Exploiting a Search Engine to Develop More Flexible Web Agents

Shou-de Lin
Computer Science Department,
University of Southern California
sdlin@isi.edu

Craig A. Knoblock
Information Sciences Institution
University of Southern California
knoblock@isi.edu

Abstract

With the rapid growth of the World Wide Web, more and more people rely on the online services to acquire and integrate information. However, it is time consuming to find the online services that are perfectly suited for a given task. First, the users might not have enough information to fill in the required input fields for querying an online service. Second, the online service might generate only partial information. Third, the user might only find the inverse version of the desired service. In this paper we propose a framework to develop flexible web agents that handle these imperfect situations. In this framework we exploit a search engine as a general information discovery tool to assist finding and pruning information. To demonstrate this framework, we implemented two web agents: the Internet inverse geocoder and the address lookup module.

1. Introduction

In general web agents adopt two strategies to gather information from the Internet. The first is to rely on a search engine, e.g. many question answering (QA) systems extract answers from search results [11]. The second is by querying appropriate online services, e.g. a web agent that gathers geographic data usually queries the online geocoder for the latitude/longitude (lat/long) corresponding to a given address.

In this paper, we define an online service as an Internet service that provides an interface for the users or agents to interact with its internal program for relevant information. The tasks performed in the internal program can be as simple as querying its local database or as complicated as integrating various information from different sites. Nonetheless, the web agents tend to view the internal functionality as a black box (Figure 1) since the internal process is unknown. The agent has to provide an input set $x_{1...}x_m$ and the online service will accordingly generate output set $y_1 \ldots y_n$. For instance the geocoder site[1]

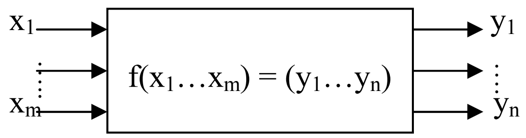


Figure 1. The online service as a black box

is a typical online service in the sense that $(x_{1...}x_m)$ represents an input address while $(y_1 \ldots y_n)$ is the corresponding latitude and longitude.

The two information-gathering strategies, either utilizing the search engine or querying an online service, are diverse in many aspects. First of all, the information found via these two strategies is different. Search engines surf through many online documents; however, the drawback is that they are incapable of acquiring specific information from online services. For example, the web agent that utilizes the search engine cannot uncovering the lat/long given the address. On the other hand, online services, usually designed for providing certain types of information, supply only domain-specific data and thus cannot be applied as generally as a search engine.

Moreover, the characteristics of their inputs and outputs show some divergence. While utilizing a search engine, the web agent has flexible keywords as inputs. However, the inputs required for the agents to interact with online services are usually restricted. The online service accepts only a certain type of data (e.g. the zip code can only be a five-digit number) and sometimes there are implicit constraints or correlation among inputs (e.g. for city and state, there is no New York City in California). The outputs are also organized differently: the outputs of a search engine are arbitrary documents, structured or unstructured. On the other hand the outputs of online services usually have a structured or semi-structured format and in most cases can be extracted easily and precisely by a wrapper [9].

This paper describes the idea of the flexible web agent, whose goal is to integrate these two strategies to exploit the strength of each. There are two potential prospects of integration. The first is to keep using online services as the core information-seeking approach in a web agent and apply the search engine as an auxiliary tool to handle the limitations of the online sources. The second is to

[1] http://geocode.com/eagle.html

enhance the facility of a search engine for interacting with the online service to improve the recall rate of search results. In this paper we will focus on the first one.

2. Limitations of online services

In this section we address three potential limitations of the online services. Ideally an intelligent web agent should have the capability to generate high quality results even if these limitations exist.

The first limitation is the existence of required inputs. Many online services require valid inputs, but not always a web agent is provided with sufficient information to fulfill all the required fields. For instance, in many cites that provide stock information such as Yahoo Finance[2], it is required to use the "ticker symbol" as the input instead of the company name, but sometimes the users only know the latter.

The second limitation is the incompleteness of the output. In many cases the online services are incapable of returning all the information their users are looking for. The users would like, in the ideal world, an intelligent web agent that automatically fills in the missing information. For example, the web agent that utilizes the Yahoo Yellowpage site[3] can discover a company's phone number, city and state information given its name. But this agent cannot satisfy the users that need the zip code.

