Temporal Planning for Compilation of Quantum Approximate Optimization Algorithm Circuits

Davide Venturelli, Minh Do, Eleanor Rieffel, Jeremy Frank

Abstract

We investigate the application of temporal planners to the problem of compiling quantum circuits to newly emerging quantum hardware. While our approach is general, we focus our initial experiments on Quantum Approximate Optimization Algorithm (QAOA) circuits that have few ordering constraints and allow highly parallel plans. We report on experiments using several temporal planners to compile circuits of various sizes to a realistic hardware. This early empirical evaluation suggests that temporal planning is a viable approach to quantum circuit compilation.

1 Introduction

We explore the use of temporal planners to optimize compilation of quantum circuits to newly emerging quantum hardware. Over the last few decades, stunning instances of quantum algorithms that provably outperform the best classical algorithms have been designed, but only now is prototype hardware on which to run them emerging. IBM recently provided public access to a 5-qubit processor through the cloud (IBM 2017), and scalable quantum computing architectures are being manufactured as well by other groups such as TU Delft (Versluis et al. 2016), Rigetti Computing (Sete, Zeng, and Rigetti 2016), and Google (Boxio 2016). All cited groups have announced plans to build gate-model quantum processors with 40 or more qubits in the near term, as well. Previously, only special purpose quantum hardware was available - quantum annealers targeting optimization problems. Gate-model computing expands the potential applications beyond optimization, as well as enabling a broader array of quantum approaches to optimization.

Like classical algorithms, quantum algorithms must be compiled into a set of elementary machine instructions (gates) applied at specific times in order to run them on quantum computing hardware. Quantum algorithms are often specified as quantum circuits on idealized hardware since the physical hardware constraints vary from architecture to architecture. For example, emerging gate-model quantum computer hardware based on superconducting qubits have planar architectures which impose nearest-neighbor restrictions on the memory locations (qubits) to which the gates can be applied. For this reason, compiling quantum circuits to specific hardware requires adding new gates that move qubits to gates that can act on them. Optimizing this compilation process is a challenging problem due to the parallel execution of gates with different durations. Further, for quantum circuits with more flexibility in when the gates can be applied, or when some gates can be applied in a different order while still achieving the same computation, the search space for feasible compilations is larger than for less flexible circuits. Even though it's more challenging to find the optimal compilation on circuits with more flexibility, there is also a greater potential win from improved compilation optimization than for less flexible circuits. While there has been active development of software libraries to synthesize and compile quantum circuits from algorithm specifications (Wecker and Svore 2014) (Smith, Curtis, and Zeng 2016) (Steiger, Häner, and Troyer 2016) (Devitt 2016) (Barends et al. 2016), few approaches have been explored for compiling idealized quantum circuits to realistic quantum hardware (Beals et al. 2013) (Brierly 2015) (Bremner, Montanaro, and Shepherd. 2016), leaving the problem open for innovation. An analogous issue arising when compiling classical programs is the *register allocation* problem, in which program variables are assigned to machine registers to improve execution time; this problem reduces to graph coloring (Fu, Wilken, and Goodwin 1960).

In this paper, we use temporal planning techniques to solve the compilation of quantum algorithm on the gate-model quantum machine. Specifically, we model machine instructions as PDDL2.1 durative actions, enabling domain-independent temporal planners to find a parallel sequence of conflict-free instructions that when executed can achieve what the high-level quantum algorithm intends to achieve. While our approach is general, we focus our initial experiments on circuits that have few ordering constraints and thus allow highly parallel plans. We report on experiments using a variety of temporal planners to compile circuits of various sizes to an architecture inpired by those currently being built. This early empirical evaluation suggests that temporal planning is a viable approach to quantum circuit compilation.

