Swarm and Evolutionary Computation 69 (2022) 101030

Contents lists available at ScienceDirect

Swarm and Evolutionary Computation

journal homepage: www.elsevier.com/locate/swevo

Quantum circuit compilation by genetic algorithm for quantum approximate optimization algorithm applied to MaxCut problem

Lis Arufe[a], Miguel A. González[a,*], Angelo Oddi[b], Riccardo Rasconi[b], Ramiro Varela[a]

[a] Department of Computing, University of Oviedo, Campus of Gijón, Gijón 33204, Spain
[b] Istituto di Scienze e Tecnologie della Congnizione, Consiglio Nazionale delle Ricerche, Via S. Martino della Battaglia, 44, Rome, Italy

ARTICLE INFO

Keywords:
Quantum circuit compilation
Scheduling
Makespan
Optimization
Genetic algorithm

ABSTRACT

The Quantum Circuit Compilation Problem (QCCP) is challenging to the Artificial Intelligence community. It was already tackled with temporal planning, constraint programming, greedy heuristics and other techniques. In this paper, QCCP is formulated as a scheduling problem and solved by a genetic algorithm. We focus on QCCP for Quantum Approximation Optimization Algorithms (QAOA) applied to the MaxCut problem and consider Noisy Intermediate Scale Quantum (NISQ) hardware architectures. Based on the fact that these algorithms apply a set of basic quantum operations repeatedly over a number of rounds, we propose a genetic algorithm approach, termed Decomposition Based Genetic Algorithm (DBGA), that in each round extends the partial solutions obtained for the previous ones. DBGA is compared to the state of the art across a set of conventional instances. The results of the experimental study provided interesting insight in the problem structure and showed that DBGA is quite competitive with the state of the art. In particular, DBGA outperformed the best current method on the largest instances and provided new best solutions to most of them.

1. Introduction

It is widely accepted that quantum computing may represent the next big step in the field of computation as it could contribute to solve some extremely hard problems. Much in the same way as classic computers operate on logical gates, quantum computing relies on quantum gates operating on quantum bits (qubits), each qubit representing a quantum state (qstate), i.e., a superposition of the two pure qstates, denoted $|0\rangle$ and $|1\rangle$. Executing a quantum algorithm on a quantum hardware entails evaluating a set of quantum gates on qubits (one or two in this study) representing the proper qstates, this process being subject to some constraints imposed by both the algorithm and the hardware. We consider herein the hardware technology termed Noisy Intermediate Scale Quantum (NISQ) processors [1]. As an example, Fig. 1 shows four NISQ quantum chip designs inspired by Rigetti Computing Inc. [2] with different number of qubits ($N = 4, 8, 21, 40$) distributed in weighted and undirected graphs. Each qubit is identified by an integer and it is located in a node. Two qubits connected by an edge are adjacent, which represents that a 2-qubit gate may be executed on those qubits. The type of connecting line, either dashed or continuous, has to do with the processing time of the gates on these qubits. A quantum algorithm may be viewed as a series of quantum gates that must be applied on the qubits over time (see Fig. 3(b)).

Due to the fact that binary gates can only be applied to adjacent qubits, when a gate must be applied on two particular qstates, we have to ensure that they are located on adjacent qubits. To this end, it may be necessary moving qstates from one qubit to another, which is done by means of *swap* gates, i.e., gates that just swap the qstates of two adjacent qubits. Furthermore, two gates cannot be applied on the same qubit at the same time. These facts raise the problem of distributing the calculations on the specific hardware, which is known in the literature as Quantum Circuit Compilation Problem (QCCP) and may be formulated in the planning/scheduling framework.

In this work, we investigate the use of genetic algorithms to solve the QCCP. In particular, we will focus on the model described by Venturelli et al. in [3], where the authors proposed a benchmark - called Venturelli's set herein - and explored the use of temporal planners to synthesize compilation plans characterized by minimum makespan (i.e., the circuit's depth). This model was considered in further proposals [4–6] and is characterized by: (i) the class of Quantum Approximate Optimization Algorithm (QAOA) applied to the MaxCut problem [7], and (ii) the specific hardware architecture depicted in Fig. 1.

* Corresponding author.
E-mail addresses: arufelis@uniovi.es (L. Arufe), mig@uniovi.es (M.A. González), angelo.oddi@istc.cnr.it (A. Oddi), riccardo.rasconi@istc.cnr.it (R. Rasconi), ramiro@uniovi.es (R. Varela).

https://doi.org/10.1016/j.swevo.2022.101030
Received 20 October 2020; Received in revised form 15 December 2021; Accepted 1 January 2022
Available online 7 January 2022

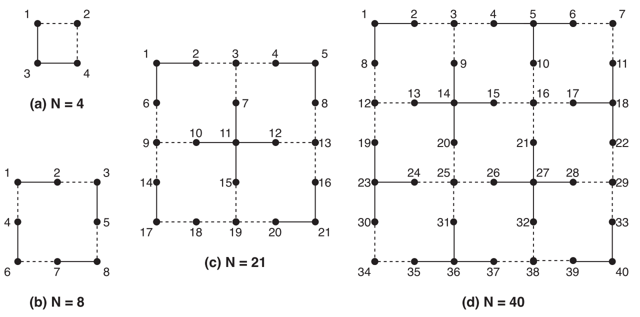


Fig. 1. Four quantum chip designs with different number of qubits.

Due to the fact that QAOA is based on the application of the same set of gates over a number of rounds, we propose a decomposition based approach termed Decomposition Based Genetic Algorithm (DBGA), which builds solutions incrementally. It runs for a number of rounds and, in each one, it applies a genetic algorithm to the subproblem given by the current round, starting from solutions to the previous rounds. Our approach is quite different in many aspects from the genetic algorithm proposed in [5] for the same problem; among others the chromosome encoding, the decoding algorithm, the heuristic population initialization, the diversification operator or the decomposition strategy are specific to DBMA.

One of the key points of DBMA is the coding/decoding schema: in each round r, a chromosome is composed by the sequence of quantum gates in the round r together with a partial solution to the subproblem of rounds $1, \dots, r-1$. Each gate operates on a pair of qstates, and the chromosome indicates the pair of adjacent qubits in which it must be executed. The decoding algorithm inserts the minimum number of *swap* gates to move the qstates to the corresponding qubits. Other elements such as the diversification mechanism or the initialization procedure also contributed to the algorithm performance; but it was the decomposition approach the element that most clearly contributed to make DBMA an outstanding algorithm.

In the experimental study, we analysed the contribution of the components of DBGA and compare it to the best methods in the state of the art that, as far as we know, are the genetic algorithm proposed in [5] and the rollout heuristic proposed in [6]. In these experiments, DBGA shows better performance than these two methods; in particular, it reached new best solutions for 48 of the 50 largest instances of the considered benchmark set.

The remainder of the paper is organized as follows. In the next section, a literature review is presented. In Section 3 the formal definition of the QCCP for MaxCut is given. Section 4 describes the Decomposition Based Genetic Algorithm (DBGA) proposed to solve the QCCP. Section 5 reports the experimental study. The main conclusions and some ideas for future research are summarized in Section 6. Additionally, we include two appendices. Appendix A describes the basis of the Quantum Approximate Optimization Algorithm (QAOA), and Appendix B describes how QAOA may be applied to the MaxCut problem.

2. Literature review

Despite the issue related to the compilation (a.k.a. qubit mapping [1]) of quantum circuits is relatively new, a number of works in the literature have been published in the recent years, which tackle the QCCP from different perspectives, leveraging different techniques and targeting different objectives.

After the approach presented in [3], temporal planning was also exploited in [8], where it was combined with Constraint Programming (CP) to produce *warm-start* solutions. In that paper, the authors considered two variants of the QCCP; the first one accounts for crosstalk interactions between qubits (QCCP-X), while the second includes the ability to arbitrarily initialize qubits (QCCP-I[1]). The qubit initialization problem was also analysed in [9].

In [4] the authors proposed a greedy random search heuristic to solve the same problem, where the main contribution is a lexicographic two-key ranking function for quantum gate selection. The first key acts as a global closure metric aimed to minimize the makespan, and the second one is a local metric acting as "tie-breaker" to avoid cycles. This heuristic improved the results reported in [3].

Later, in [5], the same authors proposed a genetic algorithm exploiting a novel chromosome encoding, where each gene controls the iterative selection of a quantum gate to be inserted in the solution; to this end they used the ranking function proposed in [4]. This genetic algorithm improved the results reported in [4]. Besides, they tackled the two extensions QCCP-X and QCCP-I.

In [6] three approaches were proposed to solve the problem: 1) a schedule builder guided by a priority rule, which is a variant of that proposed in [4]; this rule selects one among the eligible gates in each iteration based on distances between the current qubits of the qstates involved in the eligible *p-s* gates; 2) a rollout based sequential decision making approach, which decides on which operation to schedule next based on the makespan projection given by the aforementioned priority rule, and 3) a stochastic version of the rollout heuristic, which iteratively switches between rollout and the priority rule in order to improve the exploration capabilities of the method. As far as we know, this paper reports the best results to date on the benchmark set proposed in [3].

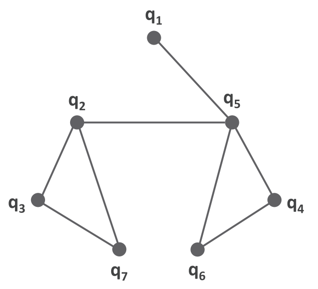
A significant amount of papers in the literature analyze quantum compilation algorithms that add swaps to the ideal circuit, focusing on IBM QX architectures, so that the circuit's behavior can be readily simulated on the IBM's Qiskit framework.[2] In particular, in [10] the analysis is targeted at minimizing the number of additional quantum gates necessary to perform the compilation, the rationale being that the fewer the gates, the shallower the circuit's depth. Similarly, depth minimization is the primary objective of the heuristic-based compilation procedure proposed in [11].

Still focusing on IBM QX architectures, the analysis carried out in [12] aims at minimizing both the number of gates in the final circuit realization (leveraging *swaps, bridges* and *reversals*), and the time occurred for the compilation process. In [13] an interesting analysis is made about the trade-off between the number of swap gates introduced to realize the compilation process and the circuit's depth, highlighting the fact that the minimization of the number of swaps and the minimization of circuit's depth may be mutually conflicting objectives. In [14], some approaches to solve the quantum compilation problem using off-the-shelf MILP solvers, such as Gurobi, have been investigated. Other heuristics were also applied with success to the QCCP as, for example, simulated annealing and lookahead heuristics [15].[3]

Another very interesting approach to the quantum circuit compilation problem is to take into account different performance parameters. In [16] for instance, the authors consider the gate error rates to produce circuit realizations that maximize the circuit's fidelity, while attempting to minimize the number of *swap* and/or *bridge* gates. A similar approach that takes into account the analysis of gate error rates has been used in [17,18].

A relatively new approach to quantum circuit compilation is based on learning. The work [19] presents a procedure to convert the approximate compilation problem into an auxiliary quantum variational algorithm native on the hardware. Whereas in [20] an iterative learning procedure is used to solve both the compilation problem and the gate synthesis problem.

[1] This problem is denoted QCCP-V in [5].
[2] https://qiskit.org/
[3] In this paper, the problem is called Quantum Circuit Transformation.



P-S$_1$	MIX$_1$	P-S$_2$	MIX$_2$
p-s(q$_1$,q$_5$)	mix(q$_1$)	p-s(q$_1$,q$_5$)	mix(q$_1$)
p-s(q$_2$,q$_3$)	mix(q$_2$)	p-s(q$_2$,q$_3$)	mix(q$_2$)
p-s(q$_2$,q$_5$)	mix(q$_3$)	p-s(q$_2$,q$_5$)	mix(q$_3$)
p-s(q$_2$,q$_7$)	mix(q$_4$)	p-s(q$_2$,q$_7$)	mix(q$_4$)
p-s(q$_3$,q$_7$)	mix(q$_5$)	p-s(q$_3$,q$_7$)	mix(q$_5$)
p-s(q$_4$,q$_5$)	mix(q$_6$)	p-s(q$_4$,q$_5$)	mix(q$_6$)
p-s(q$_4$,q$_6$)	mix(q$_7$)	p-s(q$_4$,q$_6$)	mix(q$_7$)
p-s(q$_5$,q$_6$)		p-s(q$_5$,q$_6$)	

Fig. 2. MaxCut problem instance on a graph with 7 nodes. Each node is associated with a particular qstate q_i and such associations define the compilation objectives. The right side of the figure shows the list of *p-s* and *mix* quantum gates to be planned for and executed, in this case we consider $p = 2$.

Another recent approach for quantum circuit compilation is that presented in [21]. The quantum gates are heuristically separated in layers such that all gates in the same layer can be concurrently executed, then a layer ordering is heuristically decided, and finally a compiler backend introduces *permutation layers* (i.e. swap gates) that move the qstates so that the quantum gates of the next layer can be performed. The authors extend the study in [22] by proposing the QAIM heuristic to improve the initial location of qstates, and different heuristics to perform the layer ordering, including one that takes into account the reliability of the operations in order to maximize the success probability of the circuit.

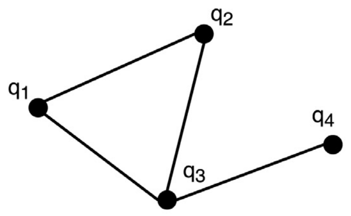
3. Quantum circuit compilation problem formulation

In the QCCP, we are given a tuple $P = \langle C_0, L_0, QM\rangle$. C_0 is the input quantum circuit, which represents the quantum algorithm to solve the problem at hand. L_0 is the initial assignment of qstates to qubits and QM is a representation of the quantum hardware as a multigraph.