The third limitation is lack of reversibility. The majority online sources provide only one-way lookup services. For instance, many online email-finding services are lack of the reversibility; that is, people can find the email from a person's name but not vice versa. Resolving inverse queries given only the forward lookup service is a challenging non-deterministic task. Theoretically it is solvable by exhaustive search given the input domain is finite but in practice it is usually computational intractable. Users would prefer to use a web agent that can handle the inverse query with only the existence of forward services.

3. Web agents that handle imperfect sources

In this section we describe a framework to develop flexible web agents that are adaptive to the above three limitations. The key idea is to exploit a search engine as an auxiliary tool that generates required information.

3.1. The assumption

Our approaches are appropriate for the agents utilizing imperfect online services that have inputs and outputs satisfying the following assumption: *Given $E=\{e_1 \ldots e_n\}$ is an input or output set of an online service, then* $\forall e_i \in E, \exists$ a non-empty set $E' \in subset(E)$ *and* $E' \neq \{e_i\}$ *s.t. all the elements in the set $\{E', e_i\}$ appear somewhere in at least one document that can be found by a search engine.*

This assumption captures the idea that the elements in the input set are correlated with each other in the sense that we can use some of them to index the others through a search engine. The same assumption applies to the output set as well. The inputs and outputs of a typical online service often satisfy this assumption. For instance, the (title, director, cast) as the inputs to a movie site; the (street number, street name, zip code) as the outputs to a theater or restaurant lookup services and inputs to a map lookup page; the (title, author, publisher) as the inputs to the electronic library.

Our assumption is similar to the fundamental assumption behind all keyword-indexed information retrieval systems. It is a reasonable assumption in view of the fact that the inputs themselves are used together to query a set of outputs. So these inputs are to the least extent correlated with each other through the outputs, and in many cases the correlations are even stronger.

However, we do not assume similar correlation to occur between inputs and outputs, which would be a much stronger assumption than the one we made and conceivably can be satisfied in fewer cases. In other words the assumption does not necessarily hold if the set E is the union of input and output sets. Take a geocoder for example: the inputs (address) and outputs (latitude/longitude) usually do not appear together in any documentation that can be found by a search engine.

3.2. Handling input and output limitations

To deal with the first two limitations of an online resource, we propose an idea of utilizing the search engine along with the known information as a preprocessor and postprocessor to generate potential candidates for the missing input and output fields.

Figure 2 shows the framework of developing a web agent that copes with missing required inputs. We exploit a search engine as the pre-processor to generate the required inputs. There are three stages for generating the required inputs. The first is the keyword-generation stage. In this stage the agent uses incomplete input data to form a set of keywords to the search engine: Given an incomplete set of inputs $x_2 \ldots x_n$ (x_1 is the missing but required input) ,the web agent can formulate the strictest keywords by putting them in one group (keyword="$x_2 \ldots x_n$"). Alternatively it is feasible to relax the keyword by putting one quote on each and combining them (keyword="x_2",…,"x_n"). It can also drop some inputs to make it less strict (keyword="x_2",…,"x_i", $i<n$).

[2] http://finance.yahoo.com/

[3] http://yp.yahoo.com

Additionally one can apply the "keyword spices" [13] approach to perform domain specific searches through a general purposed search engine by adding some auxiliary keywords in this stage. The second stage is to call the search engine with one of the generated keywords. The third stage is to extract potential candidates for missing inputs from the documents returned by the search engine. After the candidates for required input fields are generated, the agent can use them to query the online services. Note that multiple input candidates could be generated, thus the web agent will return a set of plausible results instead of one. It is preferable since the user might want more choices given some key criteria are missing.

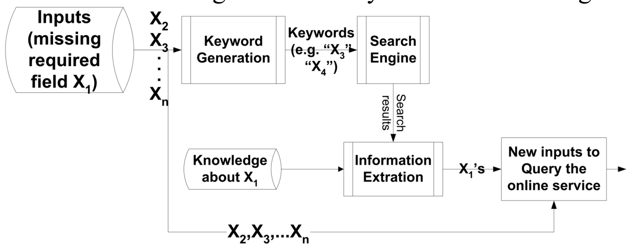


Figure 2. The framework for exploiting a search engine to handle missing required-input X1

Let us look at an example: Given a movie service that takes "director name", "leading actor", and "leading actress" as inputs and outputs the movie title. If our agent is given only partial inputs "leading actor=P1, leading actress=P2" but not the required "director name", it will formulate different keywords ("P1 P2", "P1" "P2", "P1", or "P2") to the search engine and extract all the potential director names Dk from the returned documents. Then the input sets (P1,P2,D1),…,(P1,P2,Dk) will be applied to query the movie service one by one and the user will be able to see all the associated directors and movie titles returned based on P1 and P2.

