Greedy Randomized Search for Scalable Compilation of Quantum Circuits

Angelo Oddi and Riccardo Rasconi

Institute of Cognitive Sciences and Technologies (ISTC-CNR),
Via S. Martino della Battaglia, 44, 00185 Rome, Italy
{angelo.oddi,riccardo.rasconi}@istc.cnr.it
http://www.istc.cnr.it

Abstract. In this work, we investigate the performances of a Greedy Randomized Search approach to optimize the compilation of quantum circuits. Quantum computing is based on the manipulation of quantum bits (*qubits*) rather than conventional bits, and such manipulation is performed by executing a set of operations on the qubits called *quantum gates*. Unfortunately, technological constraints for many technologies and applications limit the interaction distance between qubits so as to allow the execution of gates between adjacent (i.e., *nearest-neighbor*) qubits only. However, nearest-neighbor compliance can be obtained by adding a number of so-called *swap* gates between adjacent qubits, whose effect is to mutually exchange the physical locations of the involved qubits. Moreover, current quantum computational hardware suffers from a known phenomenon called *decoherence*, which degrades the performance of quantum algorithms with time. For this reason, it is essential that the duration (makespan) of the quantum circuit compilation plan be as short as possible, so as to minimize the decoherence effect and guarantee more stability to the computation.
Our planning procedure has been tested on a set of quantum circuit benchmark instances of increasing sizes available from the recent literature and characterized by a high number of quantum gates. We demonstrate that the heuristic approach we present outperforms the solutions obtained in previous research against the same benchmark, both from the CPU efficiency and from the solution quality standpoint.

Keywords: Quantum Computing, Optimization, Scheduling, Planning, Greedy Heuristics, Random Algorithms

1 Introduction

In this work, we investigate the performances of Greedy Randomized Search techniques [1–3] to the problem of compiling quantum circuits to emerging quantum hardware. Quantum Computing [4] represents the next big step in realizing computing machines. The impact of this new technology on theoretical/applicative aspects of computation as well as on the society in the next decades is considered to be huge. The possible applications of quantum computing ranges from classical optimization (e.g., airline scheduling, financial analysis, and web search), to machine learning (e.g., object detection in a scene, labeling stories and images), or Monte Carlo simulation, used in many industries (e.g., energy or manufacturing). Emerging gate-model processors are universal in that, once scaled up, they can run any quantum algorithm. For example, IBM recently provided public access to a 20-qubit gate-model processor through the cloud; in addition, they are currently working on a new 50-qubit processor prototype[1].

A quantum circuit is composed by a number of interactions between qubits (called quantum *gates*) whose execution (circuit compilation) realizes the execution of the quantum algorithm relative to the circuit. In this work, we present an Greedy Randomized Search (GRS) procedure that synthesizes compilation plans for quantum circuits. Our procedure has been tested on a set of quantum circuit benchmark instances of different size belonging to the *Quantum Approximate Optimization Algorithm* (QAOA) class [5] for the MaxCut problem for an architecture proposed by Rigetti Computing Inc. [6]. These benchmark problems are characterized by a high number of commuting quantum gates (i.e., gates among which no particular order is superimposed), which allows for great flexibility and parallelism in the solution, which makes the corresponding optimization problem very interesting and guarantees greater makepan minimization potential for decoherence minimization [7]. We demonstrate that the meta-heuristic we present outperforms the solutions obtained in previous research against the same benchmark, both from the CPU efficiency and from the solution quality standpoint.

The paper is organized as it follows. Section 2 gives a brief introduction to Quantum Computing an describes the relevant literature to this paper. Next Section 3 proposes a formal statement of the solved problem, whereas subsequent Section 4 and Section 5 provide the given heuristic solving algorithm and the Greedy Randomized Search approach, respectively. Finally, an empirical validation based on the results proposed in [7] and some conclusions close the paper.

2 Quantum Computing

Quantum computing is based on the manipulation of quantum bits (*qubits*) rather than conventional bits, and such manipulation is performed by executing a set of quantum operations on the qubits, called *gates*. A gate whose execution involves k qubits is called *k-qubit quantum gate*. In this work we will focus on quantum circuits composed of 1 or 2 qubits (1-qubit and 2-qubit quantum gates, respectively). As anticipated earlier, in order to be executed on quantum computing hardware, quantum algorithms must be compiled into a set of elementary machine instructions (i.e., the gates), which are subsequently applied at specific times. According to [7], the problem of finding a sequence of gates that efficiently realizes the quantum compilation fits perfectly into a temporal

[1] Quantum Computing - IBM Q, `https://www.research.ibm.com/ibm-q/` last accessed December 13, 2017

planning problem. However, the Achilles' heel of quantum computational hardware is *decoherence*, which degrades the performance of quantum programs over time. Thus, it is important to produce compilation plans that minimize the duration of the resulting circuits, so as to minimize the decoherence effect. In [7], the authors modelled machine instructions as PDDL2.1 durative actions [8], enabling domain independent temporal planners [9] to find a quantum compilation plan in terms of a parallel sequence of conflict-free operators (i.e., the gates), characterized by minimum completion time (makespan).

