Quantum Walk / LanQ

Knowles Atchison, Jr
Fall 2013
Johns Hopkins University
Quantum Computation
Research Paper
December 10, 2013

Abstract

The study of quantum algorithms is a relatively new field when compared with traditional Computer Science. Due to the complication of constructing an actual physical quantum computer, attention has turned to simulation of one in a classical environment. To facilitate such experimentation, quantum programming languages of many shapes and varieties have been invented. One such language is LanQ. LanQ is an imperative high level quantum programming language that allows combining of quantum and classical computations [Mlnarik 2006]. This paper will introduce the LanQ programming language at a software developer level and will develop a program for a quantum walk.

1 Introduction and Contributions

LanQ is an imperative high level quantum programming language that allows combining of quantum and classical computations The language offers the ability to implement quantum protocols through process management. This paper will introduce the important bits of the LanQ language as necessary for implementation of a quantum algorithm. The Deutsch-Jozsa algorithm will be used as the quintessential "hello world" program. A quantum walk algorithm will then be introduced and compared with traditional walk algorithms.

2 LanQ Basics

Syntax of LanQ is incredibly similar to C, one of the key reasons this language was adopted in this paper. In that regard, code should be relatively self explanatory syntactically speaking. A more comprehensive look at the language can be found in the appendix of *Introduction to LanQ – an Imperative Quantum Programming Language*.

2.1 Deutsch-Jozsa Algorithm

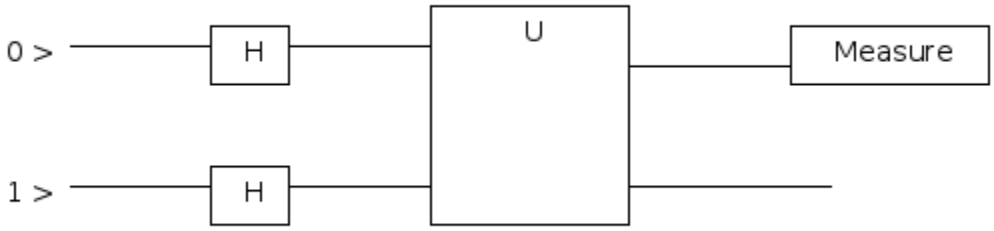

Figure 1: Quantum Circuit for Deutsch-Jozsa Algorithm

Consider the following code snippet for DJ:

```
#library "deutsch.libq"
#library "library.libq"

void main() {
        qbit x,y;

        x = new qbit();
        y = new qbit();
        ProjTo0(x);
        ProjTo1(y);

        Had(x); //Hadamard gates
        Had(y);

        U_balanced_x(x,y);
//      U_balanced_not_x(x,y);
//      U_const_0(x,y);
//      U_const_1(x,y);
```

```
	Had(x);

	if (measure(StdBasis,x) == 0) {
		print("U is a constant function\n");
	} else {
		print("U is a balanced function\n");
	}
}
```

Two qbits are created (LanQ allows for q'd'bit, where 'd' is the number of dimensions of the qbit. qbit is shorthand for q2it, qtrit for q3it, etc) [Mlnarik 2007], their value set, run through a Hadamard gate, and then passed into function U.