The same concept can be applied to find the missing outputs by utilizing these three stages as the post-processor to the online service.

In the third stage we perform an Information Extraction (IE) task, which is the most challenging step in our framework with the goal to extract relevant facts from a document [10]. One traditional method for IE is to apply natural language process techniques [15]. Alternatively we can use wrapper technology to automatically wrap semi-structured pages [9]. Additionally, machine learning techniques are also applicable for both semi-structured or non-structured sources [14].

In general extracting precise information from arbitrary web documents is a challenging problem. However, in our framework, we search for only required inputs or incomplete outputs, which themselves usually form some pattern (e.g. there are patterns for addresses, emails and names) and can be handled by similar techniques applied to named-entity tagging problems. The methods using Hidden Markov Models [4], Rule-based systems [7], or Maxima Entropy Models [1] to extract names, time, and monetary amounts are applicable approaches for our IE stage. Another reason that makes our IE stage not as difficult as a typical IE problem is that the precision of the result is not critical. The backend online service can be treated as a precise evaluation engine that filters out the incorrect or irrelevant inputs generated by the IE engine.

Due to the fact that the pattern is known and the precision is not as important as recall in our IE stage, we present a suitable IE method as formatting the pattern instantiation problem into an AI Constraint Satisfaction Problems (CSP) by modeling the pattern as a set of constraints. The advantage of formatting an IE problem into a CSP is that we do not need to explicitly program how to extract each individual field in the pattern. Instead we tell the CSP engine what the pattern looks like and the CSP engine will look for all the matched instances for us. In addition we can easily control the recall and precision rate by manipulating the constraints: strict constraints imply high precision (and low recall) while sparse and loose constraints raise the recall at the cost of precision. This CSP approach simplified the implementation of our IE stage since, for a recall-driven problem, it is not necessary to exhaust ourselves to conceive all the precise constraints. As will be shown in section 4.2, with a backend online service as a verification component, we can fill in all the missing address fields without exactly knowing how to extract the street name from a document.

3.3. Handling inverse queries

The third limitation of online services is that most of them accept only one-way queries. In this section we propose a framework to construct a web agent that resolves inverse queries from the forward services.

The challenge of constructing a reversible service lies in the fact that the original resource (online service) is an **unknown one-way function**. We first give a working definition, borrowed from cryptography, to the one-way function [2]:

*Definition****:*** *A function f from a vector space X to a vector space Y is called a one-way function if f(x) is "easy" to compute for all vectors* $x \in X$ *, but for a random vector* $y \in f(x)$ *it is computationally infeasible to find any* $x \in X$ *satisfies* $f(x) = y$

In general there is no shortcut to find out the x that satisfies f(x) = y given y if the f(x) is a black box. The only way is to try the candidates of X one by one until a match is found. This is also the basic assumption behind information security and key encryption/decryption [2]: the non-deterministic inverse function plus a immense input domain limit the chance of successful cracking (find x that satisfies f(x) = y) to almost zero.

Not knowing what is inside the black box for an online service, what we can do to improve the performance of inverse mapping is to reduce the "trial and error" testing domain. In this scenario the search engine plays a role as a heuristic generator, which provides the most plausible input candidates.

Originally the "trial and error" method has input cardinality as large as the cardinality of the cross product of all input fields $|x_1|*|x_2|*\ldots*|x_n|$, where $|x_k|$ stands for the cardinality of a certain input field. There are two steps for reducing the search domain in our framework. **The first step** is to check if there exist online services that map the output y to some individual input field. If there are services that takes y or a subset of y as inputs and generate partial set of x, say x_1 to x_k, then the "trial and error" cardinality will be cut to $|x_{k+1}|*|x_{k+2}|*\ldots*|x_n|$. **The second step** is to utilize the identified input fields $x_{1\ldots}x_k$ to indicate remaining input fields $x_{k+1\ldots}x_n$ in a search engine.

