QSimov: Quantum Computing Framework

Hernán Indíbil de la Cruz Calvo
Departmento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Albacete, Spain

José Javier Paulet González
Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid, Madrid, Spain
Departmento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Albacete, Spain

Fernando Cuartero Gómez
Departmento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Albacete, Spain

Fernando López Pelayo
Departmento de Sistemas Informáticos, Universidad de Castilla-La Mancha, Albacete, Spain

Abstract

In this paper, we present QSimov: the Quantum Computing Framework we have developed. This tool has been designed having academic environments in mind, focused on being simple (immediate translation from the drawing of a circuit to code written for QSimov), lightweight (able to be run on a low-end laptop), potent (able to run big experiments) and open sourced. A couple of samples on how to implement some algorithms are provided, as well as graphs showing the execution times of the simulation, which in the worst cases will be exponential. QSimov is still under development: we are currently adding new features and changes asked by the community, as well as fixing any bug that might be detected.

2012 ACM Subject Classification Theory of computation → Quantum computation theory

Keywords and phrases Quantum computing, Simulation

Funding This work has been supported by the IPI Conocimento y flexibilidad SL project ELABORACIÓN DE UNA PLATAFORMA DE COMPUTACIÓN CUÁNTICA, Y ACTIVIDADES FORMATIVAS (210296UCTR)

1 Introduction

QSimov (Figure 1) is a Python module designed to let the user developing and practising quantum computing algorithms regardless the underlying architecture. Since its conception, it was designed to be light weighted to run in a low-end laptop and as powerful as possible in terms the amount of qubits that can handle.

QSimov is based on the circuit model, so allowing the user to implement algorithms not

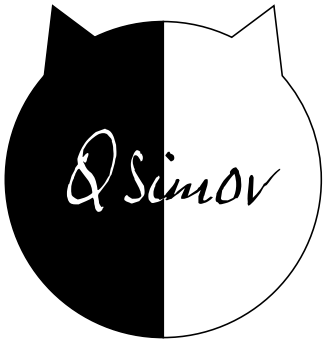

Figure 1 QSimov logo.

restricted but for the number of entangled qubits. It also automatises the task of iterating a piece of code a whole number of times. Moreover, it provides operating seeds for random issues.

QSimov is not just a simulator since it is open-sourced and modular structured in order to provide the developer with the possibility to run circuits defined in QSimov over some other simulators as either IBM's Qiskit or Rigetti's Forest.
Last released QSimov's version is 4.3.0.

2 Architecture

QSimov 4.3.0 is structured into two sub-modules:

- Drewom (Figure 2) is the module in charge of translating circuits designed over QSimov into other code suitable to be executed over some other simulators and, once executed the output is also translated into QSimov fashion. It is embedded into QSimov's python package.
- Doki (Figure 3) is a quantum computer state vectors simulator written in C. It serves as the default target simulator for QSimov. It is included as another open-sourced python package.

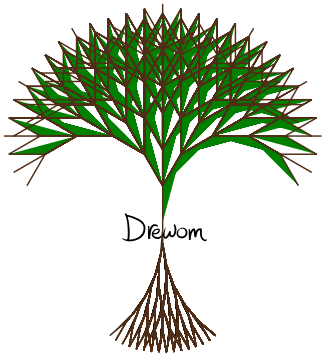

Figure 2 Drewom logo.

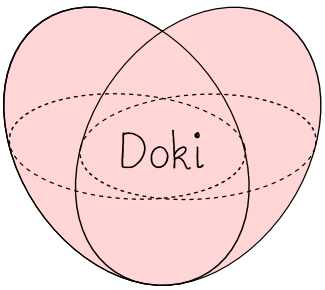

Figure 3 Doki logo.

The usual application flow is:

1. The user writes a circuit according to QSimov
2. The user instantiates an executor with Drewom (Doki connector is used by default)
3. The user requires the executor to run the corresponding circuit
4. Drewom translates the circuit into Doki instructions and executes them

5. Drewom translates the results of the simulations from abroad into QSimov API format
6. The executor returns the user with the output

3 Computational complexity

The computational complexities here stated apply when using the default simulator (Doki).

