Fact Repository Population for Intelligent Agents

Benno Lee

**Abstract**

**Section 1. Introduction**

The primary mechanism for intelligent agent reasoning using natural language is the generation and manipulation of Text Meaning Representation (TMR) objects [McShane Unpublished]. In order to populate the information these TMRs encapsulate, a fact repository needs to be maintained of the long term information required by the agent to simulate perfect long term memory. This is currently implemented in this project with a database accessing information for restaurants using the CityGrid API [CityGrid 2013]. However, not all the information on a restaurant can be gleaned from the CityGrid database and so many of the entries within the project ontology are left blank. These fields were also deemed non-essential to the current requirements of the project.

The database used in the project is a SQLite3 database, implemented through the Python module packaged with Python 2.7. The primary motivation behind this decision was that SQLite provides a SQL-like interface to database manipulation, but does not require a server in order to provide access. In addition, the scope of the project is relatively small compared to most production level projects and thus only requires a small database. The query language is also relatively small and simple, resulting in a small learning curve [SQLite 2013].

Restaurant data was retrieved using the CityGrid Places API with the query restricted to restaurants within the Troy, NY, city area. Due to the structure of the query API, a detailed query must be made for each restaurant to appropriately acquire all the pertinent information. The primary data stored within the detailed query is the category tags that CityGrid provides with each result.

In the rest of this paper, I outline the table organization (Section 2), unique challenges in populating business hours (Section 3)

**Section 2. Table Structure**

There are three types of mappings between values within the database: one-to-one, one-to-many, many-to-many. One-to-one mappings are very simple to model as the values should be entered on the same row in the table. An example of this would be the restaurant and its phone number. For values such as location, a second table was provided to encapsulate a few values together into a grouping.

One-to-many mappings require a second table that holds individual entries for the “many” values with a column in the table that refers to row id of the “one” value. This improves the run-times of joins made between these tables. This is required for entries such as the business hours of a restaurant. The schedule table will have an entry for each day that a restaurant is open and a value for the restaurant id that the hours correspond. It is interesting to note that the schedule table has a field to determine whether a schedule corresponds to a human's day or to a restaurant's day. This is because human schedule and restaurant schedules share the same structure but may not share independent row\_ids.

Many-to-many mappings require a third table map the rows of two tables together. In order to map restaurants to categories, three tables are needed. One table each is required for restaurants and categories, however, since a restaurant can have many categories and a category may be attributed to many restaurants, a third table is needed to keep track of their pairings.

For a complete outline of tables and columns, see the RestDef.txt file in TMRMaker/Knowledge Database/citygrid/ directory.

**Section 3. Business Hours**

One of the major challenges of the values within the database creation is the population of business hours. The format for business hours is not standardized within the CityGrid API. Andy took care of the particulars in parsing the text into floats as his parsing language is better at handling text manipulation than any of the languages available to me. We agreed to represent the business hours as floats on the domain of [0, 48] since there are restaurants which are open until, perhaps, 1 or 2 am in the morning the next day. The writer feels that this more appropriately models human perception of time spans as being continuous. Furthermore, the values are stored as floats instead of text because doing so will allow simpler comparisons or calculations done with the business hours.

**Section 4. Summary of Personal Contribution**

The resulting database is the primary commitment for this project. Since the entries usually only require addition only once, the addition process is usually taken care of interactively. This is unlike the rest of the project, which is dynamically generated each time the program is run. The makedb.py script creates the LEIA database and the matching tables. It is useful to use it as a reference to determine what columns are within the database. Additionally, the search.py script is a modified version of the sample citygridv2 demonstration script. It will grab fifty results and insert them into the database. It is currently hard-coded to pull a specific page or sub-set of 50 results. To make this script scalable, it is simple to modify it to cycle through all the available pages. Finally, the business hours python script takes in as input, a text file where every three lines are the opening hours, closing hours, and break symbol, respectively. Opening hours and closing hours are represented as seven floating point numbers separated by a “, “. Since each set corresponds to a specific restaurant, it is easy to hard-code the restaurant id into the business hours.

My primary job was to create and maintain the database. The primary tasks I performed were that all entities required by the program were defined and available within the database. I wrote several scripts to generate the database and to populate it with values. I made sure that the entities corresponded with the fields attributed to them in the ontology. I searched for and pulled values from the CityGrid database using their API. Since most of the work need only be done once, the work was largely not redundant.

**Section 5. Results**

The primary complexity in creating the fact repository is the proper modeling of the tables and their relationships. Since most of the entries of the database will not need to be changed (type-wise), most of the population can be performed interactively. A complexity of the CityGrid API means that many queries need to be made to the server in order to get the appropriate amount of information which needs to be addressed with appropriately placed timeouts. The business hours are difficult to model because of inconsistent formatting. However, a 48-hour schedule is necessary to represent time spans that restaurants are open that do not neatly fit within a single day.

**References**

1. CityGrid Developer Center. November 25, 2013. Places API. Retrieved from http://docs.citygridmedia.com/display/citygridv2/Places+API.

2. McShane, M., Nirenburg, S., Beale, S. *Meaning-Centric Language Processing*. Unpublished.

3. SQLite. November 25, 2013. SQL As Understood By SQLite. *SQLite*. Retrieved from http://www.sqlite.org/lang.html