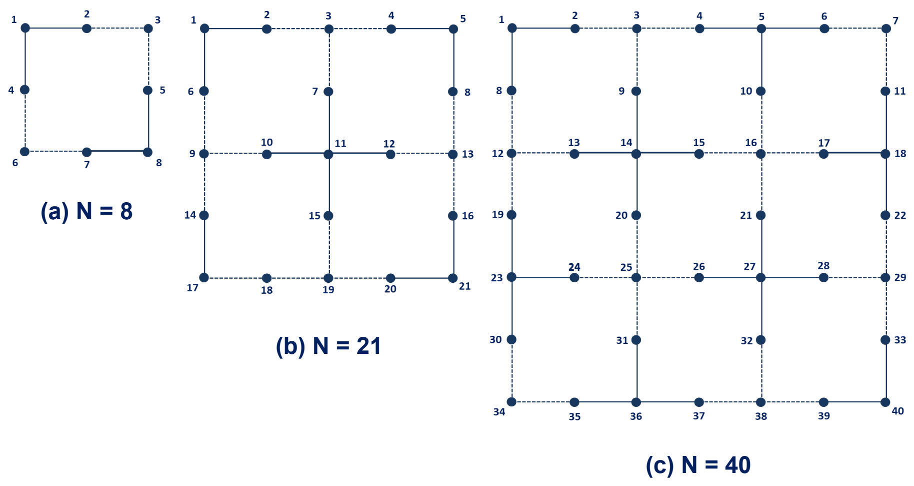


Fig. 1. Scheme for three quantum chip designs characterized by an increasing number of qubits ($N = 8, 21, 40$). Every qubit is located at a different location (node), and the integers at each node represent the chip location identifier. Two locations connected by an edge are adjacent, and each edge represents a 2-qubit gate (*p-s* or *swap*) that can be executed between the qubits located at those locations. *p-s* gates executed on continuous edges have duration $\tau_{p-s} = 3$, while *p-s* gates executed on dashed edges have duration $\tau_{p-s} = 4$.

In order to be compiled, a quantum circuit must be mapped on a quantum chip that determines the circuit's hardware architecture specification ([10]). The chip can be generally seen as an undirected weighted multigraph whose nodes represent the qubits current locations (quantum physical memory locations) and whose edges represent the types of gates that can be physically implemented on the qubits of the physical hardware (see Figure 1 as an example of three chip topologies of increasing size). The association between a qubit and its current location is defind as the qubit state (or *qstate*). The quantum circuit thus described is called *ideal* in that it assumes that all quantum gates involving any qubit can be reliably executed regardless the qubits' currently separating distance in the chip. Unfortunately, technological constraints for many technologies and applications limit the qubit interaction distance to the extent of allowing the execution of gates between adjacent (i.e., *nearest-neighbor*) qubits only. However, nearest-neighbor compliance can be obtained by adding a number of so-called *swap* gates between adjacent qubits, whose effect is to mutually exchange the physical locations of the involved qubits. Consequently, by properly *planning* the necessary *swap* gates, every pair of qubits can be eventually made adjacent, allowing all quantum gates to be safely executed. In addition, current quantum computational hardware suffers from a known phenomenon called *decoherence*, which degrades the performance of quantum algorithms with time. In order to minimize the negative effects of decoherence and guarantee more stability to the computation, it is essential that the duration of the circuit's execution that carries out the quantum computation be as short as possible.

From all of the above, the importance of synthesizing compilation plans that: (i) ensure nearest-neighbor interaction on qubits, and (ii) are characterized by a minimal completion time (makespan), becomes clear.

3 Problem Definition

From the Planning & Scheduling point of view, quantum gates can be seen as elementary operations characterized by durations, ordering constraints and absolute start times. In other words, given a set of gates operating on a set of qubits, the execution of a compilation plan that satisfies all the imposed constraints corresponds to the execution of the algorithm (represented as a quantum circuit) that generates the output solution. However, as in the case of classical CPU, ideal quantum circuits must be compiled on a specific quantum hardware architecture. The compilation process generates an hardware specific quantum circuit containing additional gates and satisfying additional constraints due to the use of the specific hardware device. In this paper we will consider the same framework used in [7]:

- we consider the class of *Quantum Approximate Optimization Algorithm* (QAOA) circuits ([5]) to represent an algorithm for solving various instances of the MaxCut problem (see below);
- the ideal QAOA circuits will then be compiled on a hardware architecture proposed by Rigetti Computing Inc. [6].

As opposed to [7], where the hardware-specific compilation problem is reduced and solved as a PDDL ([8]) temporal planning problem, in this work we propose a scheduling-oriented formulation of the same compilation problem. The choice of utilizing the QAOA class of circuits demands the planning capability to manage a high number of commuting quantum gates (i.e., gates among which no particular order is superimposed), which allows for great flexibility and parallelism in the solution, therefore making the temporal planning problem very interesting and guarantees greater makepan minimization potential for decoherence minimization [7]. Moreover, the Rigetti hardware architecture selected for this quantum compilation problem (see Figure 1) is particularly interesting as it allows for two types of nearest-neighbor relations, which directly reflects on two different durations for the *p-s* gates, depending on the chip's particular edge (depicted as continuous or dashed in the figure) on which the *p-s* gate is executed.

The rest of this section is dedicated to the purpose of: (i) describing the Max-Cut problem and (ii) providing a formulation of the Quantum Gate Compilation Planning problem.

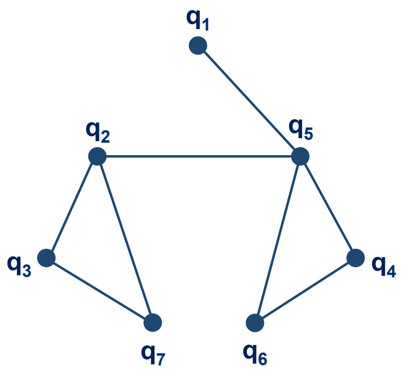
MaxCut Problem: Given a graph $G(V, E)$ with $n = |V|$ nodes and $m = |E|$ edges, the objective is to partition the node set V in two subsets V_1 and V_2 such that the number of edges that connect every node pair $\langle n_i, n_j \rangle$ with $n_i \in V_1$ and $n_j \in V_2$ is maximized.