3.1 Spatial complexity

We set the size of the problem to n as the number of qubits used in the simulated quantum system.

- $\Omega(n)$ is the lower bound complexity order for the amount of memory required in single-qubit operations, i.e. entanglement free scenario.
- $O(2^n)$ is the upper bound complexity order under the worst case scenario conceived for this, i.e. maximum entanglement among the qubits (which involves two-qubits gates)

3.2 Time complexity

Gate application

The size of the problem $n - m + t$, where n is the addition of the number of target qubits (included their tentative entangled qubits) and control qubits (included their tentative entangled qubits), m is the number of control (either control or anti-control) qubits in the gate, and t is the number of target qubits in the gate.

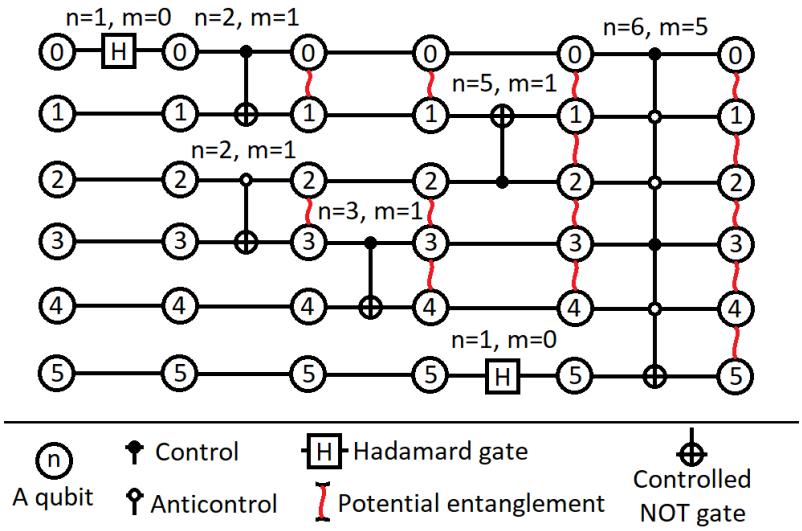

Figure 4 n and m values associated with their respective operations. $t = 1$ for all the examples

Figure 4 shows an example on how to calculate n and m. In this example we know none of the initial value of the six qubits. Therefore, any gate of more than one qubit can potentially entangle every affected qubit (including controls and anti-controls). Red curved lines represent a possible entanglement relationship between two qubits. n is calculated by counting all the qubits connected by a red line to those affected by the gate. Since all the gates used only target one single qubit, $t = 1$ for all the operations in this example.

- $\Theta(2^{n-m+t})$

Measurement

The size of the problem is n, the number of qubits potentially entangled with the one we want to measure.

- $\Theta(2^n)$

4 Use cases

Several quantum algorithms have been implemented on QSimov in order to give the reader a clear idea about how to use it, as well as to summarise the time taken at computing in some graphics.

4.1 BB84 key exchange algorithm

BB84 is a key exchange algorithm proposed by C. H. Bennett and G. Brassard in [1] as a quantum-resistant way to exchange a key between two parties. Since it only makes use of one-qubit gates, no qubits would be potentially entangled. That means this case covers the best-case scenario therefore it allows thousands of qubits to be simulated.

The code written in Listing 1 makes use of QSystem, one of the low-level data structures in QSimov.

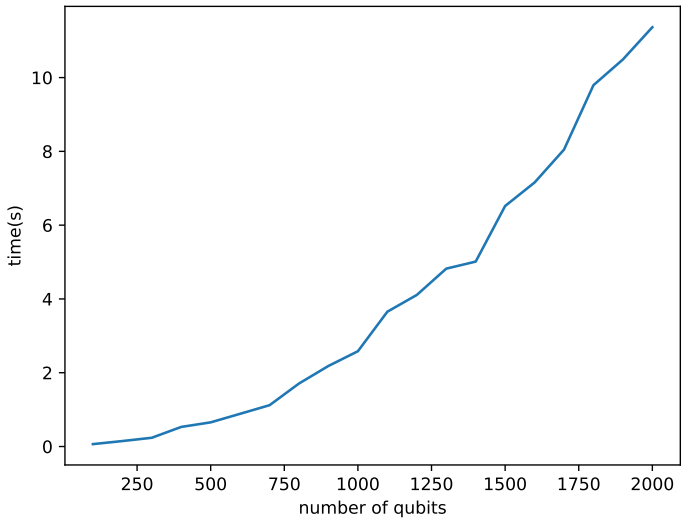

Figure 5 BB84 execution time.

