AINT512 – Science and Technology of Human Robot Interaction

Talking Plant Finder Report

By Tom Queen

Contents

[Justification 2](#_Toc513673602)

[1. Task domain vocabulary specification 2](#_Toc513673603)

[2. Interaction scenario specification / or defined from conducted observations. 2](#_Toc513673604)

[Both Scenarios 2](#_Toc513673605)

[Scenario One – Normal Search 2](#_Toc513673606)

[Scenario Two – Reverse Search 3](#_Toc513673607)

[Network architecture 3](#_Toc513673608)

[3. Personality specification 4](#_Toc513673609)

[4. Grammar design 4](#_Toc513673610)

[5. Dialogue design 4](#_Toc513673611)

[6. Error catching / handling 5](#_Toc513673612)

[Error catching 5](#_Toc513673613)

[Error handling 5](#_Toc513673614)

[Reverse search 6](#_Toc513673615)

[7. Action / response generation 6](#_Toc513673616)

[8. Evaluation 7](#_Toc513673617)

[Further Improvements 7](#_Toc513673618)

[Bibliography 9](#_Toc513673619)

[Appendix 1 – Full system flowchart 10](#_Toc513673620)

# Justification

Interaction with standard databases are only possible if the user knows how to execute SQL queries. GUIs designed to allow untrained users to request information from a database generally fall in one of two categories: overwhelming or inefficient.

Spoken interfaces can provide a faster, more intuitive method for accessing complex databases by untrained users, whilst still providing methods of searching and exploring the database for users who don’t completely know what they are searching for yet [1].

This projects focus is the development of an interactive speech interface to a database of plants compiled by Plants For A Future (PFAF) [2] with the aim of producing a system in which untrained users can find a plant faster than they could with a traditional GUI interface.

# 1. Task domain vocabulary specification

The vocabulary for this system was generated via the bootstrap method. Some obvious sentences were constructed in the grammar which was then tested on people other than the author. Whenever an out-of-grammar sentence was spoken, it was written down and added to the grammar. The iterative cycle was then repeated.

# 2. Interaction scenario specification / or defined from conducted observations.

This program is intended to serve as an assistant in garden centres. One or many kiosks would be setup around the garden centre, with either a built-in microphone and speaker or a tethered headset. The system developed in this report has been designed to address two main scenarios: finding a plant to match an environment, and finding an environment to match a plant. An example dialogue of each scenario is presented below.

## Both Scenarios

In both scenarios a user will either discover the kiosk or be shown to it by a member of staff at a centre. System voice outputs are highlighted in blue and user responses are highlighted in grey.

“Welcome to the talking plant finder. You are currently searching for plants that will grow in Britain. Would you like to change that?”

If the user says yes they will be asked which country they would like to change to, if not the program will continue.

"Would you like to find a plant to match an environment (normal search), or see what environments match a plant (reverse search)?"

## Scenario One – Normal Search

In this scenario, a customer has an idea of where they are planning on growing a plant but hasn’t yet found a plant that will grow well in these conditions.

“How can I help?”

The user can begin entering search criteria, or can ask for help.

“Do you have any suggestions?” or “Can I have an example please?”, or simply “Help!”

“Why not try searching for where the plant will grow, such as **East Wall**, or **Cultivated Beds**. Or you could try searching by type of plant, like **Bulb**, **Bamboo** or **Biennial**. You can also specify how hardy you want the plant to be.”

(Words in **bold** are randomly generated each time the user asks for suggestions)

“Okay, can you help me find an extremely hardy tree to grow against a shady edge?”

Blue writing extracted from speech, SQL query automatically constructed:

**SELECT** `Common name`**,**`Latin name` **FROM** `plantlocations` **WHERE** `Habit`**=**'Tree' **AND** `ShadyEdge`**=**1 **AND** `Hardyness` **BETWEEN** 9 **AND** 10

Error checking and other validation performed here (see error handling / catching for more information). After error checking:

“Okay, here are your results. Common name and then Latin name:

Puahou - Pseudopanax arboreus “

“Would you like more information on this plant?”

