Entering the spoken dialog challenge
Research report

Msc in Artificial Intelligence

MERIGAUD Nicolas
Supervisor: Oliver Lemon
4/17/2010
Statement of Non-Plagiarism

i. By this statement, I declare that this submitted work is all my own work and is in my own words

ii. The sources I referred to and from which I have quoted or from which I got a substantial amount of information directly relevant to this work are fully listed in the references section of this document. Moreover I have not handed in an essay, paper or thesis with similar content elsewhere.

iii. I discussed the project and shared ideas with the following people:
   • Helen Hastie, Interaction Lab, MACS, Heriot Watt University
   • Oliver Lemon, Interaction Lab, MACS, Heriot Watt University
   • Xingkun Liu, Interaction Lab, MACS, Heriot Watt University

Name:

Date:

Signature:
Abstract:

Spoken dialog systems are more and more widely used. There is a range of different applications for them, starting from simple command based agents like the one you would have in your voicemail, to more complicated conversational agents. We tend to produce better and better command-based dialog systems but it is still far from being perfect. Representing human language in a computer language is a tough task, building systems that can understand human language and speak human language is much tougher. In most cases we must restrict the language used by the user and the system in order to build a simple but effective dialog system.

This document will cover two different aspects of spoken dialog systems. The first part will be describing the 2010 spoken dialog challenge, where we had to design and develop a dialog system providing bus information for the city of Pittsburgh, USA. The second part will introduce a developed evolutionary dialog manager and compare it against a simple hand written dialog manager.
# Table of contents:

1. **Introduction:**
   
2. **Literature review**
   
   2.1 Spoken dialog systems, an introduction:
   
   2.2 Spoken dialog systems, basic architecture:
   
   2.3 Spoken dialog systems, an insight into each component:
      
      2.3.1 Automated Speech Recognition
      
      2.3.1.1 Definition
      
      2.3.1.2 Techniques:
         
         2.3.1.2.1 N-gram models:
         
         2.3.1.2.2 The Noisy channel model:
      
      2.3.2 Natural language understanding
      
      2.3.3 Dialog management
      
      2.3.4 Natural language generating
      
      2.3.5 Text-To-Speech synthesis
   
   2.4 The DUDE system
      
      2.4.1 VoiceXML
      
      2.4.2 DUDE’s basic architecture
      
      2.4.3 Developing spoken dialog systems using DUDE
         
         2.4.3.1 DudeADT: the development environment
         
         2.4.3.2 Deploying a dialog system
   
   2.5 Evaluating spoken dialog systems
   
   3. **Bus Information System**
      
      3.1 A word on the Let’s go challenge
      
      3.2 System’s scope and specifications
         
         3.2.1 Initial System’s specifications:
         
         3.2.2 Initial dialog flow:
         
         3.2.3 Final System’s specifications:
         
         3.2.4 Final dialog flow:
      
      3.3 Software Design:
         
         3.3.1 Database design:
         
         3.3.1.1 Original database:
         
         3.3.1.2 Database redesign:
         
         3.3.2 System design:
         
         3.3.2.1 Top level layer:
         
         3.3.2.2 Sub-level layer:
   
   3.4 Software development
      
      3.4.1 Database implementation:
      
      3.4.2 Dialog manager implementation:
### Table of figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Spoken Dialog Systems architecture</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Bi-grams probabilities table</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>The noisy channel model</td>
<td>8</td>
</tr>
<tr>
<td>4</td>
<td>bus frame example</td>
<td>9</td>
</tr>
<tr>
<td>5</td>
<td>Semantic grammar representation</td>
<td>9</td>
</tr>
<tr>
<td>6</td>
<td>Semantic grammar interpretation</td>
<td>9</td>
</tr>
<tr>
<td>7</td>
<td>a sample of the UNISYN lexicon</td>
<td>12</td>
</tr>
<tr>
<td>8</td>
<td>VoiceXML architecture</td>
<td>13</td>
</tr>
<tr>
<td>9</td>
<td>VoiceXML, an example</td>
<td>14</td>
</tr>
<tr>
<td>10</td>
<td>The DUDE Architecture</td>
<td>15</td>
</tr>
<tr>
<td>11</td>
<td>City-Guide top level BPM</td>
<td>16</td>
</tr>
<tr>
<td>12</td>
<td>Restaurants BPM</td>
<td>17</td>
</tr>
<tr>
<td>13</td>
<td>database relation and information presentation for the restaurant subtask</td>
<td>18</td>
</tr>
<tr>
<td>14</td>
<td>Cuisine subtask</td>
<td>18</td>
</tr>
<tr>
<td>15</td>
<td>Architecture of a simple deployed dialog system</td>
<td>20</td>
</tr>
<tr>
<td>16</td>
<td>Maximizing user satisfaction using PARADISE</td>
<td>21</td>
</tr>
<tr>
<td>17</td>
<td>Initial Call flow of the system</td>
<td>27</td>
</tr>
<tr>
<td>18</td>
<td>Final call flow of the system</td>
<td>29</td>
</tr>
<tr>
<td>19</td>
<td>Initial database structure</td>
<td>31</td>
</tr>
<tr>
<td>20</td>
<td>single table structure</td>
<td>31</td>
</tr>
<tr>
<td>21</td>
<td>Insert statements for a bus run</td>
<td>33</td>
</tr>
<tr>
<td>22</td>
<td>Top level start node</td>
<td>35</td>
</tr>
<tr>
<td>23</td>
<td>top level buses bpm</td>
<td>36</td>
</tr>
<tr>
<td>24</td>
<td>sub level buses bpm</td>
<td>37</td>
</tr>
<tr>
<td>25</td>
<td>Synonyms in DUDE</td>
<td>38</td>
</tr>
<tr>
<td>26</td>
<td>Removing unwanted routes</td>
<td>39</td>
</tr>
<tr>
<td>27</td>
<td>database pre-processing, main loop</td>
<td>40</td>
</tr>
<tr>
<td>28</td>
<td>dialog system test cases</td>
<td>42</td>
</tr>
<tr>
<td>29</td>
<td>extract of dialogActs.xml</td>
<td>45</td>
</tr>
<tr>
<td>30</td>
<td>Tree view of the system acts graph</td>
<td>46</td>
</tr>
<tr>
<td>31</td>
<td>Matrix view of the system acts graph</td>
<td>47</td>
</tr>
<tr>
<td>32</td>
<td>matrix view of the user model</td>
<td>48</td>
</tr>
<tr>
<td>33</td>
<td>Loggers class diagram</td>
<td>49</td>
</tr>
<tr>
<td>34</td>
<td>SystemData class diagram</td>
<td>50</td>
</tr>
<tr>
<td>35</td>
<td>evolutionaryDM class diagram</td>
<td>52</td>
</tr>
<tr>
<td>36</td>
<td>the checkForConflicts() method</td>
<td>59</td>
</tr>
<tr>
<td>37</td>
<td>folder structure</td>
<td>62</td>
</tr>
</tbody>
</table>
1 Introduction:

Developing spoken dialog systems is not an easy task to do. They usually rely on several different modules, each one of them being as important as the others.

The 2010 spoken dialog challenge is a challenge organised by Carnegie Mellon University, Pittsburgh which aims to compare different dialog systems of the same task in order to investigate our strengths and weaknesses in the domain as well as different evaluation techniques.

As we will see later on, this year challenge’s task was to build a spoken dialog system that is giving bus information schedules. 4 research labs were taking part in the challenge and evaluations have been performed on site by CMU.

This thesis will be separated into 2 main parts:

- The first one will emphasize on the spoken dialog challenge and its developed system, including testing and evaluations
- The second one will focus on another system that has been developed just after the bus information system. This system has been developed in order to compare a basic dialog manager against a newly written evolutionary dialog manager based on genetic algorithms

We will then end up this dissertation with a presentation of the evaluation results and what has been learned in the past few months.
2 Literature review

2.1 Spoken dialog systems, an introduction:

Spoken dialog systems are computer systems that interact with humans using voice on both sides. In a perfect world you would talk to a computer the same way you would with a human.

It is however a very idealistic view of the subject, indeed most spoken dialog systems are restricted to a specific domain. They commonly are classified into two main categories:

- Command based systems:
  - In this category the user will treat the system as an interface to an application; examples can go from a simple weather interface system asking for the city in which you want the forecast to more complicated interfaces like a flight booking/check-in system or in our case a bus information system.

- Conversational systems:
  - On the other hand, conversational agents can be regarded as a conversational partner. Such systems are way more difficult to produce as they need a much larger domain knowledge.

However, most systems now can be regarded as being part of these 2 categories at the same time. In other words, people try to build user-friendly spoken dialog systems that can perform more or less complicated tasks.

Spoken dialog systems are not the only dialog systems found in our everyday life. Other systems might only use the recognition and understanding parts without having to answer the user. Such systems are commonly defined as dialog systems as they still have to process an audio input in order to achieve a goal. Examples of these systems can be:

- Microsoft speech recognition: Here the voice recognition is used in order to issue commands to a computer. The computer has to recognize what the user says and understand it in order to perform the required task.

- In car speech recognition: In this case the speech recognition is also used in order to issue commands like controlling the CD-player etc ... Note that it can also be a complete spoken dialog system.
2.2 Spoken dialog systems, basic architecture:

Now that we have seen what spoken dialog systems are, we will highlight their basic architecture. Here is a diagram showing the architecture of such systems.

As we can see on figure 1, a spoken dialog system is usually made of 5 different modules:

- **The Automated Speech Recognition module**
  - Takes the user’s voice as input, and translates the utterance into a textual hypothesis.

- **The Natural Language Understanding module**
  - Parses this hypothesis into plain text and generates a semantic representation out of it.

- **The Dialog Manager**
  - These semantics are then handled by the dialog manager, in order to decide what to do next in the conversation. Once a decision has been taken, it generates new semantics and sends them to the Natural Language Generation module.

- **The Natural Language Generation module**
  - It parses these new semantics into a textual form and sends them to the speech synthesizer.

- **The Text-To-Speech Synthesis module**
  - Taking plain text as input it generates an audio output to the user.
Now, this architecture sounds natural but it can of course be different, it usually depends on the designer's appreciation.

Some different architecture might contain other modules such as discourse modelling for example, which would be contained in the dialog management module anyway.

Other ones might just decide to use a blackboard instead of a standard pipe to pass information between modules. A blackboard is basically a logic component where each module would read and write to.

Finally, the overall architecture of such a system could be reduced to only three components, where NLU and ASR would be merged, as well as NLG and TTS. Merging such components can be interesting as we may lose information by sending data from one component to another.

However, modularity is usually the answer, as modules are easier to implement when they remain small than when they are too big. Another good thing is that module’s implementation can be done by different people on the same team using different technologies and programming languages.

As long as they're fully implemented it remains easy to link them together even though the technologies used are different.
2.3 Spoken dialog systems, an insight into each component:

In this section we will focus on the different techniques used in each module. How can we understand or generate natural language, how can we recognize or generate speech. How do we know what to do after recognizing and understanding the user’s utterance?

We will define each one of these modules and highlight a few of their techniques, such as N-grams and Grammars, if necessary.

2.3.1 Automated Speech Recognition

2.3.1.1 Definition

Automated speech recognition is a technology that allows the user to give instructions to a computer system using voice rather than a keyboard for example.

This technology focuses on recognizing the words in a sentence rather than generating semantics and understanding it.

Speech recognition is quite a tricky task; indeed, its performances usually depend on how big the vocabulary is. Recognizing words on a 200 words vocabulary is obviously much easier than in a 50 000 words vocabulary. Other factors like the speaker’s accent and talking speed can alter greatly their performances.

A speech recognizer that is set up for the American-English language might have troubles recognizing Irish-English or Scottish-English for example. Also, an utterance is much easier to recognize when there is a tiny pause between each word. This is called isolated word recognition while saying words without any pause is called continuous speech recognition.

Finally another issue in speech recognition is about the users themselves. Most systems are built in for a specific user. The first time the user runs the system, a calibration occurs in order to set up the recognizer to the user’s specific voice.

That done, recognition becomes way easier for the system. However, most spoken dialog systems are built for many different users. For example a bus information system could be called by a thousand different people on the same day. Having a calibration phase is simply impossible; In this case the recognition is called speaker independent and is much more difficult to perform than in a speaker dependant domain.
2.3.1.2 Techniques:

2.3.1.2.1 N-gram models:

Before actually talking about speech recognition we need to define what is called the language model. In fact, as most speech recognition techniques are based on probabilities over a language, we need a mean to represent our language.

The English language, and any other language, being way too large to represent, language have to be modelled in more or less restricted domains in order to get a fast and reliable recognition.

N-grams are probabilistic models used for word prediction. A bi-gram would be “please give”, a 3-gram “please give me”.

In other words given a sentence W with N words, N-gram models allow us to predict the next words in the sentence according to the previous ones. Let’s assume W = “I need a bus ...”; it is likely that the next word would be “from” or “to”, however “the” would probably not follow the word “bus”.

This model is called a language model.

They are essential in any noisy environment such as speech recognition, but are also very useful in Natural Language Generation as we will see later on.

So, how would we compute the probability of a sentence using N-grams?

First, to compute the probability of a word following a sequence of N words, we would need to find a large enough corpus, count the number of times we find the given sentence, count the number of times we find it with the following word and divide both values.

That seems easy but it becomes really tricky while dealing with large sentences such as “I am looking for a bus in Oakland going to Downtown early”. How would we compute the probability that early is followed by “in”?

In language, rephrasing is very common the sentence “I want a bus in Oakland going to Downtown early” does mean exactly the same as the previous one; however it will not help calculating the probability of “in” following “early”. We therefore need another method for calculating probabilities to get a sentence in a specific language model.

Using smaller N-grams seem to be the answer. In fact, Given a Corpus C, using bi-grams to compute the probability of a sentence in C becomes more reliable.

