Final Project Thoughts: Multistep Vocal Command

Edward L. DeCoste & Andrea Youwakim

Freshman Research Initiative
Department of Computer Science
University of Texas at Austin

Task

The problem we wished to resolve was multistep voice recognition delivered orally by any user. A systematically variable and iterative algorithm divided into a three level hierarchy allows the Building Wide Intelligence (BWI) robots to assess data auditorily. This task has a multitude of applications because it would allow the robot to interpret a set of tasks instead of accepting only a single task at a time. Our project's goal was to build upon the BWI Robots' existing speech to command functionality by building a framework to allow the robots to perform sequential operations based off of natural speech patterns. Queueing subtasks in the same order by which they are received allows this implementation of the scheduler to maintain initial command integrity. By enhancing the speech-to-text recognition abilities, we were able to enhance the robot's ability to operate on a more autonomous level.

Implementation

To achieve our goals, we created a hierarchically intelligent control system. We chose to divide the algorithm into three hierarchical levels: Pre-Organization, Organization, and Command & Control. With the strict sectors and properties of these three levels, we are able to

implement a far more intelligent and versatile algorithm. This allows for an increased level in algorithm output precision even when input precision is not intact.

Pre-Organization Level

The Pre-Organization level handles the interfacing with the speech-to-text (STT) libraries. The current implementation interfaces with the PocketSphinx library. We chose this one as it is a completely offline solution that can be used with a custom language. The main purpose of our SpeechAPI class is to provide an interface that can handle and desired Speech-To-Text (STT) API that the user desires to utilize. The implementation we utilized interfaces with the PocketSphinx library and the various APIs which the library implements. This implementation uses a custom language model to parse the vocal commands the robot acquires via auditory extensions. Custom Language models are often written about in published papers on human-computer interaction and structured language models. In the paper “Structured Language Modeling,”[1] the use of syntactic structure in natural language is described for the implementation of more sophisticated language models for speech and command recognition. Our implementation model merges techniques in parsing and language using an original parameterization of command types. In the paper “Natural language processing (almost) from scratch”[2], the goal of chunking is to label segments of a sentence with its syntactic meanings such as noun or verb phrases. Each word is assigned a single unique tag. This information is directly represented in the project by using a custom language model in conjunction with PocketSphinx.

Organization Level

This level of the library handles the parsing of and creating the subtasks from the initial input. Inside of the parser, it checks the validity of the sub task through checking the keyword command and its required parameters. To create extract each subtask, we employ regular expressions.

```
turn_command = ("turn", "\\bleft\\b|\\bright\\b")
```

Fig. 1 Turn command definition with the first option of the tuple being the command title and the second option as the parameters of the command.

This definition defines the “turn” command and a regular expression that defines the input parameters. For this command, we accept a single word parameter, either “left” or “right.” The “\\b” signifies the word boundary within the expression, and the “|” character is the logical OR.

Once the command is detected as valid, we must serialize it to send to the next level of the hierarchy. This is done through a simple string representation of the command in a form that the scheduler can decipher and understand. This keeps the package small and versatile as less code is needed to transfer the data between the nodes. If a command is not recognized by the system, an informational message is printed in the ROS Console and the algorithm attempts to recover by continuing the parsing of the rest of the input.

Command & Control Level

The final level is how the library interfaces with ROS. Once a serialized command is received, it looks though its command-function dictionary and finds the appropriate logic to

perform. We are able to make the assumption that every received command is valid as it is the Organization Level's job to verify each command.

```
@staticmethod
def turn(direction):

    goal = MoveBaseGoal()
    goal.target_pose.header.stamp = genpy.Time()
    goal.target_pose.header.frame_id = "/base_link"

    dirs = {
        'left': 90,
        'right': -90
    }

    quaternion = transformations.quaternion_from_euler(0, 0,
dirs[direction])

    goal.target_pose.pose.orientation.x = quaternion[0]
    goal.target_pose.pose.orientation.y = quaternion[1]
    goal.target_pose.pose.orientation.z = quaternion[2]
    goal.target_pose.pose.orientation.w = quaternion[3]

    return Command('/move_base', MoveBaseAction, goal)
```

Fig. 2 The definition of the turn action within the scheduler.

