Problem Set 4
CPSC 440/640 Quantum Algorithms
Andreas Klappenecker

The deadline for this assignment has been extended to Thursday, Oct 5, 11:59pm. The signed hardcopy is due on Friday, Oct 6, in class.

The goals of this assignment are (a) to make you familiar with the simulation of quantum circuits on a classical computer, and (b) to give you the opportunity to get familiar with lex and yacc (use flex and bison if possible).

You are given a considerable amount of time for this assignment so that this exercise does not provide a conflict with your research. I recommend that you start early. You can earn 150 points instead of the usual 100 points. Some source code is provided in the tar-ball `alfred.tgz`.

0) Modify the source code so that it will compile on the system of your choice. For instance, on Windows systems, you might need to include `windows.h`. If your system does not support the `drand48` random number generator, then you might want to supply your own pseudo-random number generator.

1) The core simulator is contained in the file `sim.c`. Supplement the missing code in the procedures `measure_state` and `applygate`.

2) Write a small test file that uses the procedures given in `sim.c` to create the state $0.707|00) + 0.707|11)$ from input state $|00)$. Do this by simulating the action of one Hadamard gate and one controlled-not gate with `applygate`. This should be followed by measuring the state with `measure_state`. Monitor the evolution of the state after each step by `print_state`.

3) Get familiar with lex and yacc, or, rather, with flex and bison. Read the manuals and implement some small example that allows you to grasp the main concept of the interaction between lex and yacc.

4) Correct all errors so that you get a fully functional simulator for the language Alfred.

You will receive a little Alfred program and you need to demonstrate that your simulator works. The details will be discussed in class.

Remarks. i) The simulator assumes that you will not simulate more than 20-30 quantum bits. For efficiency reasons, information about the position of control and target qubits are encoded by setting the corresponding bits in an integer. For example, if the target qubit position `pos` is the least significant bit, then this is represented by `pos = 1<<0`, if the target qubit is the most significant qubit in a system of 3 qubits, then this is encoded by

`pos = 1<<2`. Review the bit operations of the language C to see the benefit of this convention.

ii) Write the code for the procedure `measure_state`. The input for this procedure is an integer `pos`, which has a bit set at the position of the quantum bit, which will be observed. The measurement is done with respect to the computational basis. Your procedure should directly modify the input state vector `state`. You can address the content of this state vector by `state[`i`]`, where $0 \leq i < $ `1<<Nbits`. Use the random number generator `rand` in your implementation. Make sure that your implementation will reflect the behaviour of quantum mechanics. You might need to change the initialization of the random number generator on you system to obtain the desired results.

iii) Write the code for the procedure `applygate`. Your code should realize an implementation of a multiply conditioned gate, as explained in the lecture. The conditions are provided in terms of integers. A bit set in `ocnd` means that this bit must be 0, a bit set in `icnd` means that this bit must be set to 1. The integer `gpos` has a single bit set at the target bit position of the gate.

iv) Before writing the code for `applygate`, I suggest that you re-read the lecture notes on multiple control quantum gates.

v) Update: Fill in the verbatim output of the five challenges below.

vi) Use the GNU gcc compiler. The source code uses some language extensions provided by gcc. If you use some Windows operating system, then I suggest that you have a look at cygwin, which provides you with all the Unix utilities such as lex and yacc (or rather flex and bison),...

vii) You are welcome to solve this problem set in a different language such as Ruby or C++.

viii) You are allowed to discuss problems related to compilation of the code and other technical difficulties. You should avoid sharing code or showing your solution to others.

Problem 1. Include a verbatim copy of your implementation `measure_state`. Include a rational for your implementation that documents your code well.

Solution. My `measure_state` code is given by

`INCLUDE CODE HERE`

Include rational for your implementation here . . .

Problem 2. Include the verbatim output of `./alfred < chlg1.alf`

Solution.

Problem 3. Include the verbatim output of `./alfred < chlg2.alf`

Solution.

Problem 4. Include a verbatim copy of your implementation `applygate`. Include a rational for your implementation that documents your code well.

Solution. My `applygate` code is given by

`INCLUDE CODE HERE`

Include rational for your implementation here . . .

Problem 5. Include the verbatim output of `./alfred < chlg3.alf`

Solution.

Problem 6. Include the verbatim output of `./alfred < chlg4.alf`

Solution.

Problem 7. Include the verbatim output of `./alfred < chlg5.alf`

Solution.