As an example; Using bi-grams, let us calculate the probability of the sentences “I want a bus” and “I want a bus in”.

We will first need the probabilities of each word in the corpus following any word in the same corpus. This is represented in figure 2; note that these are completely random probabilities and are just here to illustrate the method as they are actually not taken from any corpus.

<table>
<thead>
<tr>
<th></th>
<th>I</th>
<th>want</th>
<th>a</th>
<th>bus</th>
<th>to</th>
<th>in</th>
<th>dinner</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>0.0</td>
<td>0.65</td>
<td>0.0</td>
<td>0.0</td>
<td>0.01</td>
<td>0.05</td>
<td>0.0</td>
</tr>
<tr>
<td>want</td>
<td>0.005</td>
<td>0.0</td>
<td>0.5</td>
<td>0.05</td>
<td>0.6</td>
<td>0.15</td>
<td>0.2</td>
</tr>
<tr>
<td>a</td>
<td>0.0012</td>
<td>0.002</td>
<td>0.0</td>
<td>0.7</td>
<td>0.003</td>
<td>0.01</td>
<td>0.3</td>
</tr>
<tr>
<td>bus</td>
<td>0.001</td>
<td>0.01</td>
<td>0.001</td>
<td>0.0</td>
<td>0.45</td>
<td>0.45</td>
<td>0.002</td>
</tr>
<tr>
<td>to</td>
<td>0.001</td>
<td>0.01</td>
<td>0.05</td>
<td>0.002</td>
<td>0.0</td>
<td>0.03</td>
<td>0.006</td>
</tr>
<tr>
<td>in</td>
<td>0.01</td>
<td>0.002</td>
<td>0.6</td>
<td>0.1</td>
<td>0.0003</td>
<td>0.0</td>
<td>0.025</td>
</tr>
<tr>
<td>dinner</td>
<td>0.1</td>
<td>0.07</td>
<td>0.002</td>
<td>0.0001</td>
<td>0.25</td>
<td>0.7</td>
<td>0.0</td>
</tr>
</tbody>
</table>

This table can be read as such:

\[
P(\text{want} | \text{I}) = 0.65, \text{means that the probability of the word “want” following the word “I” is equal to 0.65. In other words, for 100 “I”s in the corpus we have 65 “I”s that are followed by the word “want”}.\]

We also need to introduce two more probability values:

- \( P(w | S) \): The probability of a word starting a sentence
- \( P(E | w) \): The probability of a word ending a sentence

Let us say that \( P(\text{“I”} | S) = 0.3, P(E | \text{“bus”}) = 0.2 \) and \( P(E | \text{“in”}) = 0.06 \)

We can now calculate the probability of both sentences using bi-grams probabilities:

\[
P(\text{“I want a bus”} E) = P(\text{“I”} | S) * P(\text{“want”} | \text{“I”}) * P(\text{“a”} | \text{“want”})
\* P(\text{“bus”} | \text{“a”}) * P(E | \text{“bus”})
= 0.3 * 0.65 * 0.5 * 0.7 * 0.2
= 0.01365
\]

\[
P(\text{“I want a bus in”} E) = P(\text{“I”} | S) * P(\text{“want”} | \text{“I”}) * P(\text{“a”} | \text{“want”})
\* P(\text{“bus”} | \text{“a”}) * P(\text{“in”} | \text{“bus”}) * P(E | \text{“in”})
= 0.3 * 0.65 * 0.5 * 0.7 * 0.45 * 0.06
= 0.00184275
\]

Thus, we can see that the first sentence is more likely to appear than the second one.
2.3.1.2.2 The Noisy channel model:

As we have seen so far, speech recognition is about taking an audio input and translating it into a text output containing a string of words.

The noisy channel model assumes that the received audio is a noisy input and should be treated as such. Basically, every possible strings of the language are compared to the noisy input and the best matching sentence is then recovered as the output. This is shown in figure 3 below.

Doing such, 2 problems may arise. How do we find the best match out of all the set of possible matches? How do we make sure the search won’t be too long?

To answer question 2, one way of doing that would be to first try to reduce the search space as much as possible and then use efficient search algorithms.

Answering question 1 is a bit more complicated, we will present here one technique used to calculate the probability of a sentence to be the input.

At first, the acoustic input can be segmented into a sequence of smaller acoustic observations:

$$ O = o_1, o_2, o_3...o_N $$

Same is done with every possible sentence:

$$ W = w_1, w_2, w_3...w_N $$

After a couple of Mathematical equation simplifications, computing the probability of a sentence being the input can be done as such:

$$ p_W = \text{argmax } P(O | W) P(W) $$

Where $P(O | W)$ is computed by the acoustic model and techniques such as Hidden Markov Models, and $P(W)$ is simply computed using N-grams as it represents the probability of a sentence in the language.
2.3.2 Natural language understanding

As we’ve said earlier, the goal of the natural language understanding component is to parse a string of words into semantics. In other words given a string of words W, the NLU would have to give meaning to this sentence in order to correctly understand it.

There are a few ways of doing such. In typical command based spoken dialog systems frames are used. Frames can be defined as a set of information that needs to be filled thanks to the input string. Here is an example:

| Bus route: |
|-----------------|------------------|
| Departure stop: | Airport          |
| Arrival stop:   | Downtown         |
| Departure time: | 10 AM            |

We can see in this frame that 3 main slots have to be filled in order to complete the bus route frame, and generally send a database query. But how do we fill in these frames?

One method would be to use what is called Semantic grammars. Semantic grammars are a kind of context-free grammar that assigns semantic to a set of words. Let us follow on this example and write one such grammar.

\[
\begin{align*}
\text{DEPARTURE TIME} & \rightarrow \text{at (one | two | ... | 12 (AM | PM))} \\
\text{DEPARTURE STOP} & \rightarrow \text{from [stop]} \\
\text{ARRIVAL STOP} & \rightarrow \text{to [stop]} \\
\text{STOP} & \rightarrow \text{Downtown | Airport | Oakland |...}
\end{align*}
\]

We can see on figure 5 that we defined a set of simple rules. For example, we would get the departure stop if we find in the input sentence the word “from” followed by the name of a stop that is included in the STOP list. In order to get the departure time, the word “at” has to be followed by a number and either “AM” or “PM”

The following sentence would be interpreted like that:

\[
\text{I want to go from Downtown to Oakland at 7 PM.}
\]

From Downtown => Departure stop
To Oakland => Arrival stop
At 7 PM => Departure Time
2.3.3 Dialog management

The dialog manager component is maybe the most important component of all. It takes a semantic representation as input from the Natural language understanding module and decides which action to take next. Upon deciding the action to perform, it sends its result to the natural language generation module.

In general, the dialog manager must answer two trivial questions:

- What to say next?
- How to say it?

Answers to these questions will greatly vary depending on the input and other parameters. If the dialog manager is sure of what the user said and all the needed slots are filled it may decide to run a database query on the different slots and present the information as an answer to the user’s need.

However it is rarely the case, as a dialog is made of several utterances. There can be two kinds of answers the dialog manager decides to take:

- Ask a question
- Present an Answer

Usually, when getting an input from the user, the recognition process gives a confidence score associated with it. This score between 0.0 and 1.0 represents how confident the system is on what has just been recognised. Dialog managers will use that number in order to decide whether to accept the facts or to try to get a confirmation by the user. Such confirmations can be of two kinds:

- Implicit confirmation:
  - An implicit confirmation may be as such: “Going to Downtown, Where do you want to take your bus? “ In that case, the dialog manager decides to acknowledge the fact that the user said he wanted to go Downtown. The user may still want to change it if it sees that the system recognized the wrong input.

- Explicit confirmation:
  - An explicit confirmation is a directed question over what the user said previously. “Did you say you wanted to go to Downtown?” Such confirmations may be used when the confidence score is too low, but they have to be used only when necessary.

In Human/Human conversations, implicit confirmations are used most of the time. Always asking if you got the information right can annoy the user very quickly and explicit confirmations have to be used with care as it can make dialog systems not as user-friendly as they should.
2.3.4 Natural language generating

The Natural Language Generation Module is here to represent what the dialog manager decided to do next. It basically takes semantics as input and will convert it into a string of words that will be sent to the Text-To-Speech synthesizer.

In command-based systems the natural language generation module may work with predefined utterances that you have to fill with values the dialog manager sends.

An example of such a template can be:

Going to [ARRIVAL STOP], where do you want to take your bus?

where [ARRIVAL STOP] is a slot in the bus route frame.

Now that is a pretty simple way of generating utterances, but it can be very effective in many command-base dialog systems.

For more complicated systems, we need a way of generating better language, for example asking question always in the same way: “Please say X” can get fairly annoying to the user.

When asking many questions, a good and simple way is to make questions shorter and shorter in time.

Finally, N-grams can also be used here to generate language. In fact, they are a good way of being sure a sentence is well formed. Using N-grams may allow us to generate multiple sentences for the same goal without having to write all of them down in basic templates.
2.3.5 Text-To-Speech synthesis

The goal of TTS synthesis is to transform a text input into an acoustic wave form, the audio output. There are two main strategies of doing such.

In simple command-base systems sentences and slots are pre-recorded, mixed together and the audio output is generated. Here is a simple example:

“Going to [destination], at what time do you want to take your bus?”

“Going to” and “At what time do you want to take your bus?” would be pre-recorded sentences while [destination] is a filled slot that is mapped to an audio representation of it.

Text-To-Speech synthesis is a field that is improving greatly, and this first method tends to be used less than it used to. The second method can be divided into two processes:

- Translation from an orthographic text to a phonetic string
- Generation of an audio wave from this phonetic string.

The first phase is usually done with a pronunciation dictionary. This dictionary contains a list of words associated with their different pronunciations. Here is an example taken from the UNISYN pronouncing dictionary, a freely available pronouncing dictionary for research purposes:

Going: \{g * ou\}.> i ng

Dictionary: \{d * i k . sh @ . n ~ e . r ii\}

Figure 7: a sample of the UNISYN lexicon

This dictionary allows us to model any kind of phonemes as well as accentuation, stress etc...

Once the string has been translated into a phonetic string, the synthesizer will finally use this phonetics in order to generate an acoustic wave of the sentence.
2.4 The DUDE system

Now that we have seen what spoken dialog systems are and how they work we will focus on a system called DUDE. It has been implemented by the University of Edinburgh from January 2008 to September 2009 and is now being held in the interaction lab, Heriot Watt University, Edinburgh.

This system is a development environment for creating spoken dialog systems. After a system has been deployed, DUDE automatically generates the Voice XML pages needed by the dialog system.

In this section we will introduce the VoiceXML format and present the most important parts of DUDE system and demonstrate its use on a city guide spoken dialog system.

2.4.1 VoiceXML

VoiceXML is a standard for developing voice applications. As it name says it is based on the XML structure, and is used to easily create voice applications.

If we were to compare it with HTML which is the standard for web browsing applications, instead of having a Keyboard+mouse as input and a screen as output VoiceXML replaces both of them with audio features.

In other words, the input becomes an audio input (the user's voice) and the output an audio signal (pre-recorded sentences or speech synthesized sentences). As we will see on figure 8, the VoiceXML voice web browser is situated on a specialised gateway fully connected to the internet.

![Figure 8: VoiceXML architecture](image-url)
Here is a sample example of how a simple VoiceXML page looks like, and how it works.

```xml
<form id="getPhoneNumber">
  <field name="PhoneNumber">
    <prompt>What's your phone number?</prompt>
    <grammar src="../grammars/phone.gram" type="application/srgs+xml" />
    <help>Please say your ten digit phone number.</help>
  </field>
</form>
```

**Figure 9: VoiceXML, an example**

In this simple example, we can see that it is a simple form like you would get in basic html. The field wanted relates to a phone number, the only difference is that instead of typing it in a web browser the user will have to say it.

The system will first prompt the user with the `<prompt>... </prompt>` property. It will basically ask his phone number. Then the field “PhoneNumber” will be filled by the user’s answer.

If the user asks for help, the system will then say what is in `<help>... </help>` and prompt again for his phone number. Allowed answers are always defined by a grammar file. Here the grammar is phone.gram and its type `srgs+xml`.

VoiceXML already provides a handful of grammars, but developers can easily create them and add them to their VoiceXML applications.
2.4.2 DUDE’s basic architecture

So, what is the relation between DUDE system and VoiceXML?

While designing spoken dialog systems with DUDE you first draw what is called BPM diagrams (Business process models), designing the point of entry of the system as well as its basic logic. Once these BPMs are created and each node defined, DUDE generates all the necessary grammar that is usually linked to a database; it then generates voiceXML pages that are hosted on a Voxeo server. We will see more details about the BPMs in section 2.4.3.1

As we can see on figure 10, the developer designs the dialog system using the development environment, this application is linked to a database, a dialog manager and a VXML generator, which defines the designed spoken dialog system.
2.4.3 Developing spoken dialog systems using DUDE

In this section we will introduce how to create spoken dialog systems using the DUDE development tool.

Note that, all images from this section are taken from “DUDE Dialogue Development Environment: A tutorial in the City Guide Domain”, Helen Hastie, Xingkun Liu, Oliver Lemon, 2010

2.4.3.1 DudeADT: the development environment

DudeADT is DUDE’s development environment. As seen before, it allows developers to link databases and business resources to a spoken dialog system.

Upon creating a new project, the developer will be able to generate more or less complicated dialog systems without having to write a single line of code or grammar. The first steps will be to link the new project with its related database, BPM’s have then to be designed. Figure 11 shows the top level of a simple dialog system.

![Figure 11: City-Guide top level BPM](image)

Here, 2 top level tasks are defined: “restaurants” and “hotels”, which mean that from the top level dialog, say: “Hello, how may I help you today?” the user will be able to ask information about either restaurants or hotels.