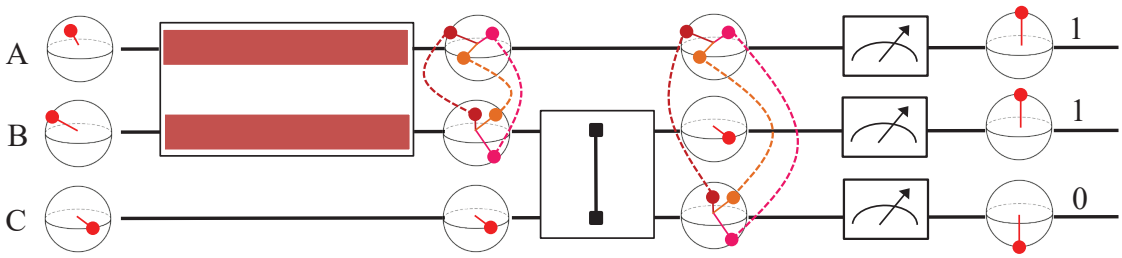

Figure 1: A generic representation of an example quantum circuit involving three qubits. Lines indicate the flow of time from left to right. The box with the red stripes represents a two-qubits gate which generically entangles A and B, generating co-existing alternatives of combination of quantum states. The second box is a gate that swaps the physical location of B and C, preserving the correlations (pictorialized as dashed lines). The third operation is the end of the algorithm where the qubits are individually measured. The quantum states and the correlations determines the probability that the final results is a particular bit-string among the $2^3 = 8$ possible results.

2 Background

Quantum algorithms process information stored in qubits, the basic memory unit of quantum processors. Quantum gates are the building blocks of quantum algorithms, just as instructions on registers are the building blocks of classical algorithms. For the purposes of this paper, one can formalize the compilation problem for quantum circuits entirely classically in terms of memory locations (qubits) and gates (operations on memory locations) together with specific constraints on them. In the rest of this section, we give a brief review of gate-model computing. Impatient readers can skip to Section 3; for a review of quantum computing, see (Rieffel and Polak 2011).

A qubit stores a quantum state, specified by three real numbers, corresponding to a vector in 3D space, with the up and down corresponding to classical bit values 0 and 1. These values, encoding *probability amplitudes*, are not addressable directly, but the qubit states can be fed as input to elementary *quantum gates*, operations that change the state of either a single qubit or a pair of qubits jointly.

In the process of executing the gates, the state of the entire set of qubits becomes correlated, even quantum correlated (i.e. *entangled*). Even when quantum correlations exist, for the algorithms we consider, we obtain a classical output by probabilistically projecting each quantum state to up or down (0 or 1). The objective of such quantum algorithms is to obtain, with high probability, a state which can be associated to a classical bitstring that solves the problem of interest. Quantum computational hardware suffers from "decoherence" which degrades the performance of quantum algorithms over time. Especially for near-term hardware, which will not be able to mitigate decoherence, it is important to *minimize* the *duration of the circuit* that carries out the quantum computation to minimize the decoherence experienced by the computation.

3 Architecture-specific compilation problem

Gate-model quantum computers work on a clock, like digital classical computers. In a quantum-circuit, a gate has a start time and a duration in integer units of the clock cycle. Quantum circuits for general quantum algorithms (see Fig. 1) are often described in an idealized architecture in which any 2-qubit gate can act on any pair of qubits. Physical constraints impose restrictions on which pairs of qubits support gate interactions in an actual physical architecture. For superconducting qubit architectures, qubits in a quantum processor can be thought of as nodes in a planar graph, and 2-qubit quantum gates are associated to edges. Gates can operate concurrently when they do not operate on the same qubits[1]. Furthermore, there are different types of quantum gates, each taking different durations, with the duration depending on the specific physical implementation.

In order for the computation specified by the idealized circuit to be carried out, we require a particular type of 2-qubit gate, the *swap* gate, which exchanges the state of two qubits. A sequence of swap gates moves the contents of two distant qubits to a gate where a desired operation can be carried out. Swap gates may be available only on a subset of edges in the hardware graph, and swap duration may depend on where they are located. For the purposes of this study, we will consider the case in which swap gates are available between any two adjacent qubits on the chip and all swap gates have the same duration, but our approach can handle the more general cases.

Compilation examples: Figure 2 shows a hypothetical chip design that we will use for our experiments on circuit compilation. It is inspired by the architecture of the machine envisioned by Rigetti Computing Inc. (Sete, Zeng, and Rigetti 2016). Qubits are labeled with n_i and the colored edges indicate the types of 2-qubit gates available, in this case swap gates and two other types of 2-qubit gate (further described in Section 4). Given an idealized circuit consisting only of the non-swap gates, used to define general quantum algorithms, the circuit compilation problem is to find a new architecture-specific circuit by adding swap gates and ordering gates when required. The

[1]There may be additional restriction on which operations can be done currently (e.g. see paper by Google Inc. (Boxio 2016)).