If the user says yes, they are directed to the reverse search.

## Scenario Two – Reverse Search

In this scenario the user has a plant in mind but wishes to find out more about its ideal growing conditions. By knowing the common or Latin name of a plant, the user can perform a ‘reverse search’.

“Reverse search.”

"Please say the common or Latin name of a plant you want to search for."

“Garlic”

Generated query:

**SELECT** **\*** **FROM** `plantlocations` **WHERE** `Common name`**=**'Garlic' **GROUP** **BY** `Latin name`

Vocalised response:

“Latin name Allium sativum”

“Habitat Not known in a truly wild situation.”

“Habit Bulb” “Height 0.6” “Width 0.15” “Hardyness 8”

The user is then asked if they wish to use the service again. If yes, they are returned to a state where they can select ‘normal’ or ‘reverse search’.

## Network architecture

Network architecture would depend on the size of the garden centre. For small centres, the program and database would be hosted on a relatively lightweight computing platform (such as a NetTop). For medium-sized garden centres, the database would be hosted on a central SQL server and NetTops would be distributed throughout the centre, each processing its own speech recognition and text-to-speech. For larger garden centres, a centralised SQL server and centralised nuance server would be setup, and users would interact with the system via thin clients placed around the garden centre.

To increase reliability, headsets would be the primary user interface. To further develop the system, microphone arrays could be used to cancel background noise and isolate an individual user [3], removing the need for a headset.

# 3. Personality specification

Research has shown that users will assign personalities to machines. It is the developer’s incentive to build personalities into their products. If they do not, then users will still assign personalities onto a developer’s machines but they may not be favourable. [4]

Care has been taken to reduce user frustration at every stage of the dialogue, however no official personality has been specified for this system.

# 4. Grammar design

For the first prototype, a grammar was developed which filled one slot with a string corresponding to a spoken search term. More search terms could be spoken sequentially by the user which would update this slot with a new string to be added to the search.

(canopy) {<env\_said canopy\_said>}

(east wall) {<env\_said east\_wall\_said>}

For the “free-speech” version of this program, this type of slot filling was no longer appropriate as a slot could only be filled once per user sentence. To remedy this, each potential search term was assigned its own slot, returning an integer ‘1’ if filled.

(canopy) {<canopy\_said 1>}

(east wall) {<east\_wall\_said 1>}

This allowed multiple slots to be filled from a single sentence.

This approach required the generation of an if statement for each potential search term. As there are more than thirty search terms in the primary category (and many categories), python code was written to generate this C code. A set of column headings were exported from MySQL into a text file. Macros were written to convert this list of criteria into a python list structure. Python loops were then appended to this text file which iterated though the list, generating a series of C if statements as well as parts of the grammar. This method was reapplied to multiple parts of the C program.

# 5. Dialogue design

The initial dialogue was a menu-based, system initiative, finite state dialogue which revolved around the system printing a list of all potential search terms to the screen. A user could verbally select a term and it would be added to the search. Search terms were added sequentially, so error handling functionality was limited to allowing the user to remove the last term only.

This system was less complex and so it was easier to add multiple types of search term. Prototypes included functionality for searching for plants by medical use and as replacements for foods. This functionality did not make it into the final version due to time constraints.

The system worked reasonably well but relied upon visually prompting the user with potential search terms, and required the user to pre-specify the category a search term belonged to. It was suggested that a touchscreen GUI would be a more effective interface to this menu-based solution. In order to better utilise the potential advantages of speech interfaces the decision was made to move to a “free-speech” type system where a user could speak any number of search terms in any order, even though this technically reduced the functionality of the system by removing access to several search categories.

The second prototype used a system initiative driven finite-state machine to direct the user to an Information State Update (ISU) based free-speech state. When the user gets to the free-speech state, user initiative dialogue takes over, with suggestions and error handling offered by the system. The ISU approach allowed the user to say multiple pieces of information at once, and then prompted the user if they had missed any critical information (i.e. the user has not said any environmental categories).