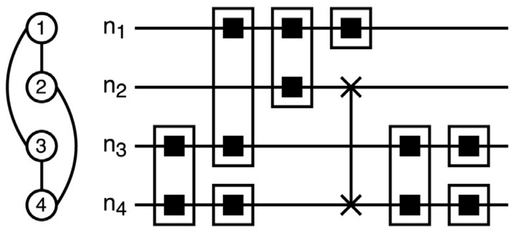
If the problem at hand is MaxCut, the input quantum circuit can be defined as $C_0 = \langle Q, P\text{-}S, MIX, \{g_{start}, g_{end}\}, TC_0\rangle$, where $Q = \{q_1, \dots, q_N\}$ is the set of qstates which, from a planning and scheduling perspective, represent the resources necessary for each gate's execution. *P-S* and MIX are, respectively, the set of *p-s* and *mix* gate operations, for the first one some mathematical formulations are given in Eqs. (B.7), (B.8) and (B.9), while for the second the formulation is that of Eq. (B.6). As they operate on two and one qstates respectively, we use the notations $p\text{-}s(q_i, q_j)$ and $mix(q_i)$. Two fictitious gates g_{start} and g_{end} are also considered that do not operate on any qstate. Every quantum gate requires the uninterrupted use of the involved qstates during all its processing time, and each qstate q_i can be processed by at most one gate at a time.

TC_0 is a set of precedence constraints imposed on the *P-S*, MIX and $\{g_{start}, g_{end}\}$ sets. Every gate in *P-S* and MIX must be executed after g_{start} and before g_{end}. According to the number of rounds p of the QAOA algorithm (see Appendix A), *P-S* and MIX sets are organized into p steps that must be interleaved as $P\text{-}S_1$, MIX_1, $P\text{-}S_2$, MIX_2, ..., $P\text{-}S_p$, MIX_p. All the gates belonging to the step $P\text{-}S_r(MIX_r)$ involving a specific qstate q_i must be executed before all the gates $MIX_r(P\text{-}S_{r+1})$ involving the same qstate q_i, for $r = 1, 2, \dots, p$ (for $r = 1, 2, \dots, (p-1)$). Therefore, p is the number of compilation passes. The *p-s* gates are commutative, so they may be executed in any order. Figure 2 (left side) shows an example of a graph upon which the MaxCut problem is to be executed, corresponding to the problem instance no. 1 of the subset characterized by $N = 8$, $u = 0.9$ and $p = 2$ (see Section 5.1). The list of *p-s* and *mix* quantum gates that must be executed during the compilation procedure is shown on the right side of the figure. The compilation problem depicted in the figure requires two compilation passes (i.e. $p = 2$).

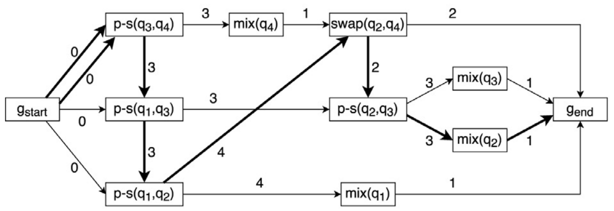
QM is a representation of the quantum hardware as an undirected graph $QM = \langle V_N, E_{p-s}, E_{swap}, \tau_{mix}, \tau_{p-s}, \tau_{swap}\rangle$, where $V_N = \{n_1 \dots n_N\}$ is



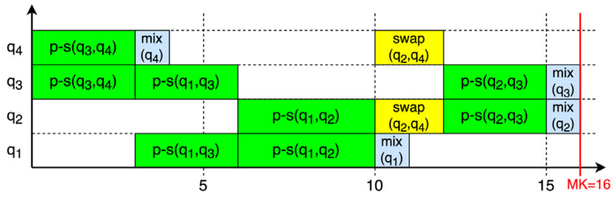
(a) MaxCut instance.



(b) Quantum circuit for the MaxCut instance in Figure 3(a). Rectangles represent *p-s* gates, squares represent *mix* gates and single lines represent *swap* gates. Initially each qstate q_i is on qubit n_i. The quantum hardware is represented on the left. Each horizontal line represents the operations on one qubit over time.



(c) Solution graph. Bold arcs indicate the critical path, whose length (16) corresponds to the makespan. The node g_{start} represents the initial situation, i.e., the qstate q_i in qubit n_i. Double arcs means that a binary gate takes the two inputs from the same source (the g_{start} node or another binary gate).



(d) Gantt chart, representing the operations on each qstate over time.

Fig. 3. Example of MaxCut instance (a) and one possible solution, considering only one round (i.e. $p = 1$), represented by a quantum circuit (b), a solution graph (c) and a Gantt chart (d).

the set of qubits (nodes) of the hardware. E_{p-s} and E_{swap} are the sets[4] of undirected edges (n_k, n_l) which represent adjacent locations where the qstates q_i and q_j of the gates $p\text{-}s(q_i, q_j)$ or $swap(q_i, q_j)$ can be allocated to; *mix* gates can be executed at any node. τ_{mix}, $\tau_{p\text{-}s}$ and τ_{swap} represent the durations for the different types of gates. In our study, $\tau_{mix} = 1$, $\tau_{swap} = 2$ and τ_{p-s} is 3 if the gate is executed in a continuous edge and it is 4 when it is executed in a dashed edge (see Fig. 1).

A feasible solution is a tuple $S = \langle SWAP, TC\rangle$, which extends the initial circuit C_0 with a set $SWAP$ of *swap* gates, added to guarantee the adjacency constraints for the qstates in *P-S* gates, and a set TC of additional precedence constraints such that for each qstate q_i a total order

[4] In this study, both sets are in fact the same.

is imposed among the set of operations requiring q_i. TC must guarantee that the qstates (q_i, q_j) of all *p-s* and *swap* gates are allocated on adjacent qubits. The makespan of a solution corresponds to the maximum completion time of the gate operations in S.

A schedule may be viewed as a solution graph $G_S = (V_S, E_S)$, where $V_S = \{g_{start}, g_{end}\} \cup P\text{-}S \cup SWAP \cup MIX$ and E_S represents the partial order on the operations in V_S defined by TC_0 and TC. Figure 3(c) shows an example, in some cases there are two equivalent arcs between two nodes, this is just to emphasize that they correspond to binary gates sharing the two qstates. The costs in the arcs are the durations of the operations in the origin. The makespan of the schedule is given by the longest path cost from g_{start} to g_{end}, which is called *critical path*.

The goal of the QCCP is to find a feasible solution with the minimum makespan. In [23] QCCP is proven to be NP-hard.

As an example, consider a toy instance in the chip with $N = 4$ (see Fig. 1(a)), in which the following four gates: *p-s*(q_1, q_2), *p-s*(q_1, q_3), *p-s*(q_2, q_3) and *p-s*(q_3, q_4) must be executed. Figure 3a shows a graph of a MaxCut instance with 4 nodes (the number of nodes must be lower than or equal to the number of qubits in the quantum chip). Figure 3b shows a solution in form of compiled quantum circuit, considering only one round, i.e. $p = 1$ (be aware that a $swap$ gate was required to ensure adjacency compliance). Figure 3c shows the equivalent solution graph, whose critical path has cost 16. Notice that *p-s* and $swap$ gates have two predecessor nodes and two successor nodes each (unless they operate on the same qstates of the predecessor or successor gates, which is represented by means of double arc, as mentioned), which correspond to the previous (resp. next) gates executed on the corresponding two qstates. mix gates have only one predecessor and one successor, since they operate on only one qstate. In any case, if a gate is the first (resp. last) one executed on a qstate, then its predecessor (resp. successor) is the node g_{start} (resp. node g_{end}). Finally, Fig. 3d shows the Gantt chart of the solution, with makespan 16.

4. The decomposition-based genetic algorithm

Given the particular structure of the problem, specifically the fact that a problem instance is composed by a set of p rounds and that in each round the same sets of *p-s* and *mix* gates must be scheduled, we propose an incremental procedure that in each step $r \in \{1, \dots, p\}$ solves the subproblem defined by the rounds $1, \dots, r$. To be more precise, the proposed DBGA works in a decomposition-based approach and tries to optimize the scheduling of the *p-s* gates in round r, given a number of solutions to the subproblem defined by rounds $1, \dots, r-1$ (*mix* gates in round r are trivially scheduled after all *p-s* gates that involve the corresponding qstate).

Hence, in each round DBGA takes solutions from the previous round and “extends” them by adding the corresponding gates of the new round. This is done by means of a genetic algorithm, and so it first creates an initial population of solutions, which are evolved by using selection, recombination, replacement and diversification operators, until a termination condition is met. The flow chart of DBGA is depicted in Fig. 4, and its main components are detailed in the following subsections.

4.1. The coding scheme

As pointed, one execution of the GA starts from a set of solutions to the subproblem $1, \dots, r-1$ calculated in previous steps, and builds schedules to the subproblem $1, \dots, r$. Therefore, a chromosome has to include a solution to the subproblem $1, \dots, r-1$ and the encoding of a candidate schedule for the *p-s* gates in round r. To this end, we propose encoding a chromosome by a triplet of the form $(sg_{r-1}, ch1, ch2)$, where sg_{r-1} is a solution graph for the subproblem $1, \dots, r-1$ and $ch1$ and $ch2$ are two chains of symbols; $ch1$ is a permutation of the set of the *p-s* gates in round r and $ch2$ is a sequence of pairs of adjacent qubits of the same length as $ch1$. This encoding must be understood so as the gate *p-s*(q_i, q_j)

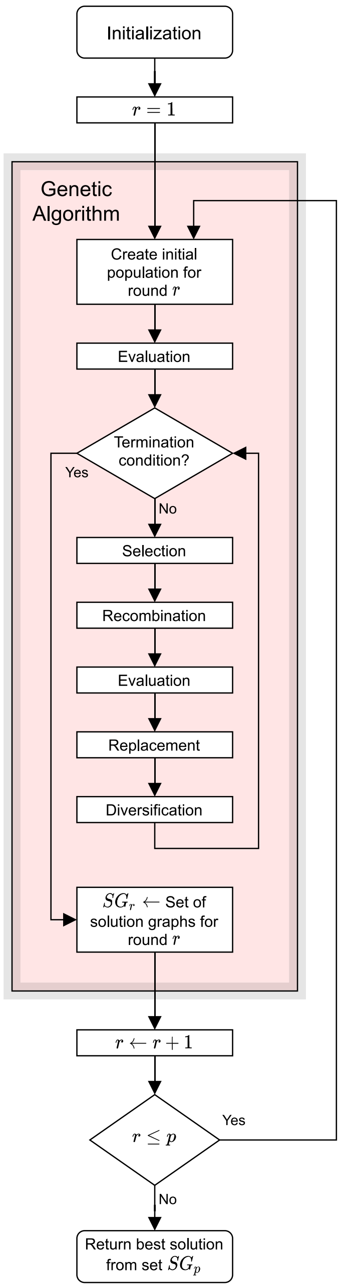


Fig. 4. Flow chart of DBGA.

in a position of $ch1$ must be executed on the pair of qubits $\{n_k, n_l\}$ in the same position in $ch2$; and the schedule must be built taking into account the earliest starting times of the quantum gates on the qubits after the partial solution to the subproblem $1, \dots, r-1$, as we will see in Section 4.3.

4.2. Recombination and mutation

Regarding crossover and mutation, any operator devised for permutation encoding may be adapted. The crossover operates on components $ch1$ and $ch2$ at the same time to generate two offspring. Each offspring takes the sg component from one parent. In our experiments, we extended the well-known partial mapping crossover (PMX). This operation is illustrated in Fig. 5. Besides, we consider a single mutation (*mut1*) that consists in swapping two positions (the gate and the pair of qubits) chosen at random.

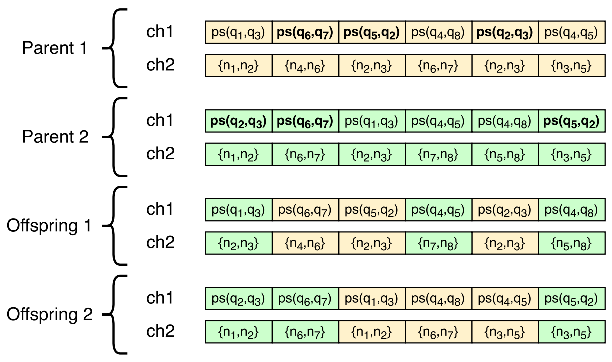

Parent 1
ch1 ps(q1,q3) ps(q6,q7) ps(q5,q2) ps(q4,q8) ps(q2,q3) ps(q4,q5)
ch2 {n1,n2} {n4,n6} {n2,n3} {n6,n7} {n2,n3} {n3,n5}
Parent 2
ch1 ps(q2,q3) ps(q6,q7) ps(q1,q3) ps(q4,q5) ps(q4,q8) ps(q5,q2)
ch2 {n1,n2} {n6,n7} {n2,n3} {n7,n8} {n5,n8} {n3,n5}
Offspring 1
ch1 ps(q1,q3) ps(q6,q7) ps(q5,q2) ps(q4,q5) ps(q2,q3) ps(q4,q8)
ch2 {n2,n3} {n4,n6} {n2,n3} {n7,n8} {n2,n3} {n5,n8}
Offspring 2
ch1 ps(q2,q3) ps(q6,q7) ps(q1,q3) ps(q4,q8) ps(q4,q5) ps(q5,q2)
ch2 {n1,n2} {n6,n7} {n1,n2} {n6,n7} {n3,n5} {n3,n5}

Fig. 5. Illustration of the crossover operator. A subset of *p-s* gates with their destination qubits are taken from the first parent and put in the same positions in one offspring. The remaining *p-s* gates with their destination qubits are taken from the second parent and inserted in the remaining positions of the offspring keeping their relative order.

With the above operators, only the pairs of qubits allocated to the *p-s* gates in the initial population would be considered in the evolutionary process. So, some other operators should be considered that may introduce new genetic material in the form of new candidate pairs of qubits for the *p-s* gates. To this end, we propose using another single mutation ($mut2$) that changes the pair of qubits for one *p-s* gate by a random pair of adjacent qubits sharing one qubit with the original pair. When a chromosome is mutated, one of the operators is chosen with equal probability.