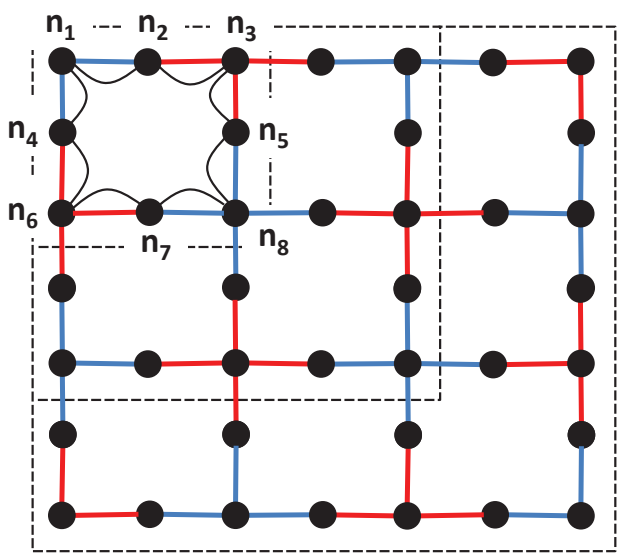

Figure 2: A schematic for the hypothetical chip design used in our numerical experiments, with available 2-qubit gates represented by colored arcs in a weighted multigraph. Each color is associated to a specified, distinct gate-type and duration: SWAP gates (black) and two other types of 2-qubits gates (red and blue). The 1-qubit gates are present at each qubit (black dot). Dashed boxes indicate the 3 different chip sizes used in our empirical evaluation (see Sec. 6). For visual clarity, only the label locations and the SWAP-gates for the smaller chip size are shown.

main objective is to minimize the overall duration to execute all gates in a new circuit.

To illustrate the challenges of finding effective compilation, we present some concrete examples, with reference to the 8-qubit section in the top left of Fig. 2. Suppose that at the beginning of the compilation, each qubit location n_i is associated to the qubit state q_i. Let us also assume that the idealized circuit requires the application of a red gate to the states q_2 and q_4, initially located on qubits n_2 and n_4. One way to achieve this task would be to swap the state in n_4 with n_1, while at the same time swapping n_2 with n_3. Another swap, between n_1 and n_2, positions q_4 in n_2 where a red-gate connects it to q_2 (which is now in n_3).

The sequence of gates to achieve the stated goal are:

$$\{\text{SWAP}_{n_4,n_1}, \text{SWAP}_{n_2,n_3}\} \rightarrow \text{SWAP}_{n_1,n_2} \rightarrow \text{RED}_{n_2,n_3} \equiv \text{RED}(q_2, q_3) \quad (1)$$

The first line refers to the sequence of gate applications, while the second corresponds to the algorithm objective specification (a task defined over the qubit states). The sequence in Eq. (1) takes $2\tau_{swap} + \tau_{red}$ clock cycles where $\tau_\star$ represents the duration of the $\star$-gate.

As the second example, the idealized circuit requires $\text{BLUE}(q_1, q_2) \wedge \text{RED}(q_4, q_2)$, in no particular order. If $\tau_{blue} > 3 \times \tau_{swap}$, the compiler might want to execute BLUE_{n_1,n_2} while the qubit state q_4 is swapped all the way clockwise in five SWAPs from n_4 to n_3 where RED_{n_2,n_3} can be executed. However, if $\tau_{swap} < 3 \times \tau_{blue}$, it is preferable to wait until the end of BLUE_{n_1,n_2} and then start to execute the instruction sequence in Eq. (1).

Problem definition: An idealized quantum circuit consists of a set of nodes (qubits), which can be thought of as memory locations, and a specification of start times of operations (gates), each acting on a single node or a pair of nodes. Operations have specified durations and there exist specifications to operations whose time order can be reversed, either individually or as blocks.

Ideal to hardware-specific quantum circuit compilation problem: The problem input is an idealized quantum circuit and a hardware multigraph and the output is a hardware-specific circuit that implements the idealized quantum circuit. The objective is to minimize the makespan (the circuit duration) of the resulting circuit.

4 Compiling QAOA for the MaxCut problem

While our approach can be used to solve a wide range of quantum circuits and architectures, in this paper we concentrate on one particular case: QAOA circuits for MaxCut on an architecture inspired by the machine manufactured by Rigetti Computing Inc. (Sete, Zeng, and Rigetti 2016). As described below, there are a relatively small number of gate ordering constraints for this architecture. We choose to work with QAOA circuits because they have many gates that commute with each other (i.e., no ordering enforced). Such flexibility in the ordering of the gates means that the compilation search space is larger than for other circuits. This makes finding the optimal compilation challenging, but there is great potential from improved compilation optimization, compared to other classes of circuits. QAOA circuits have also been the focus of recent research (Farhi, Goldstone, and Gutmann. 2014a) (Farhi, Goldstone, and Gutmann. 2014b) (Farhi and Harrow 2016) (Wecker, Hastings, and Troyer 2016) (Yang et al. 2016) (Guerreschi and Smelyanski 2017) (Jiang, Rieffel, and Wang 2017) in the quantum computing community since their introduction by Farhi *et al.* in (Farhi, Goldstone, and Gutmann. 2014a). We chose (Sete, Zeng, and Rigetti 2016) (see Figure 2) as a template because it has two different classes of qubit pair gates, red and blue. The durations we assign to the gates are not derived from actual designs, but are realistic and serve to illustrate possible future designs.

