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 difficult and time consuming to find the online services that are perfectly appropriate 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 want to query the information about B by some input set A, but he or she can only find the online services that generate A from B. Ideally one would like an intelligent web agent to still unearth complete and accurate information despite these imperfect sources. 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 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 [6, 19]. 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 users or web agents tend to view the internal functionality as a black box (see Figure 1) since the internal process is unknown. The agent has to provide an input set $x_1 \ldots x_m$ and the online service will accordingly generate output set $y_1 \ldots y_n$. For instance the geocoder site[1] is a typical online service in the sense that $(x_1 \ldots x_m)$ represents an input address while $(y_1 \ldots y_n)$ is the corresponding latitude and longitude.

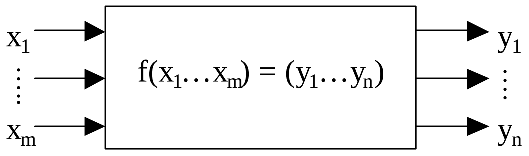


Figure 1: The Online service as a black box

These 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 information from online services. For example the web agent that utilizes the search engine cannot uncover the lat/long given the address. On the other hand, an online service, usually designed for providing certain types of information, supplies only domain-specific data and thus cannot be applied as general as a search engine.

Moreover, the characteristics of inputs/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 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 [16].

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 service as a core information-seeking approach in a web agent and apply the search engine as an auxiliary tool to handle the

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

limitations of the online sources. The second is to 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 outputs even if these limitations exist.

The first limitation is the existence of required inputs. Most online sources require valid input fields to initialize the service. For instance in AnyWho[2] white page site, the last name and the residential state of a person is required and no phone number will be generated if any one of them are left blank. These input fields are defined as "required inputs". An intelligent web agent has to handle the situation of missing required inputs in the sense that its user, even short of some required input information, might be capable of providing auxiliary data with the hope to still get results with a certain level of accuracy. For instance, the users might expect an intelligent web agent to find the potential phone numbers of their childhood neighbors even they do not know which state these neighbors are currently in.

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 Yahoo Yellowpage site[3] can get a company's phone number, city and state information given its name. But this agent cannot satisfy the users that require the zip code. An intelligent web agent could automatically exploit other sources to improve its recall rate if the current results are not sufficient for the users.

The third limitation is lack of reversibility. The majority of the web agents have online sources that provide only one-way lookup services. For instance, although there are online Geocoder services that can transform an address to its corresponding lat/long, we find no online service performing the inverse task to get an address from the lat/long. Resolving inverse queries given only the forward lookup service is a challenging non-deterministic task. Theoretically it is solvable by the exhaustive search since the discrete input domain is finite, but in practice it is usually computation intractable. The users, again, prefer a magic web agent that can somehow handle the inverse query even though there exists only the forward service.

3 Web agents that handles 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/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* $E' != \{e_i\}$ *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 usually 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 and outputs to an 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 source, we propose an idea of utilizing the search engine

[2] http://www.anywho.com/wp.html

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

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="$x_1, x_2 \ldots x_n$"). Alternatively it is feasible to relax the keyword by putting one quote on each and combining them (keyword="x_1","x_2",…"x_n"). It can also drop some inputs to make it less strict (keyword="x_1","x_2",…,"x_i", $i<k$). Additionally one can apply the "keyword spices" [20] approach to build 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 the potential candidates for the missing inputs x_1 from the documents returned by the search engine. While the candidates for x_1 are generated, the agent can then query the online services. Note that multiple candidates of x_1 could be generated, thus the web agent will return a set of plausible results instead of just one. It is preferable since the user might want more choices given some key criteria are missing.