The following formula describes a quadratic objective funcion fo the MaxCut problem:

$$U = \frac{1}{2} \sum_{(i,j) \in E} (1 - s_i s_j) \tag{1}$$

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 compilation process of the MaxCut problem on idealized QAOA circuits is rather simple, and it is composed of a *phase separation* (P-S) step and a *mixing step* (MIX), which entails the execution of a set of identical 2-qubit gates (one 2-qubit gate for each quadratic term in the objective function of Eq. (1)) called *p-s* gates, followed by the execution of a set of 1-qubit gate for each node of the graph G, called *mix* gate ([5]).



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 as a set of *p-s* and *mix* gates to be planned for and executed (note that the qstate q_8 does not appear in this instance, and therefore it will not participate to any gate).

Figure 2 (left side) shows an example of the graph G upon which the MaxCut problem is to be executed, corresponding to the problem instance **#1** of the **N8_u0.9** benchmark set that will be used in the experimental section of this paper (Section 6). In the right side of the figure, the list of *p-s* and *mix* quantum gates that must be executed during the compilation procedure is provided. In particular, the list of *p-s* gates under the *P-S 1* and *P-S 2* labels correspond to the *phase separation* steps, while the list of *mix* gates under the *MIX 1* and *MIX 2* labels correspond to the *mixing* steps. The compilation problem depicted in the figure requires the execution of two phase separation steps (*P-S 1* and *P-S 2*), interleaved by two mixing steps (*MIX 1* and *MIX 2*). In other words, all the quantum compilation instances considered in this work require that the *phase speration* step + *mixing* step (compilation pass) are repeated $p = 2$ times.

Quantum Gate Compilation Planning Problem formulation: The Quantum Gate Planning Compilation problem is a tuple $P = \langle Q, P\text{-}S, MIX, TC, QM, L_0\rangle$, where $Q = \{q_1, q_2, ..., q_N\}$ is the set of *qubits* which can be thought of as the *resources* necessary for each gate's execution; *P-S* and MIX are, respectively, the set of *p-s* and *mix* gate operations such that: (i) every *p-s*(q_i, q_j) gate requires two qubits q_i and q_j for execution (2-qubit type); (ii) every *mix*(q_i) gate requires one qubit q_i only (1-qubit type). The execution of every quantum gate requires the uninterrupted use of the involved qubits during its processing time, and each qubit q_i can process at most one quantum gate at a time.

TC is a set of simple precedence constraints imposed on the sets *P-S* and *MIX*, such that for each compilation pass $cp \in [1, .., p]$, all *mix* gates that involve a specific qubit q_i must be executed after all the *p-s* gates involving q_i have already been executed. Similarly, every *p-s* gates belonging to the compilation pass *cp* that involves a specific qubit q_i must be executed after the *mix* gate belonging to the compilation pass $cp - 1$ that involves the same qubit q_i has been executed.

At each time during the computation, every qubit q_i is located on a specific node n_j of the quantum chip (see Figure 1). QM is a representation of the quantum hardware as an undirected multi-graph $QM = \langle V, E_{p\text{-}s}, E_{swap}, d_{mix}, d_{p\text{-}s}, d_{swap}\rangle$, where $V = \{n_1, n_2, \ldots, n_k\}$ is the set of qubit locations (nodes), $E_{p\text{-}s}$ (E_{swap}) is a set of undirected edges (n_i, n_j) representing the set of *adjacent* locations to which the qubits q_i and q_j of the gates *p-s*(q_i, q_j) (*swap*(q_i, q_j)) can be assigned. In addition, the *labelling* functions $d_{p\text{-}s} : E_{p\text{-}s} \rightarrow \mathbb{Z}^+$ and $d_{swap} : E_{swap} \rightarrow \mathbb{Z}^+$ represent the durations of the gate operations *p-s*(q_i, q_j) and *swap*(q_i, q_j), respectively, when their qubits are assigned to the corresponding adjacent locations. Similarly, the *labelling* function $d_{mix} : V \rightarrow \mathbb{Z}^+$ represents the durations of the *mix* gate (which can be executed at any location).

For example, with regard to the quantum hardware represented in Figure 1, the duration of each *p-s* gate is $d_{p-s} = 3$ or $d_{p-s} = 4$ depending on the particular edge on which the gate is executed, the duration of each *mix* gate is $d_{mix} = 1$, and the duration of each *swap* gate is $d_{swap} = 2$; Finally, L_0 represents the location $n_i \in V$ of each qubit q_i at $t = 0$ (initial location).

Algorithm 1 Find Feasible Plan

```
Require: An ideal circuit $P$, quantum hardware $QM$
  $S \leftarrow$ InitSolution($P$);
  while not all the $P$-$S$ and $MIX$ operations are inserted in $S$ do
    $op \leftarrow$ SelectExecutableOperation($P$, $S$);
    $S \leftarrow$ InsertOperation($op$, $S$);
  end while
  return $S$
```

A feasible solution S of the Quantum Gate Planning Compilation problem is a tuple $S = \langle SWAP, s\rangle$ where: (i) $SWAP$ is the set of additional *swap*(q_i, q_j) gates added to S to guarantee the adjacency constraints for all the *p-s* gates that must be executed, and (ii) $s : GO \rightarrow [0, H]$, with $GO = P\text{-}S \cup SWAP \cup MIX$, is an a assignment of the gates' start-times satisfying all the above constraints. In particular, the assignments $s(op)$ with $op \in GO$ represent a separate total order imposed on the set of gate operations that use the same qubit q_i; we call this subset the *chain* ch_i for the qubit $q_i \in Q$.

The makespan of a solution S corresponds to the maximum completion time of the plan, that is $makespan(S) = \max_{op \in GO}\{s(op) + d_{op}\}$. An optimal solution S^* is a feasible solution characterized by the minimum makespan.