Idealized QAOA circuits alternate between a *phase separation* phase (PS) and a *mixing* phase. The phase-separation phase for QAOA for MaxCut (defined in the later part of this section) is simpler than for other optimization problems, consisting of a set of identical 2-qubit gates that must be applied between certain pairs of qubits depending on the graph of the MaxCut instance under consideration. We will refer to these as *p-s* gates, and the main goal of the compilation is to carry them out. The p-s gates all commute with each other, implying that they can be carried out in any order (subject to constraints on the chip, as noted previously). In the mixing phase, a set of

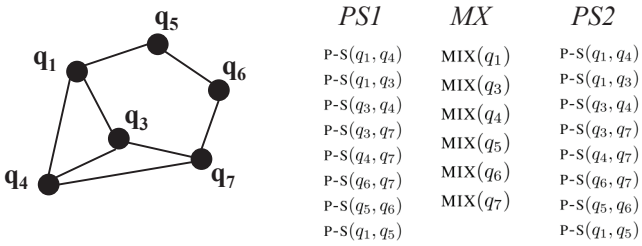

Figure 3: Example of a 6-vertex MaxCut problem on a randomly generated graph (qstates q_2 and q_8 are not appearing in this instance). The association of quantum states to every node allows the definition of the compilation objectives in terms of gates, as exemplified for QAOA $p = 2$.

1-qubit operations are applied, one to each qubit. All p-s gates that involve a specific qubit q must be carried out before the mixing operator on q can be applied. These two phases are repeated p times. We consider $p = 1$ and $p = 2$ in our experiments (detailed in Section 6).

The constraints on the compilation problem can be understood, with reference to Fig. 2, as:

- SWAP gates are located at every edge with $\tau_{swap} = 2$.
- there are two kind of non-swap gates: P-S gates are 2-qubit gates and MIX gates are 1-qubit gates.
- P-S gates are located at every edge of the grid, but their duration τ_{p-s} can be 3 or 4 depending on their location (respectively blue or red edges in Fig.2).
- MIX gates are located at every vertex with τ_{mix}=1.
- In an initialization stage, which is not considered as part of the compilation problem, a quantum state is assigned to each qubit.

MaxCut Problem: Given a graph $G(V, E)$ with $n = |V|$ vertices and $m = |E|$ edges. The objective is to partition the graph vertices into two sets such that the number of edges connecting vertices in different sets is maximized.

For every vertex $i \in V$, QAOA for MaxCut requires a quantum state q_i to be assigned on a qubit on the chip, and for every edge $(i, j) \in E$, the PS step of QAOA requires executing a gate corresponding to P-S(q_i, q_j). In Fig. 3 an idealized circuit for QAOA Maxcut with $p = 2$ is shown on a 6-vertex MaxCut instance. We ignore the final mixing phase since it is trivial to compile by just applying the 1-qubit mixing gate to each qubit as the last operation.

5 Compilation of a Quantum Circuit as Temporal Planning Problem

Planning is the problem of finding a conflict-free set of actions and their respective execution times that connects the *initial-state* I and the desired *goal state* G. We now introduce some key concepts that provide the background for the compilation of QAOA to a temporal planning problem.

Planning Domain Description Language (PDDL): PDDL is a modeling language that was originally created to standardize the input for planners competing in the International Planning Competition (IPC). Over time, it has become the de-factor standard modeling languages used by many domain-independent planners. We use PDDL 2.1, which allows the modeling of temporal planning formulation in which every action a has duration d_a, starting time s_a, and end time $e_a = s_a + d_a$. Action conditions *cond*(a) are required to be satisfied either (i) instantaneously at s_a or e_a or (ii) required to be true starting at s_a and remain true until e_a. Action effects *eff*(a) may instantaneously occur at either s_a or e_a. Actions can execute when their temporally-constrained conditions are satisfied, and when executed, will cause state-change effects. The most common objective function in temporal planning is to minimize the plan *makespan*, i.e. the shortest total plan execution time. This objective matches well with the objective of our targeted quantum circuit compilation problem. To enable reuse of key problem features present in an ensemble of similar instances, the PDDL model of a planning problem is separated into two major parts: (i) the *domain* description that captures the common objects and behaviors shared by all problem instances of this planning domain and (ii) the *problem instance* description that captures the problem-specific objects, initial state, and goal setting for each particular problem.