Speech recognition reliability decreased as more search terms were said in one sentence. It was rare to have the system successfully recognise a four-term search without adding in extra terms that the user did not say. Because of this, a more thorough error checking subsystem was developed.

# 6. Error catching / handling

The primary error catching is performed by the “envCheck1” state, and error handling is performed between the “envCheck” and “envErrorFix” states. Code for these states can be found in env\_states.c. Error handling also appears in the full system flowchart in appendix 1.

## Error catching

Error catching is primarily performed in the “envCheck1” state. This state accesses the arrays which have search terms stored in them and vocalises these search terms to the user. The user is then asked whether the spoken terms are correct. If the terms are correct, the number of matching search results are queried and vocalised. If there are any mistakes, the “envPreFixCheck” state is called which asks the user if a spoken term was missed (false negative detected), or if a term was recognised that the user did not say (false positive detected). For false positives, “envCheck” is called. For false negatives, the user is returned to the “freeSpeech” state where they can add search terms.

## Error handling

“envCheck” starts by asking the user which of the recently vocalised terms was misheard. The user then says which terms were incorrectly identified. Once the user has finished speaking, the system will change to the “envErrorFix” state which removes incorrect terms from its arrays.

To allow the user to speak multiple corrections in one go, grammar slots were created for each possible search term which are set to a ‘1’ when they get filled. An individual if statement was written for each of these slots. Below is an example of one such if statement:

NLGetIntSlotValue(AppGetNLResult(app), "perennial\_climber\_said", &perennial\_climber);

if (perennial\_climber == 1) {

printf("%s\n", "You said Perennial Climber.");

strcpy(errorStoreArray[errorEnvStringCounter], "Perennial Climber");

errorEnvStringCounter = errorEnvStringCounter + 1;

}

To detect when the user had finished speaking corrections, the following code was written:

if ((AppGetRecognitionStage(app) == 4)) {

AppGoto(app, "envErrorFix");

}

Where “ …) == 4)” corresponds to “AFTER\_SPEECH\_STAGE” in the enumerated “RecognitionStage” structure returned by “AppGetRecognitionStage”.

## Reverse search

The reverse search grammar contains approximately 4000 common plant names. Due to the vast number of words in this grammar it was rare for nuance to not find a matching word, although it was common for this word to be incorrect.

Error checking for this function was trialled by checking the confidence of the word found by nuance. It was thought that, by introducing a confidence threshold for terms spoken to the reverse search, it would be possible to decrease the rate of false positives in the reverse search. However, nuances confidence did not often correlate with the word being detected correctly.



Figure - Confidence checks - Garlic said every time

Figure 1 shows the detected word and its confidence value. For all three of the lines above, the word “Garlic” was said by the user, yet the spoken word was only found once and when it was found it had a lower confidence than the erroneous terms. For this reason the confidence threshold was removed from the reverse search.

# 7. Action / response generation

This program relies heavily on the MySQL Connector/C (MySQL C API). This API provides a programming interface (in C) to communicate with an SQL server hosted locally or remotely. Data is requested by constructing SQL query strings and submitting them via a MySQL object created from the API. If the user had asked the system “Can you help me find a Climber plant that will grow well against an East Wall”, the following SQL query could be generated:

**SELECT** **COUNT(**`Latin name`**)** **FROM** `plantlocations` **WHERE** `Climber`**=**1 **AND** `East Wall`**=**1

“Climber” and “East wall” are both column titles from the same `plantlocations` table, so this query is simple to construct in code. As the user selects categories from multiple tables the queries quickly get more complex. Please see the “constructAdvancedString” function in env\_states.c and all functions in sql\_functions.c for code on how the queries were constructed.

To generate these queries, search terms spoken by a user are stored in various arrays depending on the category the search term belongs to. When the user requests their results (or a count of how many results match their currently selected search terms), the program will construct strings from these arrays in the format of SQL queries and submit them to the MySQL server containing the database. Results from the database are generally vocalised via nuances AppAppendTTSPrompt function.

