An Innovative Genetic Algorithm for the Quantum Circuit Compilation Problem

Riccardo Rasconi, Angelo Oddi
Institute of Cognitive Sciences and Technologies, National Research Council of Italy (ISTC-CNR)
Rome, Italy
{riccardo.rasconi, angelo.odd}@istc.cnr.it

Abstract
Quantum Computing represents the next big step towards speed boost in computation, which promises major breakthroughs in several disciplines including Artificial Intelligence. This paper investigates the performance of a genetic algorithm to optimize the realization (compilation) of nearest-neighbor compliant quantum circuits. Current technological limitations (e.g., decoherence effect) impose that the overall duration (makespan) of the quantum circuit realization be minimized, and therefore the makespan-minimization problem of compiling quantum algorithms on present or future quantum machines is drawing increasing attention in the AI community. In our genetic algorithm, a solution is built utilizing a novel chromosome encoding where each gene controls the iterative selection of a quantum gate to be inserted in the solution, over a lexicographic double-key ranking returned by a heuristic function recently published in the literature.

Our algorithm has been tested on a set of quantum circuit benchmark instances of increasing sizes available from the recent literature. We demonstrate that our genetic approach obtains very encouraging results that outperform the solutions obtained in previous research against the same benchmark, succeeding in significantly improving the makespan values for a great number of instances.

Introduction
Quantum Computing represents the next big step towards speed boost in computation, which promises major breakthroughs in several disciplines including Artificial Intelligence for the resolution of important problems, for example the optimization of complex systems and drug discovery. The impact of quantum computing technology on theoretical/applicative aspects of computation as well as on the society in the next decades is considered to be immensely beneficial (Nielsen and Chuang 2011). While classical computing revolves around the execution of logical gates based on two-valued bits, quantum computing uses quantum gates that manipulate multi-valued bits (qubits) that can represent as many logical states (qstates) as are the obtainable linear combinations of a set of basis states (state superpositions). A idealized quantum circuit is composed of a number of qubits and by a series of quantum gates that operate on those qubits, and whose execution realizes a specific quantum algorithm. Executing a quantum circuit entails the chronological evaluation of each gate and the modification of the involved qstates according to the gate logic. Unfortunately, real current quantum computing technologies like ion-traps, quantum dots, super-conducting qubits, etc. limit the qubit interaction distance to the extent of allowing the execution of gates between adjacent (i.e., nearest-neighbor) qubits only (Cirac and Zoller 1995; Herrera-Marti et al. 2010; Yao et al. 2013). This has opened the way to the exploration of techniques and/or heuristics aimed at the compilation of ideal circuits to real quantum hardware through the synthesis of additional gates that bring the qstates involved in a gate to satisfying nearest-neighborhood compliance (Quantum Circuit Compilation Problem - QCCP). In addition, the Achilles’ heel of quantum computational hardware is the problem of decoherence, which degrades the performance of quantum programs over time. In order to minimize the negative effects of decoherence and guarantee more stability to the computation, it is therefore essential to produce circuits whose overall duration (i.e., makespan) is minimal.

The field of applying evolutionary algorithms to quantum computing follows two major directions: designing quantum inspired evolutionary algorithms (da Silveira, Tanscheit, and Vellasco 2017; Talbi and Draa 2017) and designing quantum computers by means of evolutionary algorithms. Specifically, while the research in quantum circuit design focuses mainly on using genetic algorithm (or genetic programming) for automatically discovering or synthesizing quantum algorithms given a desired output function (Mukherjee et al. 2009; Ruican et al. 2007), to the best of our knowledge there are no examples of genetic algorithms used to tackle the problem of compiling idealized quantum circuits to real quantum machines focusing on swap insertions.

In this work, we investigate the performance of a genetic algorithm (GA) applied to the problem of compiling quantum circuits to near-term quantum hardware. Experimenting on a set of benchmark instances of different size belonging to the Quantum Approximate Optimization Algorithm (QAOA) class (Farhi, Goldstone, and Gutmann 2014; Guerreschi and Park 2017) tailored for the MaxCut problem and devised to be executed on top of a hardware architecture proposed by Rigetti Computing Inc. (Sete, Zeng, and Rigetti...
2016), we demonstrate that our genetic algorithm outperforms the approach used in previous research against the same benchmark, both from the CPU efficiency and from the solution quality standpoint. In particular, we compare our approach against the QCCP benchmark originally proposed in (Venturelli et al. 2017), and subsequently expanded in (Booth et al. 2018) by considering: (i) variable qubit state initialization (QCCP-V), and (ii) crosstalk constraints that further restrict parallel gate execution (QCCP-X).

