" below the default value if "non–smoking area" is chosen.

However, not all weights should be decreased. For example, if a search query concerns a restaurant ontology, then the system should not decrease weights in the golf ontology. To avoid such situations, the user profile is divided into sections according to the individual ontologies. Therefore, the system can decrease the weights in a specific category.

There are several strategies concerning updating weights

1. Weights can be updated based on all actions
2. Weights can be updated based on deliberate actions

The first strategy has a drawback concerning the weights of actions. Specifically, if we consider the select and the browse actions, we can agree that the first action is stronger than the second. However, the browse action can determine two different things; an item is either good (the item is not selected but feedback is positive) or bad (feedback is negative). Thus, this approach forces the user to rate each observed item. In the second strategy, interaction with the user is more flexible. Namely, instead of the browse action (which is done automatically when the user navigates through results), the system provides another action called deselect which always indicates negative feedback.

In this approach the user can select (weights are increased), deselect (weights are decreased) or do not nothing (weights remain the same). Unfortunately, there is a problem with the deselect action. The user can deselect a recommendation because of one of its properties. However, the system cannot easily determine, which property caused this action.

Taking the previous considerations into account, the system updates weights based on the following rules:

- In the system, the user gives positive feedback by selecting results. The selected result is assumed to be the best.
- The user is not able to give negative feedback, but negative weights are calculated based on the structure of an ontology

4.2. Reading a structure of an ontology

OntoReader is used to dynamically generate a user interface based on an ontology. It is a simplified version of QueryBuilder from the project called Agent in Grid [29]. There are several factors which justify the decision to reimplement this software artefact:

1. QueryBuilder was developed for the Play! Framework and its PC version is not available. Thus, the already existing implementation had to be slightly modified.
2. OntoReader uses only OWLAPI [24] (a JAVA API for creating, manipulating and serialising ontologies) to read an ontology where QueryBuilder depends also on JENA.

Because OntoReader is used on the device (not on the server), it is convenient to remove unnecessary dependencies.

3. OntoReader was implemented with the Travel Support System in mind. Specifically, OntoReader obtains only a brief structure of an ontology (typically a set of triples describing only one level of an ontology). Therefore, the system preserves battery power because only a small part of the ontology is analyzed at the given time.

During the implementation of OntoReader for the Android system, special attention had to be devoted to several problems:

- The Android system does not allow to send normal JAVA objects via Intents. Therefore, it was impossible to send a part of the ontology encapsulated within a JAVA class from a dedicated service.
- The device has limited memory. Thus, the ontology, especially a big one, should be loaded into system memory only once.

To deal with this problem, the Travel Support System implements the Parcelable interface [4]. The Parcelable interface provides a mechanism to write and read simple data types (for example String, Integer, Double, List) from a bundle, which can be sent via an Intent. The mechanism has two phases – writing and reading. In the writing phase, an object is decomposed into a set of simple types and the mechanism calls a method, which fills a bundle with pairs consisting of a key (String) and a value (simple data type). Then, the bundle is attached to an Intent. After receiving the Intent, an Activity has to obtain the Parcelable object for the attached bundle. The reconstruction method is automatically called by the Parcelable mechanism when the object is read. Both deconstruction and reconstruction methods have to be provided by the developer.

The OntoReader reads three ontological elements: Class (a concept), DataProperty (a relation between two concepts), and Individual (an instance of a concept). Each element is represent by a JAVA class – OntoClass, OntoDataProperty and OntoIndividual respectively. When the user interface is created, the system asks OntoReader for a parent Class (for example Restaurant). OntoReader returns the specific instance of OntoClass which also contains a list of all associated DataProperties. The user can navigate through the DataProperties and choose them one by one. If the user selects a DataProperty containing only Individuals, then the system creates a window which allows multiple selection (for example, the user can choose several different cuisines for DataProperty called hasCuisine). Otherwise, the system stores the current user selections and recreates the user interface for the given OntoClass. Similar operations are performed for all levels of the ontology.

