qSonify: Decompiling quantum algorithms through sonification

Joseph Iosue

10 December 2018

https://github.com/jiosue/qSonify

Motivation

For my final project, I've created a Python package called *qSonify* to study the usefulness of using music to help find patterns in quantum algorithms. My motivation for this idea came from many sources. Firstly, in computer science people often talk about the idea of abstraction. Computer programmers today do not still need to code in terms of elementary circuit operations, ie AND, OR, and NOT gates. Instead, we have abstractions that get compiled down to a sequence of these elementary gates. Abstraction is an incredibly powerful tool. In classical computer science, we are usually interested in compiling our abstraction down to machine language. However, quantum computing is currently at the stage where quantum programmers code in terms of elementary quantum gates. For example, researchers often use machine learning techniques to learn the parameters that go into building a quantum circuit (Cincio, et al. 2018). In this technique, researchers optimize over parameters that define gate operations, and the learned algorithm is simply machine optimized in terms of fundamental operations so there is no abstraction at all. This makes it difficult to extend the result of the learning procedure to, for example, larger computation with more qubits. If we can abstract, or *decompile* our algorithm, then we can hope to have an overall view of what is actually happening and learn of more general algorithms.

Another motivation for creating qSonify was my researcher report. I will quickly summarize their paper here (Putz and Svozil 2017). They discuss the possibility of quantum music and mapping quantum states to a musical representation. I extended their idea to quantum algorithms,

and using quantum computers to represent music. However, they do not mention any real practical advantage of quantum music, and I could not think of any reason that using quantum algorithms to represent songs could be of any practical use. So I instead went the opposite route and tried to find a practical advantage of using the music of quantum algorithms to learn about the algorithms themselves.

Following this thought, I remembered a popular audio file released by LIGO after their first discovery of gravitational waves. During the last moments of the inspiral of two black holes into each other, their rotation frequency increases very quickly, and this produces a so-called *chirp*. By converting their gravitational wave measurements into audio, they can hear the increase in frequency, and just that audio reveals a lot of interesting information about colliding black holes. I hope that in a similar way qSonify can help to reveal information about quantum algorithms.

Methodology

qSonify is a Python framework. Please see the presentation or the qSonify_HelloWorld jupyter notebook to see how it works. The package consists of many parts that I needed to code.

1. Code to interface with the quantum computer. I use *qiskit*, IBM's framework to interface with their simulators and actual quantum hardware. I programmed a way to represent algorithms as a list of strings of gates, and then convert that to the way in which qiskit runs algorithms. I then wrote functions to run the algorithms, sample from them, and/or probe their probability distribution. I included functionality for both markovian sampling (the output of the previous run is used as a starting point for the next algorithm run; this allows us to probe more of the Hilbert space) and non-markovian sampling (just running the algorithm many times with the same starting point; this allows us to just look at one

particular probability distribution). I found that markovian sampling was more effective in finding patterns in the algorithms. I also programmed the functionality to run the quantum circuits on IBM's hardware (as long as you have an API token). To do this, though, you must sit in queue in order for your algorithm to be run, and the queue is often quite long, so I haven't actually tested any sonfication results on the actual hardware. They would seemingly be worse results, since the current hardware is very noisy and error prone.

2. Code to deal with the sonification aspect. I decided to encode the music information in terms of midi because this provided an easy way to both play back the file and view the file on staff. I created the *Song* class to deal with all the details adding notes, rests, tracks, etc to the midi file, so that it was easy to perform the sonification in the end.
3. Code to deal with mappings. The beauty and difficulty of sonification is finding useful ways to map data to audio. The mapping aspect takes the output of the quantum computer and transforms it to a Song object, so that the sonification can take place. Unfortunately, I was limited by midi, so all mappings must map to discrete notes. I was hoping to also include methods to map the output to continuous audio, but ran out of time. However, there are many different ways that a mapping from quantum computer output to notes can be done (see the *qSonify/maps* folder on the GitHub page for some examples I implemented).
4. Code to put it all together. I allow user defined mappings to be easily implemented and used because, as stated in point 3, the real study and work that needs to be done is in figuring out which mapping helps us to perform the decompilation.

Results

It is hard to characterize audio results in a written report. The GitHub page has some midi results that you can listen to; follow the qSonify_HelloWorld jupyter notebook to see what they correspond to. I always review some of the audio results in my presentation video. But in attempt to summarize the results, I found that, at least for some algorithms, I could listen to them and then listen to other algorithms which had the original embedded within it and still hear that the original algorithm was there.

For example, I could sonify two algorithms, alg1 and alg2. I could then embed them within a larger algorithm, call the results alg1_emb and alg2_emb. When listening to alg1_emb, I could often still hear patterns that I heard in alg1, and similar for alg2_emb and alg2. In addition, I could embed alg1 and alg2 together in some larger algorithm, call this alg12_emb. When listening to alg12_emb, I could hear that alg1 and alg2 were definitely within the alg12_emb.

Reflection

I started out very optimistic about this project, so in some sense I did not get quite the results I was hoping for; I was optimistic that I would learn more about quantum algorithms from the sonification process. However, while implementing qSonify, I became more and more pessimistic as I started to see no hope for anything useful. Thus, I was pleasantly surprised in the end when I was able to "hear" algorithms embedded within other algorithms for some examples (see my presentation or qSonify_HelloWorld jupyter notebook). I am pleased with the results, and I believe I provided at least a starting framework for facilitating the study of quantum algorithms through sonification and a reason to be motivated and encouraged to study them in this way.

As I mentioned previously, I am most disappointed that I was not able to sonify the algorithms in a more continuous manner, but instead mapped only to discrete midi notes. In the future, I would experiment with continuous audio and focus on things such as specialization of audio and other perceptible audio features that would help to recognize patterns and changes in quantum algorithms.

Supplementary Materials

Please see *https://github.com/jiosue/qSonify*.

References

Cincio, Lukasz, Yigit Subasi, Andrew Sornborger, and Patrick Coles. 2018. "Learning the quantum algorithm for state overlap." *arXiv:1803.04114.*

Putz, V, and K Svozil. 2017. "Quantum Music." *Soft Computing* 21 (6): 1467-1471.