For example, assume there is a one-way online movie service that enables the users to search for a movie title by its leading actor, actress and director. To perform the inverse query, the very naïve way is to test all the combination of actors, actresses and directors in the world and check which combination generates the given movie title. This naïve method has the testing domain as large as |Director|*|Actor|*|Actress|. However, in step one we can first check if there are online services that map the movie title to its individual inputs (director, actor, or actress). Assuming we have found a service that maps the movie title to its director, then the cardinality of search space can be reduced to |Actor|*|Actress|.

According to our fundamental assumption that the inputs are more or less correlated, heuristically in the "trial and error" period we would like to give a higher priority to the input set that has elements associated with one another. In our second step the search engine plays a role as this heuristic engine in the following manner. First use the identified director name as a keyword to indicate and extract the associated actors in the search engine. Afterwards, each pair of (director, actor) can be used as the keyword again to index the associated actresses. Finally a set of plausible inputs fields will be generated and the cardinality of this set is |Actor given Director|*|Actress given Director and Actor)|. The |X given Y| represents the cardinality of X returned by the search engine given Y is used as the keyword. Conceivably the number of actors associated with a director is much smaller than the total number of actors in the world. The number of actresses associated with a given director and actor should be even smaller. In this scenario the search engine acts as a heuristic function to guide the "trail and error" testing. The size of test domain to the least extend can be reduced to $|x_1|*|x_2$ given $x_1|*\ldots*|x_n$ given $x_{n-1}\, x_{n-2\ldots}\, x_1|$ by applying only the second step even if no suitable online service can be found in the first step.

4. Case Studies

Two web agents, the inverse geocoder and the address lookup module, are implemented to demonstrate our framework.

4.1. The inverse geocoder

The inverse geocoder is a web agent realizing the idea of developing inverse service by its forward source. We developed it by integrating the search engine with the online resource (Mapblast[4]) to transform the geocode into its equivalent address including the closest street number.

The inverse geocoder consists of three parts: The zip finder, the street name finder, and the street number locator since a typical address in the United States can be uniquely identified by these three types of information.

Zip finder: The corresponding zip code of a given geocode can be found in the Mapblast site. Mapblast_Maps has the feature of displaying the map centered at a geocode given by its user. While checking the source code of this map page, we can find a hidden field "zip" that contains the zip code. Our zip finder sends the lat/long to this Map service and wraps the zip code.

Street name finder: The street name finder discovers the street name of a given geocode by manipulating the inputs to the Mapblast_Direction. Mapblast_Direction is a service that returns the driving direction (in both text and graph format) from a user-specified starting point to a specified ending point. In its advanced search it allows users to use latitude and longitude to identify the points.

The street name finder uses the original latitude and longitude as the starting point to the Mapblast_Direction. For the ending point, it uses the same latitude but slightly modifies the longitude to longitude-0.001 (see Figure 3).

By slightly modifying the geocode as the destination point, it essentially asks the system to produce the driving direction from the original lat/long to a place that is really close to it. Conceivably the street name returned in the driving direction is the street name of that lat/long (see Figure 4). The system also extracts the street direction since it is useful in "street number locator".

The zip finder and street name finder realize the idea proposed in the step 1 of section 3.3: to use online services to acquire partial inputs from the outputs. Although there is no service that explicitly provides the mapping from lat/long to zip code or street name, we exploit some related services to acquire the information.

[4] http://www.mapblast.com

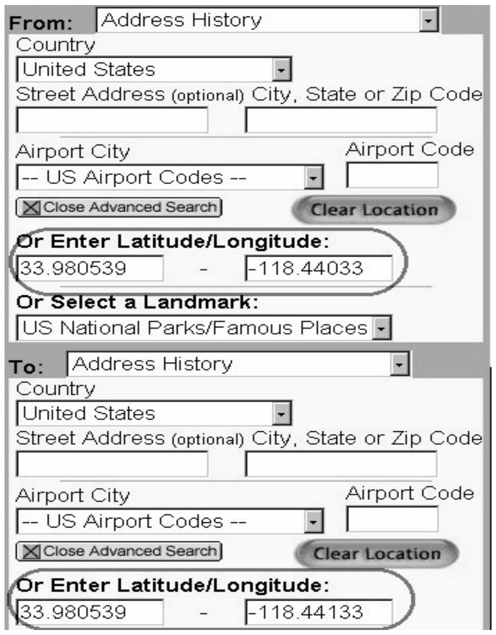

From: Address History
Country
United States
Street Address (optional) City, State or Zip Code
Airport City Airport Code
-- US Airport Codes --
Close Advanced Search
Clear Location
Or Enter Latitude/Longitude:
33.980539 - -118.44033
Or Select a Landmark:
US National Parks/Famous Places
To: Address History
Country
United States
Street Address (optional) City, State or Zip Code
Airport City Airport Code
-- US Airport Codes --
Close Advanced Search
Clear Location
Or Enter Latitude/Longitude:
33.980539 - -118.44133