The paper is structured as follows. We first provide some background information as well as a description of the QCCP (and variants), then we present a formal representation of the solved problem. Subsequently, we describe the proposed genetic algorithm in details, and exhibit the results of our empirical evaluation. Finally, some concluding remarks end the paper.

The Quantum Circuit Compilation Problem

Quantum computing is based on the manipulation of qubits rather than conventional bits; a quantum computation is performed by executing a set of quantum operations (called gates) on the qubits. A gate whose execution involves \(k\) qubits is called a \(k\)-qubit quantum gate. In this work we will focus on 1-qubit and 2-qubit quantum gates. In order to be executed, a quantum circuit must be mapped on a quantum chip which determines the circuit’s hardware architecture specification ([Maslov, Falconer, and Mosca 2007]). The chip can be generally seen as an undirected weighted multi-graph whose nodes represent the qubits (quantum physical memory locations) and whose edges represent the types of gates that can be physically implemented between adjacent qubits of the physical hardware (see Figure 1 (a) (b) (c) as an example of three chip topologies of increasing size). Since a 2-qubit gate requiring two specific qstates can only be executed on a pair of adjacent qubits, the required qstates must be conveyed on such qubit pair prior to gate execution. Nearest-neighborhood compliance can be obtained by adding a number of swap gates so that every pair of qstates involved in the quantum gates can be eventually made adjacent, allowing all gates to be correctly executed.

Problem Description

The problem tackled in this work consists in compiling a given quantum circuit on a specific quantum hardware architecture. To this aim, we focus on the same framework used in (Venturelli et al. 2017): (i) the class of Quantum Approximate Optimization Algorithm (QAOA) circuits ([Farhi, Goldstone, and Gutmann 2014; Guerreschi and Park 2017]) to represent an algorithm for solving the MaxCut problem (see below); (ii) a specific hardware architecture inspired by the one proposed by Rigetti Computing Inc. (Sete, Zeng, and Rigetti 2016). The compilation process of the MaxCut problem on QAOA circuits is composed of a phase separation (P-S) step and a mixing (MIX) step. This entails the execution of a set of 2-qubit gates (called \(p-s\) gates) on a set of goals that depend on the particular MaxCut problem instance, immediately followed by the execution of a set 1-qubit gates (called mix gates), in case the phase separation step must be re-applied ([Farhi, Goldstone, and Gutmann 2014]). The number of applications of the phase separation steps (passes) is denoted by \(P\) (in this work, \(P \in \{1,2\}\)). The goals define all the qstate pairs on which a \(p-s\) gate must be applied to solve the problem instance.

Figure 2 (left side) shows an example of a graph upon which the MaxCut problem is to be executed. Given a graph \(G(V,E)\) with \(n = |V|\) nodes and \(m = |E|\) edges, the objective is to partition the node set \(V\) in two subsets \(V_1\) and \(V_2\) such that the number of edges that connect every node pair \((n_i,n_j)\) with \(n_i \in V_1\) and \(n_j \in V_2\) is maximized. The following formula: \(U = \frac{1}{2} \sum_{(i,j) \in E}(1 - s_is_j)\) describes
a quadratic objective function $U$ for the MaxCut problem, where $s_i$ is a binary variable corresponding to the $i$-th node $v_i$ of the graph $G$, that takes the value $+1$ if $v_i \in V_1$ or $-1$ if $v_i \in V_2$ at the end of the partition operated by the algorithm.

The list of $P$-s and mix gates (i.e., the goals) that must be executed during the compilation procedure in order to solve the MaxCut instance considered in this example is shown on the right side of the figure. As shown, the compilation problem requires the execution of the two phase separation steps $P$-$S$ 1 and $P$-$S$ 2, interleaved by the two mixing steps $MIX$ 1 and $MIX$ 2 ($P = 2$).