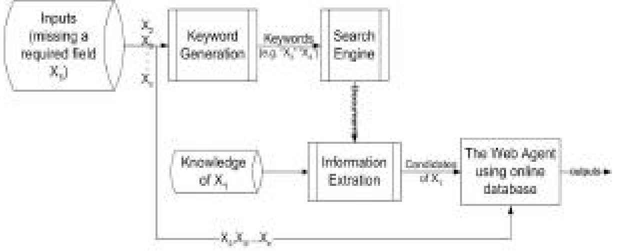


Figure 2: The framework for exploiting a search engine to handle missing required-input X1

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 (conceivably the service will not return anything meaningful for the wrong combinations) and the user will be happy to see all the directors and movie titles returned based on P1 and P2.

The same concept can be applied to discover 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, whose goal is to extract relevant facts from a document [17]. It is the most challenging step in our framework thus we would like to address some applicable IE approaches. One traditional method for IE is to apply natural language process techniques, which have been studied for decades [23]. Alternatively we can use wrapper technology to automatically wrap semi-structured pages [15]. Besides, machine learning techniques are widely used in learning the extraction rules and it is applicable for both semi-structured or non-structured sources [9, 16, 21].

In general extracting precise information from arbitrary web documents is challenging. However, in our framework, we need only required inputs or incomplete outputs. These input and output data usually follow some patterns (e.g. there are patterns for address, email and phone) and can be extracted by similar techniques applied to named-entity tagging problems. The methods using Hidden Markov Models [3], Rule-based systems [11], or Maxima Entropy Models [4] to extract names, time, and monetary amounts are applicable approaches to our IE stage. Another factor that makes our IE stage not as difficult as a typical IE problem is that the precision of the IE result is not critical since the backend online service can be treated as a evaluation engine that filters out the incorrect or irrelevant inputs generated from 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 set of constraints. The advantage of formatting an IE problem into a CSP is that we don't 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. Also 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 successfully generate the missing address fields by this approach.

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 answers 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* $x \in X$ *such that 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 in 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, the only thing 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 huge 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 M is an 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 (in other words to find out the leading actor, actress and director given a movie title), 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 some of its individual inputs (director, actor, or actress). Assume we have found a service that maps the movie title to its director. Then the cardinality of search space is 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. Eventually 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 smaller as well. In this scenario the search engine plays a role as a heuristic function to guide the "trail and error" testing, we can also say that it reduces the size of testing domain. The size of test domain in general 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 no suitable online service can be found in the first step.

In practice for each value indexed by the search engine, we have to record the keywords that were used to index it (e.g. Jean $_{\text{director=John,actor=Tom}}$ represents that the actress Jean is indexed by the director John together with actor Tom by the search engine). Finally the complete inputs (e.g. John=director, Tom=actor, Jean=actress) are generated to query the online service, one after the other, until a match of the output is found.

4 Case Studies

We implemented two web agents, the internet inverse geocoder and address lookup module 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 Unite 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 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. Mapblase_Direction is a service that returns the driving direction (in both text format and graph) from a user-specified starting point to an ending point. In its advanced search it allows the user to use latitude and longitude to identify a point.

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).

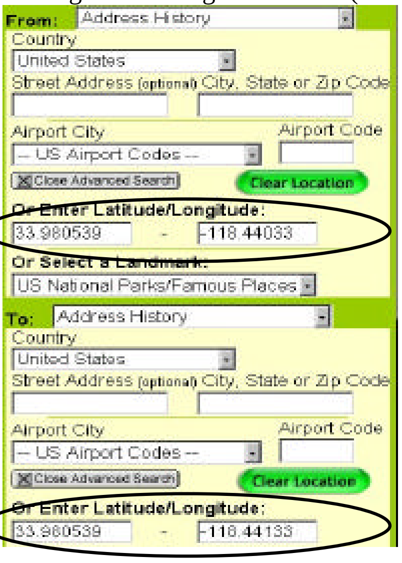

From: Address History
Country
United States
Street Address (optional) City, State or Zip Code
Airport City Airport Code
-- US Airport Codes --
Close Advanced Search
Clear Location
Or Enter Latitude/Longitude:
33.980539 - -118.44033
Or Select a Landmark:
US National Parks/Famous Places
To: Address History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

By slightly modifying the departure 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".