4 A Greedy Procedure

According to previous Section 3, the introduced solving procedure represents a solution S by using the concept of chain ch_i for each qubit state q_i. Given a partial solution S and the corresponding set of chains ch_i, $last(ch_i)$ is the last operation in the chain ch_i according to the imposed total order and $n(op)$ is the qubit location where the operation op of the solution S is executed. It is worth noting that $n(op)$ is the location at which the execution of the gate op terminates; *p-s*(q_i, q_j) and *mix*(q_i) gates leave unchanged the locations of the qubits q_i and q_j, whereas *swap*(q_i, q_j) gate swaps their locations. Finally, we define the state L_S of a partial solution S as the tuple $L_S = \langle n(last(ch_1)), n(last(ch_1)), \ldots, n(last(ch_N))\rangle$ containing the N last operations according to each chain ordering.

Algorithm 1 finds a feasible solution starting from an ideal circuit P and a quantum hardware QM representation. As a first step, it initialises the partial solution S; in particular, it sets the state L_S to the init value L_0 by initializing the locations of every qubit q_i (i.e., of every chain ch_i) at $t = 0$. The core of the procedure is the function SelectExcutableOperation(), which returns at each iteration either one of the gates in the set $P\text{-}S \cup MIX$ or a *swap*(q_i, q_j) gate necessary to guarantee *nearest-neighbor* compliance as described in the previous Section 3.

The proposed solving strategy proceeds as follows: starting form the solution state $L_S = L_0$, Algorithm 1 ranks the possible set of executable gates according to a determined evaluation function (described below). The gate with the lowest (minimal) evaluation is inserted in the partial solution S at the first available starting time and the state L_S of the partial solution is updated. The process continues iteratively until all the *P-S* and *MIX* gates are inserted in S; at the end, the produced solution will contain a set of additional *swap(q_i, q_j)* gates necessary to satisfy the *nearest-neighbor* constraints.

As introduced above, an optimal solution S^* has the minimal possible makespan; intuitively this results can be obtained by optimizing the level of concurrency among the gates, selecting the proper number of *swap* gates. As we will see shortly in this section, the above mentioned ranking function is specifically devised to reach this objective.

Given the above multi-graph QM, we consider the distance graph $G_d(V, E_{p\text{-}s})$, so as to contain an undirected edge $(n_i, n_j) \in E_{p\text{-}s}$ when QM can execute a *p-s* gate on the pair (n_i, n_j). In the graph G_d, an undirected path p_{ij} between a node n_i and a node n_j is the list of edges $p_{ij} = ((n_i, n_{j1}), (n_{j1}, n_{j2}), \ldots, (n_{jk}, n_j))$ connecting the two nodes n_i and n_j and its lenght l_{ij} is the number of edges in the path p_{ij}. Let d_{ij} represent the minimal length among the set of all the paths between n_i and n_j. Given a $p\text{-}s(q_i, q_j)$ gate requiring two qubits q_i and q_j, its *distance* d^{L_S} w.r.t. the state L_S of the partial solution S is defined as:

$$d^{L_S}(p\text{-}s(q_i, q_j)) = d(n(last(ch_i)), n(last(ch_j))) \tag{2}$$

Hence, given a $p\text{-}s(q_i, q_j)$ gate and a partial solution S, the value $d^{L_S}(p\text{-}s(q_i, q_j))$ yields the minimal number of swaps (in excess of 1) for moving the two qubits q_i and q_j to adjacent locations on the machine QM. If $d^{L_S}(p\text{-}s(q_i, q_j)) = 1$, then the two quibits q_i and q_j are in adjacent locations in the state L^S. The concept of distance defined on a single gate operation $p\text{-}s(q_i, q_j))$ can be extended to a set of gate operations. In particular, let S a partial solution and $\overline{P\text{-}S}^S$ the set of $p\text{-}s(q_i, q_j))$ gates which are not yet scheduled in S, we propose two different functions to measure the *distance* separating the set $\overline{P\text{-}S}^S$ from the *adjacent state*. The first one sums the set of the distances $d^{L_S}(p\text{-}s(q_i, q_j))$:

$$D^S_{sum}(\overline{P\text{-}S}^S) = \sum_{p\text{-}s \in \overline{P\text{-}S}^S} d^{L_S}(p\text{-}s(q_i, q_j)) \tag{3}$$

The second one returns the minimal value of the distance $d^{L_S}(p\text{-}s(q_i, q_j))$ in the set $\overline{P\text{-}S}^S$:

$$D^S_{min}(\overline{P\text{-}S}^S) = MIN_{p\text{-}s \in \overline{P\text{-}S}^S} d^{L_S}(p\text{-}s(q_i, q_j)) \tag{4}$$

Subsequently, we propose a two-dimensional distance function:

$$D^S_{min}(\overline{P\text{-}S}^S) = (D^S_{sum}(\overline{P\text{-}S}^S), D^S_{min}(\overline{P\text{-}S}^S)) \tag{5}$$

Finally, given the above two-dimensional distance function, we can evaluate the impact of the selection of a gate operation *op* on the whole solution S, according to the following lexicographic ranking function:

$$\mathit{\Delta}(S, op, \overline{P\text{-}S}^S) = \begin{cases} (D^S_{sum}(\overline{P\text{-}S}^S \setminus \{op\}), 1) & op \text{ is a p-s gate;} \\ (D^S_{sum}(\overline{P\text{-}S}^S), 1) & op \text{ is a mix gate;} \\ (D^S_{sum}(\overline{P\text{-}S}^S), D^S_{min}(\overline{P\text{-}S}^S)) & op \text{ is a swap gate.} \end{cases} \quad (6)$$

Hence, in Algorithm 1, given a partial solution S, the function SelectExecutableOperation() returns one gate operation *op* (*p-s*, *mix* or *swap*) which minimises the value $\mathit{\Delta}(S, op, \overline{P\text{-}S}^S)$ according to a lexicographic ordering.

