Approximate SPARQL for Error Tolerant Queries on the DBpedia Knowledge Base

Gregory Tauer, Ronald Rudnicki, Moises Sudit
CUBRC
Buffalo, NY

Abstract— The Resource Description Framework (RDF), a language for describing resources, is being used more commonly in information fusion systems. SPARQL is a standard query language that enables knowledge extraction from data encoded in RDF. A SPARQL query is, in essence, an exact subgraph matching problem. Unfortunately, many of the techniques that produce data in RDF (such as manual data entry, social network analysis, natural language processing, etc.) make annotation mistakes, which result in dirty RDF data. SPARQL performs suboptimally on RDF data containing errors since, as an exact graph matching tool, it is not designed to cope with noisy data. To improve knowledge extraction under these conditions, we propose an extension to SPARQL that permits approximate graph matches. This allows queries to cope with errors in the RDF graph, both on the attribute level (such as misspelled names) as well as on the structural level (missing or extra edges). We use the TruST heuristic algorithm to solve the underlying approximate graph matching problem and demonstrate the benefit it brings to answering questions on the DBpedia knowledge base.

Resource Description Framework; SPARQL; Approximate Graph Matching, TruST; DBpedia

I. INTRODUCTION

Information fusion systems deal with large quantities of heterogeneous data. To be useful, the knowledge present in such data must be processed, and processing the data requires a robust framework for storage, organization, and retrieval. The Resource Description Framework (RDF) [1] together with the linked data paradigm [2], which uses RDF as a base, is an increasingly popular framework of choice for such systems.

RDF-compliant data can be interpreted as a graph. A variety of objects and domains are naturally represented as graphs, for example social network analysis and road systems. Such domains are obvious candidates for being described with data structured according to the RDF model. The advantage of the RDF data model is that it is generic, flexible, and capable of representing data from any domain even those which may not initially appear suitable. Some examples include digital images [3], robot navigation [4] and natural language [5]. Many tools which may produce a graph as output can be adapted to instead produce RDF instance data (since RDF instance data is also a graph).

The standard query language for the retrieval of data expressed in RDF is SPARQL [6]. SPARQL queries are expressed as a description of a graph, where some variables of interest are linked to other variables of interest by edges. Execution of a SPARQL query closely resembles the problem of graph matching, especially the problem of subgraph isomorphism, which we will refer to as exact graph matching. As with exact graph matching, SPARQL relies on clean data. Although SPARQL contains a number of features which make it flexible to some inconsistency, the data to be queried must be largely free from noise and errors. This requirement is in conflict with the reality that data processed by information fusion systems almost always contain noise. Often times this noise is the result of imperfect sensors and data processing but may also result from incomplete data. Regardless of its source, searching for results in noisy data is an important problem for information fusion as it improves a system’s ability to find relevant results.

This problem of searching for information on noisy data is illustrated by the DBpedia knowledge store, which is an “…effort to extract structured information from Wikipedia…” [7]. The DBpedia project extracts knowledge from Wikipedia using a variety of template and other rule based approaches. After extraction, the knowledge is stored as linked RDF data. DBpedia covers a broad range of topics. In this paper we will focus on identifying nuclear power plants that are notable risks due to natural disasters and their proximity to population centers.

DBpedia contains multiple layers of error and incompleteness. In practice, any errors (including omissions) from the infoboxes on Wikipedia will be reflected in DBpedia, and we will show that this increases the difficulty of running queries against the data.

The main contribution of this paper is to discuss the various options available in the SPARQL language to help adapt to errors in a specified query or in data using the RDF model, and to compare these options against approximate graph matching. Ultimately we will show that, while SPARQL is an extremely powerful query language, in some circumstances aspects of inexact graph matching can be used to extend the ability of standard SPARQL to find important results in noisy data.

This work differs from other work that has studied graph matching on semantic databases [8,9], in that our work does not focus on advancing the state of inexact graph matching, but rather on the techniques available through SPARQL for matching inexactly and how they are different from, and complement, inexact graph matching.
The remainder of this paper is organized as follows. Section II contains a review of the Resource Description Framework, SPARQL, and Graph Matching. Included in Section II are discussions of how SPARQL and graph matching cope with noisy data. Section III contains a selection of SPARQL and graph matching queries run against DBpedia, their results, and a discussion on these two related tools can work together to improve search results.