**Problem Representation**

Formally, the Quantum Circuit Compilation Problem (QCQP) is a tuple $P = (C_0, L_0, QM)$, where $C_0$ is the input quantum circuit, representing the execution of the MaxCut algorithm, $L_0$ is the initial assignment of the $i$-th $q$state $q_i$ to the $i$-th qubit $n_i (q_i \leftrightarrow i)$, and $QM$ is a representation of the quantum hardware as a multigraph.

1. The input quantum circuit is a tuple $C_0 = (Q, P$-$S, MIX, \{g_{start}, g_{end}\}, TC_0)$, where $Q = \{q_1, q_2, \ldots, q_N\}$ is the set of qstates from which, a planning & scheduling perspective (see for example (Nau, Ghallab, and Traverso 2004), Chapter 15) represent the resources necessary for each gate’s execution. $P$-$S$ and MIX are, respectively, the set of $P$-s and mix gates operations such that: (i) every $P$-$s(q_i, q_j)$ gate requires two qstates for execution; (ii) every $mix(q_i)$ gate requires one qstate only. $g_{start}$ and $g_{end}$ are two fictitious reference gates operations requiring no qstates. The execution of every quantum gate requires the uninterrupted use of the involved qstates during its processing time, and each qstate $q_i$ can process at most one quantum gate at a time. Finally, $TC_0$ is a set of simple precedence constraints imposed on the $P$-$S$, $MIX$ and $\{g_{start}, g_{end}\}$ sets, such that: (i) each gate in the two sets $P$-$S$, $MIX$ occurs after $g_{start}$ and before $g_{end}$; (ii) according to the total order imposed among the steps $P$-$S_1$, $MIX_1$, $P$-$S_2$, $MIX_2$, $\ldots$, $P$-$S_P$, $MIX_P$, all the gates belonging to the step $P$-$S_k$ ($MIX_k$) involving a specific qstate $q_i$ must be executed before all the gates belonging to the next step $MIX_k$ ($P$-$S_{k+1}$) involving the same qstate $q_i$, for $k = 1, 2, \ldots, P$ (for $k = 1, 2, \ldots, (P - 1)$).

2. $L_0$ is the initial assignment at the time origin $t = 0$ of qstates $q_i$ to qubits $n_i$.

3. $QM$ is a representation of the quantum hardware as an undirected multi-graph $QM = (V_N, E_{P-S}, E_{swap}, \tau_{mix}, \tau_{P-S}, \tau_{swap})$, where $V_N = \{n_1, n_2, \ldots, n_N\}$ is the set of qubits (nodes), $E_{P-S}$ ($E_{swap}$) is a set of undirected edges $(n_i, n_j)$ representing the set of adjacent nodes the qstates $q_i$ and $q_j$ of the gates $p$-$s(q_i, q_j)$ ($swap(q_i, q_j)$) can potentially be allocated to. In addition, the labelling functions $\tau_{P-S}: E_{P-S} \rightarrow \mathbb{Z}^+$ and $\tau_{swap}: E_{swap} \rightarrow \mathbb{Z}^+$ respectively represent the durations of the gate operations $p$-$s(q_i, q_j)$ and $swap(q_i, q_j)$ when the qstates $q_i$ and $q_j$ are assigned to the corresponding adjacent nodes. Similarly, the labelling function $\tau_{mix}: V \rightarrow \mathbb{Z}^+$ represents the durations of the mix gate (which can be executed at any node $n_i$). Figure 1 shows an example of quantum hardware with gate durations.

A feasible solution is a tuple $S = (\langle SWAP, TC \rangle$, which extends the initial circuit $C_0$ with: (i) a set $SWAP$ of additional $swap(q_i, q_j)$ gates added to guarantee the adjacency constraints for the set of $P$-$S$ gates, and (ii) a set $TC$ of additional simple precedence constraints such that:

- for each $q$state $q_i$, a total order $\preceq_i$ is imposed among the set $Q_i$ of operations requiring $q_i$, with $Q_i = \{op \in P$-$S$ $\cup$ $MIX$ $\cup$ $SWAP: op$ requires $q_i\}$.
- all the $P$-$s(q_i, q_j)$ and $swap(q_i, q_j)$ gate operations are allocated on adjacent qubits in $QM$.