We observe that the two-dimension distance function used for the gate selection ranking has a twofold role. The D_{sum} component acts as a **global** closure metric; by evaluating the overall distance left to be covered by all the qubits still involved in gates yet to be executed, it guides the selection towards the gate that best favors the efficient execution of the remaining gates. Conversely, the D_{min} component acts as a **local** closure metric, in that it favors the mutual approach of the closest qubit pairs. In more details, the role played by the D_{min} component is essential as a "tie-breaker", to avoid the selection of swap gates that may induce deadlock situations (cycles), which are possible in case we based our ranking solely on the D_{sum} component.

5 A Randomized Approach

The FindFeasiblePlan() resolution procedure, as defined above, is a deterministic polynomial solution procedure able to find a solution to the problem of compiling Quantum Approximate Optimization Algorithms (QAOA) defined in Section 4. To provide a capability for expanding the search in such cases without incurring the combinatorial overhead of a conventional backtracking search, we define a random counterpart of our conflict selection heuristic (in the style of [1–3]) and embed the result within an iterative random sampling search framework. This choice is motivated by the observation that in many cases systematic backtracking search can explore large subtrees without finding any solution. On the other hand, if we compare the whole search tree created by a systematic search algorithm with the non systematic tree explored by repeatedly restarting a randomised search algorithm, we see that the randomized procedure is able to reach "different and distant" leaves in the search tree. This latter property could be an advantage when problem solutions are uniformly distributed within the set of search tree leaves interleaved with large subtrees which do not contain any problem solution.

To make FindFeasiblePlan() suitable to random greedy restart its function SelectExecutableOperation() is modified according to the following rationale. The core FindFeasiblePlan() procedure is transformed into a random procedure by redefining SelectExecutableOperation() to proceed in two-steps: (1) at each solution step, a set of "equivalent" gate operations (*p-s*, *mix* or *swap*) are first identified, and then (2) one of these is randomly selected. As in the deterministic variant, the selected gate operation *op* is then inserted

Algorithm 2 Greedy Random Sampling

```
Require: An ideal circuit $P$, quantum hardware $QM$, stop criterion
  $S_{best} \leftarrow$ FindFeasiblePlan($P$, $QM$);
  while (stopping criterion not satisfied) do
    $S \leftarrow$ FindFeasiblePlan($P$, $QM$)
    if (makespan(S) < makespan($S_{best}$)) then
      $S_{best} \leftarrow S$;
    end if
  end while
  return ($S_{best}$)
```

in the current partial solution. The set of equivalent operations is created by identifying one operation *op** associated with the minimal lexicographic value $\Delta(S, op^*, \overline{P\text{-}S}^S) = (D^*_{sum}, D^*_{min})$ and by considering equivalent to *op** all the operations *op* such that $\Delta(S, op, \overline{P\text{-}S}^S) = (D_{sum}, D_{min})$ with $D_{sum} = D^*_{sum}$ and $D_{min} = D^*_{min}$. Subsequently, the operation to be inserted in the partial solution is randomly selected from this set, resulting in a non-deterministic yet heuristically-biased choice. Successive calls to FindFeasiblePlan() are intended to explore heuristically equivalent paths through the search space.

Algorithm 2 depicts the complete iterative sampling algorithm for generating a feasible solution, which is designed simply to invoke the FindFeasiblePlan() resolution procedure until a stopping criterion not satisfied. Given that the broader objective in this paper is makespan minimization, each restart provides a new opportunity to produce a different feasible solution with lower makespan.

6 Experiments

In this section, we present the results obtained with our Gredy Random Sampling (GRS) procedure against the same quantum circuit benchmark set utilized in [7]. The benchmark is composed of instances of three different sizes, based on quantum chips with $N = 8, 21$ and 40 qubits, respectively (see Figure 1). The production of these chips should be concretely realized within the next two years. In [7], the authors base their experimentation on two problem classes for each chip size, depending on the number of passes to be executed during compilation ($p = 1$ or $p = 2$). In this work, we will mostly focus on the $p = 2$ problem class because it is the most computationally challenging; we will analyze the $p = 1$ case for the $N = 40$ qubit size only, exclusively for comparison purposes with the results obtained in [7].

The utilized benchmark[2] contains 100 different problem instances for each chip size, where each instance is representative of a graph G to be partitioned by

[2] In this work we use the same benchmark employed in [7], which is available at: `https://ti.arc.nasa.gov/m/groups/asr/planning-and-scheduling/VentCirComp17\_data.zip`

the *MaxCut* procedure to be realized (an example of graph is given in Figure 2). The benchmark instances are divided in two subsets composed of 50 instances each, depending on the "utilization level" (u) of the available qstates over the circuit. In particular, the 50 instances characterized by $u = 0.9$ are built randomly choosing 90% of the available qstates to allocate over the N edges of the instance graph G, while the other 50 instances ($u = 1.0$) are built by possibly allocating *all* the qstates over the N edges of the graph G. Larger sizes and higher p values will lead to more complex problem instances.