4.3. The decoding algorithm

Building a schedule from a chromosome is the most critical issue given the particular characteristics of the QCCP. To this end, the decoding algorithm iterates over the $ch1$ component of the chromosome and takes the actions required to schedule the $p\text{-}s(q_i, q_j)$ gate in each position on the pair of qubits $\{n_k, n_l\}$ in the same position in $ch2$. These actions introduce the necessary $swap$ gates to move the qstates q_i and q_j to the qubits n_k and n_l. In general there may be many possibilities to do that and it is not clear which is the most appropriate in each circumstance. Besides, it is not easy to encode in the chromosome one among the full set of options. For these reasons, we opted to use a strategy that minimizes the number of $swap$ gates in each scheduling decision. In this way, the proposed encoding/decoding schema does not guarantee that at least one of the feasible chromosomes can encode an optimal schedule to the *p-s* gates in round r (after a schedule for the subproblem defined by rounds $1, \dots, r-1$); in other words, the search space is not dominant.

Let $D(x, y)$ be the length of the shortest path between the qubits x and y in the quantum hardware QM, trivially, this distance is the minimum number of $swap$ gates required to move a qstate in qubit x to qubit y.

Let $n(q)$ be the qubit holding the qstate q at a given time. To move the qstates q_i and q_j from their current qubits $n(q_i)$ and $n(q_j)$ to the target qubits n_k and n_l we choose the option that minimizes the overall number of $swap$ gates; namely q_i is moved to n_k and q_j is moved to n_l if $D(n(q_i), n_k) + D(n(q_j), n_l) < D(n(q_i), n_l) + D(n(q_j), n_k)$ and the alternative is taken otherwise. The chosen paths are called a *pair of minimal paths*. The number of $swap$ gates required is given by the length of these paths as it is proven in the following result, where $d(q_i)$ and $d(q_j)$ denote the "destination" qubits of qstates q_i and q_j respectively.

Proposition 1. *Let* $n(q_i), \dots, d(q_i)$ *and* $n(q_j), \dots, d(q_j)$ *be a pair of minimal paths chosen to move the qstates* q_i *and* q_j *from the qubits* $n(q_i)$ *and* $n(q_j)$ *to the qubits* n_k *and* n_l*, then the number of* $swap$ *gates required is* $D(n(q_i), d(q_i)) + D(n(q_j), d(q_j))$.

Proof sketch. If the two paths are disjoint, i.e., they have not any qubit in common, the result fulfils trivially as each path is translated into a set of gates that is independent from the gates of the other path. However, if both paths have some qubit in common, it is possible that one $swap$ gate in one path involves two qubits holding q_i and q_j, which could prevent one of these qubits from being moved with just the minimum number of swaps. However, if this were the case, instead of scheduling a gate swapping q_i and q_j, we may change their destinations instead, and so the $swap$ gate is not actually necessary and the qubits may go to the new destinations with no additional swaps. □

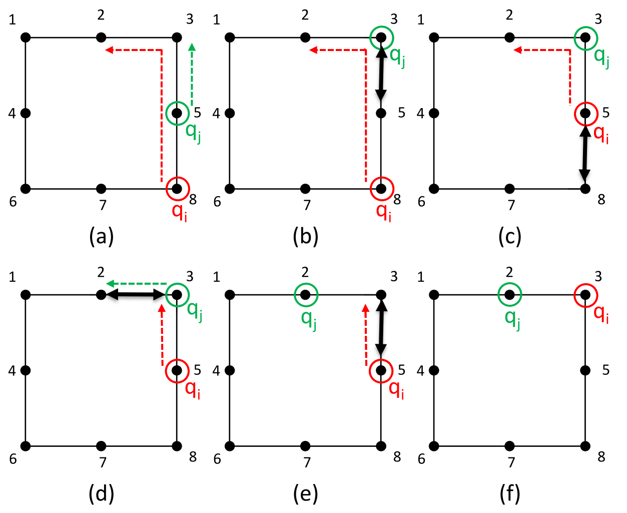


Fig. 6. Illustration of how Algorithm 1 could move two qstates q_i and q_j from their initial qubits 8 and 5 to the destination qubits 2 and 3.

Algorithm 1 shows the decoding algorithm that is applied to a chromosome $(sg_{r-1}, ch1, ch2)$ to obtain a schedule (a solution graph sg_r) to the subproblem defined by the rounds $1, \dots, r$. It iterates on the $p\text{-}s(q_i, q_j)$ gates in $ch1$ and each one is scheduled in the pair of qubits $\{n_k, n_l\}$ in the same position in $ch2$. To do that, it starts calculating a pair of minimal paths $path_1$ and $path_2$ from the current qubits of the qstates q_i and q_j to the destination qubits n_k and n_l. Then, the arcs in these paths are considered following the order in each path, and for each one the associated $swap$ gate is introduced in the partial solution. In this process the arcs from the two paths may be interleaved trying to avoid reversing two adjacent qubits holding q_i and q_j. However, if in one iteration this is the only option then the remaining paths from the qstates to their destinations are swapped instead of swapping the qubits. In this way, the number of $swap$ gates can be maintained to the minimum (see Proposition 1).

Figure 6 depicts the decisions that could be taken by Algorithm 1 in the case of a single $p\text{-}s(q_i, q_j)$ gate such that the destination pair of adjacent qubits is $\{n_3, n_2\}$, $path_i = n_8 \to n_5 \to n_3 \to n_2$ and $path_j = n_5 \to n_3$ (graphically represented by the dashed lines), in a 8-qubit quantum machine (see Fig. 1(b)). Let us suppose that the first swap to be selected by the algorithm brings q_j from qubit n_5 to qubit n_3 (Fig. 6(b)), and the second swap brings q_i from qubit n_8 to qubit n_5 (Fig. 6(c)). It is now evident that the condition $\{n_3, n_2\} = \{n(q_i), n(q_j)\}$ holds (if q_i continued its path towards qubit n_2, it would have to swap its position with q_j, thus reversing the latter's position); therefore, the algorithm enters the "else" branch performing a destination swap (Fig. 6(d)), ultimately dispatching q_j to qubit n_2 (Fig. 6(e)) and q_i to qubit n_3 (Fig. 6(f)).

After introducing all the *p-s* gates, the *mix* gates are inserted on each qubit holding a qstate.

We have also implemented a small improvement in the described decoding procedure: each time a *swap* gate is inserted, we check if one of the involved qstates has executed a *mix* gate that ends just when the *swap* gate starts (see Fig. 7(a)). If that is the case, we reverse the order of the *mix* and the *swap* gates (see Fig. 7(b)), as in this way we may be able to improve the makespan (if $SWAP_j$ of Fig. 7(b) is on the

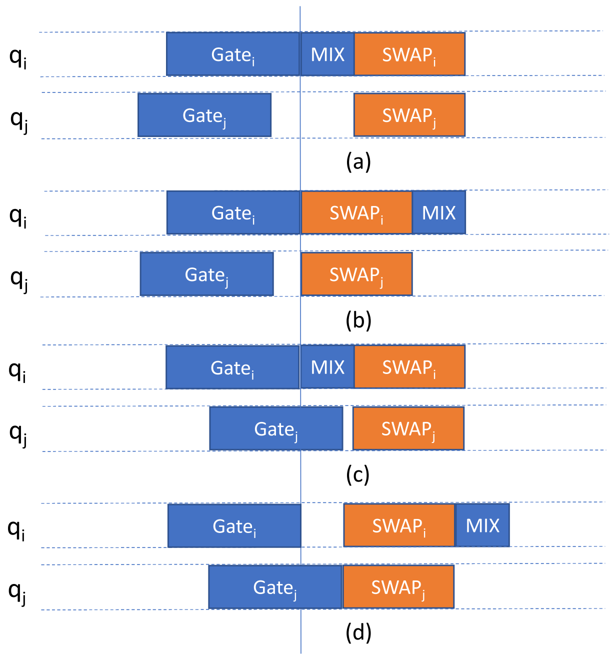


Fig. 7. Additional improvement step within the decoding procedure of Algorithm 1: in case (a) the reversal of the gates MIX and $SWAP_i$ may improve the makespan (see subfigure (b)) if $SWAP_j$ is on the critical path of the resulting solution. Whereas in case (c) the makespan might be worse if the MIX gate is on the critical path of the resulting solution, as subfigure (d) shows.

critical path of the resulting solution) as much as the duration of the *mix* gate. However, the described reversal is only performed if the last gate executed by the other involved qstate ends at a time lower than or equal to the starting time of the *mix* gate (see Fig. 7(a)), as in that case we can ensure that the makespan cannot be worse. On the contrary, Fig. 7(c) shows a case where the above described reversal might not be convenient (see Fig. 7(d)), particularly if that MIX gate is on the critical path of the resulting solution.

4.4. General structure of DBGA

The general structure of the proposed DBGA is shown in Algorithm 2.

```
Require: A QCCP problem instance $P$ with a set of qstates $Q$, $p$ rounds
    and $P\text{-}S_r$ and $SWAP_r$ sets of gates in each round $r \in 1,\dots,p$. The quantum hardware $QM$. The set of $Parameters$ of the GA: $popSize$, $P_c$, $P_m$.
Ensure: A schedule $H$ for instance $P$ on circuit $QM$.
    Distribute the qstates in $Q$ on the qubits of $QM$;
    $SG_0 \leftarrow$ set of $popSize$ initial solution graphs;
    for r = 1 to p do
        $SG_r \leftarrow GA(SG_{r-1}, P\text{-}S_r, MIX_r, Parameters)$;
    end for
    return The best schedule in $SG_p$;
```

Algorithm 2 : The Decomposition-Based Genetic Algorithm. It builds a schedule to a QCCP instance P iterating on the rounds $1,\dots,p$ of P. In each iteration r it builds schedules for round r compatible with partial solutions of the subproblem $1,\dots,r-1$.

It takes as input a QCCP instance P having p rounds and a quantum hardware QM, and produces a schedule H for P on QM. The algorithm starts distributing the qstates on the qubits; here we assume that the initial distribution is the same for all candidate solutions. Then the initial solution graphs SG_0 are calculated. These are trivial solution graphs of the form $g_{init} \rightarrow g_{end}$ representing solutions of the fictitious round 0. The algorithm iterates on the number of rounds, and in each iteration it calls the genetic algorithm (GA) to extend the solutions from round $r-1$ to round r. DBGA returns the best solution graph obtained in the last round.

4.5. The genetic algorithm

Algorithm 3 shows the GA proposed to obtain solutions in each round

```
Require: A set $SG_{r-1}$ of $popSize$ solution graphs for subproblem
    $1,\dots,r-1$, the sets of gates $P\text{-}S_r$ and $MIX_r$. The $Parameters$: $popSize$, $P_c$, $P_m$. The quantum circuit $QM$.
Ensure: A set $SG_r$ of $popSize$ solution graphs for subproblem $1,\dots,r$.
    Initial Population: for each $sg_{r-1} \in SG_{r-1}$ generate an initial chromosome $(sg_{r-1}, ch1, ch2)$, where $ch1$ is a random permutation of the gates in $P\text{-}S_r$ and $ch2$ is a set of connections in $QM$, which may be calculated either at random or by some heuristic;
    Evaluation: The decoding algorithm is applied to the initial chromosomes to extend the solution graphs $sg_{r-1}$ to the first series of solution graphs $sg_r$ in accordance with components $ch1$ and $ch2$ in each chromosome;
    while not Termination Condition do
        Selection: The chromosomes in the population are organized into random pairs;
        Recombination: Each pair of chromosomes is mated and the offsprings mutated in accordance with probabilities $P_c$ and $P_m$ respectively;
        Evaluation: The decoding algorithm is applied to the offsprings to extend the solution graphs $sg_{r-1}$ to new solution graphs $sg_r$;
        Replacement: for each pair of two parents and its two offspring, the two best solutions are selected for the next population;
        Diversification: After a number of generations without improvement, some of the chromosomes sharing the same fitness are mutated a number of times;
    end while
    $SG_r \leftarrow$ solution graphs $sg_r$ of the $popSize$ chromosomes in the population;
    return The set of solution graphs $SG_r$;
```

Algorithm 3 : The Genetic Algorithm. It builds a number of schedules for round r of a QCCP instance P after partial solutions to the subproblem defined by the rounds $1,\dots,r-1$.

r. It starts from a set $popSize$ of solution graphs SG_{r-1} for the subproblem $1,\dots,r-1$ and produces a set of solution graphs SG_r of the same size for the subproblem $1,\dots,r$. Initially, it builds the initial $popSize$ chromosomes of the form $(sg_{r-1}, ch1, ch2)$, for each sg_{r-1} in SG_{r-1}; taking in principle $ch1$ as random permutations of the *p-s* gates of round r and $ch2$ as a sequence of the same length as $ch1$ of random pairs of connected qubits in QM (in Section 4.5.1 we will see a heuristic alternative). The initial chromosomes are evaluated to obtain the first candidate solution graphs sg_r for each one. Each chromosome registers its own solution graph sg_r along the algorithm's iterations.

Then, GA iterates until the *Termination Condition* is fulfilled. This condition being that there is no improvement in a given number of consecutive generations. In each iteration, the chromosomes in the population are organized into random pairs, which are then mated and mutated. The resulting offsprings are evaluated (Algorithm 1) to obtain new solution graphs sg_r. Finally, in the replacement step two chromosomes are selected using tournament between each two parents and their two offspring.

We have observed in some preliminary experiments that sometimes the algorithm tends to converge prematurely. So, we introduced a diversification step in which after a number of consecutive generations without improvement, all but one of the chromosomes with the same fitness value are mutated a number of times in order to reintroduce variety in the population.

Finally, GA returns a set of $popSize$ solution graphs SG_r which will be used as input values for the next iteration, if $r < p$, or they represent solutions to the whole problem, if $r = p$.