II. BACKGROUND

This section reviews the Resource Description framework (A), including SPARQL which is commonly used to query it (B). Both exact and inexact graph matching (C) are also discussed as an alternative to SPARQL for querying noisy and incomplete graph data.

A. Resource Description Framework

Resource Description Framework (RDF) is designed to be a language for making assertions about resources, where a resource is literally anything whatsoever, an image on a web page, the title of a journal article, a human being, a city government, a mathematical formula or a spatial relationship between parts of a machine. (cf. definition of “Resource” in [10]).

When data is represented according to the rules of RDF it takes the form of a triple composed of subject, the resource being described, predicate, the specific aspect, characteristic, attribute, or relation used to describe the subject and the object, the resource, or literal, that describes the subject. The rules of RDF require also the use of Unique Resource Identifiers (URIs) or blank nodes (an identifier that has a scope local to the source in which it is used) to act as representatives of the resources a given triple is about. The specification is such that subjects can be either URIs or blank nodes, predicates must be URIs, and objects can be either URIs, blank nodes or literals. Examples of literals being any of the standard XML schema datatypes: string, datetime, integer, decimal, etc. An example of an RDF triple expressing the fact that the Indian Point Energy Center is located in Buchanan, New York is given by Table I. Where “dbpedia:” is the namespace prefix of http://dbpedia.org/resource and dbpprop: is the namespace prefix of http://dbpedia.org/property.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Predicate</th>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia: Indian Point Energy Center</td>
<td>dbpprop: locale</td>
<td>Buchanan, New York</td>
</tr>
</tbody>
</table>

Generally, a namespace is any unique grouping of identifiers or symbols, but in contexts where RDF is used, a namespace will more often than not refer to a vocabulary of terms that are used to describe some domain of discourse. Sometimes the namespace will refer to an ontology, a vocabulary in which the terms used to denote the entity types of the domain are arranged in a taxonomy (a class hierarchy based upon the subsumption relation (e.g. Reactor is a Structure)) and the terms have both a human-readable and a machine processable definition, enabling machine based reasoning (e.g. Indian Point Energy Center is of type Reactor, Reactor is a Structure, therefore Indian Point Energy Center is of type Structure). Where RDF provides a syntactic structure of data, ontologies add the additional layer of semantics.

The great advantage of an RDF based store of data over that of other storage methods is that it makes use of no database schema other than the generic triple form of subject, predicate and object. One does not need to create some physical data model as would, for example, be needed in a relational store and which would then need to be altered as new sources of data were added to the store. Functions such as search that have been traditionally facilitated by schemas are instead facilitated by vocabularies and ontologies in an RDF knowledge store. A resource (represented by a URI) is annotated with terms from one or more vocabularies and ontologies and then related (via predicates that are also associated with such terms) to other annotated resources. The vocabularies and ontologies can change without requiring a change to the structure of the data.

Increasingly, the advantages of RDF make it the data model of choice for large knowledge stores especially those that span multiple domains of interest and are therefore described using multiple ontologies. DBpedia is one such knowledge store the scale of which is described on the project’s wiki:

“The English version of the DBpedia knowledge base currently describes 3.77 million things, out of which 2.35 million are classified in a consistent Ontology, including 764,000 persons, 573,000 places (including 387,000 populated places), 333,000 creative works (including 112,000 music albums, 72,000 films and 18,000 video games), 192,000 organisations (including 45,000 companies and 42,000 educational institutions), 202,000 species and 5,500 diseases.” [8].

Not only does DBpedia contain a significant portion of the data from Wikipedia, it provides links from that data to data from other websites as well making it one of the important information hubs of the internet [11]. Querying DBpedia provides an experience of the vision of the semantic web in which data from different locations is linked together in ways that are tractable to a machine. Querying using the SPARQL standard also provides an experience of how difficult it is to have a data store of such vast proportions be completely arranged according to exact patterns. The use of the TruST heuristic algorithm as a solution for the underlying approximate graph matching problem is thus an effort to further the value of the semantic web.