In Figure 5 we can see the amount of time needed in order to compute the BB84 algorithm with an increasing number of qubits. It takes polynomial time to execute it.

4.2 Deutsch-Jozsa algorithm

The Deutsch-Jozsa algorithm was proposed by David Deutsch and Richard Jozsa in 1992 [2]. It was one of the first algorithms to solve an actual computationally complex problem

Listing 1 BB84 code.

```
def bb84(size):
    # Alice generates the random bit array
    a = [rnd.randint(0, 1) for i in range(size)]
    # Alice randomly picks the basis
    b = [rnd.randint(0, 1) for i in range(size)]
    s = qj.QSystem(size)
    for id in range(size):
        # Alice encodes the generated value
        if a[id]:
            s = s.apply_gate("X", targets=id)
        # Alice uses the basis she picked
        if b[id]:
            s = s.apply_gate("H", targets=id)
    # Alice sends the qubits to Bob
    # Bob then measures the received qubits using a random basis
    b_prime = [rnd.randint(0, 1) for i in range(size)]
    for id in range(size):
        # Basis change before measuring
        if b[id]:
            s = s.apply_gate("H", targets=id)
    # Measurement
    _, a_prime = s.measure([id for id in range(size)])
    # Bob publishes b'. Alice answers with b.
    # Both discard the elements where b[i] != b'[i]
    new_a = [a[i] for i in range(size) if b[i] == b_prime[i]]
    new_a_prime = [int(a_prime[i])
                   for i in range(size) if b[i] == b_prime[i]]
    k = len(new_a)  # Numbers remaining
    # Alice publishes half of the values
    picked = rnd.sample([i for i in range(k)], k//2)
    # Bob compares published values with his
    errors = sum(new_a[i] != new_a_prime[i] for i in picked)
    alice_key_str = "".join(str(new_a[i])
                            for i in range(k) if i not in picked)
    bob_key_str = "".join(str(new_a_prime[i])
                          for i in range(k) if i not in picked)
    # Keys as integers
    alice_key = int(alice_key_str, 2)
    bob_key = int(bob_key_str, 2)
    return alice_key, bob_key, len(alice_key_str), errors
```

(hard for classical machines) in polynomial time over a quantum computer (EQP complexity class). The statement of the problem is as follows, how to determine whether a function is constant or balanced, given that it can only be one of them?

The code written in Listing 2 makes use of QRegistry, the lowest-level data structure in QSimov, to just start from the case of all the qubits under entanglement condition. The use of this structure greatly reduces the maximum number of qubits, so making the simulator assuming that all the qubits could be potentially entangled, in other words the worst-case scenario. Many different oracles have been used on this use case: $f(x) = 0$ and $f(x) = 1$ as the only possible constant oracles, and $f(x) = x_i$ with x_i being the value of one of the input bits of x as the balanced oracles.

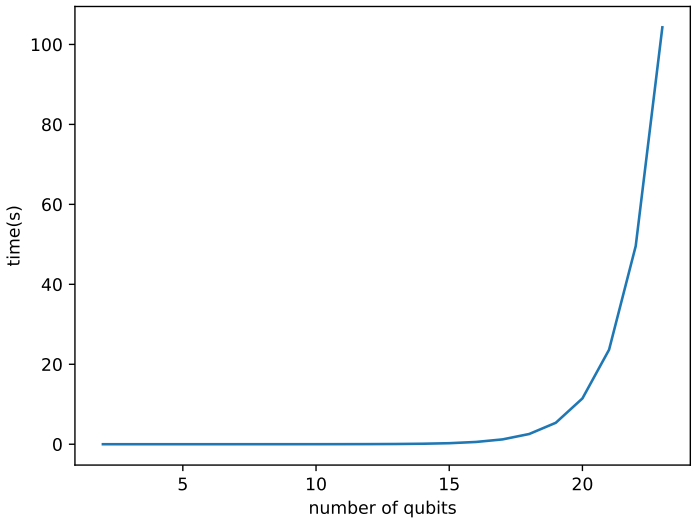

Figure 6 Deutsch-Jozsa execution time.