4.5.1. *Heuristic initial population*

Random generation of initial chromosomes is able to obtain diversity in the genetic material of the population, but it often happens those random chromosomes are so bad that a genetic algorithm can hardly converge to good solutions after a large number of generations. For this reason, it could be better to exploit some heuristic to seed at least some part of the population with individuals representing good solutions. This may help reaching better solutions quickly, at the risk of premature convergence. In any case, heuristic seeding demonstrated to be good in some well-known problems as Travelling Salesman Problem [24], where the Nearest-Neighbour heuristic is exploited, or the Job Shop Scheduling Problem [25], where the authors used variable and value ordering heuristics for some specific types of instances.

We propose to use here a kind of Nearest-Neighbour heuristic to build the $ch2$ component of the chromosomes. The idea is quite simple: the *p-s* gates in $ch1$ are visited from left to right and each gate $p\text{-}s(q_i, q_j)$ is allocated to one of the closest pair of adjacent qubits (n_k, n_l), considering the notion of minimal pair of paths as distance between pairs of qubits; so the candidate destination qubits are of the form

$$(n_k, n_l) = argmin\{min(D(n(q_i), n) + D(n(q_j), n'),\\ D(n(q_i), n') + D(n(q_j), n)), (n, n') \in E_{p-s}\}$$

In case of ties, a continuous edge of the graph is chosen, as the *p-s* gate takes lower execution time.

It is clear that in general the candidate destination is not unique, in that case one of them is chosen at random. Besides, we may give priority to arcs joining qubits not selected previously to favour parallel execution of gates. In this way, the initialization procedure may obtain a diversity of heuristic $ch2$ components.

4.6. *Analysis of DBGA*

As mentioned, we believe that the key point of DBGA that makes a difference w.r.t. previous methods is that of decomposition. In doing so, DBGA tries a number of solutions for round r that are created from evolved solutions to the subproblem $1, \dots, r-1$; therefore it has more chance of success when evolving good solutions to the subproblem $1, \dots, r$ than a greedy heuristic as stochastic rollout, which only tries a solution for round r after each partial solution for the subproblem $1, \dots, r-1$. On the other hand, as the decomposition approach is a kind of local optimum strategy that reduces the search space, it is different from other strategies as the rollout heuristic proposed in [6] or the genetic algorithm (GA) proposed in [4] that consider the whole problem at once, and so it may lead to suboptimal solutions. We believe that this issue has to be clarified through experimental study. Another point that is worth mentioning is the encoding/decoding schema used in DBGA. The double chain encoding allows a *p-s* gate to be executed on any pair of adjacent qubits disregarding the current positions of its qstates. Clearly, this may lead to really bad solutions if the initial chains $ch1$ and $ch2$ are uniformly generated. For this reason, we have proposed a heuristic strategy that tries to locate the *p-s* gates in the proximity of the current positions of their qstates. Then, from this initial situation, DBGA has the flexibility to move the *p-s* gates towards more convenient qubits that might be in farther locations.

5. Experimental study

In order to analyse the components of the proposed DBGA and to make a comparison with the state of the art, we performed an experimental study trying to follow good practices and recommendations outlined in [26], as for example that of considering multiple independent runs so that statistical methods can deliver significant conclusions, or that of publishing results in public repositories. Our algorithms were implemented in C++ and the target machine was Intel Core i5-7400 CPU at 3.00 GHz with 16 GB RAM. As the proposed algorithms are stochastic, we performed 10 independent runs per instance in order to obtain statistically significant results. For each run we registered the best solution reached and the time taken. Hence, for each instance we report the average makespan and some dispersion measure, as the standard deviation, or the best and worst of the 10 solutions, as well as the average time taken in one run.

In the next subsection we describe the benchmark instances considered. Then we detail the running parameters of DBGA and analyze the contribution and effectiveness of each of its components. Afterwards, we compare the results from DBGA with those of the best state-of-the-art methods, which as far as we know are those reported in [6] and [5]. Finally, we also compare our method with those presented in [21] and [22].

5.1. *Benchmark set*

The experimental study was carried out across the instances proposed in [3], which may be downloaded from the web.[5] The benchmark consists of instances of several sizes, depending on the number of qubits of the quantum chip considered ($N \in \{4, 8, 21, 40\}$ (see Fig. 1)). For each chip size, different utilization levels ($u \in \{0.9, 1.0\}$) and number of required compilation passes ($p \in \{1, 2\}$) are considered. The benchmark includes 50 instances for each combination of $\{N, p, u\}$; each instance is representative of a graph G to be partitioned by the *MaxCut* procedure. The instances with utilization level $u = 0.9$ are built randomly choosing 90% of the available qstates to allocate over the N qubits of the instance graph G, while the instances with $u = 1.0$ are built by possibly allocating all the qstates over the N qubits of the graph G. In our experimental study we only consider those instances with utilization level $u = 1.0$ and required compilation passes $p = 2$, as they are the most challenging.

5.2. *Running parameters*

From some preliminary experiments, we propose the following set of parameters for DBGA for all instances: population size of 1000 chromosomes, crossover rate P_c=100% and mutation rate P_m=5%. The heuristic method described in Section 4.5.1 for initializing the population is applied to all chromosomes.

To establish the remaining parameters, we looked at the experiments performed in [6], where the proposed rollout heuristic was implemented in Matlab and run on Intel i7 processor, 16GB RAM and Windows 7 operating system. In [6], all runs for instances with $N = 8$ were limited to a maximum of 60 s, whereas for all remaining instances a maximum time limit of 300 s was imposed. The genetic algorithm proposed in [5] was implemented in Java and given 60, 300 and 600 s for instances of size 8, 21 and 40 respectively.

In order to take similar or less time than the above values, the stopping condition for each compilation pass was established to 800 consecutive generations without improving the best solution found so far. Finally, the diversification operator is performed every 10 generations without improvement, considering a maximum of 5 mutations per chromosome.

Although we have to be aware that, as pointed in [26], the time complexity can be influenced by multiple factors, including the hardware in which the experiments are run and the software of the implementation in use, such as the operating system, the programming language and/or the compiler/interpreter. A change in any of these factors could significantly alter the performance estimation of the algorithms under comparison, and so making a totally fair comparison is difficult.

[5] https://ti.arc.nasa.gov/m/groups/asr/planning-and-scheduling/VentCirComp17_data.zip

Table 1
Results of DBGA compared with versions without problem decomposition and/or without the diversification procedure.

	DBGA no dec. no div.			DBGA no dec.			DBGA no div.			DBGA		
Instance	Avg.	σ	Time	Avg.	σ	Time	Avg.	σ	Time	Avg.	σ	Time
						$N = 8$ Instances						
1	**37.0**	0.0	3.1	**37.0**	0.0	3.6	**37.0**	0.0	4.1	**37.0**	0.0	4.9
2	**34.6**	0.7	3.5	35.1	1.1	3.9	34.8	0.63	4.1	34.7	1.16	5.2
3	32.2	0.42	3.0	32.2	0.42	3.7	32.2	0.42	4.2	**32.0**	0.0	5.2
4	33.0	0.82	3.3	33.3	0.95	3.8	33.4	0.7	4.2	**32.9**	0.57	5.2
5	**28.0**	0.0	3.2	**28.0**	0.0	4.0	29.2	0.92	4.1	28.2	0.42	5.1
6	**33.6**	0.84	3.2	34.2	1.03	3.9	34.0	0.0	4.3	34.3	0.48	5.0
7	30.9	0.32	3.0	30.6	0.52	3.7	31.0	0.0	4.0	**30.4**	0.52	5.0
8	**33.3**	0.48	3.4	**33.3**	0.48	4.3	34.0	0.0	4.6	33.7	0.48	5.3
9	35.8	0.92	3.9	36.1	0.99	4.5	36.2	1.03	4.6	**35.1**	0.32	5.5
10	35.9	0.74	4.1	35.1	0.32	4.7	**34.0**	0.0	4.7	35.8	1.32	5.7
#best	5			3			2			5		
						$N = 21$ Instances						
1	56.1	4.93	37.2	53.3	2.21	40.4	48.6	2.17	37.4	**47.6**	1.07	53.9
2	51.5	2.55	31.7	52.1	2.33	33.5	50.3	1.57	34.0	**48.7**	1.64	43.8
3	53.3	2.06	42.5	50.3	3.30	47.7	**46.7**	1.42	38.6	46.9	1.52	48.1
4	48.0	3.59	34.0	**45.9**	2.69	35.7	48.1	2.56	37.2	47.6	1.96	54.5
5	56.0	3.86	32.7	56.7	3.53	39.3	52.3	2.00	38.4	**50.9**	2.28	56.1
6	50.7	2.26	32.6	50.6	2.72	34.5	**49.8**	2.78	36.0	51.4	1.96	41.2
7	61.0	2.94	34.4	61.8	5.65	36.6	55.8	2.57	39.6	**55.0**	3.02	47.4
8	52.8	4.76	35.1	48.7	2.83	38.5	**47.4**	0.84	36.4	**47.4**	1.51	45.6
9	57.6	2.01	31.7	54.6	2.37	36.6	**52.9**	2.08	36.0	53.1	2.02	44.3
10	63.1	4.15	37.5	59.4	3.37	40.8	**54.1**	2.73	40.9	55.2	2.15	47.1
#best	0			1			5			5		
						$N = 40$ Instances						
1	84.0	4.42	122.9	73.3	7.76	146.4	68.0	4.37	127.2	**62.3**	3.13	151.7
2	96.3	4.81	151.7	90.9	6.23	203.4	80.3	6.41	148.8	**72.5**	2.72	177.1
3	100.3	7.04	157.7	84.1	9.27	158.4	78.2	6.63	152.3	**70.2**	3.43	170.5
4	106.7	10.98	184.7	95.3	6.17	177.7	86.2	8.73	146.6	**75.4**	6.06	198.4
5	100.9	8.72	149.2	88.7	6.11	172.3	81.1	3.78	143.4	**72.0**	5.44	183.1
6	110.2	8.89	136.6	96.1	10.19	166.5	90.4	6.59	147.3	**78.7**	5.06	189.6
7	100.0	8.60	153.9	91.3	10.58	155.8	80.4	7.49	140.1	**70.9**	4.91	172.1
8	88.2	3.43	146.4	78.9	8.29	152.3	76.0	4.88	124.2	**65.3**	3.47	151.8
9	99.3	7.90	162.8	83.1	9.69	179.1	77.4	5.85	139.8	**67.9**	2.42	178.8
10	102.7	6.15	136.9	86.3	8.51	154.5	84.7	4.08	131.2	**75.8**	6.11	188.0
#best	0			0			0			10		

We mark in **bold** the lowest average result in each instance.

5.3. Analysis of DBGA

Before summarizing the full results from the experimental study, we will visualize some results with regards to the contribution of the components of DBGA to its performance. To analyse the performance of the decomposition strategy and the diversification strategy, we have run three variants of DBGA on the first 10 instances in each subset with $u = 1.0$ and $p = 2$, considering $N = 8$, $N = 21$ and $N = 40$. In the first variant, the problem is not decomposed into p rounds but it is solved at once, in the second one the diversification phase is removed, and in the third one neither decomposition nor diversification are exploited. In either case, the stopping condition was adjusted in order to obtain similar running times. The results of these experiments are reported in Table 1. In particular, we show the average makespan obtained in the 10 runs, which is arguably the most relevant performance measure in order to compare variants, and also the standard deviation and the average time in seconds taken in one run.

We can see that both the diversification operator and the problem decomposition strategy get DBGA to perform better and better with increasing size of the instances, as the average result generally improves while the standard deviation is comparable or even lower. The fact that decomposing the problem produces better results than solving the problem at once suggests that the difficulty of the whole problem is really high.

Regarding the diversification operator, Fig. 8(a) and (b) show the evolution of two executions of DBGA for the instance no. 4 of the subset characterized by $N = 40$, $u = 1.0$ and $p = 2$ with and without using this operation. As we can see, when it is not used, the average and best makespan tend to be quite similar after a number of generations in each round. However, the diversification operator produces clear differences between average and best values, in turn improving the final solution. Notice the large jump in makespan around generation 1600, which is caused by the algorithm switching from the first round to the second.

Figure 8 (c) shows the convergence pattern for the same instance when the whole problem is solved at once, i.e., without decomposition in two steps. In this case, the encoding includes two permutations $ch11$ and $ch12$ of *p-s* gates and two permutations $ch21$ and $ch22$ of pairs of destination qubits for the first and second rounds respectively, and the crossover operates on the permutations of each round. We can observe that the algorithm is unable to converge over the first 200 generations, and that finally it reaches a worse solution than the original DBGA.

Finally, Fig. 8(d) shows the convergence pattern with neither diversification nor decomposition. In this case both the convergence pattern and the final solution reached are the worst of the four versions.

5.4. Comparison to the state of the art: GA and RH

As pointed, the best results so far on the considered instances are those produced by the genetic algorithm (GA) proposed in [5] and the rollout heuristic (RH) proposed in [6].

Tables 2, 3 and 4 summarize the best, average and worst makespan in 10 runs obtained by GA, RH and DBGA on instances with $N = 8$, $N = 21$ and $N = 40$ respectively. Besides, we show the average time taken (in seconds) for each algorithm in one run.

Table 2
Detailed results (in makespan) from GA, RH and DBGA on the instances with $N = 8$. For GA and DBGA, the values marked in **bold** are better or at least equal than the values obtained by RH. The time limit for RH and GA is 60 s.