Figure 12 shows the BPM for the subtask restaurant.
Looking to this BPM, we can guess that the application will ask for a location and a type of cuisine. These are the slots to be filled, recall section 2.3.2 on frames and slots. Once these 2 fields have been filled the system will be able to query the database and present results to the user; in this case it would be restaurants in the city the user chose and with the type of cuisine he wanted.

However, these two diagrams are just the architecture of the dialog system but nothing have been given yet regarding how the dialog system will interact with the user, how it will get the information from the database, how they will be presented etc ...

Figures 13 and 14 will show you how to link a subtask with a database table or column and how to present the information.
Figure 13: database relation and information presentation for the restaurant subtask

Here, we can see that once starting the task, the system will prompt on “Sure let’s find you somewhere to eat” in order to acknowledge that the user wanted to get restaurants information. Once every slot have been filled, it will query the database on both fields and look for name, address, description and phone number of the resultant restaurant(s) in order to present the information as in the “pattern to Present info” field.

Figure 14: Cuisine subtask
Figure 14 shows the information to type in each of the subtasks of a particular subtask. Here we are filling the “Cuisine” subtask in “Restaurants”.

The prompt field is used to prompt the user on the type of cuisine the user is looking for. An implicit confirmation is used here to acknowledge the user and somehow make sure that the speech recognizer got it right. Implicit confirmations can be turned off/on or changed to Explicit confirm or both using the Advanced button.

User response and DB columns are here to map with the respective database column, here the cuisine column. In other words, the grammar is going to be generated from the cuisine column in the restaurants table, and the system is going to expect an answer that is part of this grammar.

Grammars can also be edited or enhanced as we can easily add synonyms for any value we want.

Now that we have seen how to create BPMs we will see what happens when a dialog system is deployed on the internet.
2.4.3.2 Deploying a dialog system

One important feature of DUDE is that it contains its own http server. This http server is really important as it is hosting all the generated grammars and VoiceXML pages of the application.

Once the application is deployed, the web server will be accessed by a remote Voxeo Server. Voxeo is the “Voice XML server” used by DUDE, it can interpret Voice XML Pages accessed from DUDE’s web server; it also features a speech recognizer and a speech synthesizer.

Basically, Voxeo is sending strings of words to DUDE’s web server, the recognized utterances according to some grammar it got from DUDE. It also receives the generated Voice XML pages and launches the next utterance accordingly.

Figure 15 shows the architecture of a deployed spoken dialog application on a mobile phone. We can easily see the interaction between the DUDE platform and the Voxeo server.

![Figure 15: Architecture of a simple deployed dialog system](image)
2.5 Evaluating spoken dialog systems

Now that we have seen the characteristics of spoken dialog systems and how to build such systems using DUDE, we need to find a way to evaluate them accurately. Doing such a thing is not as easy as it looks, dialog systems performances can rely on how well the recognizer recognizes the right words, but usually what matters the most is how the user feels like when using such systems.

For examples, very good systems can be avoided by users just because they are not as user friendly as they should. In this section we will introduce the PARADISE method of evaluating spoken dialog systems as it seems to be a very accurate one.

The AIM of paradise is to compare multiple systems doing the same task. It provides a method for developing predictive models of the usability of such systems and a technique for making generalizations across systems in order to find which properties of a system impact on usability. I.e. Figure out what really matters to the user.

As stated in “Towards developing general models of usability with PARADISE”, Marilyn Walker, Candace Kamm, Diane Litman, 1999; the main goal of PARADISE is to “maximize user satisfaction”

Figure 16 summarizes how this is done using Paradise.

General models of usability

<table>
<thead>
<tr>
<th>MAXIMIZE USER SATISFACTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAXIMIZE TASK SUCCESS</td>
</tr>
<tr>
<td>MINIMIZE COSTS</td>
</tr>
<tr>
<td>EFFICIENCY MEASURES</td>
</tr>
<tr>
<td>QUALITATIVE MEASURES</td>
</tr>
</tbody>
</table>

Figure 16: Maximizing user satisfaction using PARADISE
To do so, PARADISE uses two kinds of metrics:

- **Subjective metrics**: focused on task completion
- **Objective metrics**: focused on performances of spoken dialog system’s components such as the automated speech recognition module for example.

We will focus now on the different metrics (objective or subjective) that have to be collected in order to fully evaluate a system.

All these metrics can be separated into 4 categories; we will go through each of these categories in order to show exactly what have to be recorded:

- **Dialogue Efficiency**
- **Dialogue Quality**
- **Task Success**
- **User Satisfaction**

Dialogue efficiency metrics represent how efficient was a specific dialogue between a user and the system. 3 main metrics have to be collected.

- The elapsed time shows how long the phone call from start to end was
- System turns and User turns show how many turns the user and the system took during the dialogue.

Dialog Quality metrics represent the performances of each of the modules using system recordings. In total, 6 metrics have to be collected.

- **Mean Recognition Score**:
  - The M.R.S is the overall performances of the speech recognizer and can be calculated using confidence scores for each recognized (or not) utterance
- **Timeouts**:
  - The number of timeouts is the number of system prompts that have been played because the user was not answering quickly enough
- **Rejections**:
  - The number of rejections is the number of times the system did not get what the user said, in other words the recognizer failed to recognize the user’s utterance
- **Helps**:
  - The number of times the user asked for help
- **Cancels**:
  - The number of time it asks to cancel or start over
- **Bargeins**:
  - The number of times the user interrupted the system when it was saying something.
Task success metrics represent how well the user completed his task. Data is collected through a survey answered by the user.

User satisfaction is also a survey to be completed by the user after using the system. It should contain questions about each module’s performance, how easy was the task to complete, how well was the interaction pace with the system, how well and fast was the system to answer the user’s demands, if the system worked as expected by the user and if he would use it again. Another thing that has to be collected is the User expertise. Indeed, expert users would not interact the same way with a spoken dialog system as beginners.
3 Bus Information System

3.1 A word on the Let’s go challenge

The aim of this Master’s project is to take part in a challenge held by CMU’s dialog research centre (Carnegie Mellon University).

This challenge’s goal is to compare different spoken dialog systems of the same task. This task being to develop a bus schedule information spoken dialog system.

Here is the timeline of this challenge, taken from the dialog’s challenge webpage: http://www.dialrc.org/sdc/

- **Oct 15th 2009**: Call for Participants
- **Nov first week 2009**: The CMU Dialog Research Center will conduct web broadcast tutorial(s) on Let’s Go system
- **Nov 22nd 2009**: End of Participant registration
- **Jan 1st 2010**: Benchmark Let’s Go Software released and Data made available
- **Jan 1st 2010**: Start of Challenge
- **Jan-Apr 2010**: System Development
- **May 1st-Jun 15th 2010**: Evaluation
- **Jun 30th 2010**: Results release to participants
- **Jul 16th 2010**: Paper submission for SLT 2010
- **Jul 2010**: Start plans for SDC 2011
- **Dec 15th 2010**: Special Session at SLT 2010

The deadline for system development was initially on the 1st of May, but due to certain circumstances, it was moved to the 15th of May. Both these deadlines were quite an issue though as the masters projects system development was supposed to start early May.
3.2 System’s scope and specifications

In this section we will review the initial and final systems specifications and requirements, introducing user and system level specifications and ending up with predicted and actual dialog flows of the system.

3.2.1 Initial System’s specifications:
To define the initial system’s scope we first have to answer a couple questions:

- What do we have to design?
- What should be present in the system? What should be avoided / removed?
- What tools do we have in place yet?

As seen before, a bus information spoken dialog system has to be designed. It has been stated by the CMU that the system should give information about buses running only in the Pittsburgh area; moreover it should only answer queries for 8 bus routes.

The information that has to be given will always be about direct lines, no indirect routes have to be covered by the system. So, questions like “When is the next bus from downtown to the airport?” have to be covered only if there is a direct route between these 2 destinations and if this route is included in the list of 8 routes the system has to cover.

Finally, CMU have provided us with an updated database of bus routes in the city of Pittsburgh. This one had to be modified in order to reflect only the required routes but also DUDE’s architecture. We will see more details about the database pre-processing phase later on.

We will now discuss more about what the user may ask and what the system is required to answer, which will yield to the anticipated dialog flow.
Questions the user may ask mostly depends on the time factor, they may ask for:
- The next bus
- A bus on a specific date at a specific time
- A bus before a specific time
- A bus after a specific time
- They always have to provide the departure and arrival stops but are free to give either a departure time or arrival time
- Departure and arrival destinations may be:
  - A full address of a bus stop, eg: “Main street at Penn”
  - A neighbourhood name, eg: “Downtown”
  - A keyword related to a bus stop, eg: “the museum”

The system has to provide an answer to the user’s needs. If there is a route between the 2 points provided and at the time provided it then would have to answer something like: “There is a [routenumber] between [departure stop] and [arrival stop] at [time]”. If no routes have been found it will simply say so and ask for another route to search.

**3.2.2 Initial dialog flow:**

As we can see on figure 17, after the “welcome utterance” the user could basically say anything, therefore filling any required slots that he/she wants as well as asking for help.

After each user turn the system would implicitly confirm the user act, e.g.

- If the user is saying: “I am living from Downtown” the system would confirm using the utterance: “Right, leaving from Downtown”

Once all the needed information to query the database are provided, the system would confirm the trip using an overall explicit confirmation that would look like this: “Taking your bus in [departure place] on [date] at [time] and going to [arrival place], am I right?”

- If the user confirms, then the system will query the database and present the results
  - If the system finds multiple buses the user will be able to ask for the previous or next bus, leave or start a new query.
  - If it does not find anything the user will be able to start a new query or leave.

- If the user does not confirm, then the system would start over from the beginning.

Lastly, the user may say “start over” at any time to start again from the beginning.
Figure 17: Initial Call flow of the system
3.2.3 Final System's specifications:

Due to a very tight deadline and DUDE’s limitations in database queries and grammar generations a couple of changes had to be made in the specifications in order to get a fully working system in time.

First, because DUDE could not make the difference between the departure location and arrival location, both generated grammars being the same, we could not allow the user to fill more than one slot at any one time. Therefore, the overall system had to be fully system initiative.

For the same reason, after the final explicit confirm, if the user wanted to change either the departure place or the arrival place then the system would not be able to work out which one the user wants the change. It’s been decided later on to change the overall confirmation strategies as such:

- After a user turn, if the automated speech recognition confidence level is higher than 0.6 then the system would go to the next system act. However if it is below this threshold then the system would fire an explicit confirm e.g. “Did you mean Downtown”.
- The final explicit confirmation had to be totally removed, but as everything is being explicitly confirmed (if necessary) after each turn, that was not an issue.

Finally instead of asking for previous and next buses, the system would output a list of plausible buses, and the user would simply give the option number in order to get all the relevant details for this bus trip. The “help” utterance was also removed, and replaced by a “help” message if the user could not answer the question after 3 tries.
3.2.4 Final dialog flow:
Figure 18 shows these changes in the system specifications.

Figure 18: Final call flow of the system
3.3 **Software Design:**

Software design is one of the most critical phases of the software engineering life cycle. In this section we will highlight the most relevant aspects of this spoken dialog system, from the database design up to the high level design that the DUDE platform provides.

3.3.1 **Database design:**

As we will see, the design of the database was the most important part of the software. In fact, Not only has it to provide bus schedules, but it does provide most of the grammar used in the system as well.

As the initial database was using multiple tables, it had to be pre-processed and redesigned to fit DUDE’s specifications. The current version of DUDE could not handle multiple tables.

Here we will present how the database was structured initially and then follow up on how and why it had to be redesigned.

3.3.1.1 **Original database:**

Initially, the SDC database is an SQL database containing 6 tables as illustrated in Figure 19. The busstop table lists the bus stops in terms of an address, e.g. “Forbes Avenue at Murray” is busstop.stoppart1+busstop.intersectiontype+busstop.stoppart2.

Route refers to the bus route, e.g. route.number is 61C. The times table lists the times a bus leaves a busstop. Routessequence links buses and busstops on the same route.

To access a schedule for a specific bus, assuming that both the departure location and arrival location are busstop addresses, the SQL query would involve all of the above-mentioned tables.

In addition, there are two other tables: the stopneighborhoods table represents the different neighbourhoods in Pittsburgh, e.g. “Shadyside”; finally wordstops are simple keywords associated with certain busstops, for example, “Carnegie Mellon University” or CMU would be associated with the busstop address “Forbes Avenue at Morewood Carnegie Mellon”.

3.3.1.2 Database redesign:

As the current version of DUDE does not handle multi-table databases, some pre-processing was needed.

The 6 table database had to be transformed into a single large table where each line would represent a single bus trip at a specific time. Figure 20 shows the structure of this new database while figure 21 gives four example insert statements out of the 282,305 insert statements in its modified version.

```sql
CREATE TABLE `buses` (  
`id` int(11) NOT NULL DEFAULT '0',  
`fromloc` char(144) DEFAULT '',  
`fromwordstops` char(144) NOT NULL,  
`fromtime` char(10) NOT NULL DEFAULT '',  
`fromtimeappr` char(64) NOT NULL DEFAULT '',  
`toloc` char(144) DEFAULT '',  
`towordstops` char(144) NOT NULL,  
`totime` char(10) NOT NULL DEFAULT '',  
`totimeappr` char(64) NOT NULL DEFAULT '',  
`routenumber` char(11) NOT NULL DEFAULT '0',  
`routename` char(75) NOT NULL DEFAULT '',  
`routepattern` char(2) NOT NULL DEFAULT '',  
`routedaytype` char(8) NOT NULL DEFAULT '',  
`url` text NOT NULL,  
`url_list` text NOT NULL,  
UNIQUE KEY `buses_id_key` (`id`) )
```

Figure 20: single table structure
As we can see in figure 20, the ‘buses’ table is made up of 15 different columns. We’ll see here a few details on some of these entries.