B. SPARQL

Query languages enable information extraction from knowledge bases and such languages are specific to the data model for which they are designed. RDF is no different in this regard and the standard query language for RDF is SPARQL Protocol and RDF Query Language (SPARQL). Queries in SPARQL that select information from RDF-based knowledge stores do so by containing a set of one or more triple patterns in which any of the subject, predicate, object can be a variable. A SPARQL query returns all of the triples from the knowledge base that match the patterns of the query. For example, the simple query:
SELECT ?location WHERE {
  dbpedia:Indian_Point_Energy_Center dbpprop:locale ?location .}

Returns http://dbpedia.org/resource/Buchanan,_New_York for the value of the variable ?location. The SPARQL query returns this value because the only the triple from Table I in the DBpedia knowledge store matches the triple expressed in the WHERE clause of the query. At the time of this writing, DBpedia makes available a public SPARQL endpoint at http://dbpedia.org/sparql where interested readers may verify this result.

It is useful to see SPARQL queries as graph matching problems that are “solved” by the SPARQL query engine. For example the query:

SELECT ?location ?subdivision WHERE {
  ?location dbpprop:subdivisionName ?subdivision .}

can be thought of as a request to the query engine to find all graphs within the knowledge store that match the pattern in Figure 1.

![Figure 1: Query as a graph](image)

There are three matches to this pattern depicted in Figure 2.

![Figure 2: Query pattern matches in data as a graph](image)

The result set of the query contains the values from these three matching graphs that were requested in the SELECT clause of Table II.

<table>
<thead>
<tr>
<th>?location</th>
<th>?subdivision</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia:Buchanan,_New_York</td>
<td>dbpedia:Westchester_County,_New_York</td>
</tr>
<tr>
<td>dbpedia:Buchanan,_New_York</td>
<td>dbpedia:United_States</td>
</tr>
</tbody>
</table>

The full SPARQL query language is well suited to forming complex graph patterns that will retrieve data from a knowledge store if that pattern exists within the store. If it is limited at all, it is limited in allowing inexact searches, that is, graph patterns that are close to, but not matches of the pattern in the WHERE clause of the query. This is not to say that SPARQL has no such capability.

As mentioned, the rules of RDF allow the object of a triple to be a URI, a blank node or a literal. SPARQL queries can be created that make use of standard numeric and text functions that return a range of literal values rather than an exact value. For example the following query requests 25 places that are located between 39 and 43 degrees latitude North.

SELECT ?facility ?latD WHERE {
  ?facility dbpprop:latD ?latD .
  FILTER (?latD > 39 && ?latD < 43) .
  ?facility dbpprop:latNs ?latNs .
  FILTER regex(?latNs, "N") }

The OPTIONAL keyword provides another way within the SPARQL standard to search for inexact matches. Using the OPTIONAL keyword in a query forms a result set that includes all requested data if it is available and some portion of that data if it is not all available. For example, the following query returns all resources annotated as being a power station and the reactor type if one is associated. Of course, not every power station will have a reactor (e.g. wind farms) and others that should have a reactor type specified may not due to a mistake in annotation. Without the OPTIONAL keyword the query would only return nuclear power plants that have an associated reactor type.

SELECT ?powerstation ?reactorType WHERE {
  ?powerstation rdf:type dbpedia-owl:PowerStation .
  OPTIONAL {?powerstation dbpprop:reactorType ?reactorType} .
}

It must be said that we don’t claim to have provided an exhaustive list of the ways in which SPARQL might be used to perform inexact graph matching. For example, we have not considered some of the external libraries that contain functions that can be used within a SPARQL query. Still we are confident that the types of matching that can be performed easily with an inexact graph matching heuristic algorithm would be very difficult if not impossible to mimic using only the SPARQL language.

C. Graph Matching

The problem of graph matching is to search for a correspondence between two graphs. In exact subgraph matching (subgraph isomorphism) the problem is to search for
instances of a given template graph in a second, larger, data graph [12]. A template graph is said to have an exact match to a data graph if the template graph is isomorphic to some subgraph of the data graph.