Figure 3. Inputs to the Mapblast_Direction

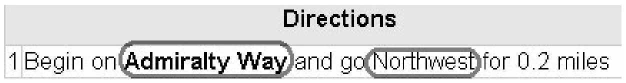
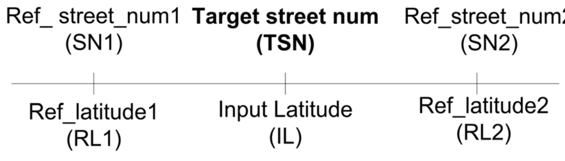

Directions
1 Begin on Admiralty Way and go Northwest for 0.2 miles

Figure 4. Outputs of the Mapblast_Direction

Street number locator: This locator brings the search engine into play to prune the size of "trial and error" domain of the street number given the street name and zip code. It realizes the idea of the second step in section 3.3 by applying the search engine as a heuristic function to guide the "trial and error" procedure. The search engine was applied to find two **valid** street numbers as reference points and use interpolation and extrapolation method to locate the precise street number of the desired geocode.

Once the street name and zip code are known, a straightforward method to locate the street number is to use two valid street numbers as reference, geocode them and apply the interpolation (given the address is in between two reference points) or extrapolation (given the address is not in between two reference points) method. For example, to locate the street number of the geocode (33.9803, -118.4402) given the known street name "Admiralty Way" and zip code "90292", we first use the available forward service (e.g. http://geocode.com) to find the lat/long for two reference addresses on the same street. For instance, "4000 Admiralty Way, 90292" has geocode (33.9815, -118.4599) and "5000 Admiralty Way, 90292" has (33.9791, -118.4522). The interpolation method can then be applied on latitude or longitude[5] to find the target street number as 4511. The interpolation equation is shown in Figure 5. Since in the real world the street number is not uniformly distributed, it is necessary to repeat the same procedure iteratively until it converges.



TSN=SN1+(IL-RL1)*(SN2-SN1)/(RL2-RL1)

Figure 5. Interpolation on latitude

The tricky part of this approach lies in choosing the first two valid reference points. Randomly picking street numbers is not efficient due to the variety of the street numbers. Some streets have valid street numbers only from 1 to 100 (e.g. Mason St, Coventry, CA) while others have valid numbers between 34000 to 38000 (e.g. Ridge Rd, Willougby, OH). To resolve this problem we applied the search engine as proposed in section 3.3 to reduce the cardinality of the street number domain. The idea arises from the fact that the street numbers indexed by the street name and zip code through the search engine are usually valid street numbers for that street name and zip. Figure 6 shows one of the results returned by the Yahoo[6] search engine while using a street name "Admiralty Way" and zip code "90292" as the keyword. The street number locator extracts the street numbers returned by Yahoo (4100 in this case) as the reference points.

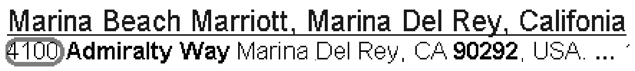

Marina Beach Marriott, Marina Del Rey, Califonia
4100 Admiralty Way Marina Del Rey, CA 90292, USA. ...

Figure 6. Relations among street name, street number and the zip code in Yahoo search

Performance: We tested our inverse geocoder on 100 different lat/long. For each lat/long our program has to call the Mapblast_Maps once, Mapblast_Direction once, Yahoo Search once, and Mapblast_forward_geocoder on the average 4.7 times. For most of our test cases the search engine found two valid street numbers, which enabled the system to perform the forward geocoder as few as three times (two for geocoding the reference points and one for verifying the result). If each service takes two seconds, then it takes on the average (1+1+1+4.7)*2=15.4 seconds to accomplish the inverse task. There are three cases that the street number cannot be found by a search engine. Thus our agent had to perform binary search for valid street numbers, which takes the agent to execute the forward geocoder on the average 12 times to discover one valid reference. The results show that utilizing the search engine can significantly shrink the size of "trail and error" domain.