Due to Dude’s grammar generation restrictions the 3 parts of a busstop address (stoppart1, intersection type and stoppart2) had to be merged together. Indeed, to generate grammar for a certain slot, DUDE is looking at all different values in a single column but cannot generate a single grammar located in more than 1 column. These addresses are the “fromloc” and “toloc” columns, which represent respectively the departure and arrival place.

Moreover, as we wanted the neighbourhoods to be in the same grammar as the busstop addresses, we also had to put them in these 2 fields. That explains why the 4 statements in figure 21 shows exactly the same bus run with a departure stop at “McKeesport at bay 2” (Neighbourhood : “McKeesport”) and an arrival stop at “Second street at grant” (Neighbourhood : “Duquesne”). Concretely, the four example statements describe the same 61C bus leaving McKeesport at 4.15pm. This is to enable the user to say a mix of neighbourhood locations and bus stop locations. For one specific bus there are the following statements:

- From neighbourhood to neighbourhood
- From neighbourhood to busstop
- From busstop to busstop
- From busstop to neighbourhood

Lastly, a single line in the database now says: “There is a [routenum] from [departure location] at [departurtime] on [daytype]. It will arrive in [arrival place] at [arrivalttime]”. In our case, the first example insert statement translates to “There is a 61C from MCKEESPORT at 4:15 AM on SATURDAY. It will arrive in DUQUESNE at 4:21 AM.”
Another important part of the database redesign was to allow DUDE to use a time grammar for bus times: In speech recognition and synthesis using pure numbers can be a problem and yield to unexpected behaviours like system crashes. But we could not simply translate it into a set of strings representing each time a bus runs. The grammar would have been way bigger than needed; but the more importantly for speech recognition purposes that would have been an issue at the user level. Admitting that a bus X is leaving the airport at 2:29 PM and the user wants a bus around 2:30 PM then how would the ASR recognize that the 2:29 PM is the right bus? As much as 29 and 30 are only a matter of minute, the user would probably say something like “half past two” or “two thirty”; linguistically speaking the system would not understand that saying “thirty” or “half past” may mean “twenty nine”.

So, keeping the absolute timings was increasing the size of the grammar but it was also pushing the user to say the exact time of their bus. But how could they? That is exactly what they want to know in the first place.

To reduce the size of this grammar we divided an hour into 3 time slices, and synonyms were attached to each slice like this:
The nil slice: from (H-1):51 to H:10. Ex: 3:51 AM to 4:10 AM
  - Entry in the grammar: “four A M”
  - Possible synonyms: “five of three fifty five A M”, “five after four A M”

The twenty slice: from H:11 to H:30. Ex: 4:11 AM to 4:30 AM
  - Entry in the grammar: “four twenty A M”
  - Possible synonyms: “four fifteen A M”, “four thirty A M”

The forty slice: from H:31 to H:50. Ex: 4:31 AM to 4:50 AM
  - Entry in the grammar: “four forty A M”
  - Possible synonyms: “four thirty five A M”, “ten of five A M”

A simple script was written to generate such grammar in an xml format that was then read by DUDE and associated to the dialog system’s time slot. This xml file can be found in appendix 1.

Also, because the current version of the DUDE platform can only use basic sql query statements with “=” operations, using time slices was enabling it to find more buses for the user while still giving the real times stored in the “fromtime” and “totime” columns.

Finally, here is a simple description of the other fields:

- **Routenumber**: the number of the route, e.g.: “61C”
- **Routename**: its name, e.g.: “McKeesport Homestead”
- **Routepattern**: its pattern, e.g.: “I” for inbound.
- **Routedaytype**: its daytype e.g.: “WEEK”, “SATURDAY” or “SUNDAY”
3.3.2 System design:

In section 2.4.3 we introduced how to develop and design spoken dialog systems using the DUDE graphical user interface. We will describe here how that has been done for the bus information system. We will first focus on the top level layer, then on the sub-level layer and finally on the added synonyms.

3.3.2.1 Top level layer:

Figure 22 shows the top level layer of the developed bus information system. This diagram shows the entry point of the dialog system.

Figure 22: Top level start node

Here we have to define the dialog start prompt which in our case is: “Welcome to the Pittsburgh bus information system. You can say start over anytime during the dialog. Where are you going to take your bus from?”

The top level node “Buses” is launched when the user answers the question with something that matches in the Departure location grammar e.g.: a departure place that can be found in the database. Figure 23 shows how the properties have been edited to allow the system to enter the buses task node.
The DB-table Mapping parameter maps the task to the single table in the database, the “buses” table. A pattern to present the information is then defined. This pattern will be used at the end of the dialog to present the results to the user. It uses the columns selected above. In our case we want to tell the user:

- The route number (or bus number) that he has to take.
- Its departure location and departure time,
- Its arrival place and arrival time.

Finally, task spotters have been defined in order to allow DUDE to know it has to enter the buses task. It is composed of key-words like “bus”, “I need a bus” but also all the generated grammar of the departure location.
3.3.2.2 Sub-level layer

After entering the “buses” task we had to define the main dialog in the sub level BPM. This diagram represents the sequence of dialog acts needed to fill in all the information required by the spoken dialog system in order to query the database.

Figure 24 shows this sub level diagram and an example of properties needed for one sub-task: the “to” task, or arrival place filler.

We can see that the dialog manager starts with asking for the departure place and then asks for an arrival place. When both these slots are filled, it will ask for the departure time. If the user answers “next” then the dialog manager skips the date (“on”) node and goes straight to the query. However, if a time is provided, then the system will ask for a travel date and then process the query.

Although all the tasks are different, they are all designed the same way so we will only present the arrival place node here.

We can see here, that the opening prompt of the dialog will be “Where do you want to go?” By choosing the “toloc” db column we are allowing DUDE to generate grammar from the arrival place column in the database. The ASR and NLU will then try to match the user input to any content in this grammar.

Finally, we also defined a set of synonyms for some important entries in the database. Figure 24 shows an example of location synonyms while appendix 1 shows the generated xml synonym files for the time grammar.
Editing such synonyms allows us to stress the recognizer on some important or complicated values in the grammar. It finally increases the recognition rate in most cases.

DELETE FROM route
WHERE route.number NOT IN ('28X', '54C', '61A', '56U', '61B', '61C', '61D', '64');

DELETE FROM times
WHERE times.routeid NOT IN (select id from route);

DELETE FROM routesequence
WHERE routesequence.routeid NOT IN (select id from route);

DELETE FROM busstop
WHERE busstop.id NOT IN (select stopid from routesequence);

DELETE FROM stopneighborhoods
WHERE stopneighborhoods.stopid NOT IN (select id from busstop);
3.4 Software development

As the DUDE development environment has built-in development tools using its graphical development environment ADT, the main part of software implementation was purely design like we’ve seen in section 3.3.2

Here we will focus on the methods used for the database implementation and a few others that had to be used to modify DUDE’s code in order to fit the requirements.

3.4.1 Database implementation:
The Port Authority buses database was quite messy and made it impossible to merge every table into a single one using MySql.

First, using the MySql statement below, unwanted bus routes were removed from the database in order to only keep the routes 28X, 54C, 61A, 56U, 61B, 61C, 61D and 64.

```
DELETE FROM route
WHERE route.number NOT IN ('28X', '54C', '61A', '56U', '61B', '61C', '61D', '64');

DELETE FROM times
WHERE times.routeid NOT IN (select id from route);

DELETE FROM routesequence
WHERE routesequence.routeid NOT IN (select id from route);

DELETE FROM busstop
WHERE busstop.id NOT IN (select stopid from routesequence);

DELETE FROM stopneighborhoods
WHERE stopneighborhoods.stopid NOT IN (select id from busstop);
```

Figure 26: Removing unwanted routes

Note, that not only the unwanted routes had to be removed but also any values in the database that are linked to these routes and not linked with the wanted routes. By doing so the size of the database is as small as possible, and so is the size of the generated grammars.

Now that our database was ready for processing, it has been dumped into a mysql dump (or recovery) file. This file was then used in a Ruby script to merge all tables together.

Ruby was the chosen language simply because it is a very powerful scripting language and it comprises a lot of very useful and straightforward features that allows the programmer to quickly get to the point.
Now, we will simply highlight the main function and show what the script is doing.

Figure 27 shows the main function of this script, readdump(). We can see that at first it has to handle special characters. In grammar generation, DUDE does not like special characters such as "[" for example. All of them were removed using the handlespecialcharacters() function and a new dump file was created and used afterwards in the program.

```ruby
def readdump
    puts "replacing special characters ...
    handlespecialcharacters()
    puts "writing new dump without special characters ...
    writenewdump()
    @inputfile = File.new("H:\Scripts\dumpParser\busDB1.sql", "r")
    puts "creating hashes ...
    createhashes()
    puts "merging busstops and neighborhoods tables ...
    mergestopsneighborhoods()
    puts "merging busstops and wordstops tables ...
    mergewordsstops()
    #Now busstophash contains the entire busstop table + the neighborhoodname
    puts "merging busstops routes and times tables ...
    mergetimesstops()
    #Now stoptimehash contains the entire busstop table neighborhoodname + route table + times table +
    wordstops
    puts "merging new busstop and sequence tables ...
    mergesequence() #everything is merged .... write new dump!
    puts "maping buses with their following stops..."
    mapbuseswithnextones()

    puts "writing new dump ...
    writemassivedump()
    puts "new dump correctly created, size : #{File.size(@outputfile2)}"
end
```

Figure 27: database pre-processing, main loop

So, when the data was ready, hashes were created; each hash containing the different values of a single table.

Then, using the merge functions each hash was linked to a specific hash using the relation between each table (mostly using bus ids and route ids). And the big hash was incrementally building up.
Once all tables were merged into a single one, we had to map all following buses of a single bus, say X, of the same route; in order to get every possible travels from a point A to a point B at time T.

The mapbuseswithnextones() function is doing so by simply looping over the current hash and adding links between buses if they actually are on the same route and the same run of the day. This function was also taking care of creating for instances of each relations (for handling neighbourhoods and complete addresses)

Finally the new dump file was created and written using the final generated hash.

Another script was written in order to generate the synonyms file in section 1. Every possible time slices were generated (nil, twenty and forty), then for each time slice synonyms were created and added to a hash table. Once all values were generated, the script was simply writing the final xml files. For more details, you can see the script in appendix 2. This one was also written in Ruby.

3.4.2 Dialog manager implementation:
As stated before, the dialog manager was implemented using the dude development tool, so not much programming was required.

However a few functions had to be written within DUDE in order to improve our system.

The first function was checking before query if the user said ‘next’ if it did then it was setting the day to ‘today’ and then launching the query. The function in DUDE that creates the sql statement had to be modified as well to handle the “next bus” possibility. Instead of using a time slice of 20 minutes we were using a time slice of 40 for the “next” case in order to hit a bus most of the time. Simple conditions have been added to the code to allow DUDE to create a query with the OR statement (‘travel time’ in this time slice OR in the next).

Finally a conversion function had to be written in order to convert decimal times into time strings, e.g. 4:05PM into “four zero five PM”, for the speech synthesizer.
3.5 Software testing

The spoken dialog system was incrementally tested. Once a functionality was added to the system, Oliver, Helen, XingKun and I were testing it using generated test cases.

Figure 28 shows an overview of these test cases. They were generated by a Ruby script that was taking the database dump as parameter. This script was taking trips at random and formatting them into a readable test case. 20 test cases were generated from the data.

<table>
<thead>
<tr>
<th>Test cases for the bus information system</th>
</tr>
</thead>
</table>
| 1) You have to catch a bus from EDGEWOOD to EAST PITTSBURGH at 5 8 P M (FIVE P M)  
  The bus route output by the system should be 61 A, its arrival time should be 5 34 P M and you should take it on A WEEKDAY |
| 2) You have to catch a bus from OAKLAND to GREENFIELD at 2 33 A M (TWO FORTY A M)  
  The bus route output by the system should be 61 D, its arrival time should be 2 43 A M and you should take it on SATURDAY |
| 3) You have to catch a bus from BOULEVARD OF THE ALLIES AT MARION MERCY HOSPITAL B to MOON TOWNSHIP at 10 37 P M (TEN FORTY P M)  
  The bus route output by the system should be 28 X, its arrival time should be 11 12 P M and you should take it on SUNDAY |
| 4) You have to catch a bus from DOWNTOWN to FORBES AVENUE AT BRADDOCK at 10 49 P M (TEN FORTY P M)  
  The bus route output by the system should be 61 A, its arrival time should be 11 11 P M and you should take it on A WEEKDAY |

Regarding test case number 1, if the user provides “Edgewood” and “East Pittsburgh” as departure location and arrival place, as well as five PM, or one of its synonyms, and a day of the week (from Monday up to Friday), then the dialog system should get the bus described in 1) as a result of its query.
3.6 Software results and evaluation

For this system we didn’t need to evaluate it by ourselves. In fact, as it was part of four different dialog systems in the Spoken Dialog Challenge, all the evaluations were done by the C.M.U. Dialog Research Centre.

The software has been tested by callers from each participant and other “real” users. Each caller interacted with each system twice using predefined cases. Each scenario was represented graphically as a departure location and an arrival location, with a time or time-frame or “now”.

For our spoken dialog system, 62% of our calls reached the stage of presenting the results to the user. Of these calls 61% gave fully correct information and 74% were correct although the times were not always entirely correct.

The remaining non correct answer can be explained with a simple fact. We explained earlier on the advantage of using time slices. However, it has a major drawback. Because DUDE could not query something like “time > 4 PM” we were restricted to a time slice of 20 minutes for each query (and 40 minutes for the “next bus” query), meaning that if the requested bus was a minute outside of the time slice, the query would not find it.

Also, 38% of the calls did not reach a query state. This can be explained because of the use of synonyms. When the recognizer was recognizing a synonym, the system was always repeating its actual value. So, let us say “Fifth Avenue at market” is the actual value and “market” is a synonym. When confirming the user input, DUDE would say “did you say fifth at market” instead of “did you say market” and so on for all synonyms. This can be quite irritating to the user, and after a while he/she may simply hang-up. This seems to have happened quite some times, but could not be changed within DUDE.