Given a solution $S$, the makespan $mk(S)$ corresponds to the maximum completion time of the gate operations in $S$. A path between the two fictitious gates $g_{start}$ and $g_{end}$ is a sequence of gates $g_{start}, op_1, op_2, \ldots, op_k, g_{end}$, with $op_j \in P$-$S$ $\cup$ $MIX$ $\cup$ $SWAP$, such that $g_{start} \preceq op_1 \preceq op_2, \ldots, op_k \preceq g_{end} \in (TC_0 \cup TC)$. The length of the path is the sum of the durations of all the path’s gates and $mk(S)$ is the length of the longest path from $g_{start}$ to $g_{end}$. An optimal solution $S^*$ is a feasible solution characterized by the minimum makespan.

**Problem Extensions**

In the original version of the QCQP problem, the initial $q$state-$qubit$ assignment $L_0$ is fixed and decided at the beginning. Moreover, the non-overlap constraint existing among quantum gates only forbids the concurrent execution of any two gates that share the same qbits. In the following, two extensions of the original QCQP introduced in (Booth et al. 2018), are presented.

**Variable Qstate Initialization (QCQP-V)** The Variable Initial State problem version, the initial assignment $L_0$ at the time origin $t = 0$ of qstates $q_i$ to qubits $n_i$ (see point 2. of the QCQP representation) is undecided, and becomes part of the search problem.

**Crosstalk (QCQP-X)** The QCQP-X (crosstalk) version of the problem forbids the concurrent execution of $P$-$s$ and/or $swap$ gate pairs that share the same neighboring qubits in the $QM$, as follows. Let $Neighbors(n)$ be a function that returns the neighboring nodes of node $n \in QM$. Given two quantum gates $op_i, op_j \in P$-$S$ $\cup$ $SWAP$ to be executed on the node pairs $(n_1^1, n_2^1)$ and $(n_1^2, n_2^2)$ respectively, if $Neighbors(n_1^1) \cup Neighbors(n_2^1) \cap \{n_1^1, n_2^1\} \neq \emptyset$ then $op_i$ and $op_j$ cannot overlap.

**The Genetic Algorithm**

Genetic algorithms define a well-known metaheuristic approach commonly used to generate high-quality solutions to optimization and search problems by relying on bio-inspired operators such as mutation, crossover and selection (Goldberg 1989) following a rather standardized pattern. The pseudocode of the genetic algorithm used in this work is depicted in Algorithm 1, and is briefly described in the following. At the beginning, the initial population $pop$
is generated and evaluated; subsequently, the evolutionary iteration starts and proceeds by: (i) identifying the best individual bestIndiv of the current population; (ii) evolving the current population into a new one (newPop), and inserting bestIndiv as the first element of newPop. At the end of the evolution phase, the new population will become the current population, and a new iteration is restarted. The evolution cycle continues until the maximum allotted time has elapsed (termination condition). The next sections will be dedicated to a detailed description of the previous algorithm.

The Encoding Scheme

Our encoding scheme is basically a value encoding, where each gene in the chromosome retains an integer value belonging to the $[1, maxValue]$ range, where maxValue is a determined value that depends on the size of each problem instance, and whose meaning will be discussed in the next section. At the beginning of the genetic algorithm, the initial population pop is created, composed of populationSize chromosomes (3-rd line of Algorithm 1) whose gene values are randomly generated from the $[1, maxValue]$ interval according to a uniform distribution.

The Decoding Scheme

As shown in Figure 3, in our encoding each gene in the chromosome retains an integer value representing the exact quantum gate position within a specific ordering computed and returned by a particular Quantum Gate Ranking Heuristic introduced in (Oddi and Rasconi 2018), and whose informal description is provided next (the reader interested in the technical details is referred to the original publication).

The Quantum Gate Ranking Heuristic (QGRH) The solution building approach presented in (Oddi and Rasconi 2018) is based on the chronological insertion of one gate operation at a time in the partial solution $S$, until all the gates in the set $P-S \cup MIX$ are in $S$. This insertion relies on the concept of chain; we define chain$i$, as a sequence of gate operations $op \in S$ that involve qstate $qi$. In (a) partial solution $S$ there exist as many chains as are the qstates involved in the problem instance. Informally, a solution is incrementally built by: (i) selecting a quantum gate $op(qi) \in S$ at each solving iteration, and (ii) inserting the selected gate in the partial solution by attaching it as the last element of the two chains associated to the qstates $q_i$ and $q_j$ involved in the gate operation (if $op(q_i) \in MIX$, only the chain associated to the qstate $q_i$ is considered).