4.3. Travel Support System – putting it all together

Figure 4.1 presents the component diagram of the Travel Support System. It is composed of two types of software agents; 1) Client Agents and 2) Server Agents. These two types of agents perform different (and disjoint) tasks and have to cooperate in order to make recommendations. The Client Agent runs on the Android system (the structure of a software agent for the Android system was described in the previous sections). First of all, the Client Agent manages the user profile on the device. During the start-up of the system, the user profile is loaded into the memory and the agent receives an I/O interface, through which it can read and write the user preferences. The Client Agent is also responsible for making

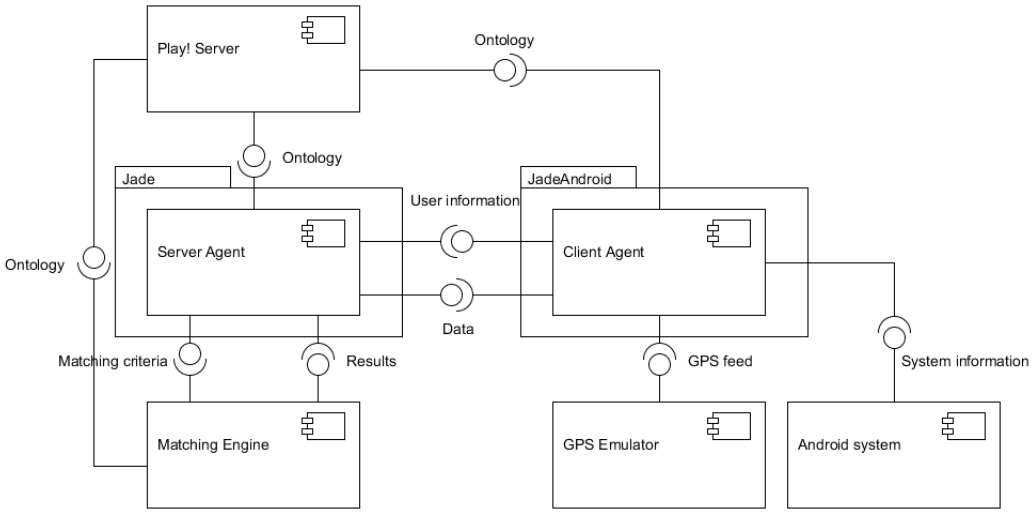

Figure 4.1: The component diagram of the Matching Engine

back-ups of the user profile and synchronizing it with the on-line server (the one which runs the Server Agent).

The Client Agent monitors the user context. Since it is a part of the Android system, the agent can read the private user data (the user is informed about it during the installation of the application, as an appropriate agreement is necessary to access data within an Android-based system) and act accordingly. The private user data includes text messages, entries in the calendar, contacts, and so on. Since the Client Agent is running on a real Android device, there is no native emulator that can be used to provide GPS feed. Therefore, the system uses its own emulator which send GPS coordinates to the Client Agent via IPv4 socket. The Client prioritizes the data sent by the emulator over data received from the Android system.

Instead of directly interacting with the Matching Engine, the Client Agent communicates with it via the Server Agent. During the communication, the Client Agent provides all information required to construct a Match Criteria. Namely, the Server Client receives a piece of the user profile corresponding to the search query and the user contextual information (GPS coordinates and the current time). The piece of the user profile contains all the (keyword, weight) pairs related to a specific topic (for example, restaurants). The Matching Criteria is constructed by the Server Agent. Then, the Server Agent sends it to the Matching Engine. Note that the Matching Engine is described in detail in section 3.4.2. The Matching Engine performs the relevance calculation and returns the results. The results are forwarded to the Client Agent who presents them to the user. Based on user selection, the user profile is modified according to the rules presented in section 4.1.

4.3.1. Matching Process Example

Let us discus in details the matching process based on finding a restaurant. In this example, the system uses the ontology from the old implementation of the Travel Support System **??**. In order to find a suitable recommendation, the Travel Support System has to perform the following steps:

1. Based on the user-related information provided by the Client Agent, a Matching Criteria $< x, q, a, g >$ is created by the Server Agent. Then, the Server Agent sends the Matching Criteria to the Matching Engine for further processing. For example, a Matching Criteria for a recommendation concerning restaurants can look as follows:

   (a) $x = UserRestaurant$

   (b) $q =$

   $PREFIXonto : \langle http : //localhost : 9000/assets/Ontologies/Restaurant/Restaurant\rangle$

   $SELECT?restaurant$

   $WHERE?restaurantisaonto : Restaurant.$

   (c) $a = \frac{3}{5}$

   (d) $g = [lat = 22.2,\ long = 44,\ dist = 2000]$:

