Problem Set 4

CPSC 440/640 Quantum Algorithms
Andreas Klappenecker

Name: Paul Gustafson
On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework.

Signature: ______________________

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 $0.707|00\rangle + 0.707|11\rangle$ from input state $|00\rangle$. 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

```
/* input: a single bit set at the position of the gate;
```

```
   this bitposition is measured and the state vector
   is changed according to the postulates of quantum
   mechanics.
   NOTE: the state must be normalized to length 1 */
void measure_state(int pos, cplx *state) {
  int i;
  double ran;
  double prob;

  srand(clock());
  ran = ((double)rand())/RAND_MAX;
  /* complete the missing code */

  /* compute the probability to observe 0,
   * determine whether 0 or 1 should be observed in this measurement
   * then modify state to produce the post measurement state
   * of the quantum computer
   */

  // the probability of getting a zero in the ith position
  prob = 0;

  // calculate prob by adding up the squared moduli of coefficients
  // with 0 in the position "pos"
  for (i = 0; i < (1 << Nbits); i++) {
    if (((i / pos) % 2) == 0) {
      prob += sq(state[i].re)+sq(state[i].im);
    }
  }

  // do the measurement
  int measuredBit;

  if(ran < prob)
    measuredBit = 0;
  else
    measuredBit = 1;

  // Apply the projection operator
  for (i = 0; i < (1 << Nbits); i++) {
    if (((i / pos) % 2) != measuredBit) {
      state[i].re = 0;
      state[i].im = 0;
    }
  }


  normalize_state();
}
```

At the beginning of the method, I pick a random number, uniformly distributed between 0 and 1. Then I calculate the probability of measuring a zero in the position *pos* by adding up the squared moduli of all coefficients of basis elements with 0 in position *pos*. If the random number is less than this probability, I measure a 0. Otherwise, I measure a 1. Then I orthogonally project the state onto the subspace compatible with the measurement, followed by normalizing the state.

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

Solution.

```
If the state is of the form
0.707|0)+0.707|1)
Then one expects that
0 will be observed with prob. 1/2
1 will be observed with prob. 1/2
1|1)
1|0)
1|0)
1|0)
1|0)
1|1)
1|1)
1|1)
1|0)
1|1)
```

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

Solution.

```
If the state is of the form
0.577|00)+0.816|10)
Then one expects that
0 will be observed with prob. 1/3
1 will be observed with prob. 2/3
1|10)
```

```
1|10)
1|10)
1|00)
1|00)
1|00)
1|10)
1|00)
1|10)
1|10)
```

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

```
/* Applying a conditioned g-gate to a state vector */
/* ASSUME: ocnd, icnd, and cpos have disjoint support */
void applygate(int Nbits, int ocnd, int icnd, int gpos,
               mat g, cplx *state  ) {
  cplx tmp0, tmp1;
  int i;
  int acnd = ( ocnd | icnd );


  /* complete the missing code */

  /* gpos is a integer of the form 2^k, where
   * k is the position of the target qubit.
   * The bits set in ocnd correspond to the
   * non-filled circles, and the bits set in icnd
   * correspond to the filled circles
   */

  for (i = 0; i < (1 << Nbits); i++) {
    if(((i & acnd) ^ icnd) == 0) {
      if(((i / gpos) % 2) == 0) {
        tmp0 = state[i];
        tmp1 = state[i + gpos];
        state[i] =  add(mult(g[0], tmp0), mult(g[1], tmp1));
        state[i + gpos] = add(mult(g[2], tmp0), mult(g[3], tmp1));
      }
    }
  }
}
```

To apply the gate, I loop through all the basis elements. The basis elements that satisfy the control bit (i.e. 0's on *ocnd* and 1's on *acnd*) form pairs based on whether they have a 0 or a 1 in position *gpos*. Each such pair forms an invariant subspace for the gate, so it's enough to just apply the gate to the pair when you hit the first of each pair in the loop.

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

Solution.

```
A Hadamard gate that acts on the
least significant bit
------------------------
|00) -->
0.707|00)+0.707|01)
------------------------
|01) -->
0.707|00)-0.707|01)
------------------------
|10) -->
0.707|10)+0.707|11)
------------------------
|11) -->
0.707|10)-0.707|11)
```

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

Solution.

```
A Hadamard gate that acts on the
most significant bit
------------------------
|00) -->
0.707|00)+0.707|10)
------------------------
|01) -->
0.707|01)+0.707|11)
------------------------
|10) -->
0.707|00)-0.707|10)
------------------------
|11) -->
0.707|01)-0.707|11)
```

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

Solution.

```
A controlled Hadamard gate that acts on the
most significant bit when the least significant
is 0 and the middle bit is 1:
------------------------
|000) -->
1|000)
------------------------
|001) -->
1|001)
------------------------
|010) -->
0.707|010)+0.707|110)
------------------------
|011) -->
1|011)
------------------------
|100) -->
1|100)
------------------------
|101) -->
1|101)
------------------------
|110) -->
0.707|010)-0.707|110)
------------------------
|111) -->
1|111)
```