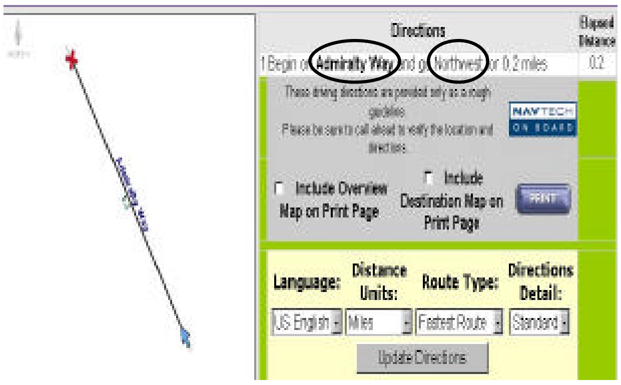

Directions | Elapsed Distance
1 Begin on Admiralty Way and go Northwest for 0.2 miles | 0.2
These driving directions are provided only as a rough guideline.
Please be sure to call ahead to verify the location and directions.
Include Overview Map on Print Page
Include Destination Map on Print Page
PRINT
Language: US English
Distance Units: Miles
Route Type: Fastest Route
Directions Detail: Standard
Update Directions

Figure 4: The output of the driving direction

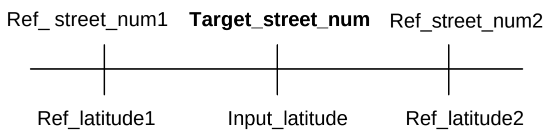
The zip and street name finder realizes the idea we proposed in the first step of section 3.3: to use online services to acquire partial inputs from the outputs. Although there is no service that explicitly provides the mapping from lat/long to zip code or street name, we are capable of manipulating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

Street number locator: This locator brings the search engine into play to prune the size of "trail and error" domain of the street number. It realizes the idea of the second step shown in section 3.3 by applying the search engine as a heuristic function to guide the "trial and error" procedure.

As 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 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 (33.980344, -118.440268) given the known street name "Admiralty Way" and zip code "90292", we can first use the available forward service to find the lat/long for arbitrary two reference addresses on the same street. For instance, "4000 Admiralty Way, 90292" has geocode (33.981569, -118.459910) and "5000 Admiralty Way, 90292" has (33.979176, -118.452240). Then we introduce the interpolation method 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 convergence.

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

[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.



Target_street_num=Ref_street_num1+(Input_latitude-Ref_latitude1) *(Ref_street_num2-Ref_street_num1) /(Ref_latitude2-Ref_latitude1)

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, CI) and 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 is 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 result 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 number returned by Yahoo (4676 in this case) as the reference points.

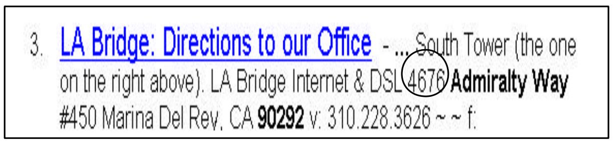

3. LA Bridge: Directions to our Office - ... South Tower (the one on the right above). LA Bridge Internet & DSL 4676 **Admiralty Way** #450 Marina Del Rey, CA **90292** v: 310.228.3626 ~ ~ f:

Figure 6: The association between street number, street name and zip

Performance: We tested our inverse geocoder on 100 different lat/long and Table 1 shows the average times each service is called. For most of our test cases the search engine found two valid street numbers, which enabled the system to call the forward geocoder as few as three times (two for geocoding the reference points and one for verifying the result). Assume it costs 5 second to query each service, it takes on the average (1+1+1+4.7)*5=38.5 seconds to accomplish the inverse task. This performance is acceptable. There are three testing cases that the street number cannot be found by a search engine and thus our agent had to perform binary search for valid street numbers, which takes the agent to execute the forward geocoder on an average of 10 more times to dscover one valid reference. The experiment data show that integrating the search engine is a promising method to handle inverse query since it significantly shrinks the size of "trail and error" domain.