6.1 Setup

Our experimental campaign is organized as follows. As anticipated earlier, we will mainly focus on the complete benchmark instences, i.e., those characterized by two compilation passes ($p = 2$). In particular, we solve the following instance sets:

- **N8_u0.9 set**: 50 instances characterized by $N = 8$ qubits, $u = 0.9$ occupation level, $p = 2$ compilation passes;
- **N8_u1.0 set**: 50 instances characterized by $N = 8$ qubits, $u = 1.0$ occupation level, $p = 2$ compilation passes;
- **N21_u0.9 set**: 50 instances characterized by $N = 21$ qubits, $u = 0.9$ occupation level, $p = 2$ compilation passes;
- **N21_u1.0 set**: 50 instances characterized by $N = 21$ qubits, $u = 1.0$ occupation level, $p = 2$ compilation passes;
- **N40_u0.9 set**: 50 instances characterized by $N = 40$ qubits, $u = 0.9$ occupation level, $p = 2$ compilation passes;
- **N40_u1.0 set**: 50 instances characterized by $N = 40$ qubits, $u = 1.0$ occupation level, $p = 2$ compilation passes;

In addition, we solve the following instance sets characterized by $p = 1$ for comparison purposes with previous results obtained in [7]:

- **N40_u0.9_P1 set**: the same 50 instances belonging to the **N40_u0.9 set**, solved considering $p = 1$ compilation passes;
- **N40_u1.0_P1 set**: the same 50 instances belonging to the **N40_u1.0 set**, solved considering $p = 1$ compilation passes.

Given that the previous sets are characterized by problem instances of increasing size, we have allotted different CPU time limits for each set, as follows. All runs relatively to the **N8*** and **N40_u*_P1** sets are limited to max 60 seconds each; all runs relatively to the **N21** and **N40** sets are limited to max 900 seconds each. 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.

6.2 Results

The plots shown in this section present the results obtained by means of our GRS procedure, compared with the results obtained in [7]. In each figure, the

Table 1. Results obtained from the **N40_0.9_P1** and **N40_1.0_P1** sets

N40_0.9_P1 # ist.	**SGPlan** (60 mins)	**LPG** (60 mins)	**GRS** (1 min)	**N40_1.0_P1** # ist.	**SGPlan** (60 mins)	**LPG** (60 mins)	**GRS** (1 min)
1	95		**64**	1	76	56	**52**
2	97	84	**61**	2	108		**66**
3	97		**68**	3	120		**60**
4	120		**65**	4	125		**83**
5	86		**65**	5	96	67	**66**
6	81		**57**	6	86		**61**
7	99		**68**	7	87		**64**
8	103	96	**65**	8	85		**67**
9	88	68	**62**	9	106		**65**
10	129		**63**	10	103		**65**
11	109		**72**	11	100		**61**
12	114		**66**	12	77		**64**
13	117	78	**70**	13	93		**66**
14	98		**68**	14	115		**66**
15	93		**60**	15	121		**61**
16	118		**58**	16	101		**59**
17	130		**65**	17	99		**64**
18	114		**76**	18	109		**70**
19	104		**60**	19	90	69	**48**
20	121	77	**70**	20	110		**75**
21	98		**59**	21	98	82	**65**
22	128		**68**	22	105		**61**
23	115		**66**	23	95		**67**
24	113		**72**	24	74	86	**59**
25	98		**69**	25	99		**72**
26	126		**66**	26	94	85	**56**
27	105		**68**	27	98		**60**
28	115		**59**	28	110		**75**
29	105		**52**	29	100		**60**
30	91		**64**	30	114		**54**
31	98		**60**	31	105	91	**59**
32	99	91	**77**	32	92		**47**
33	95		**64**	33	89	67	**61**
34	81		**64**	34	118	66	**60**
35	94	81	**67**	35	102	63	**54**
36	129		**78**	36	88		**62**
37	116		**61**	37	99	67	**65**
38	93		**49**	38	73		**59**
39	104		**53**	39	101		**71**
40	138		**68**	40	87		**60**
41	114		**62**	41	91		**65**
42	110		**68**	42	117		**63**
43	95		**62**	43	99	83	**62**
44	130		**75**	44	110	72	**64**
45	122		**73**	45	79		**61**
46	90	90	**66**	46	80	64	**65**
47	134		**72**	47	93		**56**
48	94	89	**56**	48	114		**59**
49	94		**53**	49	112		**68**
50	99	66	**63**	50	76		**61**

x-axis represents the benchmark instance id (ranging from 1 to 50), while the y-axis represents the makespan value obtained.

In particular, Figure 3 and Figure 4 report the results obtained solving the **N8_u0.9** and the **N8_u1.0** benchmark sets, respectively. The figures compare the results returned by the GRS procedure (GRS plot) with the results returned by the *Temporal FastDownward* (TFD plot) planner (see [11]), as this is the planner returning the best results in [7]. In addition, the Init plot reports the the initial values returned by the GRS procedure, i.e., before the optimization cycle starts, in order to give a visual hint of the optimization efficacy of the GRS. For these results, our GRS procedure has been allowed a max CPU time of 1 minute for each run, against the 10 minutes allowed for the TFD planner. As a summary of the performances obtained in the $N8\_u0.9$ case, the GRS procedure improved 37/50 solutions (74%), left 10/50 (20%) unchanged, and was outperformed on 3/50 (6%) solutions. The average makespan difference computed over the improved solutions (average makespan improvement) is 3.83, while the average makespan difference over the unimproved solutions is 1.3, demonstrating that the quality of the unimproved solutions is yet not too distant from the best results. In the $u = 1.0$ case, the GRS procedure improved 41/50 solutions (82%), left 7/50 (14%) unchanged, and was outperformed on 2/50 (4%) solutions. The average makespan difference computed over the improved solutions is 3.95, while the average makespan difference over the unimproved solutions is 2.

The same kind of analysis has been carried out in Figure 5 and Figure 6 for the **N21_u0.9** and the **N21_u1.0** benchmark sets, respectively. For these results, our GRS procedure has been allowed a max CPU time of 15 minutes for each run, against the 60 minutes allowed for the TFD planner. As a summary of the performances, in the $u = 0.9$ case the GRS procedure improved 46/50 solutions (92%), left 1/50 (2%) unchanged, and was outperformed on 3/50 (6%) solutions. The average makespan difference computed over the improved solutions (average makespan improvement) is 9.87, while the average makespan difference over the unimproved solutions is 1.3, again demonstrating that the quality of the unimproved solutions is not too distant from the best results. In the $u = 1.0$ case, the GRS procedure improved 40/50 solutions (80%), left 3/50 (6%) unchanged, and was outperformed on 7/50 (14%) solutions. The average makespan difference computed over the improved solutions is 11.32, while the average makespan difference over the unimproved solutions is 4.8.