Clearly, the criteria used for quantum gate selection (Quantum Gate Ranking Heuristic - QGRH) play a key role in the solving process described above. Basically, the QGRH returns a list of gate operations $op \in P-S \cup MIX \cup SWAP$, among the operators that are still to be inserted in the solution $S$. The ranking value for each gate $op$ is computed by ordering lexicographically the values returned by two different functions $f_1$ and $f_2$.

The two-dimension distance function used for the gate selection ranking is devised to produce a twofold effect. The $f_1$ component acts as a global closure metric; by evaluating the overall distance left to be covered by all the qstates still involved in gates yet to be executed, it guides the selection towards the gate that best favours the quickest execution of the remaining gates. Conversely, the $f_2$ component acts as a local closure metric, in that it favours the mutual approach of the closest qstates pairs. Lastly, the reader should note that the QGRH ranking is deterministic.

Decoding from the QGRH Ranking To wrap up, QGRH return a list of gate operations ordered by “insertion appeal”, ranking as most palatable the gates that best facilitate the efficient execution of the remaining gates while favouring the mutual approach of the closest qstates pairs.

QCCP. Given the QGRH ranking, it is now possible to fully explain the structure of the chromosomes. In our algorithm, a chromosome chr is composed of $n = |P-S| + |MIX| + |SWAP|$ genes, whose values are the pointers to the selected quantum gates from the ranking returned by QGRH, and whose positions in chr denote the chronological order in which such selections will be made during the construction of the solution realized by the decoding procedure (see Figure 3).

Given a chromosome chr, the decoding procedure constructs $S$ by iteratively reading the $i$-th chromosome gene chr$[i]$ and executing the QGRH over the partial solution $S$ (initially empty). As explained in the Quantum Gate Ranking Heuristic section, the QGRH will produce a ranking out of the quantum gates still to be inserted in $S$, and the gate $op$ pointed by chr$[i]$ will be selected for the next insertion in $S$, where $i$ is the iteration’s number of the decoding procedure. Once the selected gate is inserted in $S$, QGRH is called again to produce a new ranking, to be used at the next iteration $i + 1$, exploiting the chr$[i + 1]$ chromosome gene. The
The Crossover Operator

As shown in Algorithm 1, the current population \( \text{pop} \) is evolved by selecting two new individuals \( \text{Indiv}_1 \) and \( \text{Indiv}_2 \) by means of a Tournament Selection procedure for each individual in \( \text{pop}\{\text{bestIndiv} \} \) (where \( \text{bestIndiv} \) is the best individual found at the previous iteration), and then generates a new individual \( \text{newIndiv} \) by performing a crossover operation between \( \text{Indiv}_1 \) and \( \text{Indiv}_2 \). The Tournament Selection procedure is sketched in Algorithm 3, and is identical for all problem versions QCCP, QCCP-V, and QCCP-X. Basically, each new Individual is selected by: (i) generating a new \( \text{tournament} \) population of size equal to \( \text{tournamentSize} < \text{populationSize} \) composed of individuals randomly selected from the current population \( \text{pop} \), and (ii) returning the fittest individual among those in the \( \text{tournament} \) population.

The Fitness Function

The computation of the fitness function is rather straightforward, once the solution \( S \) has been decoded. In fact, given a plan represented by \( S \), its objective function value can be directly assessed as the plan’s makespan \( mk(S) \).

The Selection Operator

As shown in Algorithm 1, the current population \( \text{pop} \) is evolved by selecting two new individuals \( \text{Indiv}_1 \) and \( \text{Indiv}_2 \) by means of a Tournament Selection procedure for each individual in \( \text{pop}\{\text{bestIndiv} \} \) (where \( \text{bestIndiv} \) is the best individual found at the previous iteration), and then generates a new individual \( \text{newIndiv} \) by performing a crossover operation between \( \text{Indiv}_1 \) and \( \text{Indiv}_2 \). The Tournament Selection procedure is sketched in Algorithm 3, and is identical for all problem versions QCCP, QCCP-V, and QCCP-X. Basically, each new Individual is selected by: (i) generating a new \( \text{tournament} \) population of size equal to \( \text{tournamentSize} < \text{populationSize} \) composed of individuals randomly selected from the current population \( \text{pop} \), and (ii) returning the fittest individual among those in the \( \text{tournament} \) population.