	GA			RH			DBGA			
Inst.	Best	Worst	Avg.	Best	Worst	Avg.	Best	Worst	Avg.	Time
1	**35**	39	37.6	35	35	35.0	37	37	37.0	5.1
2	**33**	37	**34.7**	36	36	36.0	**33**	35	**34.5**	5.2
3	32	34	32.7	31	31	31.0	32	33	32.1	5.0
4	**32**	33	**32.7**	32	34	33.8	**32**	34	**33.1**	5.2
5	**27**	28	27.6	27	27	27.0	28	29	28.1	5.1
6	**33**	36	**34.1**	35	35	35.0	**34**	35	**34.1**	5.0
7	**31**	32	31.9	31	31	31.0	**30**	31	**30.3**	5.0
8	**32**	33	**32.9**	34	34	34.0	**33**	34	**33.8**	5.3
9	37	41	38.4	35	35	35.0	**35**	35	**35.0**	5.5
10	**37**	40	38.3	38	38	38.0	**34**	37	**34.6**	5.6
11	**36**	40	**37.2**	38	38	38.0	39	40	39.2	5.7
12	36	38	37.0	33	33	33.0	34	37	34.8	5.3
13	**30**	33	**31.8**	32	32	32.0	**32**	33	32.1	5.4
14	**31**	32	**31.9**	32	32	32.0	**32**	32	**32.0**	4.9
15	**33**	36	**34.3**	35	36	35.5	**34**	34	**34.0**	5.5
16	**31**	34	32.1	32	32	32.0	**32**	32	**32.0**	4.8
17	**36**	38	37.3	36	36	36.0	**32**	32	**32.0**	5.2
18	**29**	30	29.9	29	30	29.2	**29**	31	30.4	5.1
19	**31**	33	**32.0**	32	32	32.0	**32**	32	**32.0**	5.2
20	**31**	32	31.9	31	32	31.1	**30**	30	**30.0**	5.3
21	**25**	27	**26.4**	27	27	27.0	**26**	28	**27.0**	5.0
22	44	45	44.8	39	39	39.0	40	41	40.9	5.3
23	36	37	36.6	35	37	36.0	36	37	36.9	5.5
24	33	35	34.6	32	32	32.0	33	35	33.8	5.9
25	**37**	40	38.5	38	39	38.3	**37**	38	**37.3**	5.7
26	**26**	30	**28.3**	29	29	29.0	**29**	29	**29.0**	5.2
27	35	39	36.6	34	34	34.0	36	37	36.3	5.6
28	**30**	32	**30.6**	32	32	32.0	**30**	30	**30.0**	5.1
29	37	38	37.7	35	37	35.1	37	37	37.0	5.6
30	32	33	32.9	31	32	31.2	**30**	31	**30.9**	5.8
31	33	36	33.6	32	32	32.0	**32**	34	32.3	5.6
32	**35**	40	37.3	35	37	35.9	**35**	38	**35.5**	5.5
33	**41**	44	42.9	42	42	42.0	**42**	45	43.8	5.7
34	**34**	37	36.2	35	35	35.0	**34**	34	**34.0**	5.5
35	**34**	39	**36.1**	38	38	38.0	39	39	39.0	5.5
36	**27**	29	28.7	28	29	28.2	30	32	31.2	5.7
37	**34**	37	35.7	35	35	35.0	**35**	36	35.6	5.4
38	**28**	30	**29.1**	29	30	29.9	**28**	30	**28.8**	5.3
39	**29**	32	31.3	30	30	30.0	**28**	32	30.6	5.5
40	44	45	44.5	37	38	37.1	39	39	39.0	5.5
41	**33**	40	37.2	35	35	35.0	36	39	36.7	5.9
42	**31**	33	**32.2**	33	34	33.1	33	33	33.0	5.4
43	33	35	34.1	32	34	33.1	**32**	32	**32.0**	5.1
44	**39**	42	40.8	39	41	39.9	40	43	42.2	6.0
45	**36**	39	**37.3**	38	38	38.0	**36**	37	**36.9**	5.1
46	**32**	35	**33.5**	34	34	34.0	**34**	34	**34.0**	5.3
47	41	43	42.0	38	39	38.8	**37**	38	**37.8**	5.2
48	**32**	33	**32.9**	33	33	33.0	**33**	33	**33.0**	5.0
49	37	40	38.5	36	37	36.1	38	40	38.4	5.5
50	31	33	32.2	30	32	31.3	31	33	31.8	5.2
#bold	35		18				32		26	

At a first glance, we can observe that the three methods produce similar makespan values on the instances with $N = 8$, and that RH and DBGA are better than GA in the remaining instances. Besides, DBGA seems comparable to RH on the instances with $N = 21$.

Regarding the largest instances ($N = 40$), Table 4 shows that DBGA running with the parameters given in Section 5.2 (last columns in the table) produces the best results. In this case, DBGA established new best solutions for 48 out of the 50 instances of the set and reached the best known solution for one more instance. However, given the differences in the implementations (RH is implemented in Matlab and DBGA is implemented in C++) the time taken by DBGA may not be comparable to the time limit given to RH. Therefore, to establish a fairer comparison between the methods, we performed new experiments on these instances giving DBGA about 10% of the time given to RH in [6]. To do that, we reduced the population size to 800 chromosomes and the number of consecutive generations without improving the best solution to 200. The results of these experiments are reported in the columns labeled “DBGA (less running time)” of Table 4; even though they are slightly worse than before, we can see that they are still better than the results from RH.

We have performed some statistical tests to better analyze the differences between DBGA and RH (as GA clearly obtains overall worse results than those other methods). As we have multiple-problem analysis, we used non-parametric statistical tests, as suggested in [26]. First, we ran a Shapiro-Wilk test to confirm the non-normality of the data. Then we used paired Wilcoxon signed rank tests to compare DBGA and RH on each set of instances, depending on the chip size. In these tests, the level of confidence used was 95% and the alternative hypothesis was “the difference between DBGA and RH is lower than 0”. The p-value obtained with this test was 0.0000004929 for $N = 40$ (when giving DBGA its full

Table 3
Detailed results (in makespan) from GA, RH and DBGA on the instances with $N = 21$. For GA and DBGA, the values marked in **bold** are better or at least equal than the values obtained by RH. The time limit for RH and GA is 300 s.

	GA			RH			DBGA			
Inst.	Best	Worst	Avg.	Best	Worst	Avg.	Best	Worst	Avg.	Time
1	51	59	55.5	49	53	51.2	**46**	50	**47.6**	53.9
2	53	60	56.9	50	55	52.9	**47**	52	**48.7**	43.8
3	47	55	50.0	42	48	45.7	44	49	46.9	48.1
4	48	50	49.0	44	48	45.6	45	51	47.6	54.5
5	**52**	60	56.6	52	54	53.0	**47**	54	**50.9**	56.1
6	51	58	53.9	50	55	52.9	**48**	54	**51.4**	41.2
7	58	65	61.5	55	57	56.5	**50**	60	**55.0**	47.4
8	50	59	54.4	49	52	50.4	**45**	51	**47.4**	45.6
9	58	64	60.3	54	56	54.4	**51**	56	**53.1**	44.3
10	56	64	59.9	54	60	56.7	**52**	60	**55.2**	47.1
11	**46**	53	49.9	47	51	48.9	**43**	51	**47.1**	40.5
12	60	67	62.2	56	62	58.7	**53**	57	**55.1**	38.9
13	46	49	47.4	43	45	44.4	**42**	50	**44.0**	39.7
14	48	56	52.1	46	48	46.2	**46**	57	51.4	41.9
15	48	54	50.9	46	48	47.4	**43**	46	**44.3**	37.3
16	63	71	66.1	57	64	60.3	**52**	58	**55.2**	41.3
17	57	62	59.3	50	52	50.7	52	56	53.3	44.3
18	57	64	**58.9**	54	61	59.4	**50**	55	**52.4**	39.0
19	57	64	61.1	56	62	59.3	**49**	57	**53.1**	42.3
20	55	63	59.4	50	53	51.8	51	58	53.3	40.3
21	54	61	58.4	51	54	51.7	52	58	55.0	48.7
22	55	62	58.7	54	55	54.5	**51**	57	**53.5**	41.6
23	54	59	56.1	48	51	50.1	49	56	52.3	41.1
24	**50**	60	56.0	50	55	53.7	**49**	57	**52.9**	45.2
25	53	60	56.4	50	53	50.7	51	59	55.7	43.7
26	52	56	54.0	46	52	47.8	**43**	50	**45.9**	41.4
27	63	73	67.7	61	66	62.7	**56**	68	64.3	44.2
28	**46**	55	51.8	47	51	49.8	**46**	57	50.1	46.6
29	49	60	54.2	47	50	48.6	**47**	55	50.0	40.2
30	54	65	57.9	53	56	55.0	**51**	57	**54.2**	42.3
31	57	64	60.9	52	57	55.5	55	61	58.4	42.8
32	**51**	60	54.4	52	52	52.0	**47**	50	**48.4**	38.8
33	**52**	62	56.9	52	57	55.8	**49**	56	**50.7**	41.0
34	57	61	58.2	51	56	53.7	53	56	54.2	39.2
35	48	56	52.0	45	51	48.6	**45**	52	**46.9**	40.5
36	51	60	56.1	49	55	52.3	**49**	58	53.2	42.6
37	56	64	59.4	51	54	52.9	52	56	53.0	38.3
38	57	65	61.4	53	55	53.6	**53**	57	55.8	43.0
39	55	65	60.1	50	55	52.1	**50**	64	58.0	38.4
40	54	61	57.5	48	57	52.9	50	55	**51.9**	39.2
41	52	56	54.4	49	53	50.9	**47**	49	**48.6**	36.3
42	52	60	55.5	50	53	51.0	**49**	54	51.5	39.8
43	**45**	56	52.0	47	51	48.5	50	54	51.5	43.3
44	50	56	53.2	47	52	49.4	**47**	50	**48.9**	39.0
45	44	53	48.8	40	47	43.9	44	49	46.3	42.1
46	**42**	47	45.0	42	46	44.3	43	48	45.2	41.6
47	53	57	54.8	52	53	52.1	**51**	55	53.8	41.8
48	47	56	51.1	43	47	46.4	45	53	49.1	40.7
49	60	67	62.6	54	58	57.3	**54**	62	57.4	46.3
50	**53**	61	56.7	53	57	54.7	**50**	56	**52.7**	42.7
#bold	9		1				35		26	

running time), or 0.004589 (with reduced run time), showing that there exist statistically significant differences in the average makespan between DBGA and RH, in both short and long runs of DBGA. The *p*-value for $N = 21$ instances was 0.242 (0.761 if we consider the opposite alternative hypotheses), and so in these intermediate instances there are not statistically significant differences between the results of both methods. Finally, in instances with $N = 8$ the *p*-value is 0.9034 (0.0988 if we consider the opposite alternative hypotheses), and so in these small instances the differences are again not statistically significant. In conclusion, in large instances DBGA is significantly better than the previous state-of-the-art methods, whereas in small and intermediate instances it is not worse. Moreover, we argue that the results in larger instances are more relevant, as the size of the future quantum hardware will be progressively larger.

The detailed schedules of the best solutions found by DBGA for all instances considered in this experimental study are openly available on the web[6]

As an example, Fig. 9 shows the Gantt chart of the best schedule obtained by DBGA to the instance no. 2 of the subset characterized by $N = 8$, $u = 1.0$ and $p = 2$. This schedule has a makespan of 33, which is lower than the makespan of 36 reported in [6] for the same instance.

Regarding all previous experiments, it can be argued that the obtained makespans are not realistic, as typically a *p-s* gate requires at least 2 native gates and *swap* requires at least 3 native gates, whereas in previous experiments we assumed $\tau_{swap} = 2$ and $\tau_{p-s} = 3$ or 4. For this

[6] Repository section in http://di002.edv.uniovi.es/iscop

Table 4
Detailed results (in makespan) from GA, RH and DBGA on the instances with $N = 40$. For GA and DBGA, the values marked in **bold** are better or at least equal than the values obtained by RH. The time limit for RH is 300 s and for GA is 600 s.