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

Table 1. Number of times each service is called

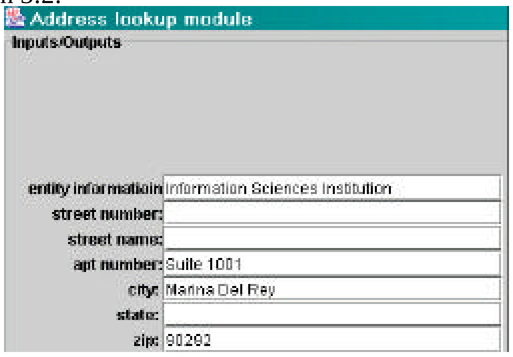
Online services	Number of Times called
Mapblast:Map	1
Mapblast: Direction	1
Yahoo Search	1
Mapblast:forward geocoder	4.7

Compared with the generic offline inverse geocoders, which tackle this problem by the geographic methods and numerical analysis based on a fairly large spatial database, our approach basically relies on integrating information from the online service and the search engine. Notably in our approach neither a local database nor intensive computation is required. Moreover, the implementation time and expenditure is much lower than generic approaches. The zip finder and street name finder also show 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

The address lookup module is built as the pre-processor or post-processor for a web-agent to fill in the missing part of any online service that has addresses as the inputs or outputs.

This module has seven optional input fields: entity information, street number, street name, apartment number, city, state, and zip code (see Figure 7). 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.



entity informatiain: Information Sciences Institution
street number:
street name:
apt number: Suite 1001
city: Marina Del Rey
state:
zip: 90292

Figure 7 The interface of address lookup module

The keyword-generation 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. Then it 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 8).

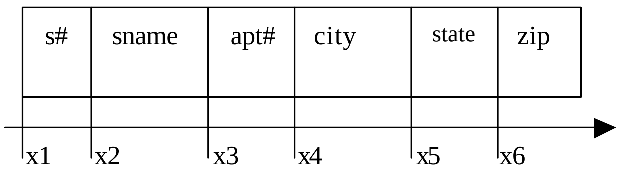


Figure 8 : The pattern of US address

Given this pattern, we can format this address extraction problem into an equivalent constraint satisfaction problem (CSP) as discussed in section 3.2. In the corresponding CSP problem, the starting positions of each field (x1….x6 in Figure 8) 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 inherent 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 pattern as a set of constraints, our CSP engine can generate all the consistent variable sets of (x1…x6) that satisfy these constraints, e.g. (4,6,10,15,22,24) and (123,124,128,128,130,130)…etc. Each solution indicates a position of a potential address pattern in the document. In our system defining precise constraints are not necessary. Our IE stage emphasizes on improving the recall since we have a backend online service as a verification tool to filter the imprecise outputs.

For evaluation, we use our module as the preprocessor to fulfill the required input address of the online WhitePage service[7], which outputs the phone-number given the address. We evaluate our module by examining if the returned phone number is correct. We have tested our module under 2 different scenarios for 50 valid addresses. Half of them missed the street name while the other half did not have both city and state information. Table 2 shows that with the street number and city/state information, in 70% of the cases our module can still find the correct phone number. The task is simpler (90% accuracy) for our module while the missing fields are city and state. We then apply our module as post-processor to Yahoo Yellowpage[8], which does not provide the zip code as the output. Table 2 shows that our module can recover the zip information perfectly. The results demonstrate that our framework is applicable in designing flexible web agents that are adaptive to the required-input and incomplete-output limitations.