Lastly, these results are still quite encouraging considering the very short amount of time that has been used for its development and the rather good evaluation results provided by C.M.U.
4 Genetically based dialog manager

Once the spoken dialog system was developed and being evaluated a new project was starting. The aim of this one was to compare a simple hand-written dialog manager with an evolutionary dialog manager that uses genetic algorithms as a mean of decision making.

We will start presenting the requirements and specification as well as the data preparation. We will the follow up with the system’s design and its implementation.

Finally, testing and evaluation results will be presented.

4.1 System requirements and specifications

First of all, this dialog manager had to be written in the same scope as the previous one, meaning that it would run on the bus information scope.

The genetic algorithm used for decision making had to generate the next dialog steps until a system query is available while still trying to keep the dialog as small as possible.

As genetic algorithms can take quite some time to compute, it also had to be fast enough as we are dealing with real-time systems.

Also, dialogs would be different every time a user calls. This is an interesting feature as the dialog system gets less than predictable and therefore the dialog would be different each time a user would call it.

To help the dialog manager deciding which system act to perform next, a system acts graph had to be generated from the 3 years C.M.U. Let’s go system logs. We will see more details on how it’s been generated later on.

Finally, for testing and evaluation purposes users were simulated using different user strategies. The evaluation script was then editing statistics about dialogs such as their length and the rate of success (system being able to query the database at the end of the dialog)
4.2 Data preparation:

Before actually designing the software, some data preparation was needed in order to build the system acts graph.

A ruby script was parsing 3 years worth of C.M.U. logs into a probabilistic model of system acts. E.g.: computing the probability of going from a state A to a state B or computing the probability of the system asking for a confirmation after asking for a departure place for example.

This Ruby script was outputting the results in an XML files that would be read by the dialog manager at initialisation. Some not so relevant system acts like “Inform too many busstop matching” were simply cut out of the scope because we wanted a small but effective running system as quickly as possible

Figure 29 shows a part of the generated xml file. The entire file can be viewed in appendix 3.

```
<SystemActs size="19">
  <Act id="4" type="ExplicitConfirm/AcknowledgeConfirm" example="Alright"
       fillers="confirmslot">
    <FollowingAct id="6" type="ProcessQuery/InformFirstProcessing" example="Hold on. Let me check that for you." fillers="none">0.25552276301011</FollowingAct>
    <FollowingAct id="5" type="GetQuerySpecs/RequestArrivalPlace" example="What is your destination?" fillers="ArrivalPlace">0.236241108199176</FollowingAct>
    <FollowingAct id="11" type="GetQuerySpecs/RequestTravelTime" example="When are you going to take that bus?" fillers="TravelTime">0.231935604642456</FollowingAct>
    <FollowingAct id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?" fillers="DeparturePlace">0.133845001871958</FollowingAct>
    <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.115499812804193</FollowingAct>
    <FollowingAct id="13" type="GetQuerySpecs/RequestDepartureStopInNeighborhood" example="Where in neighborhood.name EAST PITTSBURGH are you leaving from?" fillers="DepartureStopInNeighborhood">0.0142268813178585</FollowingAct>
    <FollowingAct id="10" type="GetQuerySpecs/RequestExactTravelTime" example="At what time do you want to travel?" fillers="ExactTravelTime">0.00992137776113815</FollowingAct>
    <FollowingAct id="2" type="GetQuerySpecs/AskHowMayIHelpYou" example="What can I do for you?" fillers="none">0.00187195806813927</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound" fillers="">0.000935979034069637</FollowingAct>
  </Act>
</SystemActs>
```

Figure 29: extract of dialog_acts.xml

We can see in figure 29 that an act is linked with a set of following acts. For example processing a query has a probability of 0.26 while following a confirmation acknowledgement.
In addition to the probabilities and the system acts id, this xml file contains some more or less relevant information such as:

- A type: the utterance type
- An example: the utterance example said by the system
- Fillers: which slot(s) this act is asking to fill. E.g. asking for a departure place would try to fill the departure place slot

Once this file was generated, it was parsed by the dialog manager at initialisation and presented as a tree and a matrix in its Graphical User Interface. Figure 30 and 31 show these two representations.

![Figure 30: Tree view of the system acts graph](image-url)
Figure 31: Matrix view of the system acts graph
4.3 System design

Once the data was prepared the system had to be designed. It was actually incrementally designed, meaning that each module was designed before implementation if necessary.

We will here review each package in more or less details and talk about the Genetic algorithm design and requirements before focusing on the development.

4.3.1 Design of the different software packages:

4.3.1.1 Graphical User Interface:

The graphical user interface is a tabbed based GUI, all 3 tabs representing dialog models, the two first tabs shown in figure 30 and 31 present the system acts probabilistic model while the last one presents the user acts probabilistic model. You can see this one in figure 32; more details will be explained in the implementation part.

As you can see in figure 32, different user strategies can be selected from the two combo boxes and the dialog manager can be ran by simply clicking on the “RunGA” button.
4.3.1.2 Loggers:

The Loggers package is here to handle logging of dialogs. It is composed of 5 classes:

- **DMLogger**: the master class for all loggers
- **EvaluationLogger**: a sub-class of DMLogger, handles logging of evaluation
- **GALogger**: a sub-class of DMLogger, handles logging of the genetic algorithm
- **HistoryLogger**: a sub-class of DMLogger, handles logging of the dialog history
- **LoggingHandler**: A static class that handles all Loggers

Figure 33 shows an overview of the class diagram for this package.

![Loggers class diagram](image)

The main program does only have access to the LoggingHandler instance but no visibility on the actual loggers; this enables us to develop as much kind of loggers as we want.
4.3.1.3 System data:

The systemData package contains all the classes that relates to system data, dialog acts, user acts, graphs and so on. It also contains the overall factory of the system. Here is a list of all the classes that it contains:

- **DActFactory**: a singleton class used to instantiate all system data, access to dialog graphs will be granted through the static method getInstance() from this class.
- **DialogAct**: the master class for dialog acts.
- **DialogGraph**: contains the user acts graph and the system acts graph
- **Evaluator**: Used to store evaluation results
- **SystemAct**: a sub-class of DialogAct, used to define system acts
- **UserAct**: a sub-class of DialogAct, used to define system acts
- **UserProfile**: class used to define user strategies

As for the loggers here is a glimpse at its class diagram:

![Class Diagram](image)

**Figure 34: SystemData class diagram**

Note that, the Evaluator is not linked to any class in this diagram, but we will see that it is directly accessed in the main loop of the dialog manager.
4.3.1.4 XML Parser:
The XMLParser contains two classes, one is used to parse the xml file representing the system acts graph, and the other one writes XML for the user acts.

4.3.1.5 Evolutionary dialog manager:
Finally, the evolutionnaryDM package is the most important of all. It contains the most important parts of the dialog manager. Here is a list of its classes:

- CandidateSolution: contains the encoding of a candidate solution, e.g.: a sequence of system acts.
- DialogHistory: contains the history of all the dialog
- DialogManager: The main class of the software. It manages the dialog
- GA: A super class defining the genetic algorithm
- Gene: It is basically an element of a candidate solution, a systemAct and some other parameters
- GenerationalSingleGene: one implementation of the genetic algorithm, it isn't used in the system as it was too slow
- QueryRuleEngine: Used to handle queries and know how far the manager is from queuing the database.
- SteadyStateSingleGene: the implementation of the genetic algorithm that is used in the system

Like in the previous packages, here is a glimpse on this package's class diagram:
Figure 35: evolutionaryDM class diagram
4.3.2 Genetic algorithm design:
We have seen the software design so far but haven’t covered the genetic algorithm. In this section we will present the few choices that made it work.

First of all, Genetic algorithms are population and evolution based algorithm. The idea is to evolve a population of a predefined number of candidate solutions so as to optimize the best solution. A fitness function is designed for that matter. The main goal of a genetic algorithm is to minimize or maximize this function depending on the problem. In our case we want to minimize it in order to get a dialog as small as possible.

In this section we will cover the candidate solution design, the fitness function design, the selection method, and the mutation operator design.

4.3.2.1 Candidate solution design:
In our system, a candidate solution is a sequence of system acts that goes from the start of the dialog to a query.

As the length of a dialog is not known in advance a candidate solution is a sequence of System Acts of length L (its DNA) + some other parameters. Therefore when randomly generating the initial population, each candidate solution can be of any length, the GA will take care of the rest.

Another important thing to think about first is how to represent genes. A gene is a single element of the DNA. It could have only been a SystemAct, but adding in probabilities and out probabilities was very handy for later mutations. In probabilities and out probabilities are the probabilities of going from the previous gene to the actual gene, and from the actual gene to the next gene in the DNA.

At initialisation, the Genetic algorithm is initialising the pool of candidate solutions randomly. However, as we are working on a probabilistic model only two system acts that have a probability p greater than 0 can be linked in the sequence. This has to be handled in the generate() function.

Finally, when the dialog manager needs to run the algorithm another time because some conflict arose in the top-level, then the generation also has to handle the dialog history. Concretely, the N first genes in the sequence will be represented by the sequence of system acts used in the dialog so far.
The fitness function:
The fitness function is used to tell the GA how good a solution is, it has to be calculated and stored in the candidateSolution object. After the entire run, the fittest solution will be the one used by the dialog manager to handle the next system acts.

This is most certainly the most important part of the GA, as it drives it to evolve its population in the right direction. We will see here the most important parameters used to decide how good a solution is:

- Size of the dna: the biggest the size, the highest the fitness
- Slots duplicated: we don’t want to ask twice for the same information, increases the fitness if discovered
- Slots missing: we don’t want to miss out a slot, increases the fitness if discovered
- How may I help you messages duplicated, we don’t want to ask how may I help you twice, except if it hasn’t been confirmed.
- Consecutive confirmations: we don’t want a sequence of more than one confirmation in a row. If more than one, the fitness is increasing
- Last gene is not a query: We want the solution to end with a query, as the mutation operator may remove it, we increase the fitness if the last gene is not a query

Selection method:
Before evolving its population the GA has to select a candidate for mutation. The chosen selection method is a very straightforward one.

The GA chooses two candidates at random in the population and return the one that is less fit. This one will then be mutated as we will see in the next section. It will then be compared against the worst candidate in the population.

If it is less fit than the worst, we discard it, if it is a better solution then we replace the worst candidate by this newly created candidate.
4.3.2.4 Mutation operator:

The mutation operator allows us to evolve solutions. It is based on three mutation operators:

- **Add gene:** adds a gene at random in the candidate’s DNA (adds a system act at a random position in the sequence of system acts)
- **Replace gene:** replace a gene at random in the candidate’s DNA (replace a system act at a random position in the sequence of system acts)
- **Remove Gene:** removes a gene at random in the candidate’s DNA (removes a system act at a random position in the sequence of system acts)

It is driven by only one heuristic per mutation:

- We cannot add a system act between two system acts if the in probability or the out probability are equal to zero (see the use of in and out probabilities here)
- We cannot replace a system act in the sequence if the in probabilities and out probabilities of the new system act are equal to zero
- We cannot remove a system act in the sequence if the in probabilities and out probabilities of its following or previous acts are zero.

Another constraint is that if the GA has to be re-run, then the mutation operators cannot possibly change the first part of the DNA that represents the dialog history.

If multiple system acts can be inserted or can take the place of a system act then an array of these possible system acts is built, one system act being chosen at random from this array and constituting part of the new gene.

If no mutation is possible in a Gene then the DNA simply stays the same and replaces (or not) the worst gene in the population.
4.4 Software implementation:
In this section we will highlight some of the key points of the software implementation, from probabilistic models loading to logging.

4.4.1 Loading probabilistic models:
We will see here how the system and user graphs are loaded into memory at system start-up, and how they can be accessed.

At initialisation, the main system is instantiating a DActFactory singleton instance. This class used to instantiate critical information will be created once and for all during the whole system run.

It is first creating an instance of an XMLParser that will be used to recover the probabilistic model from appendix 3. Parsing simply gets dialog act ids and there relevant fields (idstr, example, etc ...) and creates System Acts objects. Once these objects are instantiated it then builds up a two dimensional array of probabilities between system acts. This would be the equivalent of a directed graph.

Once this first graph has been generated the same instance of DActFactory creates a default user profile. We will talk more in details about these user profiles in the user simulation testing chapter.

Afterwards, the factory is creating the user graph based on this user profile. The difference between user and system graph is that the user graph does not rely on any previously processed data (apart from the two parameters of the user profile that we will discuss later). What it does, is that it creates a two dimensional array as well but this times building up relationships between a specific system act and the anticipated user answer. Concretely, if the system is asking for a departure place then it is expected that the user would answer ... a departure place!

However, he/she could be answering a departure place and an arrival place. But also, as we are assuming that this input comes straight from the automated speech recognition module, it is sometimes likely that we would get only a depart time and a departure date, the ASR missing out on the departure place.

Anticipating unlikely events allows the system to be more flexible, but it can as well allow it to be quicker to get to the point. So, basically, the algorithm that generates this probabilistic model is using three sorts of probabilities.
High probability: used for user acts that correspond exactly to the system act’s question e.g.: when asking for a departure place, the user answer will be a departure place.

Low probability: used for user acts that correspond more or less to the system act’s question e.g.: the asked slot is filled but there are more information and the utterance.

Very low probability (but not 0): used for user acts that don’t answer the system act’s question e.g.: When the system is asking for a departure place the user answers a departure time.

The system is ready to go, once this phase is complete.

To access these data structure, the objects that need them simply have to call the static getInstance() method of the DActfactory singleton followed by a getter of the graph or profile wanted.