As opposed to the previous figures, Figure 7 and Figure 8 offer no comparative analysis, as no planner in [7] succeded in solving any $N40$ instance with $p = 2$ within the max allotted time of 60 minutes. Therefore, we only show the results obtained with our GRS procedure (max allowed time = 15 mins), and compare such results against the makespan value of the initial solution found (Init plot), acknowledging an average makespan improvement of 33.26 and 28.14, in the $u = 0.9$ and $u = 1.0$ case, respectively.

In order to test the performances of our GRS procedure against the results obtained in [7] for the $N40$ instances, Figure 9 and Figure 10 show the results for the **N21_u1.0** and the **N40_u1.0_P1** benchmark sets respectively, both in the $p = 1$ case. In particular, the figures compare the results obtained by means of our GRS procedure against those obtained in [7] with the SGPlan planner ([12, 13]) (SGPlan plot) and the LPG planner ([14]) (LPG plot), respectively. As shown, The LPG planner seems to achieve better results over the SGPlan planner, even though it succeeds in solving only a strict minority of the instances. However, as a remarkable result, all the 50 instances have been improved by the GRS procedure, for both the $u = 0.9$ and the $u = 1.0$ case. To summarize the performances, the average makespan improvement obtained by GRS over SGPlan is 41.98 and 35.74 for $u = 0.9$ and $u = 1.0$ respectively, while the average makespan improvement obtained by GRS over LPG (computed on the subset of instances solved by LPG) is 16.3 and 13.14 for $u = 0.9$ and $u = 1.0$ respectively. Remarkably, it should be underscored that all runs executed by GRS have been allotted a max CPU time of 1 minute, against the 60 minutes allowed

