Quantum e Algoritmi in Q#

Daniel Biasiotto

[2022-05-12 Thu 19:31]

CONTENTS

1 INTRODUZIONE

Lo sviluppo di software quantistici si é confermato come un'area di grande interesse negli ultimi decenni, offrendo grandi possibilitá di superare i limiti computazionali attualmente compresi in diverse aree di ricerca. Per esempio é possibile che in futuro algoritmi quantistici possano sostituire controparti classiche in diverse applicazioni:

- crittografia
- problemi di ricerca
- simulazione di sistemi quantistici
- machine learning
- computazione biologica
- chimica generativa

Ovviamente il calcolo classico non verrá abbandonato, l'approccio classico e quello quantistico differiscono nelle loro forze e debolezze.

Mentre gli attuali computer diventano sempre piú veloci e le tecniche industriali permettono miniaturizzazioni sempre maggiori gli hardware quantistici rimangono estremamenti complicati da costruire e distribuire. Il modo attualmente piú congeniale di utilizzare questi hardware rimane quello della condivisione di risorse attraverso le tecnologie di *cloud computing*.

Ma questo non significa che non sia possibile studiare i problemi e le soluzioni algoritmiche che il calcolo quantistico offre a livello teorico e di sviluppo software. Attualmente i software sviluppati per hardware quantistici possono essere eseguiti su simulatori, sempre software, oppure sfruttando reali macchine quantistiche in in remoto. I simulatori e le macchine reali condividono interfacce condivise che permettono lo sviluppo di software che abbia la possibilitá di essere testato in maniera indiscriminata su un qualsiasi di questi.

2 AMBIENTE

Per lo sviluppo di software quantistici sono disponibili diversi ambienti e framework, tra i piú conosciuti troviamo **Microsoft Azure** con il proprio Quantum Development Kit (`QDK`) o l'ambiente di sviluppo di `IBM` **Qiskit**. Altri *Software Development Kit* che possono essere utilizzati per eseguire circuiti quantistici su prototipi di device quantistici o simulatori sono:

- Ocean
- ProjectQ
- Forest
- t|ket>
- Strawberry Fields
- PennyLane

Molti di questi progetti sono open-source e sviluppati sulla base di `Python`.

Per questo lavoro abbiamo utilizzato gli strumenti offerti da Microsoft per l'ottima documentazione consultabile sulle loro pagine web e in quanto era ció che era utilizzato dalla nostra fonte principale *Learn Quantum Computing with Python and Q#*.

L'ambiente di esecuzione `Q#` puó essere configurato sul editor `Visual Studio Code` tramite l'add-on proprietario di Microsoft. Quest'ultimo é disponibile solo sulla versione non `FOSS` del software, che é possibile installare tramite le repository opensource linux.

In alternativa o anche parallelamente é possibile sviluppare codice Q# ed eseguirlo tramite Jupyter Notebook tramite Python. Questo con i kernel necessari installati, quindi l'ultima versione di dotnet disponibile.

Tramite anaconda si crea un ambiente con il necessario:

```
$ conda create -n qsharp-env -c microsoft qsharp
 ↪  notebook
$ conda activate qsharp-env
```

L'esecuzione del software Q# puo' essere testato localmente predisponendo un ambiente di simulazione tramite il pacchetto Python chiamato qsharp.

```
import qsharp
from QsharpNamespace import Operation_One, Operation_Two
var1 = 10
print("Simulation started...")
Operation_One.simulate(par1=var1)
Operation_Two.simulate(par2=var1,par3=5)
```

Ad esempio come nel listato qui sopra utiliziamo uno script host.py per creare un ambiente di simulazione per poter eseguire le operazioni Q# definite in Operation_One e Operation_Two. Il pacchetto automaticamente va a cercare nella directory locale le definizioni.

3 ORACOLI

Gli oracoli che utiliziamo per testare gli algoritmi definiti in seguito sono:

```
operation ApplyZeroOracle(control : Qubit, target :
 ↪  Qubit) : Unit {
  }

  operation ApplyOneOracle(control : Qubit, target :
   ↪  Qubit) : Unit {
    X(target);
  }

  operation ApplyZeroOracleN(control : Qubit[], target
   ↪  : Qubit) : Unit {
  }
```

```
operation ApplyOneOracleN(control : Qubit[], target :
↪  Qubit) : Unit {
  X(target);
}

operation ApplyIdOracle(control : Qubit, target :
↪  Qubit) : Unit {
  CNOT(control,target);
}

operation ApplyXOROracleN(control : Qubit[], target :
↪  Qubit) : Unit {
  for qubit in control {
      CNOT(qubit,target);
  }
}

operation ApplyNotOracle(control : Qubit, target :
↪  Qubit) : Unit {
  X(control);
  CNOT(control,target);
  X(control);
}
```

Dove sono definiti versioni a singolo qbit e a n-qbit degli oracoli quantistici di alcune funzioni booleane costanti e bilanciate. In particolare abbiamo definito oracoli per le seguenti funzioni:

- $f(x) = 0$
- $f(x) = 1$
- $f(x) = x$
- $f(x) = \neg x$ o $f(x) = 1 - x$
- $f(x, y) = x \oplus y$
  - dove x e' l'input lungo n qbit e y e' l'output

In questi casi le prime due funzioni sono costanti e le restanti sono bilanciate.