   $lat =$ user's latitude

   $long =$ user's longitude

   $dist =$ the maximum distance (in meters) between the source (the user's current position) and a target object

   Note that the source object x, *UserRestaurant*, does not exist in the ontology. It is an information object created based on the user profile. For example, if a user profile contains pairs:

   (*cur:Polish*, 0.8)

   (*cur:Chinese*, 0.45)

   (*asc:SmokingArea*, 0.6)

   (*drs:Casual*, 0.3)

   (*rcc:FastFoodRestaurant*, 0.55)

   (*rcc:CasualRestaurant*, 0.7)

   then the information object corresponding to this user profile has the following relations:

   (a) *res:UserRestaurant res:cuisine cui:Polish*

   (b) *res:UserRestaurant res:cuisine cui:Chinese*

   (c) *res:UserRestaurant res:smoking acs:SmokingArea*

   (d) *res:UserRestaurant res:dress drs:Casual*

   (e) *res:UserRestaurant res:restaurantCategory rcc:FastFoodRestaurant*

   (f) *res:UserRestaurant res:restaurantCategory rcc:CasualRestaurant*

2. By executing the g and q queries, the *Matching Engine* determines a set of possibly relevant target objects. In our example, the Client Agent is interested in restaurants. The q query accepts all individuals which are instances of the *Restaurant* class. However, the q query does not determine which information objects are located within a certain distance from the user (here, a sample value $dist = 2000$). Therefore, the g query removes all places located further than a given distance. By combining these two queries,

the *Matching Engine* is able to obtain all restaurants located not further than 2 km from the user.

3. Finally, the *Matching Engine* calculates the semantic relevance between the source and all the target information objects found in the previous step. The matching process involves:

   (a) source instance URI = *UserRestaurant*

   (b) target objects URI's = [*PolskiePierogi*, *BarOrientalny*, *BarSmakosz*]

   (c) relevance threshold: $a = \frac{3}{5}$

   The detailed description of the relevance calculations performed by the system is presented in the next section.

4.3.2. Relevance Calculation Example

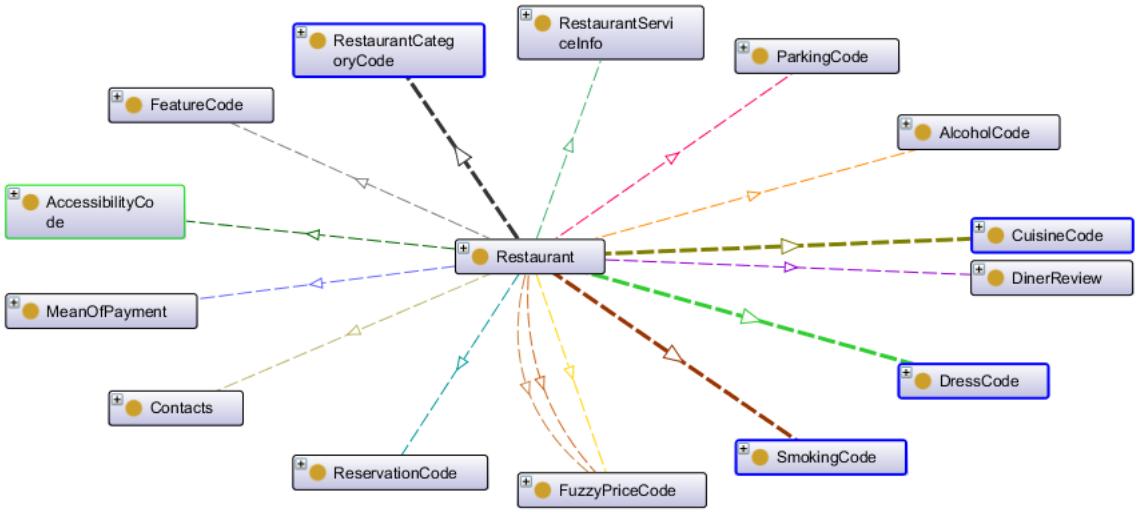

Figure 4.2: The Restaurant ontology from the old implementation of the Travel Support System

The relevance calculation starts from creating a *Relevance Graph* (described in section 3.3) which corresponds to an overview presented in figure 4.2. Since the resulting graph has no multiple relations between two adjacent information objects, the system skips the edge merging procedure. Now, let us recall that different information objects can have different individual weights stored in the user profile. For example, *Polish* and *Chinese* cuisines have their individual weights equal to 0.8 and 0.45 respectively. Therefore, the *Matching Engine* needs to scale the edges by applying equation 3.2. Without loss of generality, all the edges have their initial distances equal to a constant number $0 \leq d \leq 1$ (here, a sample value $d = 1$).