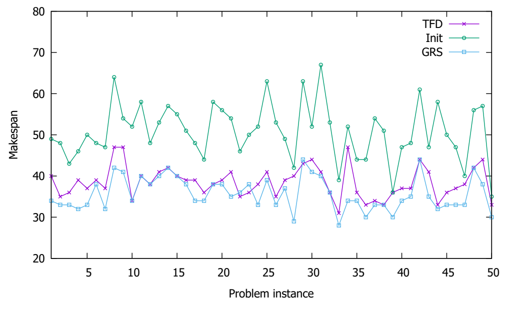


Fig. 3. Results from the **N8_u0.9** set

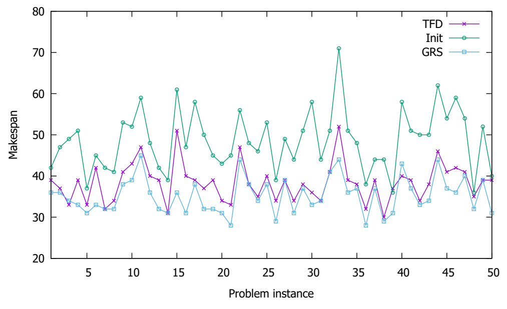


Fig. 4. Results from the **N8_u1.0** set

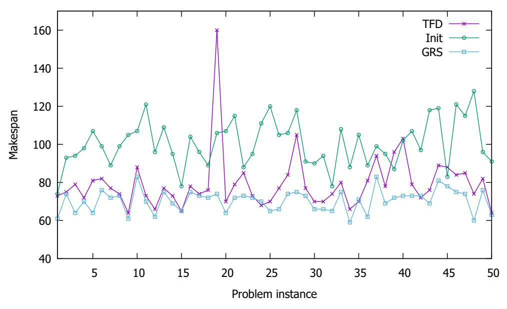


Fig. 5. Results from the **N21_u0.9** set

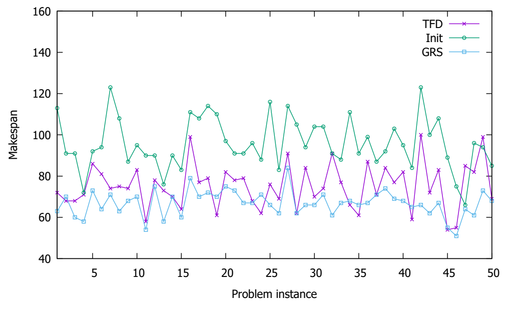


Fig. 6. Results from the **N21_u1.0** set

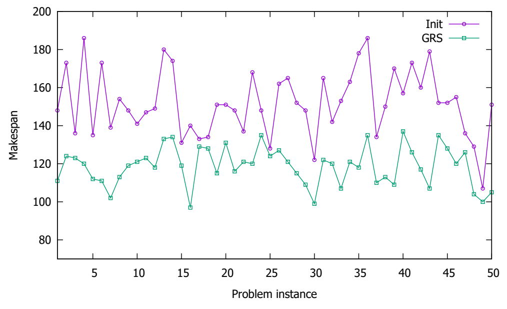


Fig. 7. Results from the **N40_u0.9** set

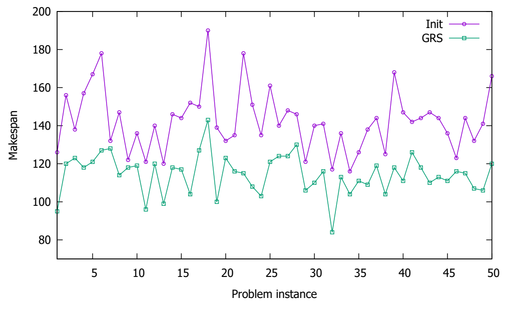


Fig. 8. Results from the **N40_u1.0** set

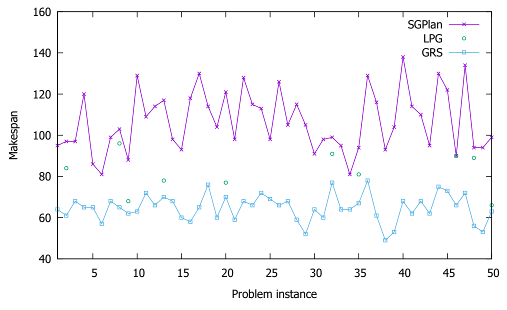


Fig. 9. Results from the **N40_u0.9_P1** set

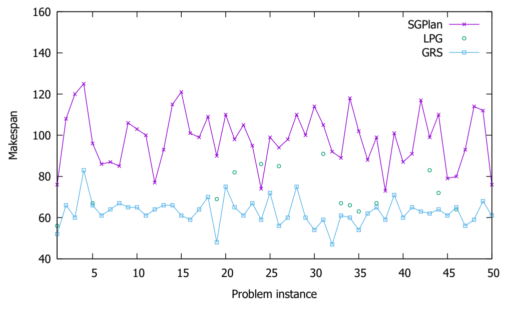


Fig. 10. Results from the **N40_u1.0_P1** set

for the SGPlan and LPG planners, as a further prove of the good performances exhibited by the GRS procedure.

Finally, to the aim of providing the reader with more precise results, Table 1 reports all the makespan values obtained for the **N40_u0.9_P1** and the **N40_u1.0_P1** benchmark sets[3].

7 Concluding Remarks

In this work we propose a greedy random search heuristic to solve the quantum circuit compilation problem, where the objective is essentially to synthesize a quantum gate execution plan characterized by a minimum makespan. We test our procedure against a number of instances from a benchmark repository publicly available, and compare our results with those obtained in a recent work where the same problem is solved by means of PDDL-based planning technology, showing that our procedure is more performing in the vast majority of cases. Despite the very good results, we consider our present contribution to the quantum compilation problem complementary to the PDDL approach. The take-home message from this comparison can in fact be wrapped up as follows. On the one hand, it is confirmed that tackling a problem with a general technique such as PDDL can be less rewarding in terms of overall solution quality than employing heuristics more tailored on the problem; such heuristics, though very simple, can remain extremely efficient when the problem size scales up significantly. On the other hand, it remains true that in more complex domains, a general approach such as PDDL-based planning could still represent a winning factor, also considering that some of the solutions we compared against have demonstrated to be of very high quality. Our conclusion is that an integration between the two techniques might be beneficial in order to enjoy the representational generality of PDDL planning, without renouncing the exploration/exploitation power of state-of-the-art constraint-based metaheuristics.

References

1. Hart, J., Shogan, A.: Semi-greedy heuristics: An empirical study. Operations Research Letters **6** (1987) 107–114
2. Resende, M.G., Werneck, R.F.: A hybrid heuristic for the p-median problem. Journal of Heuristics **10**(1) (Jan 2004) 59–88
3. Oddi, A., Smith, S.: Stochastic Procedures for Generating Feasible Schedules. In: Proceedings 14th National Conference on AI (AAAI-97). (1997) 308–314
4. Nielsen, M.A., Chuang, I.L.: Quantum Computation and Quantum Information: 10th Anniversary Edition. 10th edn. Cambridge University Press, New York, NY, USA (2011)

[3] We could not provide the complete set of tables for obvious reasons of space. In case the paper is accepted, we will make available on a web site the complete set of makespan values obtained, together with the complete set of solutions.

5. Farhi, E., Goldstone, J., Gutmann, S.: A quantum approximate optimization algorithm. arXiv preprint arXiv:1411.4028 (November 2014)
6. Sete, E.A., Zeng, W.J., Rigetti, C.T.: A functional architecture for scalable quantum computing. In: 2016 IEEE International Conference on Rebooting Computing (ICRC). (Oct 2016) 1–6
7. Venturelli, D., Do, M., Rieffel, E., Frank, J.: Temporal planning for compilation of quantum approximate optimization circuits. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17. (2017) 4440–4446
8. Fox, M., Long, D.: Pddl2.1: An extension to pddl for expressing temporal planning domains. J. Artif. Int. Res. **20**(1) (December 2003) 61–124
9. Nau, D., Ghallab, M., Traverso, P.: Automated Planning: Theory & Practice. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA (2004)
10. Maslov, D., Falconer, S.M., Mosca, M.: Quantum circuit placement: Optimizing qubit-to-qubit interactions through mapping quantum circuits into a physical experiment. In: Proceedings of the 44th Annual Design Automation Conference. DAC '07, New York, NY, USA, ACM (2007) 962–965
11. Eyerich, P., Mattmüller, R., Röger, G.: Using the context-enhanced additive heuristic for temporal and numeric planning. In: Proceedings of the 19th International Conference on Automated Planning and Scheduling, ICAPS 2009, Thessaloniki, Greece, September 19-23, 2009. (2009)
12. Wah, B.W., Chen, Y.: Subgoal partitioning and global search for solving temporal planning problems in mixed space. International Journal on Artificial Intelligence Tools **13**(04) (2004) 767–790
13. Chen, Y., Wah, B.W., Hsu, C.W.: Temporal planning using subgoal partitioning and resolution in sgplan. J. Artif. Int. Res. **26**(1) (August 2006) 323–369
14. Gerevini, A., Saetti, A., Serina, I.: Planning through stochastic local search and temporal action graphs in lpg. J. Artif. Int. Res. **20**(1) (December 2003) 239–290