Voice-Controlled Chess Game on FGPA Using Dynamic Time Warping

Submitted by Michael Kuo and Varun Chirravuri on October 31, 2008

Project Overview

Our team plans to build a real-time, voice-controlled chess game on the 6.111 labkit. To build this, we rely on a dynamic programming technique called dynamic time warping (DTW) to analyze spoken commands, and a chess game-engine built on the kit.

Michael Kuo will be building the chess game-engine as well as working on the visual display of the chess game on the VGA display. Varun Chirravuri will be building the voice command recognition module.

Voice Recognition Module (Varun Chirravuri)

Description

The voice recognition module will have two main functionalities – training and active recognition. In the training phase, the user will sequentially issue all of the possible voice commands, and samples of the voice commands will be stored on the FPGA for later comparison. During the active recognition phase, input audio samples will be compared against the samples stored from training to see if a match is detected.

Sampling

Audio will be sampled by implementing a version of the AC97 audio codec from lab 4 with only a record functionality. The samples will be passed through a low pass filter to remove any aliasing and high frequency noise that may enter as a result of the sampling process. The number of taps of the FIR filter will be maximized based on the available onboard memory and the number and size of samples we will store.

Memory Use

We initially intend to store the samples in BRAM memory, as multiple read/write accesses may be needed at every clock for the dynamic time warping algorithm. We intend to store 8 bit samples of the data to the BRAM, and the precise number of samples will be determined once the command list is finalized (it should be well under the 64k in lab 4 considering that our commands should last at most 3 seconds each).

Audio Recognition

To determine when a valid audio sample is entered versus background noise, we rely first on the low pass FIR to clean up our signal. After the filtered sample is taken in, we will compare the magnitude of the sample to the previous few samples. If the magnitude of the incoming sample is significantly greater than the previous samples, we can say that a sound event is occurring. Once a sound event is registered, we will begin storing it as a potentially valid audio clip. To be stored as a valid audio clip, not only does the magnitude of the first sample of the clip have to be significantly greater than the previous samples, but the subsequent samples must remain at an approximately equal magnitude for a certain amount of time. A valid audio clip ends when the magnitude of the incoming sample is much less than the magnitude of the previous sample, and remains that way for a specified number of clock cycles. This ensures that while loud and abrupt noises will not be recognized as valid samples, pauses or breaks in words (such as the brief pause between the *d* and *m* in the word *grandma*) will not be registered as multiple audio clips.

Dynamic Time Warping

The majority of the voice comparison will be performed using an algorithm called Dynamic Time Warping (DTW). DTW algorithms find similarity between two sequences that may vary in speed or occur for different lengths of time. The algorithm relies only on the time domain samples of the audio, and uses non-linear comparisons to remove the non-linear time variations in the two samples to find the proper sequence alignment.

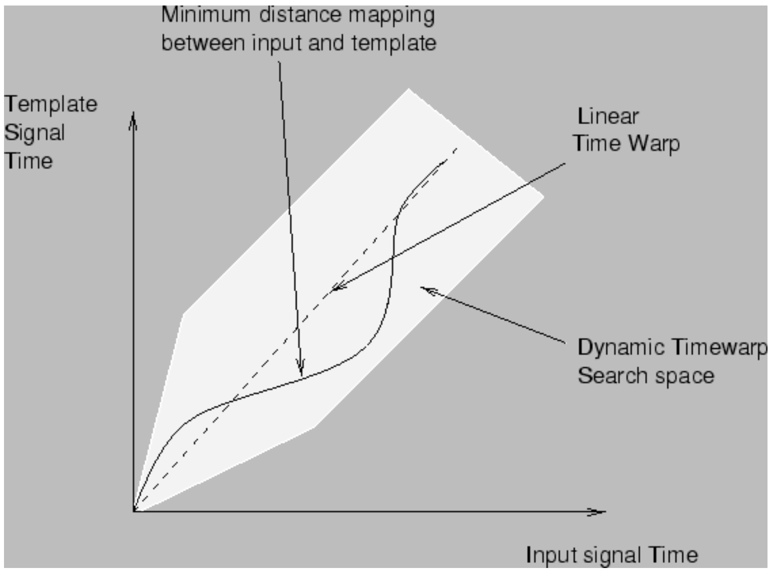
Minimum distance mapping between input and template
Template Signal Time
Linear Time Warp
Dynamic Timewarp Search space
Input signal Time