1. $distance_{Polish} = (\frac{1}{d_{cuisine}} \times w_{Polish})^{-1} = (1 \times 0.8)^{-}1 = 1.25$
2. $distance_{Chinese} = (\frac{1}{d_{cuisine}} \times w_{Chinese})^{-1} = (1 \times 0.45)^{-}1 = 2.2$

If the user profile does not specify the weight for an information object, then a default value is returned (for example, $w = 0.1$).

After creating the *Relevance Graph*, the *Matching Engine* calculates relevance values between the source (*UserRestaurant*) and the target objects (*PolskiePierogi*, *BarOrientalny* and *BarSmakosz*):

PolskiePierogi

The system utilizes the Yen and Dijkstra algorithms (described in section 3.3.1) to find all simple paths. Based on the *Relevance Graph*, we can define four paths from *UserRestaurant* to *PolskiePierogi*:

Path $1 : UserRestaurant \rightarrow Polish \rightarrow PolskiePierogi$

Path $2 : UserRestaurant \rightarrow SmokingArea \rightarrow PolskiePierogi$

Path $3 : UserRestaurant \rightarrow Casual \rightarrow PolskiePierogi$

Path $4 : UserRestaurant \rightarrow CasualRestaurant \rightarrow PolskiePierogi$

By applying equation 3.5, the respective distance values are:

$$D_{Path1} = 1.25 + (2 \times 1.25) = 3.75$$

$$D_{Path2} = 1.67 + (2 \times 1.67) = 5.01$$

$$D_{Path3} = 3.34 + (2 \times 3.34) = 10.02$$

$$D_{Path4} = 1.43 + (2 \times 1.43) = 4.29$$

Finally, the *Matching Engine* obtains the final relevance value by utilizing equation 3.6:

$$Rel_{PolskiePierogi} = \frac{1}{3.75} + \frac{1}{5.01} + \frac{1}{10.02} + \frac{1}{4.29} = 0.7992$$

BarOrientalny

All simple paths between *UserRestaurant* and *BarOrientalny*:

Path $1 : UserRestaurant \rightarrow Chinese \rightarrow BarOrientalny$

Path $2 : UserRestaurant \rightarrow SmokingArea \rightarrow BarOrientalny$

Path $3 : UserRestaurant \rightarrow Casual \rightarrow BarOrientalny$

Path $4 : UserRestaurant \rightarrow CasualRestaurant \rightarrow BarOrientalny$

By applying equation 3.5:

$$D_{Path1} = 2.23 + (2 \times 2.23) = 6.69$$

$$D_{Path2} = 1.67 + (2 \times 1.67) = 5.01$$

$$D_{Path3} = 3.34 + (2 \times 3.34) = 10.02$$

$$D_{Path4} = 1.43 + (2 \times 1.43) = 4.29$$

By applying equation 3.6:

$$Rel_{BarOrientalny} = \frac{1}{6.69} + \frac{1}{5.01} + \frac{1}{10.02} + \frac{1}{4.29} = 0.682$$

BarSmakosz

All simple paths between *UserRestaurant* and *BarSmakosz*:

Path $1 : UserRestaurant \rightarrow Polish \rightarrow BarSmakosz$

Path $2 : UserRestaurant \rightarrow Casual \rightarrow BarSmakosz$

Path $3 : UserRestaurant \rightarrow CasualRestaurant \rightarrow BarSmakosz$

By applying equation 3.5:

$$D_{Path1} = 2.23 + (2 \times 2.23) = 6.69$$

$$D_{Path2} = 3.34 + (2 \times 3.34) = 10.02$$

$$D_{Path3} = 1.43 + (2 \times 1.43) = 4.29$$

By applying equation 3.6:

$$Rel_{BarSmakosz} = \frac{1}{3.75} + \frac{1}{10.02} + \frac{1}{4.29} = 0.5996$$

Remarks concerning the relevance calculation

From the presented calculations, one can derive several noteworthy observations:

PolskiePierogi and BarOrientalny
Despite the fact that *PolskiePierogi* and *BarOrientalny* have the same number of paths which cover the same relations (*cuisine*, *smoking*, *dress* and *restaurantCategory*), the difference between their relevance values comes from the user preferences. Specifically, the user profile states that *Polish* cuisine is more likely to be chosen by the user than Chinese cuisine. Therefore, the *PolishPierogi* restaurant is more relevant to *UserRestaurant*.