Figure 2 is a continuation of the previous example of the turn functionality. We define a turn function that takes in the "direction" parameter. The passed in parameters are handled by the main scheduler code, so adding an implicit parameter is allowed. The defined function should return a Command object that takes in the parameters:

- Path: String of the ROS path to publish to
- Type: Type of the data being used in the command

- Data: Data that the command object holds

Through the use of this object, we have created a dynamic scheduler that will work with any type that can be put through the built in SimpleActionServer class.

Scalability and Modularity

A primary goal for the project was to keep it simple, modular, and scalable. We were able to achieve these goals through various optimizations in the code such as the extraction of the SpeechAPI class and the implementation of a simple way to create new functionalities.

When designing the SpeechAPI, we implemented it in such a way that it is small, easy to interface with new STT APIs, and be completely optional in the usage of the library. To interface with a new STT API, all that is required is to change the listening node's path and update any data-type dependencies. The use of the class in the library is made optional by having it separated from the core logic. A user can publish directly to the core and skip the STT parsing completely. This would be useful in a situation where a user needs to provide the robot human readable commands, but the environment is not suitable to verbally issue the commands.

Designing a new functionality was intentionally made simple to allow the continuation of the project with new developers. Throughout the paper, we gave code snippets for the turn functionality. The total lines of code for that functionality is less than 15, and that is the complete functional and linguistic definition of the feature.

Accomplished Goals and Analysis

By the end of the semester, our goal was to have the robots processing speech and performing actions independently once the initial input has been processed. Initial performance

testing included a series of phrases that were delivered at one time. Each sentence or command was separated by key delimiters. Such phrases included “Go to xyz room and say hello” and “Say xyz then spin 90 degrees.” These initial goals were modified to the following functionally complete set: forwards and backwards movement, turning left and right, and speech synthesis. Combined, this base set of commands allows for movement in all of the cardinal directions. We chose to limit the scope of the project in order to provide the most complete and functional product as possible. Our iterative algorithm written in Python is a modified version of the hierarchically intelligent control system described by Sardis in his publication “Intelligent robotic control”[3]. This type of system allows us to employ a pattern matching system to analyze and classify the input data. The implementation uses a custom language model to parse the vocal commands the robot acquires via auditory extensions. One of the main benefits of using a custom language model is the ability to limit the field of understanding for the robots. This allows for precise execution on certain commands that way the code can be further accurately developed in the future.

Testing

We tested the robot by giving it a series of commands that were parsed using the systematically variable and iterative algorithm that we implemented. For instance, the robot should be able to receive commands such as “Turn left and move forward, then say hello” and “Move backwards then turn left and say I need help”. Notice that each task must be separated by a key delimiter phrases— in this case, the words “then” and “and.” Other delimiters may include “afterward” and “followed by.” If the robot is unable to properly parse the series of commands and therefore cannot complete the given tasks, then we knew that the tests failed and revisions needed to be made.

In reference to long term applications, the most efficient way to test this system is to use different voices and dialects with each available function that has been implemented on the robot. This will allow for the most accurate speech detection and operation as well as broaden the field of accessibility to various non english-speaking users. Another testable feature is that the robot is able to complete each task in order and keep track of an accurate list of tasks that are to be completed. Other goals for the future is to incorporate this with other projects - such as the robotic tour guide project - and other applications. In reference to the use of PocketSphinx, it was an excellent starting library, however, the library has many limitations and it would be much more helpful to upgrade to a more sophisticated and versatile library.

Resources

PocketSphinx allowed us to utilize enterprise grade speech to text software. Future goals are to incorporate current top platforms such as Google Speech and Alexa Speech Service.

Works Cited

[1] Collobert, Ronan, et al. "Natural language processing (almost) from scratch." Journal of Machine Learning Research 12.Aug (2011): 2493-2537.

[2] Chelba, Ciprian, and Frederick Jelinek. "Structured language modeling." Computer Speech & Language 14.4 (2000): 283-332.

[3] G. Saridis, "Intelligent robotic control," in IEEE Transactions on Automatic Control, vol. 28, no. 5, pp. 547-557, May 1983. doi: 10.1109/TAC.1983.1103278