	GA			RH			DBGA (less running time)				DBGA			
Inst.	Best	Worst	Avg.	Best	Worst	Avg.	Best	Worst	Avg.	Time	Best	Worst	Avg.	Time
1	77	87	81.4	65	70	69.3	**61**	69	**64.7**	31.3	**58**	68	**62.3**	151.7
2	92	107	98.0	74	77	75.9	**67**	78	**73.2**	35.6	**69**	77	**72.5**	177.1
3	83	98	89.2	71	77	74.5	**68**	84	**74.5**	32.7	**65**	75	**70.2**	170.5
4	87	104	93.9	74	82	77.3	**71**	97	80.3	42.0	**68**	86	**75.4**	198.4
5	84	99	93.6	78	78	78.0	**67**	89	**77.9**	37.1	**66**	82	**72.0**	183.1
6	87	103	96.3	81	83	82.1	**76**	89	**81.5**	37.8	**72**	86	**78.7**	189.6
7	86	98	93.4	79	83	80.8	**63**	84	**73.7**	33.8	**63**	80	**70.9**	172.1
8	74	90	83.0	68	77	73.7	**63**	80	**70.3**	30.3	**61**	73	**65.3**	151.8
9	77	95	89.5	66	67	66.9	67	81	73.9	34.7	**64**	72	67.9	178.8
10	88	101	95.8	80	80	80.0	**69**	88	**79.4**	36.4	**69**	86	**75.8**	188.0
11	80	99	84.5	68	72	70.5	**64**	79	70.9	34.5	**67**	81	72.7	155.7
12	85	98	90.9	74	77	75.6	**70**	81	75.9	34.6	**70**	89	78.3	162.7
13	75	88	81.3	62	65	64.5	65	72	67.1	29.9	**59**	74	65.5	131.9
14	89	98	93.1	74	84	78.9	**72**	86	**78.9**	32.4	**68**	79	**73.7**	139.0
15	84	96	89.3	78	78	78.0	**69**	84	**75.4**	31.2	**67**	75	**72.0**	165.7
16	82	94	87.4	77	78	77.9	**68**	81	**74.6**	35.6	**65**	84	**72.5**	164.1
17	87	96	91.5	78	78	78.0	**67**	79	**72.1**	34.2	**67**	82	**73.3**	151.8
18	90	108	99.3	79	82	80.4	**73**	92	81.6	32.7	**72**	84	**78.5**	155.2
19	**70**	85	81.2	70	71	70.9	**62**	73	**66.3**	31.6	**61**	70	**65.1**	130.8
20	87	105	97.9	78	85	81.5	**68**	104	83.2	38.0	**73**	93	82.0	197.7
21	79	97	86.9	77	81	79.9	**69**	86	**76.6**	37.2	**68**	81	**73.8**	172.4
22	79	98	90.0	76	81	78.3	**64**	85	**75.1**	35.2	**67**	80	**74.6**	160.1
23	74	86	79.8	63	67	65.0	66	79	74.5	34.8	66	82	72.3	156.1
24	83	92	88.1	80	80	80.0	**61**	71	**66.6**	31.6	**63**	73	**67.0**	157.0
25	75	95	86.5	71	75	74.7	**71**	86	76.5	33.6	**64**	76	**71.6**	146.0
26	90	100	94.3	81	84	82.1	**67**	79	**72.4**	31.3	**63**	78	**69.6**	167.8
27	89	100	93.0	81	81	81.0	**72**	90	**79.7**	30.5	**72**	86	**75.8**	134.1
28	93	105	100.3	88	88	88.0	**68**	86	**76.2**	36.4	**70**	83	**76.5**	165.0
29	**74**	91	84.0	77	77	77.0	**62**	71	**66.8**	30.1	**64**	72	**66.9**	137.2
30	79	92	85.1	72	75	73.4	**64**	77	**70.6**	31.6	**60**	73	**67.3**	141.3
31	81	98	92.0	69	74	71.7	71	85	75.2	37.2	**66**	80	72.7	149.3
32	66	78	73.9	62	68	65.5	**56**	64	**60.7**	28.9	**53**	62	**58.4**	132.1
33	84	92	89.8	73	75	73.8	**64**	73	**68.4**	29.5	**60**	75	**65.6**	137.5
34	79	93	86.1	73	75	70.8	**66**	82	74.6	35.6	**60**	75	**68.2**	147.4
35	75	90	85.5	70	74	71.3	**66**	81	72.5	28.9	**63**	77	**70.9**	149.1
36	95	107	101.0	80	88	86.5	**74**	83	**79.4**	33.8	**73**	83	**76.7**	162.6
37	82	96	89.8	73	77	74.7	**69**	86	77.3	36.5	**70**	79	75.5	152.8
38	**72**	88	83.8	72	77	73.9	**65**	71	**67.9**	29.6	**62**	72	**66.5**	142.7
39	**82**	103	93.8	82	82	82.0	**69**	79	**73.4**	34.0	**66**	81	**74.9**	158.4
40	79	92	87.1	69	76	72.6	**67**	80	73	31.2	**64**	81	**71.7**	152.4
41	98	105	101.5	76	76	76.0	**75**	91	82	30.8	**68**	84	**74.8**	138.0
42	81	91	85.9	65	68	67.1	66	74	69.4	33.6	**61**	83	70.9	154.5
43	81	96	89.1	72	75	73.6	**63**	76	**70.5**	31.4	**63**	77	**68.1**	142.1
44	78	97	90.3	68	82	76.0	**65**	85	**75.9**	32.1	**65**	82	**75.6**	166.3
45	89	101	95.9	69	81	73.4	72	84	77.6	31.1	**69**	85	75.4	153.6
46	91	99	95.1	78	78	78.0	**68**	82	**73.6**	32.5	**65**	75	**70.0**	154.3
47	89	105	99.2	78	82	78.3	**76**	83	78.5	31.4	**69**	79	**74.1**	147.8
48	**69**	91	83.9	75	77	76.5	**66**	82	**72.1**	34.1	**63**	75	**69.6**	149.5
49	89	101	93.3	73	83	80.1	**70**	88	**76.9**	30.0	**71**	88	**77.5**	145.2
50	78	95	87.6	74	85	76.1	**66**	78	**70**	31.9	**64**	79	**69.3**	142.0
#bold	5		0				44		32		49		40	

reason, we have redone the experiments with DBGA using $\tau_{swap} = 3$ in order to see the differences in makespan. Table 5 shows the obtained makespans, and we do not show the computational times for clarity, as they are similar to those reported in Tables 2, 3 and 4. As expected, the best makespan reached is worse in 149 of 150 instances (being the same in the other instance). It is also remarkable that the best makespan obtained is in average 10.6% worse in $N = 8$ instances, 17.7% worse in $N = 21$ instances, and 23.7% worse in $N = 40$ instances. This makes sense, as bigger quantum architectures usually require more swaps, as qstates travel longer distances, and so increasing τ_{swap} from 2 to 3 produces the most difference.

5.5. Comparison to the state of the art: CFH

In this section we compare the performance of the proposed DBGA algorithm with the performance obtained by several configurations of the methods proposed in [21] and [22], denoted CFH (Compilation Flow Heuristics) in the following. Table 6 shows the results obtained by using three different sets of 5 adapted instances (a total of 15 instances) taken respectively from the three benchmark sets introduced in Section 3 and characterized by the values $N = 8, 21, 40$, $u = 1.0$, and $p = 2$. It should be noted that, as opposed to our approach which is based on an abstract representation of the quantum gates (whose durations are generic and not related to any particular quantum architecture), the approach used in [21,22] is built on top of the Qiskit[7] framework and therefore follows a specific gate model in which the used gates (*p-s, swap*, and *mix*) are expressed in terms of sequences of native gates made available by the Qiskit backend simulator, where each native gate has a conventional unit time duration. In particular, all *p-s* and *swap* gates are ex-

[7] https://qiskit.org/

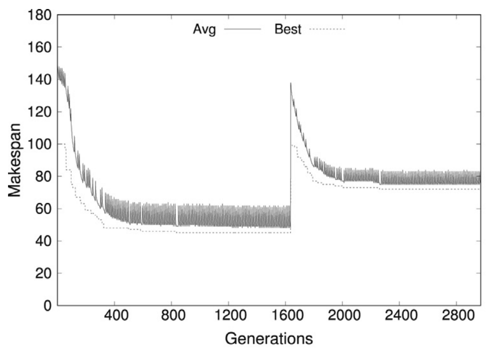
Table 5
Detailed results (in makespan) of DBGA when using $\tau_{swap} = 3$ instead of $\tau_{swap} = 2$.

	$N = 8$ Instances			$N = 21$ Instances			$N = 40$ Instances		
Inst.	Best	Worst	Avg.	Best	Worst	Avg.	Best	Worst	Avg.
1	39	39	39.0	53	60	56.1	74	85	79.7
2	37	39	38.0	52	62	57.0	83	94	88.8
3	36	37	36.7	51	60	56.3	83	101	91.3
4	35	36	35.5	55	61	57.3	87	115	99.6
5	33	33	33.0	59	68	62.5	81	99	88.4
6	38	38	38.0	55	61	57.9	87	107	99.6
7	33	33	33.0	61	71	65.6	85	107	94.8
8	37	37	37.0	55	61	57.1	79	92	84.3
9	41	41	41.0	58	68	62.9	78	108	89.9
10	37	37	37.0	60	68	63.7	86	111	98.0
11	44	45	44.5	51	57	55.1	76	102	85.4
12	39	40	39.9	61	68	64.1	87	109	94.0
13	36	37	36.4	50	54	51.1	72	90	78.1
14	36	37	36.2	55	69	59.9	85	99	92.0
15	36	36	36.0	50	53	51.4	82	97	89.0
16	36	36	36.0	63	68	65.9	82	99	90.8
17	33	33	33.0	60	62	60.8	78	108	89.1
18	33	34	33.8	58	67	61.5	85	98	92.3
19	35	36	35.1	59	73	65.4	70	92	77.4
20	32	33	32.2	58	67	62.8	90	111	100.4
21	30	30	30.0	55	69	63.3	81	102	91.5
22	46	47	46.8	61	65	62.9	83	92	87.8
23	40	43	40.3	59	64	62.0	86	96	91.3
24	34	37	34.9	57	66	61.2	74	89	80.3
25	39	41	40.2	60	67	64.0	76	99	89.0
26	31	31	31.0	53	60	56.0	79	89	85.5
27	38	39	38.6	72	80	74.8	85	94	89.5
28	32	32	32.0	56	63	58.2	90	102	94.7
29	37	39	37.2	55	62	59.1	72	90	81.3
30	33	35	34.2	57	68	63.5	79	89	83.0
31	34	37	35.2	62	72	68.6	87	107	94.5
32	40	43	40.5	55	61	57.7	68	83	72.9
33	45	47	46.5	54	67	59.4	77	92	83.6
34	39	40	39.3	62	66	64.1	84	94	89.2
35	42	42	42.0	52	60	56.5	78	96	85.3
36	34	35	34.7	59	68	63.6	90	106	97.2
37	39	39	39.0	60	65	62.0	83	97	90.8
38	32	33	32.5	61	68	64.6	78	94	81.9
39	32	37	33.7	62	74	69.1	86	95	90.1
40	44	45	44.9	59	65	60.0	81	96	88.5
41	39	43	40.8	53	57	54.2	83	102	91.8
42	37	37	37.0	56	63	59.4	76	90	82.3
43	37	37	37.0	59	65	60.9	76	103	86.9
44	43	44	43.9	58	60	58.5	82	101	91.2
45	39	40	39.2	51	56	52.9	82	102	92.7
46	40	40	40.0	49	56	52.7	77	94	87.3
47	41	41	41.0	61	66	63.2	84	102	91.7
48	37	37	37.0	53	60	56.2	79	100	87.7
49	42	43	42.9	68	79	72.2	85	110	93.1
50	35	37	35.3	61	65	62.3	77	88	83.6

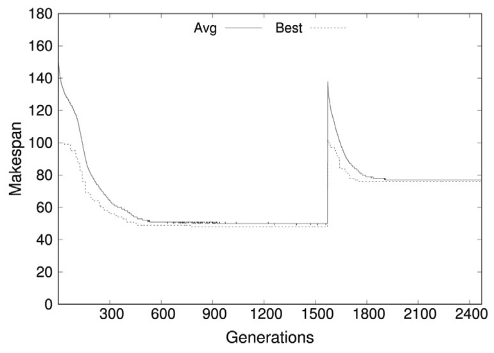
pressed in terms of 3 native gates (and hence their duration is equal to 3), while all *mix* gates are expressed in terms of one native gate. Therefore, in order to allow for a fair comparison with CFH (the code is available at https://github.com/mahabubul-alam/QAOA-Compiler), we had to make some modifications to the instances used for the comparison, adapting the gate durations used in our model to the ones used in Qiskit and described above. By allowing for the previous modifications on our side, the value of the circuit's depth coincides with the circuit's makespan, and the obtained results can be fairly compared.

Moreover, given that our approach does not entail any initial mapping decision process during the compilation (i.e., the i-th qstate is supposed to rest on the i-th qubit at the beginning) we decided to introduce an additional initial mapping option in the CFH's code that allows for the same initial conditions. We called this new option *dummy* initialization. Table 6 compares DBGA with the six different solving strategies proposed in [21,22]. In particular, we consider two different initial mapping methods: (i) *dummy*, which sets each qubit $i = 1 \ldots N$ to the corresponding qstate i, and (ii) QAIM, which applies an heuristic procedure described in [22]. All benchmark instances have been solved using the following solving strategies: IP, IC and IterC.

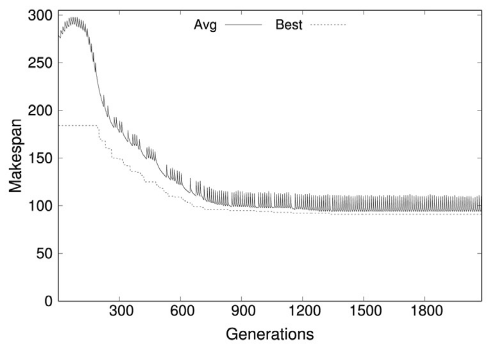
In order to make the two solving approaches comparable with regard the CPU time, we have considered the most CPU intensive solving configuration, that is the solving method IterC and QAIM as initial mapping, and we have empirically determined the following minimal integer upper bounds in the CPU time: 2 s for $N = 8$ instances, 5 s for $N = 21$ instances and 20 s for $N = 40$ instances. To achieve similar CPU times, we set the population size of DBGA to 400 chromosomes and the stopping condition at 200 generations (for each compilation pass) without improving the best solution found so far. The results with this configuration are shown in columns "DBGA (less running time)". Additionally, results with the standard set of parameters described in Section 5.2 take longer computational time and are shown in columns "DBGA".



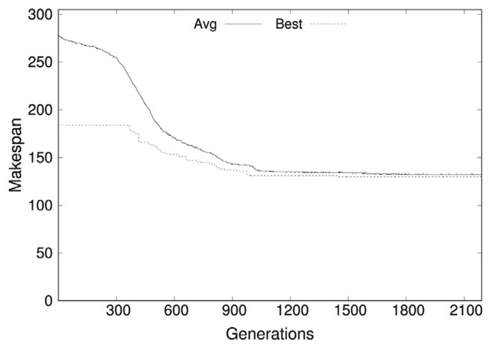
(a) DBGA.



(b) DBGA without the diversification operator.



(c) DBGA without decomposing the problem in two rounds.



(d) DBGA without the diversification operator and without decomposing the problem in two rounds.