In exact graph matching it is required that the template graph match a subgraph of the data graph exactly; both the structure of the graphs and attributes of nodes and edges must be exactly the same. Because of this, exact graph matching has limited opportunity to accommodate errors in the template or data graphs.

In its most basic form, evaluating a SPARQL query is equivalent to exact graph matching. This can be seen in the example of Figures 1 and 2. Additional features of SPARQL can be thought of as “add-ons” to exact graph matching. For example, the FILTER and OPTIONAL keywords require no modification to the matching algorithm. FILTER can be applied to the matching results to limit the results presented to a user following the filter criteria. OPTIONAL can be applied after matching to enrich results when information is available.

Exact graph matching relies on clean and complete data. When graph data does contain noise, an exact graph matching algorithm (and SPARQL by extension) may fail completely. Inexact graph matching allows a less extreme than exactly isomorphic or not isomorphic comparison of how similar two graphs are, and this concept can be usefully extended to SPARQL results.

With inexact graph matching, two graphs are no longer classified as the same or not the same, but rather placed on a continuous scale of how similar they are. Graph similarity is intended to capture how closely both the structure and attributes of two graphs match. In support of this, it is common to express the similarity between nodes (edges) from the template graph to nodes (edges) from the data graph as a numeric value and find a match using some objective function which favors selecting matches with high similarity. These similarity measures can be chosen following a probabilistic interpretation [13], however we will focus on manually selected similarity measures. The quality of a graph match is defined to be the sum of the similarities of all elements that are matched between the template graph and the data graph.

All results presented here retrieved using an implementation of the TruST heuristic for approximate graph matching [14]. The details of TruST are beyond the scope of this work. We have found TruST to be a good heuristic, however there is nothing unique about its use for this application; any heuristic for inexact graph matching could be substituted in its place.

Syntaxically, we use the “APPROXIMATE” keyword to indicate when a query should be solved using inexact graph matching. Note that this is not part of the standard SPARQL specification and so will not work with publicly available SPARQL tools. By default, two edges are defined to have similarity 0.1 if their types match exactly, otherwise they have similarity 0.0. Two nodes are defined to have a similarity of 0.1 if they are either both properties or both not properties and 0.0 otherwise. This causes the system to avoid matching property values (which we consider as a type of node) with standard nodes. We additionally use the following three similarity scorers to design queries. The use of a scorer for a node is designated by a “SIMIALRITY node scorer.” (see examples in Table III).

### Table III: Similarity scorers that are used by the inexact matcher.

<table>
<thead>
<tr>
<th>Scorer Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exact_URI_Match</td>
<td>Returns similarity of 1.0 if both the template and data nodes are not property values and their URIs matches exactly, otherwise returns 0.0.</td>
</tr>
<tr>
<td>Value_String_Contains(St)</td>
<td>Returns similarity of 1.0 if both the template and data nodes are property values and the data graph node’s value string contains the string St, otherwise returns 0.0.</td>
</tr>
<tr>
<td>Soft_Greater_Than(UL, LL)</td>
<td>Returns similarity of 0.0 if the template and data nodes are not both numeric property values. Otherwise, returns 0.0 if the data node’s value is less than LL, 1.0 if the data node’s value is greater than UL, and <a href="UL-LL">UL-(data node’s value)</a> if the data node’s value is between LL and UL.</td>
</tr>
</tbody>
</table>

To illustrate one advantage of inexact graph matching, consider the following SPARQL query to find cities with a population greater than 3 million in the United States and results in Table IV.

```sparql
```

This query enforces a strict cutoff at 3 million total population. In some cases, a softer limit which will permit some values slightly less than 3 million may be desired. This is easily accomplished with inexact graph matching, by specifying a similarity between population values that tapers off, for example the following query tapers linearly starting at 3 million and ending at 2 million.

```sparql
```

### Table IV: Query results for cities with population > 3 million.

<table>
<thead>
<tr>
<th>City</th>
<th>Population</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia:Los_Angeles</td>
<td>3792621</td>
</tr>
<tr>
<td>dbpedia:New_York_City</td>
<td>8244910</td>
</tr>
</tbody>
</table>

### Table V: Inexact query results for cities with population > 3 million.

<table>
<thead>
<tr>
<th>City</th>
<th>Population</th>
<th>Similarity</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia:New_York_City</td>
<td>3792621</td>
<td>100%</td>
</tr>
<tr>
<td>dbpedia:Los_Angeles</td>
<td>8244910</td>
<td>100%</td>
</tr>
<tr>
<td>dbpedia:Chicago</td>
<td>2695598</td>
<td>91%</td>
</tr>
</tbody>
</table>
The approximate version of this query captures Chicago, even though its population is less than the original 3 million. Similarity is expressed as a percent of the maximum possible similarity score (for example, Chicago here has a similarity of 3.096 out of a possible 3.4).