Moreover, accessing objects is quite easy, but how do we know how to access the probability of a system act following another system act. A matrix can only be accessed with integer ids, and although each act has an id, they are recognizable with their actual meaning.

Defining a hard-coded relationship between ids and id-strings would have been a bad solution. Not only we want to access them but we also want the system to work even if the data have been altered (new system act for a next version, or deletion of system acts that we find useless). Hard coding everything in the code would simply be a mess to manage. Instead the parser generates a simple array of the system acts encountered in the xml file.

Each system act having an id, we just have to place them at the same position in the array and in the graph (the position being the act id taken from the xml file). That way if acts are removed or added to the xml dialog-graph the system can still load everything and the GA would also still work perfectly as it does not need to know what is going on up in the graph, it simply evolves on the given data.

This however is not true yet for the user graph. Time running out, an xml file representing a list of user acts was hard-written and then read by the parser to build user acts objects. This file can be found in appendix 4. Instead, the xml file defining user acts could be generated from the system act graph using some basic regular expressions, for example:

If a system act is called “GetQuerySpecs/RequestDeparturePlace” the corresponding generated user act would be “AnswerQuery/Departureplace” and so on.
4.4.2 Running the core of the system:

Now that we have seen how to load the training data, we will present how the manager handles the dialog in both scenarios, the genetic algorithm one and the hand written one.

4.4.2.1 Running the evolutionary dialog manager:

So far, we have discussed on how the genetic algorithm was running and how it would generate the next steps to be taken in the dialog.

However, that is not all. Although the algorithm can find good sequences of dialog and nearly a different one every-time it runs, unexpected things can happen, some increasing the size of the dialog other ones reducing it.

The GA is only here to generate a predictable dialog; however we still need a mechanism that would allow us to handle scenarios like:

- If multiple slots have been field in a single user act, do we have to ask for the same slots again? The obvious answer is ... “no”! But if the GA computes a sequence that first asks for a departure place and then for the departure time and the user directly filled departure place and time in the first utterance, the system would still be asking for a departure time in the next utterance. That is a problem that can’t be solved by the GA unless it is ran every after each user act. We don’t want that though, because that would basically mean wasting computation time for nothing!
- If a slot is asked by the system but then not confirmed by the user we would have to go back one step and ask the same question again, but that is definitely not part of the generated sequence. That is the same problem as above ... again.

The solution used was to check after each system acts and user answers if there were any conflict. No conflict means we the dialog manager goes to the next step, however there is a conflict one of these can happen:

- Go back one step if the user did not confirm a system act.
- Go to the last system act in the sequence if the closest query has all the relevant information
- Re-generate the dialog sequence, or re-launch the GA, for any other conflict.

Figure 36 shows the checkconflict() function from the DialogManager class. It is called at the end of each turn in the dialog.
public boolean checkForConflicts(UserAct uAct){
    boolean conflicts = false;
    SystemAct nextSystemAct = new SystemAct();
    SystemAct currentSystemAct = (SystemAct) dialog.get(turn);
    if(turn < dialog.size()-1)
        nextSystemAct = (SystemAct) dialog.get(turn+1);
    if(turn == 0){
        conflicts = false;
    }
    else if(currentSystemAct.isQuery() && QueryRuleEngine.getInstance().getNumberOfMissingSlots(history.getSlotfilled())>0)
    {
        LoggingHandler.logSystemOutput("!!! Not All Slots Have Been Filled !!!");
        LoggingHandler.logSystemOutput("!!! Regenerating Solution !!!");
        conflicts=true;
    }else if(uAct == null && currentSystemAct.acknowledgingConfirm() && !nextSystemAct.askingSlot() && !nextSystemAct.isQuery())
    {
        LoggingHandler.logSystemOutput("Current act is an ack, next act is not a query !!!");
        conflicts=true;
    }else if(uAct != null && uAct.getAsrConfidence()>=0.2 && uAct.getAsrConfidence()<0.6 && !currentSystemAct.askingConfirm() && !nextSystemAct.isOfConfirmType())
    {
        LoggingHandler.logSystemOutput("!!! Current act is a slot filler with small confidence and next act is not a confirm !!!");
        conflicts = true;
    }else if(history.getSlotfilled().contains(nextSystemAct.getSlotfiller()))
    {
        LoggingHandler.logSystemOutput("next act's slot is already filled : " + nextSystemAct.getSlotfiller());
        conflicts = true;
    }
    return conflicts;
}

Figure 36: the checkForConflicts() method
We can see in this function that there are 5 different kinds of conflicts in our system so far, here is an explanation for each one of them:

- If turn is equal to 0, that means that we are in the welcome utterance (start of the dialog), no conflict is possible.
- If the next system act is a query to the database but not all slots have been filled, there is a conflict (we missed one or more slot(s)).
- If there is no user act following a system act, then we must be in a system act acknowledging a confirmation. If the next act is not a query or not a question, then there is a conflict.
- If the ASR confidence is low and the system is not asking for a confirmation, there is also a conflict. Low ASR confidence means that we always have to confirm.
- Lastly, if the next system act is asking for a slot that is already filled (that may happen sometimes) there is a conflict. We don’t want to have two times the same question.
- Otherwise, no conflict is detected and we return false.

Moreover here is what happens in the run() method of the DialogManager class, when processing a system output:

1. The system output, or dialog act, is pushed in history
2. A set of conditionals grab the purpose of this system act
   a. If act is a system act:
      i. If it’s asking for a confirmation
         1. User answer generated ("yes", or "no")
         2. User act pushed in history
         3. If the answer is yes:
            a. Push the newly slot(s) filled in history
         4. If it is no
            a. Go back to previous system act that is not a confirm
      ii. If it’s acknowledging a confirmation
          1. Simply log it
   b. If act is asking to fill-in a slot
      i. Generate user act
      ii. Push it in history
      iii. If ASR confidence is less than 0.2
          1. Go back to previous act, confidence is way to small
   c. If act is a query to the database
      i. Log it
   d. Else
      i. Log it
3. Check for conflicts
4. re-run GA or not
5. go to Next act.(step 1)
4.4.2.2 Running the hand-written dialog manager:

To compare dialog managers of the same scope, a very simple hand written dialog manager has been set up.

It uses the same run() method as the evolutionary dialog manager but instead of running any algorithm to generate the dialog sequence, it is hard-coded and never changes.

Check for conflicts are being removed too and the only thing that is actually kept is the reaction after a non confirmation, in other words going back one step if the user did not confirm.

Here is the hard-coded dialog sequence:

- Inform/GiveIntroduction
- GetQuerySpecs/AskHowMayIHelpYou
- ExplicitConfirm/RequestConfirm
- ExplicitConfirm/AcknowledgeConfirm
- GetQuerySpecs/RequestDeparturePlace
- ExplicitConfirm/RequestConfirm
- ExplicitConfirm/AcknowledgeConfirm
- GetQuerySpecs/RequestArrivalPlace
- ExplicitConfirm/RequestConfirm
- ExplicitConfirm/AcknowledgeConfirm
- GetQuerySpecs/RequestTravelTime
- ExplicitConfirm/RequestConfirm
- ExplicitConfirm/AcknowledgeConfirm
- GetQuerySpecs/RequestExactTravelTime
- ExplicitConfirm/RequestConfirm
- ExplicitConfirm/AcknowledgeConfirm
- ProcessQuery/InformFirstProcessing

User profiles are set the same way as the evolutionary dialog manager as they are only required in user simulation.
4.4.3 Logging dialogs and evaluation:

Logging dialogs was the last thing to handle when the system was fully working. It allows us to keep track of any kind of information gathered during run time, as well as to debug the system and evaluate it.

As we have seen earlier on the Loggers package is composed of 5 classes, 3 of them being actual loggers

Upon initialising a dialog manager instance, the DialogManager constructor calls the static method initLoggers() from the LoggingHandler class that will set-up Logging directories and instantiate the relevant loggers if necessary.

It creates directories depending on the time this method has been called, which type of dialog manager is running (evolutionary or hand-written) and what user strategy has been chosen. Figure 37 shows an example of these directories on a windows file system:

![Figure 37: folder structure](image)

The first series of directories represents the user strategy followed by “HW” or “GA” depending on the type of dialog manager. The second series, on the right part of the screenshot shows folders for each run with date and time.
There are 4 different logging files in each one of these folders as well as 1 evaluation file per subdirectory:

- **SystemHisto.log**: shows all system acts from the dialog, their asked slot (if any) and the total number of System acts in the dialog.
- **UserHisto.log**: shows all user acts from the dialog, their ASR confidence score as well as the total number of User acts in the dialog.
- **FullHisto.log**: shows all user and system acts in the dialog, in other words it describes the whole dialog from start to a query (if a query is possible)
- **GA.log**: shows dialog acts, as well as all kind of information concerning the Genetic algorithms: the generated solution, its properties and fitness, the best mean and average fitness of the whole pool of solutions and explanations on why the GA had to be run again.
- **Evaluation.log**: contains all sorts of information regarding the system evaluation. We will explain more about it in its later section

Example log files can be found in the appendix section. In the run() code, every time something has to be logged, a static method from the loggingHandler is called, here is a brief description of each one of them:

- **public static void logSystemOutput(Object data)**: calls the write() method from the GALogger instance. It can log any kind of object as long as it implements the toString() method.
- **public static void logHistory(DialogHistory hist)**: calls all three writing methods from the HistoryLogger instance to write in the three history log-files described above.
- **public static void logEvaluation(Evaluator eval)**: calls the write() method from the EvaluationLogger instance.
4.5 Software testing:

4.5.1 Incremental testing:

An incremental testing strategy has been used to test the software. It basically has been tested module by module. This part of testing was to make sure that the system is doing what it is supposed to do.

We will see now a few testing strategies for the most important modules or functions:

- System graph loading and xml parsing:
  - Tests have been done with different xml files. Each file, containing a different graph of the same scope. It seemed like generating the system acts probabilistic model works for any size of acts.

- User graph generating:
  - Tests have been done through the Graphical interface, to check that the user matrix was correctly updating when changing user strategies
  - We also had to make sure that the sum of probabilities in each row was equal to 1 for any user strategies and that the dialog manager was actually using the same graph. This has been done by simply using display() or toString() functions.

- Genetic Algorithm initialisation:
  - By displaying the GA properties as well as each candidate, we were making sure that the algorithm was generating solutions as expected. The quality of solutions didn’t matter as they are nearly entirely random arrays in the generation phase. What mattered was if 0.0 probabilities were in the sequence or not and if the properties displayed corresponded to the actual candidate.

- Genetic algorithm run-time
  - Some tests have been made ensuring that the properties of the candidates were being changed accordingly to their respective mutation, that the fitness was correctly being updated and that the dialog manager was indeed taking the best candidate after each turn. Intensive running of the GA coupled with some termination criteria made sure that it was never entering any infinite loop.
4.5.2 User simulation testing:

In the previous chapters we talked quite a lot of user profiles and strategies without really explaining what exactly are these profiles. We have seen how the user profile was generated using high probabilities or low probabilities depending on the probabilities. Before actually talking about “user testing”, we will present what are the characteristics of user profiles.

While simulating a user, we are assuming two things:

- It has a certain level of English from “average English speaker” to “fluent/native English speaker”
- It has a certain tendency of answering questions as they’re asked, giving more or less information in the context. E.g. answering a departure place and an arrival place when being only asked for a departure place.

Now that’s not all, the dialog manager module comes after speech recognition and natural language understanding meaning that the actual user utterance may be disturbed by extern factors. In our case fluent English or average factor may just mean that the user has a string Scottish accent while the ASR is set up for an American accent. For the same reason the system might miss out the departure place while getting other relevant slots. These two parameters are here to simulate such scenarios.

If the English level is set to fluent then the user will tend to have higher recognition scores (The ASR will get higher confidence scores)

If the tendency of answering questions as they’re asked is high then the high probability of the user matrix would be set to 0.9. At lowest, it would be 0.6.

These strategies have been used to test how the dialog manager would react with different kind of users. They have also been used to test whether the generated dialogs were corresponding to what was expected from these user profiles. E.g. a fluent English value would yield to much less no answers to system confirmations and to ASR confidence over 0.5

Testing the GA with different scenarios was a necessity but, the real purpose of these profiles was to evaluate the software, we will conclude the evaluation in the next part.
4.6 System results and evaluation:

We’ve explained how the loggers were logging relevant data for evaluation. We will now see what exactly have been used to evaluate the software and see if the evaluation was conclusive.

First, for evaluating the software 1000 dialogs were generated per user profile (4 of them). Here is a brief description of each one of them:

- Fluent English – very accurate answers: High rate of successful confirmations, high ASR confidence and high probability of 0.9
- Fluent English – vague answers: High rate of successful confirmations, high ASR confidence and high probability of 0.6
- Average English – very accurate answers: Low rate of successful confirmations, rather low ASR confidence and high probability of 0.9
- Average English – vague answers: Low rate of successful confirmations, rather low ASR confidence and high probability of 0.6

After each set of 1000 dialogs an evaluation file was logged. The created file contains a set of statistics extracted from the dialog that we will see below:

- Minimum number of user acts
- Maximum number of user acts
- Mean number of user acts in the 1000 dialogs
- Minimum number of system acts
- Maximum number of system acts
- Mean number of system acts in the 1000 dialogs
- Minimum number of dialog acts
- Maximum number of dialog acts
- Mean number of dialog acts in the 1000 dialogs
- Number of unsuccessful dialogs
- Number of successful dialogs
- Number of hang-ups

Before actually giving out these results, a few things need to be explained:

These 4 scenarios have been run for both hand-written and evolutionary dialog manager with 100 iterations each for each scenarios.

- A successful dialog can be defined as a dialog that ends up with a query to the database, therefore all the information needed has been provided.
- An unsuccessful dialog is a dialog that does not end up with a possible query.
- The user is hanging up the phone when the system has asked the same question 3 times in a row.
We will now see all these results a single table, the most relevant ones will then be presented graphically.