Fig. 8. Convergence graphs of different versions of the genetic algorithm.

```
Require: A QCCP problem instance $P$ with $p$ rounds. The quantum hardware $QM$. A chromosome $(sg_{r-1}, ch1, ch2)$. $sg_{r-1}$ is a solution graph for the subproblem $1, \dots, r-1$, $ch1$ is a permutation of the p-s gates in round $r$. $ch2$ is a chain of connections in $QM$ of the same length as $ch1$.
Ensure: A solution graph $sg_r$ for subproblem $1, \dots, r$ on circuit $QM$
  $sg_r \leftarrow sg_{r-1}$
  for each p-s$(q_i, q_j)$ in $ch1$ (and $\{n_k, n_l\}$ in $ch2$) from left to right do
    Let $path_i \leftarrow n(q_i) \rightsquigarrow d(q_i)$ and $path_j \leftarrow n(q_j) \rightsquigarrow d(q_j)$ be a pair of minimal paths from $\{n(q_i), n(q_j)\}$ to $\{n_k, n_l\}$ in $QM$;
    while $\{n(q_i) \neq d(q_i)\}$ or $\{n(q_j) \neq d(q_j)\}$ do
      $(n, n') \leftarrow (n(q_i), succ(n(q_i)))$ or $(n(q_j), succ(n(q_j)))$ if possible such that $\{n, n'\} \neq \{n(q_i), n(q_j)\}$;
      if $\{n, n'\} \neq \{n(q_i), n(q_j)\}$ then
        insert a swap gate on qubits $\{n, n'\}$ and update $sg_r$;
        $n \leftarrow n'$; // advance in $path_1$ or in $path_2$
      else
        swap in $path_1$ and $path_2$ the subpaths from the current qubits $n$ and $n'$ to their destination qubits, so that the new paths become $n, n' \rightsquigarrow n_b$ and $n' \rightsquigarrow n_a$ if the old paths were $n, n' \rightsquigarrow n_a$ and $n' \rightsquigarrow n_b$ respectively;
      end if
    end while
    insert a p-s gate on qubits $\{n_k, n_l\}$ (where qstates $\{q_i, q_j\}$ are now hold) and update $sg_r$;
  end for
  insert a mix gate on each qubit holding a qstate;
  return The solution graph $sg_r$;
```

Algorithm 1 : Decoding algorithm. Given a chromosome in round r, it produces a solution graph for the subproblem defined by rounds $1, \dots, r$. $succ(n(q_k))$ denotes the successor of qubit $n(q_k)$ in $path_k$.

Inspecting Table 6, we observe that within the imposed CPU times, the makespans[8] produced by the DBGA algorithm are always better than those obtained by CFH (even those of the "less running time" configuration); the greater the size of the benchmark sets, the larger the improvement obtained by DBGA. In our opinion the main reasons for the DGBA advantage are the following: (i) all the solving strategies proposed in [21] and [22] are heuristic *single-pass* strategies, whereas DBGA adopts a single-pass strategy at decoding level and uses a genetic-based optimization strategy (described in Fig. 4) to effectively explore the search space and find better quality solutions; (ii) the methodology proposed in [21,22] follows a loosely coupled approach, in which the *p-s* gate scheduling (in terms of assignment to the different layers) and the *swap* gate synthesis and insertion are performed in two different and separate steps, whereas in other approaches proposed in the literature (e.g., [4,5]) the previous two solving phases are inherently interleaved (and hence, one can take advantage of the decisions made by the other at all times during the solving process).

Last but not least, an element that may significantly affect the duration of the compiled circuit is the number of *swap* gates. We have observed the best solutions obtained by CFH (in terms of number of *swap* gates), compared with our DBGA solutions (considering the "less run-

[8] In the case of the algorithms described in [21] and [22] in order to calculate comparable makespan values with DBGA, we have generated a set of readable instances for the code available at https://github.com/mahabubul-alam/QAOA-Compiler. For each obtained solution represented as set of native gate for the used Qiskit backend simulator, we have subtracted the value 2 to the produced depth value; indeed, the depth represents the longest sequence of native gates in the output solution (each one has a conventional unitary duration) and the value 2 corresponds to the duration of the circuit's first and last layer, where the first layer is used to realize the initial mapping and the last layer is used to perform the measurement operations. In fact, such devices are not considered in the model proposed in Section 3.

Table 6
Results of DBGA compared with those obtained by several configurations of the CFH proposed in [21] and [22].

	CFH + Dummy			CFH + QAIM			DBGA (less running time)				DBGA			
Instance	IP	IC	IterC	IP	IC	IterC	Best	Worst	Avg.	Time	Best	Worst	Avg.	Time
							$N = 8$ Instances							
1	51	44	41	62	47	53	**32**	32	32.0	0.5	**32**	32	32.0	4.7
2	61	41	54	62	44	50	**32**	35	34.1	0.6	**32**	35	32.6	5.5
3	43	38	42	52	56	42	**32**	32	32.0	0.5	**32**	32	32.0	5.2
4	76	50	54	67	47	55	**38**	38	38.0	0.6	**38**	38	38.0	5.4
5	53	32	42	49	38	33	**29**	32	30.7	0.6	**29**	31	29.2	5.2
							$N = 21$ Instances							
1	126	152	112	143	125	101	53	59	55.4	3.0	**50**	59	54.0	22.5
2	128	152	102	123	149	113	**53**	59	54.5	3.2	**53**	56	53.3	20.4
3	138	107	105	103	125	103	56	74	60.5	3.8	**50**	62	56.7	27.1
4	108	119	110	116	116	110	**53**	62	57.3	3.1	**53**	56	54.2	23.9
5	120	104	100	157	128	101	**50**	62	55.9	3.0	**50**	59	54.8	21.4
							$N = 40$ Instances							
1	184	152	140	209	149	159	65	86	77.4	12.6	**62**	83	73.2	71.4
2	227	185	171	214	176	186	83	110	95.8	15.9	**76**	91	83.5	94.1
3	182	164	178	219	188	163	77	94	88.9	15.1	**74**	85	80.3	80.6
4	218	194	193	195	125	180	83	107	92.9	17.4	**76**	101	86.7	92.1
5	226	221	186	219	200	187	91	119	98.4	14.9	**77**	92	85.3	83.6

We mark in **bold** the best result in each instance.

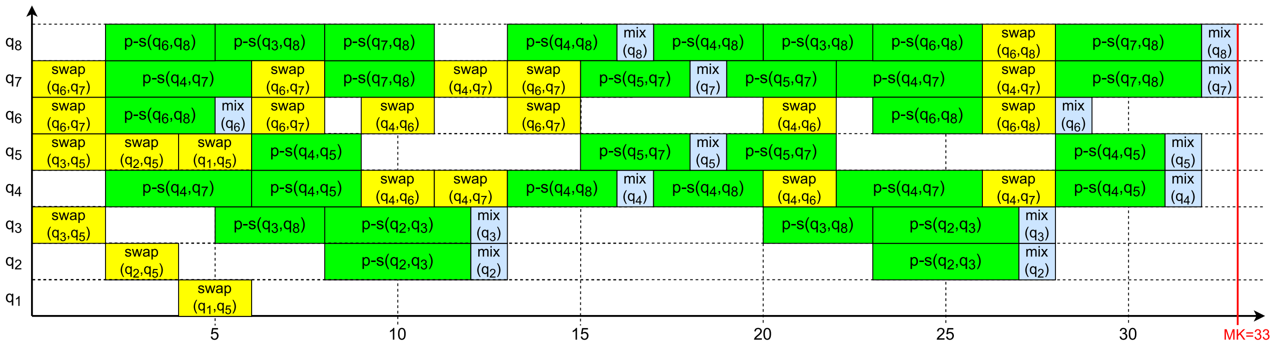


Fig. 9. Gantt representation of the best solution to the instance no. 2 of the subset characterized by $N = 8$, $u = 1.0$ and $p = 2$.

ning time" configuration). As an example, the values for the five $N = 40$ instances are as follows:

- Instance 1: 191 Vs. 110 (+ 73.6%)
- Instance 2: 223 Vs. 155 (+ 43.8%)
- Instance 3: 205 Vs. 165 (+ 24.2%)
- Instance 4: 181 Vs. 174 (+ 4.0%)
- Instance 5: 223 Vs. 174 (+ 28.1%)

From the previous values it is possible to conjecture that one of the reasons that explain the good results obtained with DBGA is the relatively lower number of *swap* gates compared with solutions from CFH.

6. Conclusions and future work

We have seen that Constraint Optimization is a suitable framework to formulate the Quantum Circuit Compilation Problem (QCCP); this problem may be naturally defined as a scheduling problem where qstates represent resources and gates are operations that require the exclusive use of one or more qstates to be performed. We focused in the class of Quantum Approximate Optimization Algorithms (QAOA), in which the same set of quantum gates must be applied for a number of rounds. This feature allowed us to develop a Decomposition Based Genetic Algorithms (DBGA) that proved to be quite competitive with the state of the art. To do that, we had to devise a suitable encoding/decoding mechanism, which was not an easy task. Due to the extreme difficulty of the problem, a coding scheme that may represent any problem solution becomes impractical. So, we opted by a simplified encoding that fixes the qubits where the qstates of a (binary) gate must be for the gate to be executed. This raised the issue of how to move two qstates from a pair of qubits to another pair of destination qubits. This may be done in many ways and we opted here to design an algorithm that minimizes the moves, i.e., the number of *swap* gates required, which may not be a globally optimal strategy. In spite of that, the proposed encoding/decoding schema proved to be really good. This is justified by the fact that it was able to outperform some of the best methods in the state of the art.

In our opinion, the key point of the proposed DBGA is that it exploits the fact that the QAOA is composed by a sequence of rounds, in each of which the same set of gates must be scheduled. Therefore, the algorithm iterates on the number of rounds and, in each one, it searches for a schedule of the gates in the current round. Of course, this is a local optimal strategy that do not necessarily converge to a global optimum. However it empirically showed to be very efficient, and it is expected that it will scale up better in the number of rounds than other strategies dealing with the whole problem at once.

This work leaves open some lines for future research. For example, more expressive coding schemas and new decoding algorithms and genetic operators could help to better explore the search space. Besides, devising neighbourhood structures to improve the solutions built by the decoding algorithm could help to reach new and hopefully better solutions that are not reachable with the current enconding/decoding schema. These structures may then be exploited in Local Search (LS) algorithms that may in turn be combined with the GA into a Memetic Algorithm [27–29]. Memetic Algorithms demonstrated to be good in solving a variety of extremely hard combinatorial problems as, for example, the travelling salesman problem [30], or a number of scheduling problems [31–35], and so they are expected to be efficient in solving the QCCP as well.

It will be also interesting to consider the application of QAOA to problems other than MaxCut, as for example to the Graph Coloring Problem, which was already considered in [36].

However, the most interesting line for future work is probably to consider the mentioned extensions of the problem, initially proposed in [8], and also considered in [5]. These extensions are: 1) variable initialization of qstates (QCCP-I) and 2) crosstalk constraints (QCCP-X). The variable initialization of qstates makes the problem harder to solve, as the initial positions of qubits would be additional decision variables. On the other hand, the crosstalk constraint forbids the concurrent execution of two gates on neighbouring qubits. The motivation is to avoid possible interferences that may be produced between qubits and so to devise more robust solutions.

Finally, it is worth to remark that quantum computing technologies are quickly evolving, and so an essential line of future work is to adapt the solving methods to the newly developed hardware architectures and emerging technologies.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Lis Arufe: Conceptualization, Software, Validation, Data curation. **Miguel A. González:** Methodology, Formal analysis, Writing – original draft. **Angelo Oddi:** Conceptualization, Methodology, Writing – review & editing. **Riccardo Rasconi:** Conceptualization, Methodology, Writing – review & editing. **Ramiro Varela:** Supervision, Methodology, Writing – original draft.

Acknowledgements

This research has been supported by the Spanish Government under research projects TIN2016-79190-R and PID2019-106263RB-I00/AEI/10.13039/501100011033, and by the Principality of Asturias under grant FC-GRUPIN-IDI/2018/000176. A. Oddi and R. Rasconi were supported by the ESA Contract No. 4000112300/14/D/MRP “Mars Express Data Planning Tool MEXAR2 Maintenance”.

Appendix A. Quantum approximate optimization algorithm

QAOA as proposed in [7] leverages quantum-gate based computing to solve combinatorial optimization problems expressed as:

$$\text{maximize:} \sum_{\alpha=1}^{m} C_\alpha(\mathbf{z}) \tag{A.1}$$

where $C_\alpha(\mathbf{z})$ are clauses on a vector of decision binary variables $\mathbf{z} = (z_1, \dots, z_n)$. So, the goal is to find the assignment of $z_i \in \{0,1\}, 1 \le i \le n$, that maximizes the number of clauses that are satisfied.

To apply QAOA to solve this problem, the user has to translate the clauses $C_\alpha(z)$ into equivalent quantum Hamiltonians $\mathbf{C}_\alpha$, by promoting each variable z_i to a quantum spin, i.e., a qubit, and then select a number of rounds p and two angles $0 \le \beta \le \pi$ and $0 \le \gamma \le 2\pi$ for each round. Then, starting from the n qubits in a qstate

$$|+\rangle = \frac{1}{\sqrt{2}}|0\rangle + \frac{1}{\sqrt{2}}|1\rangle, \tag{A.2}$$

the following problem Hamiltonian

$$H_C = \sum_{\alpha=1}^{m} \mathbf{C}_\alpha \tag{A.3}$$

and the mixing Hamiltonian

$$H_B = \sum_{j=1}^{n} \mathbf{X}_\mathbf{j} \tag{A.4}$$

where

$$\mathbf{X}_\mathbf{j} = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} \tag{A.5}$$

is the standard Pauli matrix $\mathbf{X}$, are applied alternatively over the p rounds to generate the final state

$$|\psi_p(\vec{\gamma},\vec{\beta})\rangle = \prod_{r=1}^{p} e^{-i\beta_r H_B} e^{-i\gamma_r H_C} |+\rangle^{\otimes n} \tag{A.6}$$

where

$$e^{-i\beta_r H_B} = \prod_{j=1}^{n} e^{-i\beta_r \mathbf{X}_\mathbf{j}} \tag{A.7}$$

and

$$e^{-i\gamma_r H_C} = \prod_{\alpha=1}^{m} e^{-i\gamma_r \mathbf{C}_\alpha} \tag{A.8}$$

This state is measured to obtain an approximate solution to the problem defined in Eq. (A.1), whose expectation value is given by

$$\langle \psi_p(\vec{\gamma},\vec{\beta}) | H_C | \psi_p(\vec{\gamma},\vec{\beta}) \rangle \tag{A.9}$$

If the values of p, $\vec{\beta}$ and $\vec{\gamma}$ are well selected, the state of the qubits after this transformation will represent a good solution to the problem defined in Eq. (A.1) with high probability, and the quality of the solution increases with the number of rounds. The selection of $\vec{\beta}$ and $\vec{\gamma}$ is not a trivial issue and there are some proposals in the literature [7]. A usual approach is starting from some initial values and then perform simplex or gradient based optimization.