BarOrientalny and BarSmakosz
BarOrientalny In comparison to *BarSmakosz*, *BarOrientalny* provides a smoking area. However, *BarOrientalny* serves Chinese food which is rated lower than the one served by the *BarSmakosz*. Therefore, the algorithm has to decide which factor is more important – permission to smoke or a type of cuisine. The algorithm takes advantage from analysing relations in the ontology and is able to determine that, the user can can choose a little less liked food in favour of smoking cigarettes.

Processing results

Based on these results and the proposed *Matching Criteria* ($a \geq \frac{3}{5}$), where a is the relevance threshold, *PolskiePierogi* and *BarOrientalny* will be recommended to the user. Because the relevance value of *BarSmakosz* is below the threshold, the *Matching Engine* will not include this recommendation in the results sent to the Server Agent. The recommendations will be ordered based on their relevance values.

Chapter 5

Test scenarios

To evaluate the performance of the Travel Support System, two different use case scenarios were executed.

5.1. Making proactive recommendations

One of the great features of the system is its ability to execute pro-active behaviours. This scenario tests this feature by periodically producing new recommendations for the user. For the purpose of this test, the time interval between searches initialized by the Client Agent without interacting with the user is set to a random number between 60 and 120 seconds.

The scenario proceeds as follows:

1. The user starts the Travel Support System on the Android device.
2. The user opens the *Emulator* and sets the time interval to 60–120 seconds.
3. After some time (no longer than 120 seconds) the Client Agent communicates with the Server Agent in order to obtain new recommendations. These recommendations are made based on a piece of the user profile sent by the Client Agent. The user is notified about this action by a system notification.
4. After clicking the notification, the user opens an Android activity with the results.

5.2. Context monitoring

This scenario tests the ability of the system to properly react to changing context. In this scenario, the *Emulator* is used to send a text message to the Client Agent. Then, the Client Agent analyses the content of the message and takes appropriate actions to provide the user some useful recommendations.

The scenario proceeds as follows:

1. The user starts the Travel Support System on the Android device.
2. The user opens the *Emulator* and send a text messages with the word "Chinese".

3. After receiving the message, the Client Agents starts to analyse the context in order to find some keywords.
4. The keyword recognized by the agent is "Chinese". Therefore, the Client Agent creates a new search query concerning the "Chinese" cuisine for the Travel Support System ontology. The user is notified about this action by a system notification.
5. After clicking the notification, the user opens an Android activity with the results.

5.3. Test results and summary

After successful execution of the two scenarios, we can conclude that the proposed system fulfils its requirements and is able to provide a great aid for travellers. The system takes advantage of the utilization of the newest technologies (for example, software agents on mobile devices, ontologies) and delivers functionality which overcomes the current limitation of other travel support systems (listed in section 1.1).

Moreover, by successful combining all mentioned technologies into one system, we have proved that there are some new ways to improved quality of existent solutions. Namely, the utilization of software agents and RDF/OWL ontologies can be beneficial and lead to discover new, non-trivial dependences between different resources. Hence, the system can produce more interesting and personalized recommendations. These technologies can be also available on mobile devices. This creates new, exciting possibilities to support users in more situations and gives them the opportunity to choose the most suitable device (for example, a PC at home and a mobile device on a street).

Mariusz Marek Mesjasz
Nr albumu 211615

Warszawa, 7 czerwca 2013

Oświadczenie

Oświadczam, że pracę magisterską pod tytułem „Travel Support System", której promotorem jest prof. dr hab. Marcin Paprzycki wykonałem samodzielnie, co poświadczam własnoręcznym podpisem.

.....
Mariusz Marek Mesjasz

Bibliografia

[1] Android Activity. http://developer.android.com/guide/components/activities.html.

[2] Android Intent. http://developer.android.com/guide/components/intents-filters.html.