Figure 1: Dynamic Time Warping (http://www.cse.unsw.edu.au/~waleed/phd/html/node38.html)

The algorithm will compare audio samples against all of the stored training samples and try and find the closest match. Unfortunately, DTW is a relatively slow algorithm, running in $O(N^2S)$ time, where N is the size of the sample and S is the search space (http://www.cse.unsw.edu.au/~waleed/phd/html/node38.html). To limit the search space for each sample, each command will be stored and analyzed by its smallest sub-command. For example, if the input phrase is, "eat three apples," then first "eat" would be compared to the closest match in the *verb* category, *three* would be searched in the *adjective* category, and *apples* would be searched in the *noun* category. Also, commands will be chosen to be as brief as possible and comparisons will be optimized to run on as few samples as possible such that N does not grow too large.

Output

After commands are analyzed and potential matches are found, the command will be passed to the chess game-engine as a string of bits that uniquely characterizes which piece should be moved where. The legality of the move will be determined by the game-engine.

Pre-Implementation Design and Testing

Before implementing the DTW algorithm on the FPGA, a working model will be built and tested in Matlab to find out which parameters are most important to the correct functioning of the Voice Recognition Module. Once the model is shown to work, it will be implemented in Verilog. Testing will be done by assigning stored commands a numeric value, and flashing that numeric value on the LED display when a match between samples is registered. By examining the percentage of correct matches to misses and false positives, the algorithms parameters will be corrected.

Chess Game-Engine and Graphics (Michael Kuo)

Description

The chess game will consist of three main components, a game state-machine, a constraints checker, and graphics generation. Inputs will be delivered to the chess game from either the voice recognition module or from a manual, keyboard based input, and outputs will be displayed on the VGA display.

Input

To modularize the voice recognition module and the chess game-engine, the game-engine will receive inputs from either the voice recognition module or keyboard. Input from the voice recognition module and keyboard will be converted to the same signaling schema to abstract the input source.

Game Representation

The chessboard consists of eight rows, or ranks, and eight columns, or files. The ranks are numbered 1-8 and the files are lettered A-H. After setup of the game, the white pieces occupy ranks 1 and 2 and the black pieces occupy ranks 7 and 8.

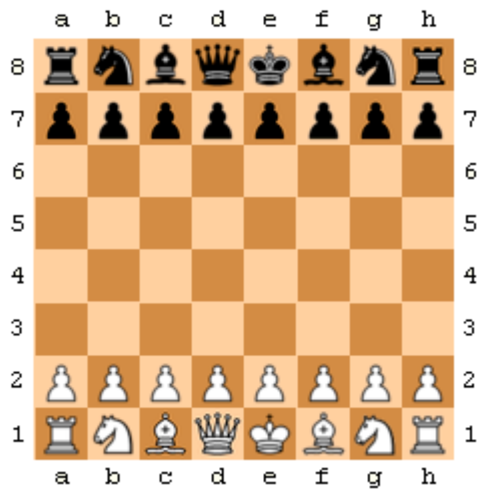

Figure 2: Chess Board
From (http://en.wikipedia.org/wiki/Chess)

Each square of the chessboard will be represented by one of 64 locations of memory. The data at the memory location will consist of the type and color of the piece occupying the corresponding square. For Kings and Rooks, an extra bit will be used to denote whether the piece has moved from its initial position. This will allow the constraint checker to determine if a castling move can be made. An auxiliary bit will also be used for Pawns to denote whether a Pawn made a two square advancement on its initial move. This denotation will only be kept for one turn and will be used to determine if the opposing player can make in en passant capture. With three bits for the type of piece, one bit for the color of the piece, and two additional bits for the auxiliary bits on the Kings and Rooks and Pawns, the piece at a square can be represented in 6 bits.