The Crossover Operator

The crossover operator procedure for the QCCP (and QCCP-X) case is sketched in Algorithm 4. The QCCP-V version of the crossover operator is only described textually.
Algorithm 4 Crossover Procedure (QCCP and QCCP-X)

```plaintext
newIndiv ← null;
for (i = 1; i <= chr.LLENGTH(); i++) do
  if (probability(xoverRate)) then
    gene ← Indiv1.GETGENE(i);
  else
    gene ← Indiv2.GETGENE(i);
  end if
newIndiv.SETGENE(gene);
end for
return newIndiv;
```

Algorithm 5 Mutation Operator (QCCP and QCCP-X)

```plaintext
for (k = 0; k < populationSize; k++) do
  Indiv ← newPop.GETINDIVIDUAL(k);
  for (i = 1; i <= Indiv.LENGTH(); i++) do
    if (probability(mutationRate)) then
      gene ← Indiv.RANDOMVALUE();
      Indiv.SETGENE(gene);
    end if
  end for
end for
return newPop;
```

**QCCP and QCCP-X case.** Given two parent chromosomes Indiv1 and Indiv2, the offspring individual newIndiv is generated by randomly selecting each gene either from Indiv1 or Indiv2 with a uniform probability xoverRate, and assigning the selected gene to newIndiv.

**QCCP-V case.** In the QCCP-V case, since the chr0 chromosome’s segment represents a permutation of values, a standard Partially-Mapped Crossover (Goldberg and Lingle 1985) has been used. The rest of the chromosome is treated as in the QCCP case.

**The Mutation Operator**

When a new population is generated at each evolution step (see Algorithm 1) every chromosome undergoes a very simple mutation process, as depicted in the Algorithm 5 for the QCCP (and QCCP-X) cases. The QCCP-V version of the mutation operator is only described textually.

**QCCP and QCCP-X case.** Each gene of each chromosome of the newly generated population is randomly modified with a probability equal to mutationRate.

**QCCP-V case.** In the QCCP-V case, the chr0 chromosome segment is mutated by swapping two randomly chosen positions $i$ and $j$ ($i \neq j$, $i, j \in [1, N]$), with mutationRate probability. The rest of the chromosome is mutated as in the previous case.

**Empirical Evaluation**

The material presented in this section describes the performance obtained with our genetic algorithm against the benchmark set originally presented in (Venturrelli et al. 2017) (QCCP problem version), and later expanded in (Booth et al. 2018) (QCCP-V and QCCP-X problem versions). Many of the results obtained in (Venturrelli et al. 2017) have been successively improved by (Oddi and Rasconi 2018) by means of an heuristic-based greedy randomized search procedure.

In this work, we compare our results with those of both the previous works. In particular, the benchmark set we tackle is composed of three benchmarks characterized by instances of three different sizes, based on quantum chips with $N = 8, 21$ and 40 qubits, respectively (see Figure 1). The $N = 8$ and the $N = 21$ benchmarks are solved considering $P = 2$ problem instances (two passes), while the $N = 40$ benchmark is solved in both flavors $P = 1$ (to allow comparison with (Venturrelli et al. 2017)) and $P = 2$ (to allow comparison with (Oddi and Rasconi 2018)). Moreover, in order to guarantee maximum fairness of comparison, we have also implemented a version of the greedy random sampling procedure proposed in (Oddi and Rasconi 2018), named GRS* in this work, to differentiate it with the original GRS procedure.

In the actual implementation of Algorithm 1, we have assigned the following values for the parameters: populationSize = 50, tournamentSize = 5, xoverRate = 0.5, mutationRate = 0.015, maxValue = 15, 30, 60 for benchmark size $N = 8, 21, 40$ respectively. To guarantee fairness of comparison also from the computing resources standpoint, all experiments have been performed on a 64-bit Windows10 O.S. running on Intel(R) Core(TM)2 Duo CPU E8600 @3.33 GHz with 8GB RAM, exactly as in (Oddi and Rasconi 2018).