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

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

Table 2. Accuracy of address lookup module

Test sets	Accuracy
Without street name	70%
Without city and state information	90%
Yellowpage without zip code	100%

5 Related Work

The idea of exploiting the functionality of search engines resembles Etzioni's information food chain [8], 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. MetaCrawler [7] is a meta-search engine built on top of several search engines. Citeseer [12] is an autonomous web agent that utilizes search engines for retrieving and identifying publications. WebSuite [5] has its own search engine that can accept not only keywords but also the criteria of the connection between pages (e.g. finding a web page that comes from page P via link N). Also most of the question and answering (QA) systems utilize the search engine [6, 14]. However, these agents operate the search engine as the major tool for inquiring information and they do not usually integrate it with the other sources. In our approach the online services are still the major tool of acquiring information while search engine plays a supporting role in providing the extra information and reducing the input cardinality for inverse service.

On the other hand many information integrating platforms made efforts toward integrating various online services such as ShopBot [18], the Information Manifold [1], Ariadne [13], and Occam [22]. These systems aim at resolving different issues of integrating data from the web such as information resource selection and modeling, view integration from distributed sources, and handling the inconsistency among sources. However none of them are focused on generally fixing the limitations (especially the non-reversibility) of the sources. In these platforms the search engine are rarely being integrated.

Although it is feasible to integrate another online resources instead of a search engine to resolve the required-input limitation, such as an information integration agent [10] does. However, our approach of integrating a search engine into a web agent is more appropriate in terms of handling the missing information: It is more flexible and effort saving since we are not necessary to find the source that fits the requirement perfectly. Furthermore, it is more robust and less risky since the search engines are more stable than the online services. It is because that every time an online service changes the contents, formats, or interfaces, people have to fix their web agents to adapt to these changes. Unfortunately, online services change every now and then.

6 Conclusions

In this paper we provide a new framework for developing flexible 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 answering the inverse query from its forward service. Our approaches can be applied not only to web agents but any type of agents that have sources of similar limitations. We implemented two web agents as the demonstration to our frameworks. Our inverse geocoder is the only web agent that accomplishes the inverse geocoding task without employing any local database or intensive computation. 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/outputs. We also present the idea of resolving a certain type of information extraction problems by translating it into an equivalent constraint satisfaction problem. Our address lookup module shows that this approach simplifies the implementation and fits perfectly in recall-driven IE tasks.

7 References

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

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

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

[4] Borthwick, e.a. *Description of the MENE named Entity System*. in *the Seventh Machine Understanding Conference (MUC-7)*. 1998.

[5] C. Beeri, G.E., T. Milo. *WebSuite – A Tool Suite For Harnessing Web Data*. in *WebDB98*. 1998. Valencia, Spain.

[6] C. Kwok, O.E., and D. S. Weld. *Scaling question answering to the web.* in *10th world wide web conference*. 2001.

[7] Erik Selberg, O.E. *Multi-Service Search and Comparison Using the MetaCrawler*. in *4th International World-Wide Web Conference*. 1995.

[8] Etzioni, O. *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.

[9] Freitag, D. *Information extraction from html: Application of a general learning approach*. in *the Fifteenth Conference on Arti cial Intelligence AAAI-98*. 1998.

[10] Genesereth, M.R., Keller, A. M., Duschka, O. *Infomaster: An Information Integration System.* in *Proceedings of 1997 ACM SIGMOD Conference*. 1997.

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

[12] 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.

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

[14] Li., R.S.a.W., *Information extraction supported question answering.* Proceedings of the 8th Text Retrieval Conference, 1999.

[15] Muslea, I. *Extraction patterns for information extraction tasks: A survey*. in *AAAI-99 Workshop on Machine Learning for Information Extraction*. 1999.

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

[17] Pazienza, M.T., *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.

[18] 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.

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

[20] Satoshi Oyama, T.K., Toru Ishida, Teruhiro Yamada, Yasuhiko Kitamura, *Keyword Spices: A New Method for Building Domain-Specific Web Search Engines.* the 17th International Joint Conference on Artificial Intelligence, 2001.

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

[22] Weld., C.K.a.D. *Palnning to gather information.* in *14th National Conference on AI*. 1996.

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