Note that these annotations mean:

- fluVacc : fluent English – Very Accurate answers
- fluVag : fluent English – Vague answers
- AvgVacc : Average English – Very Accurate answers
- AvgVag : Average English – Vague answers

Table 1: evaluation statistics table

<table>
<thead>
<tr>
<th></th>
<th>Evolutionary Dialog Manager</th>
<th>Hand Written dialog manager</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>fluVacc</td>
<td>fluVag</td>
</tr>
<tr>
<td>Min User Acts</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Max User Acts</td>
<td>21</td>
<td>21</td>
</tr>
<tr>
<td>Min System Acts</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>Max System Acts</td>
<td>25</td>
<td>25</td>
</tr>
<tr>
<td>Min Dialog Acts</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>Max Dialog Acts</td>
<td>46</td>
<td>45</td>
</tr>
<tr>
<td>Mean Dialog Acts</td>
<td>17.7</td>
<td>17.53</td>
</tr>
<tr>
<td>Unsuccessfull dialogs</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Successfull dialogs</td>
<td>981</td>
<td>981</td>
</tr>
<tr>
<td>Hang-ups number</td>
<td>19</td>
<td>19</td>
</tr>
</tbody>
</table>

From this table we can already spot some differences between the two methods. Unsuccessful dialogs are more frequent in the hand written DM while there are only 0.025% of dialogs that didn’t end up whether by a query or by a hang-up. Also, user acts, system acts and dialog acts are strongly correlated (nb user acts + nb system acts = nb dialog acts). In the next graphs we will show these selected results:

- Mean number of dialog acts in graph 1
- Unsuccessful, successful dialogs and number of hang-ups in graph 2
Table 2: Evaluation results, mean number of dialog acts

mean number of dialog acts

<table>
<thead>
<tr>
<th></th>
<th>fluVacc</th>
<th>fluVag</th>
<th>AvgVacc</th>
<th>AvgVag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolutionary Dialog Manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hand Written dialog manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Table 3: Evaluation results, end of dialog

Evaluation results

<table>
<thead>
<tr>
<th></th>
<th>fluVacc</th>
<th>fluVag</th>
<th>AvgVacc</th>
<th>AvgVag</th>
</tr>
</thead>
<tbody>
<tr>
<td>Evolutionary Dialog Manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hand Written dialog manager</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
From the first graph it seems that for the same user profiles the number of dialog acts decreases as expected when using the evolutionary dialog manager. It seems even stronger for a high fluency value. Looking at the slope, we can also say that the GA performs much better in terms of dialog length but it seems relevant enough only for users that usually get over 0.5 ASR confidence.

From the second graph, the disparity appears to be the same, much better results with “fluent” users much worst with the “average english” user. But still for each of these categories, the GA gets much better results than the hand written dialog manager.

The number of hang-ups is roughly the same for both systems which can be explained by the fact that the GA doesn’t care too much about that, as this parameter is directly related to his “fluency” level. However, as dialogs are shorter with the GA they tend to get to the point quicker and avoid unnecessary hang-ups when user’s ASR confidence (or fluency) is bad.

Lastly, it is clear enough that the GA is optimized to get all the information it needs and query the database. Not including the hang-ups that rely mostly on the user ability to be understood (or the ASR ability to understand the user ...) the number of times the GA did not get to a query varies between 0 and 1, while it varies between 29 and 123 for the hand written dialog manager.

Although the difference is not huge between the two methods, it seems like evolution does improve all aspects of the dialog flow. This was, however, expected from the beginning because of the tiny scope of the application.

Finally, the last parameter of the evaluation that we haven’t talked yet about is the Genetic Algorithm run-time. Appendix number 5 shows a mean GA run-time of about 1.3 seconds. That means that when having to re-generate the dialog system, it will take 1.3 seconds. This amount of time is acceptable. However, it still can be optimised and would certainly react in less than a second in an optimized version of the dialog manager.
5 Conclusion

In conclusion to this dissertation we can say that both these projects have been challenging projects. Deadlines were very tight but always respected.

It also allowed me to learn a lot of things about dialog management and spoken dialog systems in a very short of time. From designing one with high level tools such as DUDE to actually implement one, using evolution as a mean of dialog generation.

Results were good enough in both projects and more or less as we were expecting them. Sadly, the bus information couldn’t go to the live testing phase in Pittsburgh because the evaluation results were not good enough for our system to be used by real day-to-day users. However, for a dialog system that has been implemented in such a small time frame, evaluation results show that DUDE can allow us to build more or less reliable spoken dialog systems in very little time with very little programming skills.

The second project was defined in order to keep me busy and has also been a personal success. This time programming skills were more than required, so it would obviously take more time if the developer is not a software developer. As we’ve seen, the results of the evaluation were pretty good, as expected too. However, it is only a small system. I believe that it would also work fine in larger systems that have the same kind of dialog structure; moreover, one thing that hasn’t been tested and that would make a huge difference in the future is the ability of the dialog manager to adapt on the smallest query when many different database queries are possible. Using the same dialog manager for a much bigger system would be possible without changing too much code if the xml system-acts graph is of the same structure. However for a really huge domain, tuning the algorithm would probably be a good idea.
6 References

6.1 Books and articles:


Oliver Lemon, Helen Hastie, Xingkun Liu, and Nicolas Merigaud. 2010. “Let’s go dude!” Using the spoken dialogue challenge to teach spoken dialogue development”.

6.2 Websites:

http://www.dialrc.org/sdc/
http://www.cstr.ed.ac.uk/projects/unisyn/
http://www.cslu.ogi.edu/asr/
http://www.voicexml.org/
http://www.voxeo.com/prophecy/
http://www.microsoft.com/windowsxp/using/setup/expert/moskowitz_02september23.mspx
http://www.cs.utah.edu/~hal/courses/2009S_AI/Walkthrough/Speech/
http://caise07 idi.ntnu.no/gifs/cellphone.gif
7 Appendices

7.1 Part of synonyms.xml

```xml
<AnswerType typeName="busmergedn_buses_fromtimeappr" typeOrigin="DB">
  <AnswerValue normalizedValue="one a m">
    <SynonymicValue>one a m</SynonymicValue>
    <SynonymicValue>one five a m</SynonymicValue>
    <SynonymicValue>one ten a m</SynonymicValue>
    <SynonymicValue>five after one a m</SynonymicValue>
    <SynonymicValue>ten after one a m</SynonymicValue>
    <SynonymicValue>twelve fifty five a m</SynonymicValue>
    <SynonymicValue>five of one a m</SynonymicValue>
  </AnswerValue>
  <AnswerValue normalizedValue="one twenty a m">
    <SynonymicValue>one twenty a m</SynonymicValue>
    <SynonymicValue>one fifteen a m</SynonymicValue>
    <SynonymicValue>one twenty five a m</SynonymicValue>
    <SynonymicValue>one thirty a m</SynonymicValue>
    <SynonymicValue>a quarter after one a m</SynonymicValue>
  </AnswerValue>
  <AnswerValue normalizedValue="one forty a m">
    <SynonymicValue>one forty a m</SynonymicValue>
    <SynonymicValue>one thirty five a m</SynonymicValue>
    <SynonymicValue>one forty five a m</SynonymicValue>
    <SynonymicValue>one fifty a m</SynonymicValue>
    <SynonymicValue>ten of two a m</SynonymicValue>
  </AnswerValue>
  <AnswerValue normalizedValue="two a m">
    <SynonymicValue>two a m</SynonymicValue>
    <SynonymicValue>two five a m</SynonymicValue>
    <SynonymicValue>two ten a m</SynonymicValue>
    <SynonymicValue>five after two a m</SynonymicValue>
    <SynonymicValue>ten after two a m</SynonymicValue>
    <SynonymicValue>one fifty five a m</SynonymicValue>
    <SynonymicValue>five of two a m</SynonymicValue>
  </AnswerValue>
  <AnswerValue normalizedValue="two twenty a m">
    <SynonymicValue>two twenty a m</SynonymicValue>
    <SynonymicValue>two fifteen a m</SynonymicValue>
    <SynonymicValue>two twenty five a m</SynonymicValue>
    <SynonymicValue>two thirty a m</SynonymicValue>
    <SynonymicValue>a quarter after two a m</SynonymicValue>
  </AnswerValue>
  <AnswerValue normalizedValue="two forty a m">
    <SynonymicValue>two forty a m</SynonymicValue>
    <SynonymicValue>two thirty five a m</SynonymicValue>
    <SynonymicValue>two forty five a m</SynonymicValue>
    <SynonymicValue>two fifty a m</SynonymicValue>
    <SynonymicValue>ten of three a m</SynonymicValue>
  </AnswerValue>
</AnswerType>
```
Entering the spoken dialog challenge

</AnswerValue>
<AnswerValue normalizedValue="three a m">
  <SynonymicValue>three a m</SynonymicValue>
  <SynonymicValue>three five a m</SynonymicValue>
  <SynonymicValue>three ten a m</SynonymicValue>
  <SynonymicValue>five after three a m</SynonymicValue>
  <SynonymicValue>ten after three a m</SynonymicValue>
  <SynonymicValue>two fifty five a m</SynonymicValue>
  <SynonymicValue>five of three a m</SynonymicValue>
</AnswerValue>
<AnswerValue normalizedValue="three twenty a m">
  <SynonymicValue>three twenty a m</SynonymicValue>
  <SynonymicValue>three fifteen a m</SynonymicValue>
  <SynonymicValue>three twenty five a m</SynonymicValue>
  <SynonymicValue>three thirty a m</SynonymicValue>
  <SynonymicValue>a quarter after three a m</SynonymicValue>
</AnswerValue>
<AnswerValue normalizedValue="three forty a m">
  <SynonymicValue>three forty a m</SynonymicValue>
  <SynonymicValue>three thirty five a m</SynonymicValue>
  <SynonymicValue>three forty five a m</SynonymicValue>
  <SynonymicValue>three fifty a m</SynonymicValue>
</AnswerValue>
<AnswerValue normalizedValue="four a m">
  <SynonymicValue>four a m</SynonymicValue>
  <SynonymicValue>four five a m</SynonymicValue>
  <SynonymicValue>four ten a m</SynonymicValue>
  <SynonymicValue>five after four a m</SynonymicValue>
  <SynonymicValue>ten after four a m</SynonymicValue>
  <SynonymicValue>three fifty five a m</SynonymicValue>
</AnswerValue>
<AnswerValue normalizedValue="four twenty a m">
  <SynonymicValue>four twenty a m</SynonymicValue>
  <SynonymicValue>four fifteen a m</SynonymicValue>
  <SynonymicValue>four twenty five a m</SynonymicValue>
  <SynonymicValue>four thirty a m</SynonymicValue>
  <SynonymicValue>a quarter after four a m</SynonymicValue>
</AnswerValue>
<AnswerValue normalizedValue="four forty a m">
  <SynonymicValue>four forty a m</SynonymicValue>
  <SynonymicValue>four thirty five a m</SynonymicValue>
  <SynonymicValue>four forty five a m</SynonymicValue>
</AnswerValue>
</AnswerType>
7.2 SynonymGenerator.rb

```ruby
class SynGen

attr_accessor :outputfile,
              :hourarrayv,
              :minutearrayv,
              :valueshash,
              :typearray

def run
  initparams
  fillinarrays
  generatdbvalues
  writexmlfile
end

def writexmlfile
  @outputfile.write("<t><AnswerType typeName="busmergedn_buses_fromtimeappr"
  typeOrigin="DB">
  @valueshash.each_key(){|key|
    @outputfile.write("<t><AnswerValue normalizedValue="#{key}"/>

    @valueshash[key].each{|val|
      @outputfile.write("<t><SynonymicValue>")
      @outputfile.write("#{val}")
      @outputfile.write("</SynonymicValue>")
    }
    @outputfile.write("</AnswerType>")
  }
  @outputfile.close
end

def generatdbvalues
  hbefore = hourarrayv.last
  tbefore = typearray.last
  @typearray.each{ |t|
    @hourarrayv.each { |h|
      @minutearrayv.each{|mn|
        str = "#{h} #{mn} #{t}"
        if(str!="three forty A M" && str!="four A M" && str!="four twenty A M")
          if(mn=="")
            @valueshash["#{h} #{t}"] = Array.new
            @valueshash["#{h} #{t} "].push("#{h} #{t}"
            @valueshash["#{h} #{t} "].push("#{h} five #{t}"
            @valueshash["#{h} #{t} "].push("#{h} ten #{t}"
            @valueshash["#{h} #{t} "].push("five after #{h} #{t}"
            @valueshash["#{h} #{t} "].push("ten after #{h} #{t}"
            if(hbefore=="eleven")
              @valueshash["#{h} #{t} "].push("#{hbefore} fifty five #{tbefore}"
              @valueshash["#{h} #{t} "].push("five of #{h} #{t}"
          else
```
@valueshash["#{h} #{t}"].push("#{h} before forty five #{t} before")
@valueshash["#{h} #{t}"].push("#{h} before fifty #{t} before")
@valueshash["#{h} #{t}"].push("#{h} before fifty five #{t} before")
@valueshash["#{h} #{t}"].push("a quarter of #{h} before #{t} before")
@valueshash["#{h} #{t}"].push("ten of #{h} before #{t} before")
@valueshash["#{h} #{t}"].push("five of #{h} before #{t} before")
else
  @valueshash["#{h} #{t}"].push("#{h} before fifty five #{t}")
  @valueshash["#{h} #{t}"].push("five of #{h} #{t}")
  @valueshash["#{h} #{t}"].push("#{h} forty five #{t}")
  @valueshash["#{h} #{t}"].push("#{h} fifty #{t}")
  @valueshash["#{h} #{t}"].push("#{h} fifty five #{t}")
  @valueshash["#{h} #{t}"].push("a quarter of #{h} #{t}")
  @valueshash["#{h} #{t}"].push("ten of #{h} #{t}")
  @valueshash["#{h} #{t}"].push("five of #{h} #{t}")
end
else
  syn = case mn
    when "twenty" then "fifteentwenty five thirty"
    when "twenty" then "ten fifteen"
    when "forty" then "thirty five forty five fifty"
    when "forty" then "thirty"
  end
  @valueshash["#{h} #{mn} #{t}""] = Array.new
  @valueshash["#{h} #{mn} #{t}""].push("#{h} #{mn} #{t}")
  artmp = syn.split("_")
  artmp.each{|val|
    @valueshash["#{h} #{mn} #{t}""].push("#{h} #{val} #{t}")
  }
  syn2 = case mn
    when "twenty" then "a quarter after"
    when "forty" then "ten of"
  end
  if(syn2=\"nil\")
    artmp2 = syn2.split("_")
    artmp2.each{|val|
      if(val==\"ten of\")
        hafter = increment(h)
      if(h==\"eleven\")
        @valueshash["#{h} #{mn} #{t}""].push("#{val} #{hafter} #{t} before")
      else
        @valueshash["#{h} #{mn} #{t}""].push("#{val} #{hafter} #{t}"")
      end
    else
      @valueshash["#{h} #{mn} #{t}""].push("#{val} #{h} #{t}"")
    end
  end
end
end
def increment(h)
    hr = case h.to_s
         when "one" then "two"
         when "two" then "three"
         when "three" then "four"
         when "four" then "five"
         when "five" then "six"
         when "six" then "seven"
         when "seven" then "eight"
         when "eight" then "nine"
         when "nine" then "ten"
         when "ten" then "eleven"
         when "eleven" then "twelve"
         when "twelve" then "one"
        end
    return hr
end