In Figure 6 we can see the amount of time needed to compute the Deutsch-Jozsa algorithm as the number of qubits grows. It takes exponential time to be executed.

5 Future work

QSimov is under a continuous improvement process roughly adding new features asked by the community. As it is an open-source project, contributions from everyone who could write efficient and clean/readable code, as well as proper suggestions will be very welcome. QSimov is currently used in several different environments, which give us a very valuable feedback to keep developing and improving it, among we want to point out the following:

- UCLM
- UCM
- CESGA: FinisTerrae II/III
- Oak Ridge National Laboratory: Summit

In its initial phase, we have also colaborated with ABDProf, a company located in Valencia which is involved for years in quantum computing. It helped in including some features to be provided by QSimov. Currently we have included, or we are in process of including, the following characteristics:

Listing 2 Deutsch-Jozsa code.

```
def dj_alg(nq, verbose=False):
    my_print = lambda *args: None
    if verbose:
        my_print = print
    # We create a registry of nq qubita
    r = qj.QRegistry(nq)
    # We apply a Pauli X gate to the last qubit
    r = r.apply_gate("X", targets=nq-1)
    # We apply a Hadamard gate to each qubit
    for i in range(nq):
        r = r.apply_gate("H", targets=i)
    # We apply the oracle
    is_balanced = bool(rnd.getrandbits(1))
    if not is_balanced:
        # In this case f(x) = function_result
        # function_result is always the same value
        # for any value of x, constant function
        function_result = rnd.getrandbits(1)
        my_print(f"Using␣f(x)␣=␣{function_result}␣(constant)")
        if function_result == 1:
            r = r.apply_gate("X", targets=nq-1)
    else:
        # In this case f(x) = xj
        # (value of the bit j of the number x)
        # balanced function
        j = rnd.randrange(nq-1)
        my_print(f"Using␣f(x)␣=␣x_{j}␣(balanced)")
        r = r.apply_gate("X", targets=nq-1, controls=j)
    # We apply a Hadamard gate to all but the last qubit
    for i in range(nq-1):
        r = r.apply_gate("H", targets=i)
    # We measure all but the last qubit
    _, res = r.measure([i for i in range(nq-1)])
    dj_result = any(res[:-1])
    if dj_result:
        my_print("Result:␣f(x)␣is␣balanced")
    else:
        my_print("Result:␣f(x)␣is␣constant")
    return dj_result == is_balanced
```

- Simulating over distributed memory architectures
- Simulating over architectures operating with GPUs (NVidia)
- Data structure based on tensor networks instead of state vectors (UJI)
- Including connectors for Qiskit, Forest and QPath
- Noise/error simulation
- Graphic front-end

We are always open to contributions at:

- github.com/Mowstyl/QSimov: For the framework
- github.com/Mowstyl/Doki: For the current simulator

References

1 Charles H. Bennett and Gilles Brassard. Quantum cryptography: Public key distribution and coin tossing. *Theoretical Computer Science*, 560:7–11, 2014. Theoretical Aspects of Quantum Cryptography – celebrating 30 years of BB84. URL: `https://www.sciencedirect.com/science/article/pii/S0304397514004241`, `doi:https://doi.org/10.1016/j.tcs.2014.05.025`.

2 David Deutsch and Richard Jozsa. Rapid solution of problems by quantum computation. *Proceedings of the Royal Society of London. Series A, Mathematical and Physical Sciences*, 439:553–558, 1992. URL: `https://royalsocietypublishing.org/doi/10.1098/rspa.1992.0167`, `doi:http://doi.org/10.1098/rspa.1992.0167`.