Inexact graph matching is also less strict in handling missing links in the data. Revisiting the example from Section II.B concerning the type of a power station’s reactor, with an inexact graph matching tool this query could be expressed without the OPTIONAL keyword.

<table>
<thead>
<tr>
<th>?powerstation</th>
<th>?reactorType</th>
<th>Sim</th>
</tr>
</thead>
<tbody>
<tr>
<td>dbpedia:</td>
<td>Pressurized_water_reactor</td>
<td>100%</td>
</tr>
<tr>
<td>Koeberg_Nuclear_Power_Station</td>
<td>dbpedia:</td>
<td>Pressurized_water_reactor</td>
</tr>
<tr>
<td>Stade_Nuclear_Power_Plant</td>
<td>dbpedia:</td>
<td>Pressurized_water_reactor</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>dbpedia:</td>
<td>Heavy_water_reactor</td>
<td>61%</td>
</tr>
<tr>
<td>Karachi_Nuclear_Power_Plant</td>
<td>dbpedia:</td>
<td></td>
</tr>
<tr>
<td>Offshore_Wind_Farm</td>
<td>dbpedia:</td>
<td></td>
</tr>
</tbody>
</table>

As with the original (exact SPARQL) query, this query will return a list of power stations. The difference with approximate matching in this case is reflected by the ordering of results. In this query, power stations that have a reactor type of pressurized water reactor will be preferred over stations with other reactor types, which will themselves be preferred over reactors with no listed reactor type. This is because approximate graph matching prefers results which have larger sum of similarity over the matched elements. In the reactor type example, matches with reactor types will be preferred since there are more elements in the total similarity summation.

### III. COMPREHENSIVE EXAMPLE

We illustrate both good and bad aspects of approximate graph matching through example by searching for nuclear power plants that are located in a county with a large population and on a river which has a dam. We do not intend to imply that the resulting matches are at high risk- this exercise is presented for illustrative purposes only. Readers interested in a comprehensive risk analysis should refer to the Nuclear Regulatory Commission at “www.nrc.gov”.

The query is motivated by a recent occurrence of river flooding at the Fort Calhoun Nuclear Station on the Missouri River in Nebraska which led to substantial damage [15]. Dams are often used to control the flow of water through rivers, so clearly the failure of a dam upstream of a power plant may lead to flooding at that plant. In the case of the Fort Calhoun Nuclear Station, flooding was exacerbated by the Garrison Dam becoming overwhelmed by heavy rain.

The first result column exemplifies the intended match of the query. The Indian Point Energy Center is located on the Hudson River which is also the location of the Federal Dam. Furthermore, the Indian Point Energy Center is located in Buchanan, New York, which is itself located in Westchester County. This county has a population of 949,113 which is less than the ideal target of 1 million, but closes enough to yield a good match.

The next result column is very similar to the first, except the population is incorrectly matched with Yorktown Heights, New York. This error is a side effect of inexact graph matching’s flexibility. There is no hard constraint that population must be a numeric value. A similar occurrence is seen in the third results column for the Columbia Generating Station. This station is correctly indicated as being located on the Columbia River, however the matched dam is not itself a dam, but rather a propaganda song about dams on the Columbia River. Unlike the error of the second result (where population was matched with Yorktown Heights), this result could be interpreted as an interesting success of approximate matching.

