Grover's algorithm, amplitude amplification

Note: Grover's algorithm is Part 4.2 of the pre-bootcamp workshops

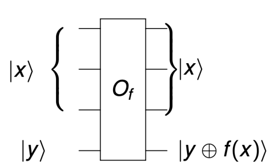
see also my workshop slides 'Quantum_Algorithms.pdf'

1

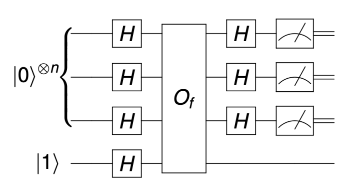
Oracle functions

- speedup based on superposition (interference) instead of testing each case

(1985 first quantum algorithm by D. Deutsch, generalized in collab. with Jozsa:
Consider a circuit (oracle) that implements a comparator function, and we are asked whether the function is constant (returns the same value for all inputs) or balanced (returns 1 on one input and 0 on another).

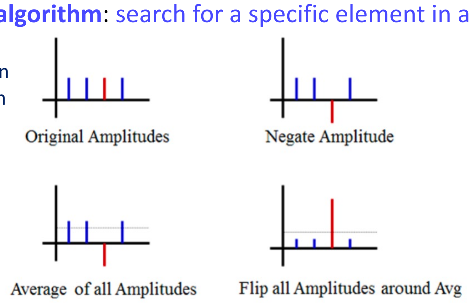


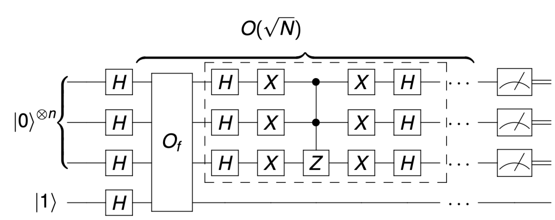
Implemented as (n+1)-qubit circuit:



- **Grover's algorithm**: search for a specific element in an unsorted list

- oracle evaluation based on inversion about the mean:





2

Are oracle functions good for anything?

- typical example: find the entry where all qubits are in state |1>
  e.g. 3-qubits system: find entry '7' (='111')
  → oracle: $f(x_0, x_1, x_2) = (x_0 == 1) \wedge (x_1 == 1) \wedge (x_2 == 1)$
  → apply CCCX gate (multi-controlled-NOT)

➢ seems like a complicated way to get something obvious (largest value you can construct with 3 bits)

Oracle functions are combinations of boolean statements

➢ efficient way to check boolean satisfiability

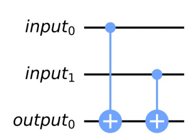
➢ it's important to set up the combination of boolean statements for a given problem and construct corresponding gates & circuits

(Sidenote: a generalized form of Grover's algorithm is the Quantum Amplitude Amplification, a basic quantum algorithm used as ingredient in many other quantum algorithms.)

3

- **Oracle functions are combinations of boolean statements**

XOR realized via
CX(in0, out)
CX(in1, out):



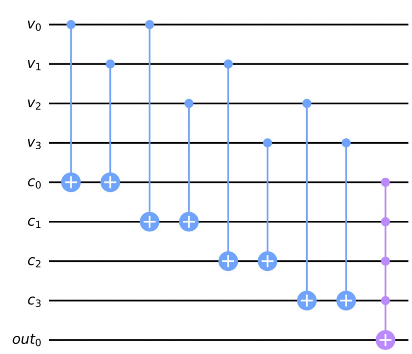
Example: 2x2 Sudoku
(see QISkit tutorial: Grover)

V_0	V_1
V_2	V_3

→ Requirements: $v_0 \neq v_1 \wedge v_0 \neq v_2 \wedge v_1 \neq v_3 \wedge v_2 \neq v_3$

→ oracle: $f(v_0, v_1, v_2, v_3) = v_0 \oplus v_1 \wedge v_0 \oplus v_2 \wedge v_1 \oplus v_3 \wedge v_2 \oplus v_3$

→ series of XORs: each output requires a new scratch qubit, which are combined via CCCCX



Important note: the full algorithm acts on v_0 to v_3 and out_0
→ any entanglement with c_0 to c_3 must be removed ("uncomputing" via applying the same gates in opposite order)
before applying the diffuser or any measurement!