The portion that allows for the relatively light work in the actual function is the headers included. In LanQ, the developer defines the mathematical operators for use via matrices, like so:

```
	/** deutsch.libq
	 *	Constant function, f(x) = 0 for x \in {0,1}.
	 *  The corresponding unitary for use in the Deutsch algorithm is |xy> \mapsto |x (y xor f(x))>
	 *  |00> -> |00>
	 *  |01> -> |01>
	 *  |10> -> |10>
	 *  |11> -> |11>
	 */
	unitary U_const_0 = [
	  [ 1, 0, 0, 0 ],
	  [ 0, 1, 0, 0 ],
	  [ 0, 0, 1, 0 ],
	  [ 0, 0, 0, 1 ]
	];

	/**
	 *	Constant function, f(x) = 1 for x \in {0,1}.
	 *  The corresponding unitary for use in the Deutsch algorithm is |xy> \mapsto |x (y xor f(x))>
	 *  |00> -> |01>
	 *  |01> -> |00>
	 *  |10> -> |11>
	 *  |11> -> |10>
	 */
	unitary U_const_1 = [
	  [ 0, 1, 0, 0 ],
	  [ 1, 0, 0, 0 ],
	  [ 0, 0, 0, 1 ],
	  [ 0, 0, 1, 0 ]
	];

	/**
	 *	Balanced function, f(x) = x for x \in {0,1}.
	 *  The corresponding unitary for use in the Deutsch algorithm is |xy> \mapsto |x (y xor f(x))>
	 *  |00> -> |00>
	 *  |01> -> |01>
	 *  |10> -> |11>
	 *  |11> -> |10>
```

```
 */
unitary U_balanced_x = [
  [ 1, 0, 0, 0 ],
  [ 0, 1, 0, 0 ],
  [ 0, 0, 0, 1 ],
  [ 0, 0, 1, 0 ]
];

/**
 *          Balanced function, f(x) = (1 - x) for x \in {0,1}.
 *  The corresponding unitary for use in the Deutsch algorithm is |xy> \mapsto |x (y xor f(x))>
 *  |00> -> |01>
 *  |01> -> |00>
 *  |10> -> |10>
 *  |11> -> |11>
 */
unitary U_balanced_not_x = [
  [ 0, 1, 0, 0 ],
  [ 1, 0, 0, 0 ],
  [ 0, 0, 1, 0 ],
  [ 0, 0, 0, 1 ]
];

/** library.libq
*******************************************************************************
  We define a Hermitian operator ProjTo0 which is a projector to |0> basis
  vector. If it is given a maximally mixed state, it renormalizes the state.
 *******************************************************************************/
hermitian ProjTo0 = [
    [sqrt(2), 0],
    [0,       0]];

 /*******************************************************************************
  We define a Hermitian operator ProjTo1 which is a projector to |1> basis
  vector. If it is given a maximally mixed state, it renormalizes the state.
 *******************************************************************************/
hermitian ProjTo1 = [
    [0,  0      ],
    [0,  sqrt(2)]];
```

Running the program as is results in the following output:

```
<1:main>U is a balanced function
<1:main, prob=1.0> terminated
```

The other various U operators may be used to illustrate the other possibilities of output for this program.

3 Quantum Walk

The classical random walk is one of the most basic concepts of probability. When formulated with a finite number of states, it consists of randomly choosing paths between vertices on a graph, and many

randomized algorithms for computing are such walks. These randomized algorithms are significant because they are often some of the simplest and fastest known ways to solve hard problems. A quantum walk is defined by the quantum mechanical movement of a particle on a graph such that if its position were measured continuously, the probability distribution would be a classical random walk. [Hoyer 2008].

There are many applications; however the remaining focus of this paper is the boolean satisfiability problem. Consider the following algorithm:

Classical Random Walk

With probability p, pick a variable occurring in some unsatisfied clause
and flip its truth assignment.
With probability p-1, follow Greedy SAT,
i.e. make the best possible local move [Atchison 2012]

Since a superposition can be created of all possible inputs, it appears that collapsing the state onto a set of literals that solve the equation can be done relatively "easily" with a quantum computer. And as mentioned above, this can be achieved by measuring the system sequentially, effectively doing the same calculation as the classical algorithm.

The quantum algorithm has three pieces: a "coin" flip [C], a shift [S], and the operator that continuously repeats the previous two parts [U]. The circuit can be seen in figure 2 below.

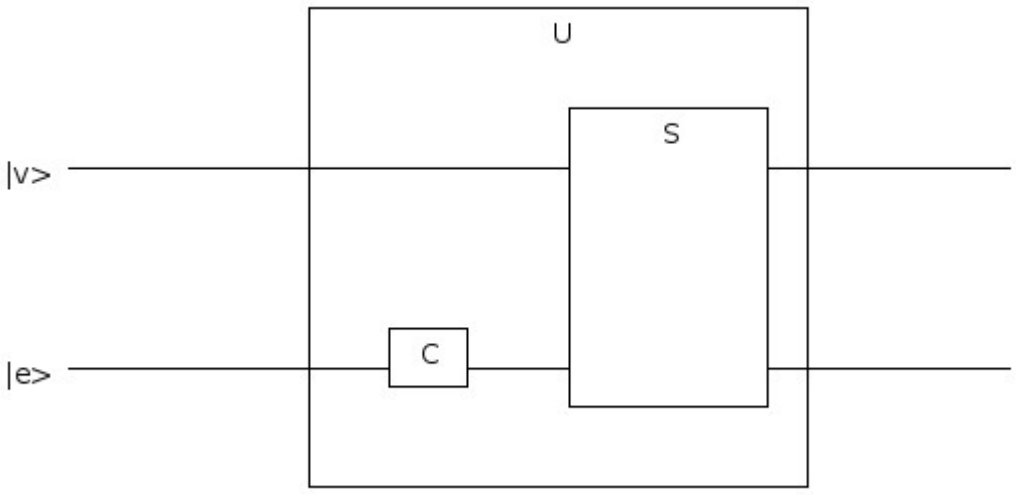

Figure 2: Quantum Circuit for Quantum Walk

The circuit accepts two states, the vectors of the graph and corresponding edges. The coin flip portion of the circuit can be overly complicated (see Hoyer paper section 4.4); however for a 2 qbit system, this transformation is the beloved Hadamard gate, as the Hadamard transform is the discrete Fourier

transform when n = 2. The Pauli matrix for Y could also be used for a perfectly balanced coin flip. The shift operator is defined by its action on each basis vector, S|v,i> = |v',i'> [Hoyer 2008].