The last matching result is also linked to a dam in an initially unintended fashion. Although there is no direct link between the Limerick Nuclear Power Plant and the Schuylkill River in DBpedia (although this power plant is, in truth, located on that river), there is a link to the Black Rock Dam (which is also located on the Schuylkill River) through a collection of buildings and structures in Montgomery County. This example shows that inexact graph matching can, in some cases, help overcome unexpected omissions in data.
Conspicuously absent from these results is the motivating example of the Fort Calhoun Nuclear Station and the Garrison Dam. Unfortunately there is currently no clear link in DBpedia between the Fort Calhoun Nuclear Station and the Missouri River. Furthermore, the Fort Calhoun Nuclear Station is located in a different county from the Garrison Dam, which prevents the match that was possible for the Limerick Nuclear Power Plant and Black Rock Dam. Due to the absence of these links, querying for the Fort Calhoun and Garrison Dam relationship does not return a result, illustrating that it is possible to use inexact graph matching to overcome errors and omissions only to a point.

### Table VII: Results from the power station and dam query.

<table>
<thead>
<tr>
<th>?Power Station</th>
<th>Indian_Point_Energy_Center</th>
<th>Indian_Point_Energy_Center</th>
<th>Columbia_Generating_Station</th>
<th>Dresden_Generating_Station</th>
<th>Limerick_Nuclear_Power_Plant</th>
</tr>
</thead>
<tbody>
<tr>
<td>?River</td>
<td>Hudson_River</td>
<td>Hudson_River</td>
<td>Columbia_River</td>
<td>Illinois_River</td>
<td>Buildings_and_structures_in_Montgomery_County_Pennsylvania</td>
</tr>
<tr>
<td>?Dam</td>
<td>Federal_Dam_(Troy)</td>
<td>Federal_Dam_(Troy)</td>
<td>Grand_Coulee_Dam_(song)</td>
<td>Starved_Rock_Lock_and_Dam</td>
<td>Black_Rock_Dam_(Schuylkill_River)</td>
</tr>
<tr>
<td>?County</td>
<td>Westchester_County_New_York</td>
<td>Westchester_County_New_York</td>
<td>Stateline_Wind_Farm</td>
<td>LaSalle_County_Nuclear_Generating_Station</td>
<td>-</td>
</tr>
<tr>
<td>?Population</td>
<td>949113</td>
<td>Yorktown_Heights_New_York</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Similarity 96% 83% 79% 79% 38%

The examples in Section II.B demonstrate that SPARQL is a rich language capable of handling errors, inconsistencies, and omissions in data in a variety of ways. Two of these ways which we discussed are the OPTIONAL and FILTER keywords. Although these keywords each give access to a powerful tool, they also carry a number of limitations which can be addressed, in part, by inexact graph matching.

The OPTIONAL keyword is primarily used to allow additional information to be returned with a query when such information is available, without requiring the additional information for a match. This is in contrast to inexact graph matching, where all elements of the query are implicitly assumed to be optional. Although optional, inexact matches are preferred when they have a larger number of matching elements. This is illustrated by the example of Table VI.

FILTER is used to restrict results which contain unwanted attribute, or property values. A query match is excluded if it does not satisfy the filter. A similar, but less extreme, behavior is present in inexact graph matching, where instead of a filter the query can specify a similarity function. Results with lower similarity will not be excluded, but will appear lower in the list of results than matches which have high similarity, as shown by the example of Table V.

Although we have largely discussed them separately, these facets of graph matching interact in complex ways. Consider the example of Table VI. Highly matching results to this query are nuclear power stations with a pressurized water reactor, but stations with any listed reactor type also appear, however at lower levels of similarity. Finally, the example of Section V shows how these various cases can work together to provide interesting results to complex queries which would otherwise match very few cases in real data.

### V. CONCLUSION

The SPARQL is a decidedly powerful and robust query language that shares basic functionality with graph pattern matching. We proposed an extension to SPARQL to allow inexact graph matching and showed how, in some cases, this can lead to an improved capacity for handling errors in the data.

### ACKNOWLEDGMENT

This material is based in part upon work supported by the Intelligence Advanced Research Projects Activity (IARPA) via Air Force Research Laboratory contract number FA8650-10-C-7062. The U.S. Government is authorized to reproduce and distribute reprints for Government purposes notwithstanding any copyright annotation thereon. Disclaimer: The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of IARPA, AFRL or the U.S. Government.
REFERENCES