Appendix B. QAOA applied to MaxCut

The MaxCut problem is a paradigm in combinatorial optimization that is particularly advantageous for QAOA for two reasons: (i) all clauses in the objective function have the same structure, hence only one quantum Hamiltonian needs to be designed, and (ii) the clauses involve only two decision variables.

In the MaxCut problem, we are given an undirected graph $G = (V, E)$, where $V = \{1, \dots, n\}$ is a set of nodes and E is the set of arcs. The goal is to establish a partition of the set V into two subsets V_{+1} and V_{-1} so that the number of arcs in E connecting nodes of the two subsets is maximized, in other words, the goal is

$$\text{maximize:} \sum_{(i,j)\in E} \frac{1}{2}(1 - \sigma_i \sigma_j) \tag{B.1}$$

$$\sigma_k = \begin{cases} -1 & \text{if } k \in V_{-1} \\ 1 & \text{if } k \in V_{+1} \end{cases} \tag{B.2}$$

Note that a single transformation $z = (\sigma + 1)/2$ converts the variables from the $\sigma \in \{-1, +1\}$ space to the $z \in \{0, 1\}$ space.

In this case, we have a Hamiltonian $\mathbf{C}_\alpha$ for each arc (i,j), which depends on just these two variables, so it may be denoted as $\mathbf{C}_{\mathbf{i,j}}$, and given that it operates on a 2-qubit state it has size 4×4. From Eq. (B.1), the computational basis $\{|00\rangle, |01\rangle, |10\rangle, |11\rangle\}$ may be considered as qstates of a physical system with energies 0,1,1,0 respectively; therefore, following [37] the Hamiltonian may be represented by the matrix

$$\mathbf{C}_{\mathbf{i,j}} = \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix} \tag{B.3}$$

which can be written as

$$\mathbf{C_{i,j}} = \frac{1}{2}(\mathbf{I} - \mathbf{Z}_i \otimes \mathbf{Z}_j) \tag{B.4}$$

where $\mathbf{Z_i}$ and $\mathbf{Z_j}$ are both the Pauli matrix

$$\mathbf{Z} = \begin{pmatrix} 1 & 0 \\ 0 & -1 \end{pmatrix} \tag{B.5}$$

Therefore, the exponential terms in Eqs. (A.7) and (A.8) can be expanded considering that

$$e^{-i\beta_r \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}} = \begin{pmatrix} \cos(\beta_r) & -i\sin(\beta_r) \\ -i\sin(\beta_r) & \cos(\beta_r) \end{pmatrix} \tag{B.6}$$

and

$$e^{-i\gamma_r \begin{pmatrix} 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}} = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & e^{-i\gamma_r} & 0 & 0 \\ 0 & 0 & e^{-i\gamma_r} & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.7}$$

The operators in Eqs. (B.6) and (B.7) have to be implemented by means of the set of quantum gates available in the target quantum computer. In particular, in the Rigetti architectures[9] the first one could be implemented as RX($2\beta_r$) and the second by means of the composition of the two 2-qubit gates CPHASE01(-γ_r) CPHASE10(-γ_r).

Alternatively, in [22] the phase separator operator is synthesized by two CNOT gates and a RX($\gamma/2$) operator on the second qubit in between them yielding the matrix

$$\begin{pmatrix} e^{-i\gamma_r/4} & 0 & 0 & 0 \\ 0 & e^{i\gamma_r/4} & 0 & 0 \\ 0 & 0 & e^{i\gamma_r/4} & 0 \\ 0 & 0 & 0 & e^{-i\gamma_r/4} \end{pmatrix} \tag{B.8}$$

Furthermore, in [38] the phase separators are performed by the $R_{ZZ}(\gamma)$ operator with the following equivalent matrix

$$\begin{pmatrix} e^{-i\gamma_r/2} & 0 & 0 & 0 \\ 0 & e^{i\gamma_r/2} & 0 & 0 \\ 0 & 0 & e^{i\gamma_r/2} & 0 \\ 0 & 0 & 0 & e^{-i\gamma_r/2} \end{pmatrix} \tag{B.9}$$

As mentioned, to execute a binary gate on two qstates, these states must be located on adjacent qubits. For this purpose, the use of a number of *swap* gates is generally necessary; a *swap* gate operates on two qubits swapping their qstates by implementing the following operator

$$\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{B.10}$$

which is performed by the SWAP gate in Rigetti systems, and may be implemented by three consecutive CNOT gates, the second one taking the second qubit as control and the first as target.

References

[1] D. Venturelli, M. Do, B. O'Gorman, J. Frank, E. Rieffel, K.E. Booth, T. Nguyen, P. Narayan, S. Nanda, Quantum circuit compilation: an emerging application for automated reasoning, in: S. Bernardini, K. Talamadupula, N. Yorke-Smith (Eds.), Proceedings of the 12th International Scheduling and Planning Applications Workshop (SPARK 2019), 2019, pp. 95–103.

[2] E.A. Sete, W.J. Zeng, C.T. Rigetti, A functional architecture for scalable quantum computing, in: 2016 IEEE International Conference on Rebooting Computing (ICRC), IEEE, 2016, pp. 1–6.

[3] D. Venturelli, M. Do, E.G. Rieffel, J. Frank, Temporal planning for compilation of quantum approximate optimization circuits, in: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI 2017), 2017, pp. 4440–4446.

[4] A. Oddi, R. Rasconi, Greedy randomized search for scalable compilation of quantum circuits, in: W.-J. van Hoeve (Ed.), CPAIOR 2018: Integration of Constraint Programming, Artificial Intelligence, and Operations Research, Springer International Publishing, Cham, 2018, pp. 446–461.

[5] R. Rasconi, A. Oddi, An innovative genetic algorithm for the quantum circuit compilation problem, in: Proceedings of the Thirty-Third AAAI Conference on Artificial Intelligence, vol. 33, 2019, pp. 7707–7714.

[6] S. Chand, H.K. Singh, T. Ray, M. Ryan, Rollout based heuristics for the quantum circuit compilation problem, in: 2019 IEEE Congress on Evolutionary Computation (CEC), 2019, pp. 974–981.

[7] E. Farhi, J. Goldstone, S. Gutmann, A quantum approximate optimization algorithm, 2014, arXiv:1411.4028.

[8] K.E. Booth, M. Do, J.C. Beck, E. Rieffel, D. Venturelli, J. Frank, Comparing and integrating constraint programming and temporal planning for quantum circuit compilation, in: Twenty-Eighth International Conference on Automated Planning and Scheduling (ICAPS 2018), 2018, pp. 366–374.

[9] A. Paler, On the influence of initial qubit placement during NISQ circuit compilation, in: S. Feld, C. Linnhoff-Popien (Eds.), Quantum Technology and Optimization Problems, Springer International Publishing, Cham, 2019, pp. 207–217.

[10] A. Zulehner, A. Paler, R. Wille, An efficient methodology for mapping quantum circuits to the IBM QX architectures, IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. 38 (7) (2019) 1226–1236.

[11] A.M. Childs, E. Schoute, C.M. Unsal, Circuit transformations for quantum architectures, in: W. van Dam, L. Mancinska (Eds.), 14th Conference on the Theory of Quantum Computation, Communication and Cryptography (TQC 2019), Leibniz International Proceedings in Informatics (LIPIcs), vol. 135, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany, 2019, pp. 3:1–3:24.

[12] M.Y. Siraichi, V.F.d. Santos, S. Collange, F.M.Q. Pereira, Qubit allocation, in: Proceedings of the 2018 International Symposium on Code Generation and Optimization, in: CGO 2018, Association for Computing Machinery, New York, NY, USA, 2018, pp. 113–125.

[13] G. Li, Y. Ding, Y. Xie, Tackling the qubit mapping problem for NISQ-Era quantum devices, in: Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems, in: ASPLOS'19, Association for Computing Machinery, New York, NY, USA, 2019, pp. 1001–1014.

[14] D. Bhattacharjee, A. Chattopadhyay, Depth-optimal quantum circuit placement for arbitrary topologies, 2017, arXiv:1703.08540.

[15] X. Zhou, S. Li, Y. Feng, Quantum circuit transformation based on simulated annealing and heuristic search, 2019, arXiv:1908.08853.

[16] S. Niu, A. Suau, G. Staffelbach, A. Todri-Sanial, A hardware-aware heuristic for the qubit mapping problem in the NISQ Era, IEEE Trans. Quant. Eng. 1 (2020) 1–14.

[17] S.S. Tannu, M.K. Qureshi, A case for variability-aware policies for NISQ-Era quantum computers, 2018, arXiv:1805.10224.

[18] P. Murali, J.M. Baker, A.J. Abhari, F.T. Chong, M. Martonosi, Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers, 2019, arXiv:1901.11054.

[19] T. Jones, S.C. Benjamin, Quantum compilation and circuit optimisation via energy dissipation, 2020, arXiv:1811.03147.

[20] S. Khatri, R. LaRose, A. Poremba, L. Cincio, A.T. Sornborger, P.J. Coles, Quantum-assisted quantum compiling, Quantum 3 (2019) 140.

[21] M. Alam, A. Ash-Saki, S. Ghosh, An efficient circuit compilation flow for quantum approximate optimization algorithm, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.

[22] M. Alam, A. Ash-Saki, S. Ghosh, Circuit compilation methodologies for quantum approximate optimization algorithm, in: 2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020, pp. 215–228.

[23] A. Botea, A. Kishimoto, R. Marinescu, On the complexity of quantum circuit compilation, in: Eleventh Annual Symposium on Combinatorial Search (SOCS), 2018, pp. 138–142.

[24] B. Freisleben, P. Merz, A genetic local search algorithm for solving symmetric and asymmetric traveling salesman problems, in: Proceedings of IEEE International Conference on Evolutionary Computation, 1996, pp. 616–621.

[25] R. Varela, C.R. Vela, J. Puente, A. Gómez, A knowledge-based evolutionary strategy for scheduling problems with bottlenecks, Eur. J. Oper. Res. 145 (2003) 57–71.

[26] E. Osaba, E. Villar-Rodriguez, J. Del Ser, A.J. Nebro, D. Molina, A. LaTorre, P.N. Suganthan, C.A. Coello Coello, F. Herrera, A tutorial on the design, experimentation and application of metaheuristic algorithms to real-world optimization problems, Swarm Evol. Comput. 64 (2021) 100888.

[27] P. Moscato, On Evolution, Search, Optimization, Genetic Algorithms and Martial Arts: Towards Memetic Algorithms, Technical Report 826, Caltech Concurrent Computation Program Report, Catech, Pasadena, California, 1989.

[28] J. Del Ser, E. Osaba, D. Molina, X.-S. Yang, S. Salcedo-Sanz, D. Camacho, S. Das, P.N. Suganthan, C.A. Coello Coello, F. Herrera, Bio-inspired computation: where we stand and what's next, Swarm Evol. Comput. 48 (2019) 220–250.

[29] F. Neri, C. Cotta, Memetic algorithms and memetic computing optimization: a literature review, Swarm Evol. Comput. 2 (2012) 1–14.

[30] L. Buriol, P. Moscato, P. França, A new memetic algorithm for the asymmetric traveling salesman problem, J. Heuristics 10 (2004) 483–506.

[31] D.C. Mattfeld, Evolutionary Search and the Job Shop Investigations on Genetic Algorithms for Production Scheduling, Springer-Verlag, 1995.

[32] P. França, A. Mendes, P. Moscato, A memetic algorithm for the total tardiness single machine scheduling problem, Eur. J. Oper. Res. 132 (2001) 224–242.

[9] see https://pyquil-docs.rigetti.com/en/stable/basics.html

[33] C. Cotta, A.J. Fernàndez, Memetic Algorithms in Planning, Scheduling, and Timetabling, Springer Berlin Heidelberg, Berlin, Heidelberg, 2007, pp. 1–30.

[34] C.R. Vela, R. Varela, M.A. González, Local search and genetic algorithm for the job shop scheduling problem with sequence dependent setup times, J. Heuristics 16 (2010) 139–165.

[35] M.A. González, A. Oddi, R. Rasconi, R. Varela, Scatter search with path relinking for the job shop with time lags and setup times, Comput. Oper. Res. 60 (2015) 37–54.

[36] M. Do, Z. Wang, B. O'Gorman, D. Venturelli, E. Rieffel, J. Frank, Planning for compilation of a quantum algorithm for graph coloring, 2020, arXiv:2002.10917.

[37] A. J., A. Adedoyin, J. Ambrosiano, P. Anisimov, A. Bärtschi, W. Casper, G. Chennupati, C. Coffrin, H. Djidjev, D. Gunter, S. Karra, N. Lemons, S. Lin, A. Malyzhenkov, D. Mascarenas, S. Mniszewski, B. Nadiga, D. O'Malley, D. Oyen, S. Pakin, L. Prasad, R. Roberts, P. Romero, N. Santhi, N. Sinitsyn, P.J. Swart, J.G. Wendelberger, B. Yoon, R. Zamora, W. Zhu, S. Eidenbenz, P.J. Coles, M. Vuffray, A.Y. Lokhov, Quantum algorithm implementations for beginners, 2020, arXiv:1804.03719.

[38] B. Tan, J. Cong, Optimal layout synthesis for quantum computing, 2020, arXiv:2007.15671.