4

- **Recipe for creating oracle functions**

(Re-)write the clauses into boolean expressions all combined by AND or all combined by OR.

For all variables mark whether they are listed or must be true or false in each clause.

Use controlled gates with a different output qubit for each clause.

Combine all outputs via a multi-controlled-NOT.

Add a Z-gate to the output qubit and uncompute (apply controlled gates in reverse order).

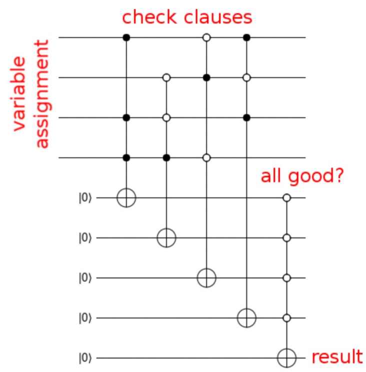
Example:

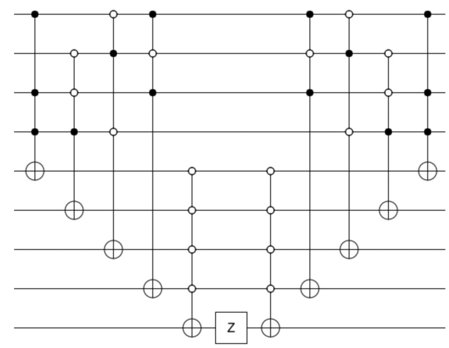
(¬x1 V ¬x3 V ¬x4) Λ
(x2 V x3 V ¬x4) Λ
(x1 V ¬x2 V x4) Λ
(x1 V x3 V x4) Λ
(¬x1 V x2 V ¬x3)

in table form:

1	2	3	4
x	_	x	x
_	o	o	x
o	x	_	o
o	o	_	o
x	o	x	_

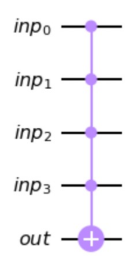
o : variable is true
x : variable is false
_ : not in clause

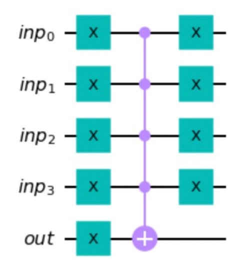




5

Note:
AND gate: multi-controlled X gate:
circuit.mcx(control_qubits, target_qubit)
OR gate: first invert each qubit, then multi-controlled X gate, then invert input quits





- Back to example: 2x2 Sudoku

→ Requirements: $v_0 \neq v_1 \wedge v_0 \neq v_2 \wedge v_1 \neq v_3 \wedge v_2 \neq v_3$

→ oracle: $f(v_0, v_1, v_2, v_3) = v_0 \oplus v_1 \wedge v_0 \oplus v_2 \wedge v_1 \oplus v_3 \wedge v_2 \oplus v_3$

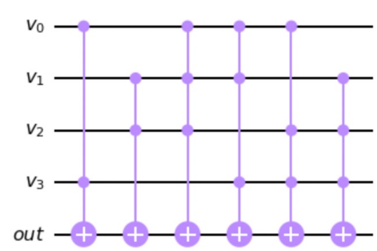
V_0	V_1
V_2	V_3

- with a little algebra *f(x)* can be rewritten as:

$$f(v_0, v_1, v_2, v_3) = (v_0 \wedge v_3) \oplus (v_1 \wedge v_2) \oplus (v_0 \wedge v_1 \wedge v_2) \oplus (v_0 \wedge v_1 \wedge v_3) \oplus (v_0 \wedge v_2 \wedge v_3) \oplus (v_1 \wedge v_2 \wedge v_3)$$

→ this oracle function does not require additional scratch qubits!

➢ best method to create oracle functions: rewrite the requirements to prevent additional scratch qubits (no need to uncompute!).



6