# 8. Evaluation

The final system was trialled with three users (one of which considered themselves knowledgeable about plants). The user who knew plants found the system easier to use as they more frequently used the same words as those in the database.

The most common source of error was users searching for categories that had not yet been integrated into the speech system, like plant height or soil conditions. This information is in the database but has yet to be incorporated into the code/grammar.

The second most common source of errors were synonyms. One user tried to search for ‘garden bed’, not knowing that the system developed only recognised ‘cultivated bed’. Synonyms could potentially be added automatically, but great care would have to be taken to avoid nonsense grammars forming. In practice, if synonyms were automatically generated, it is likely that the developer would have to check through them individually.

## Further Improvements

The system which has been developed doesn’t handle negatives. For example, “I’m looking for extremely hardly bamboo that willgrow well **away from** shade” could return “hardy = 9, bamboo, shade” if **away from** is missed by the microphone.

The current project solution is just over 10,000 lines of code, split across eight C files. 6,000 lines of this is two copies of a 3,000 line array. A series of just-in-time (dynamic) grammars coupled with a redesign of the C program could drastically increase the amount of information that can be accessed from the database. Upon start-up, the C program could access the SQL database of plants, read column titles / categories and populate arrays with this information. A series of sequential if statements could be replaced with while loops that check for strings from the populated arrays. The C program could then add the contents of these arrays to a nuance dynamic grammar. By autogenerating some grammars using categories from the database, the program could be configured to potentially access many more categories than are currently implemented whilst simultaneously reducing code size and complexity. This would also mean that data could be added to the database without having to also add additional functions and strings to the C program.

Another improvement to this system would be enabling barge-in so that nuances speech output could be interrupted, which could shorten the amount of time it takes for a user to find information they want from the database. To enable barge-in appropriate changes were made to batch files, command line arguments were fed into visual studio and enable barge-in was called from the C program via the nuance API. This generated an error in nuances log files regarding audio hardware being only half-duplex. The audio hardware used in this project supports full duplex, but the version of nuance used only allows barge-in if the developer is using one of three pre-approved hardware systems (generally telephony servers). As this hardware was unavailable, barge-in could not be enabled.

A semantic analysis stage would allow for a more natural dialogue. For example, the word ‘actually’ could be used to replace the last search term in a sentence in either the grammar or C code. E.g:

“Can you help me find a hardy climber that will grow well against a north-facing wall. Actually, east-facing”.

Slots filled: Hardy, climber, east-wall.

Thus, saving the user from having to go through the error handling dialogue to amend their search choice.

# Bibliography

|  |  |
| --- | --- |
| [1] | S.Nareshkumar, N.Mariappan and K.Thirumoorthy, “Database Interaction Using Automatic Speech,” *International Conference on Innovations in Engineering and Technology (ICIET’14),* 2014. |
| [2] | K. Fern and A. Fern, “Plants For A Future (PFAF),” [Online]. Available: https://www.pfaf.org/user/Default.aspx. [Accessed 2018]. |
| [3] | Z. Zhang and M. Etoh, “High-Speed Noise Cancellation with Microphone Array,” *NTT DoCoMo Technical Journal,* vol. Vol. 9 No. 4.. |
| [4] | B. Kraus, “Personas in Technology Design,” Adductive, 2008. |
| [5] | D. J. Litman, J. B. Hirschberg and M. Swerts, “Predicting Automatic Speech Recognition Performance Using Prosodic Cues,” *NAACL 2000 Proceedings of the 1st North American chapter of the Association for Computational Linguistics conference,* pp. 218-225, 2000. |
| [6] | G. Bugmann, “L3\_Grammars\_18,” Plymouth, 2018. |

# Appendix 1 – Full system flowchartC:\Users\Aorus\AppData\Local\Microsoft\Windows\INetCache\Content.Word\AINT512 - Full flowchart.png