**Results Analysis**

The results of our experimentation are summarized in Table 1. The results are presented in aggregated form for reasons of space; however, the complete set of makespan values, together with the complete set of solutions are available at http://pst.istc.cnr.it/~angelo/qc/. We compare our results with the best results presented in (Venturrelli et al. 2017) and obtained with the TFD (Eyerich, Mattmüller, and Röger 2009), SGPPlan (Wah and Chen 2004; Chen, Wah, and Hsu 2006) and LPG (Gerevini, Saetti, and Serina 2003) temporal planners, and with those presented in (Oddi and Rasconi 2018) and obtained by means of their GRS greedy random sampling procedure (see the respective columns in Table 1). The results of our implementation of the GRS procedure and of the genetic algorithm are presented in the GRS* and GA columns, respectively. The second row in the table reports the tackled QCCP problem version (QCCP, QCCP-V, and QCCP-X). The size of the benchmark set is reported in the column $N$, while the number of passes characterizing each benchmark instance is reported in the column $P$. For each given benchmark set, the score in every slot of the table represents the average of all the scores obtained from all the set instances, where the score of the $i$-th instance is obtained as the ratio $MK_i/MK_i(x)$, where $MK_i$ is best makespan obtained for the $i$-th instance among all competing procedures, and $MK_i(x)$ is the makespan obtained with

---

1 Benchmarks and results are available at: https://ti.arc.nasa.gov/m/groups/ast/programming-and-scheduling/VentCirComp17\_data.zip

7712
The solving procedure $x$ under analysis. Lastly, the values between brackets report the maximum CPU time (in minutes) allotted to all procedures in their respective works, for each run.

Interestingly, our implementation of the GRS procedure (GRS*) seems to clearly outperform the original counterpart proposed in (Oddi and Rasconi 2018), which was already improving significantly over the results obtained with temporal planners against the QCCP problem version. By applying some elementary math to the plan scores of Table 1, it can be seen that our version further improves the results obtained by GRS for all benchmark sets, obtaining the following score improvements: $+4.76\%$ (from 0.84 to 0.88) for the $[N = 8, P = 2]$ set, $+9.59\%$ (from 0.73 to 0.80) for the $[N = 21, P = 2]$ set, $+22.22\%$ (from 0.72 to 0.88) for the $[N = 40, P = 1]$ set, and $+21.66\%$ (from 0.60 to 0.73) for the $[N = 40, P = 2]$ set. It should also be noted that our results have been obtained by further reducing the allotted CPU time for each run w.r.t. our competitor (5 mins Vs. 15 mins for the $[N = 21, P = 2]$ instances, and 10 mins Vs. 15 for the $[N = 40, P = 2]$ instances). For these reasons, we selected the GRS* procedure as the main reference for performance evaluation.

GA Vs. GRS* The main achievement of this work is represented by the results obtained with the genetic algorithm (GA). As the table shows, the superiority of the GA over the GRS* is clear in all cases with the exception of the $[N = 8, P = 2]$ instances, where the two approaches exhibit similar performance. This can be easily explained by the fact that for the smallest instances, the search space is relatively small and therefore the GRS* random search approach is still as effective as the GA; but as the problem size grows larger (and proportionally, the search space), the random approach clearly finds more difficulties. For all the other cases (instance size and problem versions), the GA exhibits convincing score improvements ranging from a minimum of $+8.75\%$ for the $[N = 21, P = 2]$ set (QCCP case) to a maximum of $+28\%$ for the $[N = 40, P = 2]$ set (QCCP-V case).

As a comment on the previous results, we note that in (Oddi and Rasconi 2018), QGRH is used within a greedy randomized search, to conduct a broad-spectrum search that randomly selects the best ranked options only and disregards the others, the intuition being that a search process should both follow the heuristic advice and also find the best compromise with the random choices to conduct a broader search and find better quality solutions. On the contrary, the genetic algorithm starts from a set of different unbiased solutions (the initial population) and progressively converges to specific solutions still driven by QGRH, but of higher quality due to the inherent parallelism of population-based search procedures.