[3] Android Interface Definition Language. http://developer.android.com/guide/components/aidl.html.

[4] Android Parcelable. http://developer.android.com/reference/android/os/Parcelable.html.

[5] Android SDK. http://developer.android.com/sdk/index.html.

[6] Android Service. http://developer.android.com/guide/components/services.html.

[7] Android system. http://www.android.com/.

[8] Apache Jena. http://jena.apache.org/.

[9] Apache license. http://www.apache.org/licenses/.

[10] Application Components. http://developer.android.com/guide/components/fundamentals.html.

[11] Booking.com. http://www.booking.com/.

[12] Eclipse IDE. http://www.eclipse.org/.

[13] Facebook. https://www.facebook.com/.

[14] Foundation for Intelligent Physical Agents. http://www.fipa.org/.

[15] Google Now. http://www.google.com/landing/now/.

[16] Google Play. https://play.google.com/store?hl=en.

[17] Group Lens. http://www.grouplens.org/.

[18] IDL. http://en.wikipedia.org/wiki/IDL_(programming_language).

[19] Java Agent DEvelopment Framework. http://jade.tilab.com/.

[20] Java Platform, Micro Edition. http://www.oracle.com/technetwork/java/javame/index.html.

[21] Linux. http://en.wikipedia.org/wiki/Linux.

[22] Open Directory Project. http://www.dmoz.org/.

[23] OUYA . http://www.ouya.tv/.

[24] OWL API. http://owlapi.sourceforge.net/.

[25] Play Framework. http://www.playframework.com/.

[26] Principle of least privilege. http://en.wikipedia.org/wiki/Principle_of_least_privilege.

[27] Resource Description Framework. http://www.w3.org/RDF/.

[28] Semantic Web. http://www.w3.org/standards/semanticweb/.

[29] Software Agents in Grid. http://www.ibspan.waw.pl/~paprzyck/mp/cvr/research/agents_GRID.html.

[30] Software Agents in Personalized Information Provisioning. http://www.ibspan.waw.pl/~paprzyck/mp/cvr/research/agent.html.

[31] Travel Support System publications. http://www.ibspan.waw.pl/~paprzyck/mp/cvr/research/agents_TSS.html.

[32] Travel Support System Sourceforge. http://e-travel.sourceforge.net/.

[33] TripAdvisor. http://www.tripadvisor.com/.

[34] Twitter. https://twitter.com/.

[35] W3C. http://www.w3.org/.

[36] Workflows and Agents Development Environment. http://jade.tilab.com/wade/index.html.

[37] Yelp! http://www.yelp.com/.

[38] Rivest Ronald L. Stein Clifford Cormen Thomas H., Leiserson Charles E. 2011.

[39] Tarjan R. E. Fredman M. L. *Fibonacci Heaps and Their Uses in Improved Network.* Journal of the ACM, 1987.

[40] Paprzycki M. Ganzha M. Gawinecki M., Kruszyk M. Pitfalls of agent system development on the basis of a travel support system. *Proceedings of the BIS 2007 Conference*, strony 488–499. Springer, 2007.

[41] Wooldridge M. John Wiley & Sons, 2002.

[42] Andy Nauli. Using software agents to index data of an e-travel system. Master thesis, Oklahoma State University, 2000.

[43] H. S. Nwana. Software agents: An overview. *Cambridge University Press, Knowledge Engineering Review*, strony 205–244. Cambridge University Press, 1996.

[44] Martins E. Pascoal M. *A new implementation of Yen's ranking loopless paths algorithm.* 4QR - Quarterly Journal of the Belgian, French and Italian Operations Research Societies, 2003.

[45] Gruber T. R. Toward principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, strony 907–928. Elsevier, 1995.

[46] Stevens R. Prentice Hall, 1999.

[47] Park M.-W. Szymczak M. Frackowiak G. Ganzha M. Paprzycki M. Rhee S. K., Lee J. Measuring semantic closeness of ontologically demarcated resources. *Fundamenta Informaticae*, strony 395–418, 2009.

[48] Shapira B. Ricci F., Rokach L. *Introduction to Recommender Systems Handbook*. Springer, 2011.

[49] Yen Jin Y. An algorithm for finding shortest routes from all source nodes to a given destination in general networks. *Quarterly of Applied Mathematics 27*, strona 526–530, 1970.