4 ALGORITMO DI DEUTSCH-JOZSA

L'algoritmo di Deutsch-Jozsa ha interesse storico in quanto primo algoritmo quantico in grado di superare in performance il miglior al-

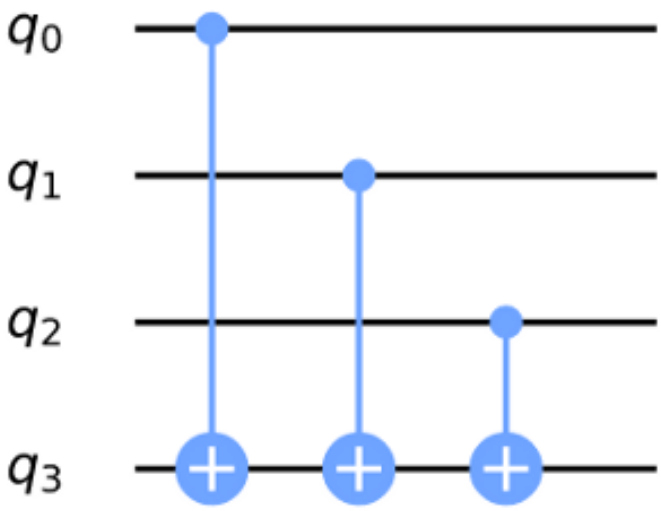

Figure 1: esempio di oracolo bilanciato utilizzando porte CNOT

goritmo classico corrispondente, mostrando che possono esistere vantaggi nel calcolo quantico. Spingendo la ricerca in questa direzione per determinati problemi.

L'algoritmo tratta la decisione di una funzione f booleana con n bit in input

$$f(\{x_0, x_1, \cdots, x_n\}) \rightarrow 0 \text{ o } 1$$

Questa funzione su cui agisce l'algoritmo ha la proprietá di essere una di due forme:

- costante
- bilanciata

E quindi restituisca per tutti gli input $\{x_0, x_1, \cdots, x_n\}$ lo stesso risultato se costante oppure restituisca *esattamente* 0 per metá degli input e 1 per metá degli input.

4.1 La Soluzione Classica

Nella soluzione classica nel **caso migliore** due *query* all'oracolo sono sufficienti per riconoscere la funzione f come bilanciata. Per esempio si hanno due chiamate:

$$f(0, 0, \cdots) \rightarrow 0$$

$$f(1, 0, \cdots) \rightarrow 1$$

Dato che é assunto che f é *garantita* essere costante oppure bilanciata questi risultati ci dimostrano f come bilanciata.

Per quanto riguarda il caso peggiore tutte le nostre interrogazioni daranno lo stesso output, decidere in modo certo che f sia costante necessita di metá piú uno interrogazioni. In quanto il numero di input possibili é 2^n questo significa che saranno necessarie $2^{n-1}+1$ interrogazioni per essere certi che $f(x)$ sia costante nel caso peggiore.

4.2 La Soluzione Quantica

Tramite la computazione quantica é possibile risolvere questo problema con un'unica chiamata della funzione $f(x)$. Questo a patto che la funzione f sia implementata come un oracolo quantico, che mappi: $|x\rangle|y\rangle$ a $|x\rangle|y \oplus f(x)\rangle$ [1]

I passi dell'algoritmo in particolare sono:

1. prepara 2 registri di `qubit`, il primo di n `qubit` inizializzato a $|0\rangle$ e il secondo di un singolo `qubit` inizializzato a $|1\rangle$
2. applica `Hadamard` a entrambi i registri
3. applica l'oracolo quantico
4. a questo punto il secondo registro puó essere ignorato, riapplica `Hadamard` al primo registro
5. misura il primo registro, questa risulta 1 per $f(x)$ costante e 0 altrimenti nel caso bilanciato

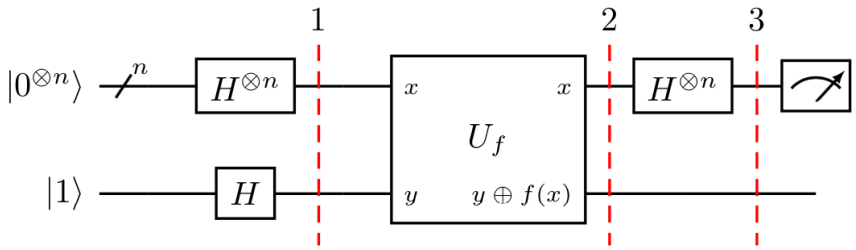

Figure 2: i passi dell'algoritmo in forma di circuito

4.2.1 *single-bit Deutsch-Jozsa*

```
operation DeutschJozsaSingleBit(oracle : (( Qubit,
↪  Qubit ) => Unit)) : Bool {
    use control = Qubit();
    use target = Qubit();

    H(control);
```

1 dove $\oplus$ é l'addizione modulo 2 o `XOR`

```
    X(target);
    H(target);

    oracle(control, target);

    H(target);
    X(target);

    return MResetX(control) == One;
}
```

4.2.2 n-bit Deutsch-Jozsa

```
operation DeutschJozsa(size : Int, oracle : ((Qubit[],
↪   Qubit ) => Unit) ) : Bool {
    use control = Qubit[size];
    use target = Qubit();

    ApplyToEachA(H, control);
    X(target);
    H(target);

    oracle(control, target);

    H(target);
    X(target);

    let result = MResetX(control[0]) == One;
    ResetAll(control);
    return result;
}
```