The effect of qubit state initialization This section is dedicated to commenting on the results obtained comparing the two problem versions QCCP and QCCP-V. As Table 1 reports, making the qstate initialization part of the decision process has the twofold effect of: (i) widening the search space (therefore making the search for good solutions harder) but on the other hand, (ii) opening the search to better solutions from the makespan minimization standpoint.

From this perspective, the obtained results prove that GA is much more effective than GRS* in exploring such enlarged search space towards good solutions. In fact, two interesting observations can be made from the obtained data that highlight opposite behaviors on behalf of GRS* and GA, respectively.

The first observation is that the the GRS* procedure weakens its optimization capability as the problem size grows larger; this can be proved by observing that the plan score improvement obtained with GRS* in the QCCP $\rightarrow$ QCCP-V passage decreases as the problem size increases, passing from $+12.5\%$ (from 0.88 to 0.99) for the $[N = 8, P = 2]$ set, to $+8.75\%$ (from 0.80 to 0.87) for the $[N = 21, P = 2]$ set, and finally to $+6.85\%$ (from 0.73 to 0.78) for the $[N = 40, P = 2]$ set.

The second observation is that the GA procedure strengthens its optimization capability as the problem size grows larger; this can be proved by observing that the plan score improvement obtained with GA in the QCCP $\rightarrow$ QCCP-V passage increases as the problem size increases, passing from $+12.64\%$ (from 0.87 to 0.98) for the $[N = 8, P = 2]$ set (a value similar to the one obtained with the GRS*), to $+14.94\%$ (from 0.87 to 1.00) for the $[N = 21, P = 2]$ set, and finally to $+21.95\%$ (from 0.82 to 1.00) for the $[N = 40, P = 2]$ set.

<table>
<thead>
<tr>
<th>$N$</th>
<th>$P$</th>
<th>TFD (QCCP)</th>
<th>SGPlan (QCCP)</th>
<th>LPG (QCCP)</th>
<th>GRS (QCCP)</th>
<th>GRS* (QCCP)</th>
<th>GA (QCCP)</th>
<th>GRS* (QCCP-V)</th>
<th>GA (QCCP-V)</th>
<th>GRS* (QCCP-X)</th>
<th>GA (QCCP-X)</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>2</td>
<td>0.78(10)</td>
<td>-</td>
<td>-</td>
<td>0.84(1)</td>
<td>0.88(1)</td>
<td>0.87(1)</td>
<td>0.99(1)</td>
<td>0.98(1)</td>
<td>0.95(1)</td>
<td>1.00(1)</td>
</tr>
<tr>
<td>21</td>
<td>2</td>
<td>0.65(60)</td>
<td>-</td>
<td>-</td>
<td>0.73(15)</td>
<td>0.80(5)</td>
<td>0.87(5)</td>
<td>1.00(5)</td>
<td>0.87(5)</td>
<td>1.00(5)</td>
<td></td>
</tr>
<tr>
<td>40</td>
<td>1</td>
<td>-</td>
<td>0.47(60)</td>
<td>0.62(60)</td>
<td>0.72(1)</td>
<td>0.88(1)</td>
<td>1.00(1)</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0.60(15)</td>
<td>0.73(10)</td>
<td>0.82(10)</td>
<td>0.78(10)</td>
<td>1.00(10)</td>
<td>0.89(10)</td>
<td>1.00(10)</td>
</tr>
</tbody>
</table>
Conclusions

In this paper, we investigate the performance of a genetic approach to solve the quantum circuit compilation problem applied to a class of QAOA circuits. The objective is the synthesis of minimum-makespan quantum gate execution plans that successfully compile idealized circuits to a set realistic near-term quantum hardware architectures.

The main contribution of this work is a genetic algorithm that leverages a specific chromosome encoding where each gene controls the iterative selection of a quantum gate to be inserted in the solution, over a lexicographic double-key quantum gate ranking returned by a heuristic function recently published in the literature. We have performed an experimental campaign, testing our algorithm against a Quantum Circuit Compilation Problem benchmark known in the literature, proving that the proposed algorithm exhibits very convincing performance compared with recently published results against the same benchmark. To make the comparison more interesting, we have also proposed a complete re-implementation of our Greedy Randomized Search originally presented in (Oddi and Rasconi 2018), demonstrating improved performance w.r.t. the previous version, despite such results remain inferior compared to the ones obtained with the genetic algorithm proposed in this work.

References