4 Quantum Programming Language Implementation

With the knowledge in hand from Hoyer and the quantum circuit in figure 2, an attempt to program the algorithm in LanQ will now be made. The library file is the same as in the previous code section.

```
/**
 *	Implementation of a random quantum walk
 */

#library "library.libq"

//the spec does not seem to explicity state if this is pass by value or reference
//based on output, it appears that it does in fact pass by reference
qbit @ qbit U(qbit v, qbit e){
	qbit one, two;
	//extra copying here otherwise compiler will complain about uninitialized variable pair
	pair aliasfor [one,two];
	one = v;
	two = e;

	//Hadamard gate, "coin flip"
	Had(two);
	//alternatively, Pauli Y matrix
	//Sigma_y(two)
	//shift operator is not implemented as this is dependent on the graph type and the initial state of
v and e
	//shift(one, two)
	return pair;
}

void main() {
	qbit v,e;
	result aliasfor [v,e];
	v = new qbit();
	e = new qbit();

	ProjTo0(v); //what should the initial states be?
	ProjTo1(e);
	int i;
	i = 0;
	while (i != 5){ //relational operators do not appear to be supported at this time
		result = U(v,e);
		dump_q(result);
		i = i + 1;
	}
	//output for debugging
	//dump_q(v);
	//dump_q(e);
	//dump_q(result);
}
```

Interestingly enough, quantum circuits do not support feedback loops. In this regard, the circuit from Figure 2 can only be executed once; however LanQ does allow for us to continually repeat portions of the circuit as the initial algorithm recommends.

The above algorithm implantation has some parts that need refinement. Issues that came up during development are noted in the code comments. There are a few unanswered questions, particularly the initial state of the qbits depending on the graph and language characteristics. While aliasing qbits to create a larger state is well defined, it caused problems in the actual code.

5 Future Research and Conclusions

One interesting feature of LanQ that is absent from most other quantum languages is the ability to simulate quantum protocols, multiparty protocols specifically. It achieves this by doing process management and forks in a similar vein to the fork() in UNIX. A simple communication example:

```
/*****************************************************************************
  Method alice sends an integer over a channel whose end is passed as
  an argument to this method.
*****************************************************************************/
void alice(channelEnd[int] c0) {
        int i;

        i = 23;
        send(c0,i);
}
 /*****************************************************************************
  Main method, from this method the program is run.
  It allocates a channel for transmitting integers. Then it creates a new
  process from method alice to which one channel end is passed.
  Finally it receives a value from the channel and displays it.
*****************************************************************************/
void main() {
        channel[int] c withends [c0,c1];
        int i;

        c = new channel[int]();
        fork alice(c0);
        print(recv(c1) . "\n");
}
```

Corresponding output:

```
<0:main>23
<1:forked from 0:main at line 28, prob=1.0> terminated
<0:main, prob=1.0> terminated
```

More complicated examples such as teleportation (by creating EPR pairs) can be found in the LanQ introduction paper.

While LanQ made for an interesting choice initially, there does not appear to be active development and a few key pieces of the language (pass by value/reference is one that comes to mind) is either undefined or not easily found in the language specification. Continuing development and working on new algorithm implementations would be an interesting endeavor given a greater amount of time. The possibilities here for implementing not only quantum algorithms, but protocols falls outside the scope of this paper, but remains an exciting unexplored area and something to play around with in the future.

References

Published Books and Papers

Atchison Jr., Knowles. 2012. *An Introduction to SAT Solvers.* Johns Hopkins University. Online. http://www.knowlesatchison.net/pdf/SATSolvers.pdf

Hoyer, Stephen. 2008. *Quantum random walk search on satisfiability problems*. Swarthmore College Department of Physics and Astronomy.

Mlnarik, Hynek. 2006. *Introduction to LanQ – an Imperative Quantum Programming Language.* Online. http://lanq.sourceforge.net/doc/introToLanQ.pdf

Mlnarik, Hynek. 2007. *Quantum Programming Language LanQ.* Online. Masaryk University, Faculty of Informatics. http://lanq.sourceforge.net/doc/lanq-thesis.pdf

WWW References

LanQ – a quantum imperative programming language http://lanq.sourceforge.net/index.php