[5] Whether using latitude or longitude for interpolation depends on the orientation of the street, for a north-south street, the latitude is used, otherwise the longitude is applied.

[6] http://www.yahoo.com

Our approach does not require a local database or intensive computation. Moreover, it can be implemented in a short time. The zip finder and street name finder demonstrate the fact that there is plentiful information hidden on the Web, only one has to create ways to find it.

4.2. Address lookup module

In this case study we demonstrate how to apply a search engine to handle the required inputs and incomplete outputs limitations. The test platforms are the WhitePage[7] service and the Yahoo Yellowpage[8]. In WhitePage it is required to fill in the complete address to obtain the corresponding phone number. Yahoo Yellowpage is a service that generates address information (except the zip code) as outputs.

We implemented an address lookup module as the preprocessor to the WhitePage to fulfill all the required input fields given only partial address information. The same module is used as a post-processor to YellowPage for the missing zip codes.

This module has seven optional input fields: entity information, street number, street name, apartment number, city, state, and zip code. It has seven identical fields as outputs. The idea is to utilize a search engine with the known values to find the missing fields. Note that in addition to the address, the user can provide any other necessary information in "entity information" field and this auxiliary information will be treated as the keyword to the search engine as well. The address lookup module has three phases as discussed in section 3.2.

The keyword-generating stage takes the inputs to generate the keywords from the strictest one to the loosest one. In the search phase the module uses the keyword to extract relevant pages. It then sends the top 100 ranked documents returned by the search engine to the third phase. The third IE phase is designed to extract the potential candidates of address from these documents. The addresses of the United States form a regular pattern: a street number followed by a street name followed by an apartment number, then the city, state, and zip (Figure 7).

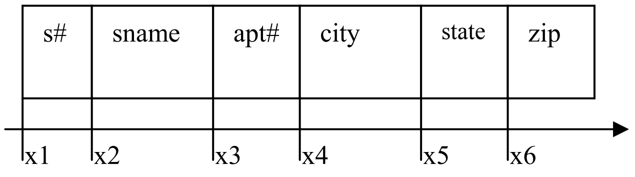


Figure 7. The pattern of US address

Given this pattern and some intrinsic characteristics of each address field, we can format this address extraction problem into an equivalent constraint satisfaction problem (CSP) (or a linear programming problem). In the corresponding CSP problem, the starting positions of each field (x1….x6 in Figure 7) in a document are variables. The constraints come from the order shown in the pattern (e.g. since city is followed by Apt#, so x4-x3>=0) as well as the characteristic of each input type (e.g. zip codes are numbers of 5 digits, or state names have at most two words). By representing this knowledge as a set of constraints, our CSP engine can generate all the consistent variable sets of (x1…x6) satisfying these constraints, e.g. (4,6,10,15,22,24). Each solution indicates a position of a potential address pattern in the document. In our system determining the precise constraints are not necessary. The major goal of our IE stage is to improve the recall since we have a backend online service as a precise verification tool.

The addresses generated by the CSP engine will then be sent to the evaluation engine for ranking. They are ranked according to their similarity to the input fields. The agent can then use the ranked outputs to query online services such as WhitePage and Yahoo Yellowpage.

The module is evaluated by examining whether the returned phone number by Whitepage is correct. We have tested our module under two different scenarios for 50 valid addresses. Half of them miss the street name while the other half are short of both the city and state information. The results show that in 70% of the cases our module can recover the missing street name. The task is simpler (90% accuracy) while the missing fields are city and state. We then apply our module as the post-processor to Yahoo Yellowpage. The results show that it can recover all the zip information precisely.

This result shows that our framework is applicable in designing flexible web agents that are adaptive to the required-input and incomplete-output limitations.

5. Related Work

The idea of exploiting the functionality of search engines resembles Etzioni's information food chain [6], in which the search engines are located in the middle of the food chain and there are goal-oriented softbots (software robots) built on top of them. Citeseer [8] is an autonomous web agent that utilizes search engines for retrieving and identifying publications. Additionally most of the QA systems utilize the search engine as well [11]. However, they operate the search engine as a major tool for inquiring information and do not usually integrate it with the other resources. In our approach the online services are still the major tool for acquiring information while search engine plays a supporting role in providing the extra information and reducing the input cardinality for inverse service.