def fillinarrays
    @hourarrayv.push("one")
    @hourarrayv.push("two")
    @hourarrayv.push("three")
    @hourarrayv.push("four")
    @hourarrayv.push("five")
    @hourarrayv.push("six")
    @hourarrayv.push("seven")
    @hourarrayv.push("eight")
    @hourarrayv.push("nine")
    @hourarrayv.push("ten")
    @hourarrayv.push("eleven")
    @hourarrayv.push("twelve")

    @minutearrayv.push(""
    @minutearrayv.push("twenty")
    @minutearrayv.push("forty")

    @typearray.push("a m")
    @typearray.push("p m")
end

def initparams
    @hourarrayv = Array.new
    @minutearrayv = Array.new
    @typearray = Array.new
@valueshash = {}
@outputfile = File.new("H:\Scripts\synonymGenerator\synonyms.xml", "w+", 0644)
end

def closefiles
  @inputfile.close
  @outputfile.close
end

generator = SynGen.new
generator.run
7.3 dialog_acts.xml

```xml
<?xml version="1.0" encoding="UTF-8"?>
<SystemActs size="19">
  <Act id="4" type="ExplicitConfirm/AcknowledgeConfirm" example="Alright"
fillers="confirmslot">
    <FollowingAct id="6" type="ProcessQuery/InformFirstProcessing" example="Hold on. Let me check that for you.">fillers="none">0.255522276301011</FollowingAct>
    <FollowingAct id="5" type="GetQuerySpecs/RequestArrivalPlace" example="What is your destination?">fillers="ArrivalPlace">0.236241108199176</FollowingAct>
    <FollowingAct id="11" type="GetQuerySpecs/RequestTravelTime" example="When are you going to take that bus?">fillers="TravelTime">0.231935604642456</FollowingAct>
    <FollowingAct id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?">fillers="DeparturePlace">0.133845001871958</FollowingAct>
    <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?">fillers="confirmslot">0.115499812804193</FollowingAct>
    <FollowingAct id="13" type="GetQuerySpecs/RequestDepartureStopInNeighborhood" example="Where in neighborhood.name EAST PITTSBURGH are you leaving from?">fillers="DepartureStopInNeighborhood">0.0142268813178585</FollowingAct>
    <FollowingAct id="10" type="GetQuerySpecs/RequestExactTravelTime" example="At what time do you want to travel?">fillers="ExactTravelTime">0.0099213777613815</FollowingAct>
    <FollowingAct id="2" type="GetQuerySpecs/AskHowMayIHelpYou" example="What can I do for you?">fillers="none">0.00187195806813927</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound">fillers="">0.000935979034069637</FollowingAct>
  </Act>
  <Act id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?">fillers="confirmslot">0.711710621824341</FollowingAct>
  <Act id="4" type="ExplicitConfirm/AcknowledgeConfirm" example="Alright"
fillers="confirmslot">
    <FollowingAct id="2" type="GetQuerySpecs/AskHowMayIHelpYou" example="What can I do for you?">fillers="none">0.0682313089765304</FollowingAct>
    <FollowingAct id="5" type="GetQuerySpecs/RequestArrivalPlace" example="What is your destination?">fillers="ArrivalPlace">0.0574643116380353</FollowingAct>
    <Act id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?">fillers="confirmslot">0.0475441567868376</FollowingAct>
    <FollowingAct id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?">fillers="DeparturePlace">0.0384708444229373</FollowingAct>
    <FollowingAct id="10" type="GetQuerySpecs/RequestExactTravelTime" example="At what time do you want to travel?">fillers="ExactTravelTime">0.029397532059037</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound">fillers="">0.0221388821679168</FollowingAct>
    <FollowingAct id="11" type="GetQuerySpecs/RequestTravelTime" example="When are you going to take that bus?">fillers="TravelTime">0.0135494797967578</FollowingAct>
    <FollowingAct id="13" type="GetQuerySpecs/RequestDepartureStopInNeighborhood" example="Where in neighborhood.name EAST PITTSBURGH are you leaving from?">fillers="DepartureStopInNeighborhood">0.00532300992015485</FollowingAct>
</SystemActs>
```
Entering the spoken dialog challenge

<FollowingAct id="12" type="GetQuerySpecs/RequestDepartureNeighborhood" example="In what neighborhood is your departure stop?" fillers="DepartureNeighborhood">0.0048390999274135</FollowingAct>

<FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.0108879748366804</FollowingAct>

<FollowingAct id="6" type="ProcessQuery/InformFirstProcessing" example="Hold on. Let me check that for you." fillers="none">0.000241954996370675</FollowingAct>

<Act id="2" type="GetQuerySpecs/AskHowMayIHelpYou" example="What can I do for you?" fillers="none">

<FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.848296079149872</FollowingAct>

<FollowingAct id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?" fillers="DeparturePlace">0.0864785635764016</FollowingAct>

<FollowingAct id="18" type="End" example="NoExampleFound" fillers="none">0.0626603151337486</FollowingAct>

<FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.00256504213997801</FollowingAct>

</Act>

<Act id="5" type="GetQuerySpecs/RequestArrivalPlace" example="What is your destination?" fillers="ArrivalPlace">

<FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.978712080894093</FollowingAct>

<FollowingAct id="18" type="End" example="NoExampleFound" fillers="none">0.0164981373070782</FollowingAct>

<FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.00372538584353379</FollowingAct>

<FollowingAct id="5" type="GetQuerySpecs/RequestArrivalPlace" example="What is your destination?" fillers="ArrivalPlace">0.00106439595529537</FollowingAct>

</Act>

<Act id="12" type="GetQuerySpecs/RequestDepartureNeighborhood" example="In what neighborhood is your departure stop?" fillers="DepartureNeighborhood"/>

<FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.868131868131868</FollowingAct>

<FollowingAct id="18" type="End" example="NoExampleFound" fillers="none">0.115384615384615</FollowingAct>

<FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.010989010989011</FollowingAct>

<FollowingAct id="14" type="GetQuerySpecs/RequestDepartureNeighborhoodAndStop" example="NoPromptFound" fillers="DepartureNeighborhoodAndStop">0.00549450549450549</FollowingAct>

</Act>

<Act id="14" type="GetQuerySpecs/RequestDepartureNeighborhoodAndStop" example="NoPromptFound" fillers="DepartureNeighborhoodAndStop"/>

<FollowingAct id="13" type="GetQuerySpecs/RequestDepartureStopInNeighborhood" example="Where in neighborhood.name EAST PITTSBURGH are you leaving from?" fillers="DepartureStopInNeighborhood">1.0</FollowingAct>

</Act>
Entering the spoken dialog challenge

<Act id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?" fillers="DeparturePlace">
  <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.857549857549858</FollowingAct>
  <FollowingAct id="9" type="GetQuerySpecs/RequestDeparturePlace" example="Where are you leaving from?" fillers="DeparturePlace">0.107549857549858</FollowingAct>
  <FollowingAct id="18" type="End" example="NoExampleFound" fillers=""></FollowingAct>
  <FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers=""/>0.00284900284900285</FollowingAct>
  <FollowingAct id="12" type="GetQuerySpecs/RequestDepartureNeighborhood" example="In what neighborhood is your departure stop?" fillers="DepartureNeighborhood">0.000712250712250712</FollowingAct>
    </Act>
  <Act id="13" type="GetQuerySpecs/RequestDepartureStopInNeighborhood" example="Where in neighborhood.name EAST PITTSBURGH are you leaving from?" fillers="DepartureStopInNeighborhood">
    <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.915254237288136</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound" fillers=""></FollowingAct>
  </Act>
  <Act id="10" type="GetQuerySpecs/RequestExactTravelTime" example="At what time do you want to travel?" fillers="ExactTravelTime">
    <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.964622641509434</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound" fillers="">0.0330188679245283</FollowingAct>
    <FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.00235849056603774</FollowingAct>
    </Act>
  <Act id="11" type="GetQuerySpecs/RequestTravelTime" example="When are you going to take that bus?" fillers="TravelTime">
    <FollowingAct id="3" type="ExplicitConfirm/RequestConfirm" example="Leaving from query.departure_place.name DOWNTOWN. Is this correct?" fillers="confirmslot">0.887490747594374</FollowingAct>
    <FollowingAct id="10" type="GetQuerySpecs/RequestExactTravelTime" example="At what time do you want to travel?" fillers="ExactTravelTime">0.0947446336047372</FollowingAct>
    <FollowingAct id="18" type="End" example="NoExampleFound" fillers=""/>0.0162842339008142</FollowingAct>
    <FollowingAct id="15" type="StartOver/AskAreYouSureStartOver" example="Are you sure you want to start over?" fillers="none">0.00148038490007402</FollowingAct>
    </Act>
  <Act id="17" type="GiveResults/RepeatSuccess" example="There is a result.rides.0.route_number 61C leaving result.rides.0.departure_stop.name MC..." fillers="none">
    <FollowingAct id="18" type="End" example="NoExampleFound" fillers=""/>1.0</FollowingAct>
  </Act>
Entering the spoken dialog challenge
7.4 User_acts.xml:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<UserActs size="17">
  <Act id="0" type="FillSlot/DeparturePlace" example="none" fillers="DeparturePlace" />
  <Act id="1" type="FillSlot/DepartureNeighborhood" example="none" fillers="DepartureNeighborhood" />
  <Act id="2" type="FillSlot/DepartureStopInNeighborhood" example="none" fillers="DepartureStopInNeighborhood" />
  <Act id="3" type="FillSlot/DepartureStopInNeighborhood" example="none" fillers="DepartureStopInNeighborhood" />
  <Act id="4" type="FillSlot/TravelTime" example="none" fillers="TravelTime" />
  <Act id="5" type="FillSlot/ExactTravelTime" example="none" fillers="ExactTravelTime" />
  <Act id="6" type="FillSlot/DepartureArrivalPlace" example="none" fillers="DeparturePlace/ArrivalPlace" />
  <Act id="7" type="FillSlot/DepartureArrivalPlaceAndTime" example="none" fillers="DeparturePlace/ArrivalPlace/TravelTime" />
  <Act id="8" type="FillSlot/DepartureArrivalPlaceAndExactTime" example="none" fillers="DeparturePlace/ArrivalPlace/ExactTravelTime" />
  <Act id="9" type="FillSlot/DepartureArrivalPlaceTimeAndExactTime" example="none" fillers="DeparturePlace/ArrivalPlace/ExactTravelTime/TravelTime" />
  <Act id="10" type="FillSlot/DeparturePlaceAndTime" example="none" fillers="DeparturePlace/TravelTime" />
  <Act id="11" type="FillSlot/DeparturePlaceAndExactTime" example="none" fillers="DeparturePlace/ExactTravelTime" />
  <Act id="12" type="FillSlot/DeparturePlaceTimeAndExactTime" example="none" fillers="DeparturePlace/ExactTravelTime/TravelTime" />
  <Act id="13" type="FillSlot/ArrivalPlaceAndTime" example="none" fillers="ArrivalPlace/TravelTime" />
  <Act id="14" type="FillSlot/ArrivalPlaceAndExactTime" example="none" fillers="ArrivalPlace/ExactTravelTime" />
  <Act id="15" type="FillSlot/ArrivalPlaceTimeAndExactTime" example="none" fillers="ArrivalPlace/ExactTravelTime/TravelTime" />
  <Act id="16" type="Confirm/AnswerConfirm" example="none" fillers="Yes/No" />
</UserActs>
```
7.5 Evaluation.log
Evaluation logged on : Sun Aug 08 14:07:30 BST 2010
For user profile : fluentEnglish_veryAccurateAnswers

minimum number of user acts : 1.0
maximum number of user acts : 21.0
mean number of user acts : 7.192

minimum number of system acts : 3.0
maximum number of system acts : 25.0
mean number of system acts : 10.593

minimum number of dialog acts : 4.0
maximum number of dialog acts : 46.0
mean number of dialog acts : 17.785

minimum GA run-time : 1.215
maximum GA run-time : 5.486
mean GA run-time : 1.3748550204918042

number of unsuccessful dialogs : 0
number of successful dialogs : 981
number of hang-ups : 19