Game Logic

The game logic keeps track of which player has the current move and handles the movement of the pieces on the chessboard. It will use the constraint checker described later to verify the legality before carrying out any moves. The game logic will also watch out for "check-mate" situations to determine the end of a game.

Constraints Checker

A player's move is represented by a tuple consisting of the piece being moved, the file being moved to, and the rank being moved to. For example <Queen, E, 5> indicates the move of the Queen to the square E5. In the case that two pieces of the same type can move to the same square, the distinguishing rank or file of the piece being moved is included in the tuple. For example <Rook, A, D, 6> indicates the move of the Rook in the A file to the square D6.

The legality of a player's move will be checked by the constraints checker before the move is carried out in the game-engine. At the basic level, move verification determines whether the desired move can be made by the type of piece being moved. However, move verification will also determine whether a move puts a player in "check". As noted before, auxiliary bits on the Kings and Rooks, as well as the Pawns, will aid in verifying castling moves and en passant captures.

The player will be notified of an illegal move on the display, as well as with one of the LEDs on the LabKit. The game-logic will then accept a new move from the player.

Graphics Generator

The configuration of the chess pieces on the chessboard will be displayed on an XVGA display (with 1024 x 768 resolution). Since the only changing elements of the graphics are the chess pieces, the chess pieces and chessboards will be represented on two different layers. The bottom layer will consist of the chessboard, while the top layer will consist of the moving pieces. Rendering of the top layer will have priority over rendering of the bottom layer. Twelve sprites, corresponding to the six types of pieces in black and white, will be used to render the chess pieces on the display.

Potential Problems

There are a few areas in designing and building the system that might pose some serious challenges.

The DTW algorithm has two main problem areas: computational requirements, debugging:

The first is due to the potentially enormous upper-bound running time of the algorithm – $O(N^2S)$ – that could require too much time or memory to run in a reasonable amount of time. One solution that was proposed earlier, to limit the search space and minimize the command sample size, may also not solve this problem. Breaking commands into smaller subcommands also means that the algorithm needs to run more times on smaller search spaces. This may still be too inefficient to complete in the time we have.

Debugging the DTW algorithm will also be increasingly difficult, as the cause of failures might not particularly apparent. Because of the complexity of computation, it may not be easy to pinpoint which metrics are creating misses, or false positives. A simple way to test if the algorithm does work is to test its ability to identify identical samples, for instance, a clip of a song or a pure tone—but how to extend this success to time varying signals is not so apparent. Also, as in most voice command applications, ours will have an optimal success rate at detecting commands, and it is possible that this rate is simply too low to implement.

The chess game-engine has one main problem area: effectively minimizing the number of commands while still enumerating every move.

To ensure that the voice recognition module does not need to store a large number of different samples to specify each chess move, we will have to be clever in designing how we name pieces and moves. For example, it would be redundant to specify which bishop moves to which spot, since one side's bishops only move on one color. On the other hands, knights and rooks can move to the same location, and pawns can attack to the same position on the board. On the other hand, if we uniquely name every piece, the number of commands increases and remembering moves during game-play becomes cumbersome.

System Integration and Extensions

The entire system will be driven by a 65 mHz clock. Since it is good engineering to run a system off of one clock to prevent timing errors and because the video module will need to be run off of the 65 mHz clock, we will adjust the timing and sampling parameters of the voice recognition module to compensate. The modularity of the two subcomponents aims to minimize any integration problems.

One possible exceed goal for the project is to allow for gameplay between lab kits. Using an RS232 interface to connect two lab kits, the goal would be to have two players looking at different screens at different lab benches play each other in voice-controlled chess. If executed properly, the game could toggle between single and multi-kit play by flipping a switch.

Another exceed goal of this project is to allow for multiple voice recognition on the box. In a sense, the game would play like a real voice-controlled game, in which players need not program their machine each time they play, and any user's voice could be adequately recognized by the system. While it is possible that the DTW algorithm is capable of doing this as is, we might have to train the system on many samples of many voices for each command, and take some sort of weighted average of them all. It is also possible that, along with the DTW algorithm, we would also need to use the frequency domain characteristics of the samples (via an fft algorithm) along with improved similarity metrics to better recognize commands from any voice.