[7] http://www.whitepages.com/address-lookup

[8] http://yp.yahoo.com/

On the other hand many information integrating platforms made efforts toward integrating various online services to achieve specific tasks, such as ShotBot [12], the Information Manifold [3], and Ariadne [5]. These systems aim at resolving different issues of integrating data from the web such as resource selection and modeling, view integration, and handling the inconsistency among sources. However, none of these frameworks is developed to generally fix the limitations (especially the non-reversibility) of the sources.

Although it is feasible to integrate other online resources instead of a search engine to resolve the required input or incomplete output limitation. However, our approach of integrating a search engine into a web agent is more suitable for this situation because it saves the time of finding appropriate sources. In addition it is more flexible and less risky since there any many available search engines and they are usually more stable than the online services.

6. Conclusions

In this paper we provide a new framework for developing web agents that overcome the required input and incomplete output limitations of sources by exploiting the search engine as the pre-processor or post-processor. Moreover, we propose an idea of applying a search engine to reduce the cardinality of the trial-and-error domain while generating the inverse service from its forward service. We also implemented two web agents to demonstrate our frameworks. The inverse geocoder is a web agent that accomplishes the inverse geocoding task without employing any local database. The address lookup module demonstrates a flexible and reusable component that can be plugged into a variety of web agents that uses addresses as inputs and outputs. We also present the idea of resolving a certain type of information extraction problems by translating it into an equivalent constraint satisfaction problem, which simplifies the implementation of the recall-driven IE tasks.

7. References

[1] A. Borthwick, J. S., E. Agichtein, and R. Grishman. NYU: Description of the MENE Named Entity System as used in MUC-7. in the Seventh Machine Understanding Conference (MUC-7). 1998.

[2] A. J. Menezes, P. C. v. O., S. A. Vanstone, Handbook of applied Cryptography. 1996: CRC Press.

[3] A. Y. Levy, A. R., J. J. Ordille. Querying Heterogeneous Information Sources Using Source Descriptions. in Intl. Conference on Very Large Data Bases (VLDB). 1996.

[4] M. Bikel, Nymble: a high-performance learning name-finder. in Fifth Conference on Applied Natural Language Processing. 1997: Morgan Kaufmann Publishers.

[5] C. Knoblock, S. M., J. Ambite, N. Ashish, I. Muslea, A. Philpot, S. Tejada, The ARIADNE Approach to Web-Based Information Integration. International Journal of Cooperative Information Systems, 2000: p. 145--169.

[6] O. Etzioni, Moving Up the Information Food Chain: Deploying Softbots on the Worldwide Web. in Proc. 13th Nat'l Conf. Artificial Intelligence (AAAI 96). 1996. San Mateo, Calif: AAAI Press.

[7] G. R Krupka , K. H. IsoQuest Inc: Description of the NetOwl "Fext Extraction System as used for MUC-7". in Seventh Machine Understanding Conference. 1998.

[8] K. D. Bollacker, S. L., and C. Lee Giles. CiteSeer: An Autonomous web Agent for Automatic Retrieval and Identification of Interesting Publications. in 2nd International ACM Conference on Autonomous Agents. 1998.

[9] N. Kushmerick, D. W., R. Doorenbos, Wrapper induction for information extraction. Proc. of 15th International Conference on Artificial Intelligence, IJCAI-97, 1997.

[10] M. T. Pazienza,, Information Extraction: A multidisciplinary Approach to an Emerging Information Technology, in volume 1299 of Lecture Notes in Computer Science, International Summer School, SCIE-97. 1997: Frascati (Rome), Springer.

[11] R. Srihari, W. L., Information extraction supported question answering. Proceedings of the 8th Text Retrieval Conference, 1999.

[12] R. B. Doorenbos, O. E., and D.S.Weld. A Scalable Comparison-Shopping Agent for the World-Wide Web. in First International Conference on Autonomous Agents (Agents'97). 1997. Marina del Rey, CA, USA.

[13] S. Oyama, T. K., T. Ishida, T. Yamada, Y. Kitamura, Keyword Spices: A New Method for Building Domain-Specific Web Search Engines. the 17th International Joint Conference on Artificial Intelligence, 2001.

[14] S. Soderland , Learning Information Extraction Rules for Semi-structured and Free Text. Machine Learning, 1999. 34(1-3): p. 233-272.

[15] Y. Wilks , Information Extraction as a core language technology. 1997, In M-T. Pazienza (ed.): Springer, Berlin.