PDDL Planners: a planner takes as input the PDDL domain and problem descriptions and returns a valid plan if one exists. Many different approaches have been implemented to find a viable plan, among them: (i) heuristically search over the possible valid plan trajectories or over the library of partial plans or (ii) compile the planning problem into another combinatorial substrate (e.g., SAT, MILP, CSP) and feed the problem to off-the-shelf solvers.

Modeling Quantum Gate Compilation in PDDL 2.1: PDDL is a flexible language that offers multiple alternative ways to model a planning problems. These modeling choices can greatly affect the performance of existing PDDL planners. For instance, many planners pre-process the original domain description before building plans; this is time-consuming, and may produce large 'ground' models depending on how action templates were written. Also, not all planners can handle all PDDL language features effectively (or even at all). For this project, we have iterated through different modeling choices with the objective of constructing a PDDL model that: (i) contains a small number of objects and predicates for compact model size; (ii) uses action templates with few parameters to reduce preprocessing effort; while (iii) ensuring that the model can be handled by a wide range of existing PDDL temporal planners.

At the high-level, in our domain, we need to model: (i) conceptually how actions representing P-S, SWAP, and MIX gates affect qubits and qubit states (qstate); (ii) the actual qubits and qstates involved with a particular compilation problem, their initial locations and final goal requirements,

```
(:durative-action swap_1_2
    :parameters (?q1 - qstate ?q2 - qstate)
    :duration (= ?duration 2)
    :condition
        (and (at start (located_at_1 ?q1))
             (at start (located_at_2 ?q2)))
    :effect
        (and (at start (not (located_at_1 ?q1)))
             (at start (not (located_at_2 ?q2)))
             (at end (located_at_1 ?q2))
             (at end (located_at_2 ?q1))))

(:durative-action mix_1_at_1
    :parameters ( )
    :duration (= ?duration 1)
    :condition
        (and (at start (located_at_1 q1))
             (over all (not (mixed q1))))
    :effect
        (and (at start (not (located_at_1 q1)))
             (at end (located_at_1 q1))
             (at end (mixed q1))))

(:durative-action P-S_1stPhaseSeparation_at_6-7
    :parameters (?q1 - qstate ?q2 - qstate)
    :duration (= ?duration 3)
    :condition
        (and (at start (located_at_6 ?q1))
             (at start (located_at_7 ?q2))
             (at start (not (GOAL_PS1 ?q1 ?q2)))

    :effect
        (and (at start (not (located_at_6 ?q1)))
             (at start (not (located_at_7 ?q2)))
             (at end (located_at_6 ?q1))
             (at end (located_at_7 ?q2))
             (at end (GOAL_PS1 ?q1 ?q2))
             (at end (GOAL_PS1 ?q2 ?q1)))))

(:durative-action P-S_2ndPhaseSeparation_at_6-7
    :parameters (?q1 - qstate ?q2 - qstate)
    :duration (= ?duration 3)
    :condition
        (and (at start (located_at_6 ?q1))
             (at start (located_at_7 ?q2))
             (at start (not (GOAL_PS2 ?q1 ?q2)))
             (at start (GOAL_PS1 ?q1 ?q2))
             (at start (mixed ?q1))
             (at start (mixed ?q2)))
    :effect
        (and (at start (not (located_at_6 ?q1)))
             (at start (not (located_at_7 ?q2)))
             (at end (located_at_6 ?q1))
             (at end (located_at_7 ?q2))
             (at end (GOAL_PS2 ?q1 ?q2))
             (at end (GOAL_PS2 ?q2 ?q1)))))
```

Figure 4: PDDL model of actions representing some exemplary SWAP, MIX, P-S gates.

(iii) the underlying graph structure (gates connecting different pairs of qubits). We follow the conventional practice of modeling (i) in the domain description while (ii) is captured in the problem description. One common practice is to model (iii) within the problem file. However, given that we target a rather sparse underlying qubit-connecting graph structure (see Figure 2), we decide to capture it within the domain file to ease the burden of the "grounding" and pre-processing step for existing planners, which can be very time-consuming. Specifically:

Objects: We need to model three types of object: qubits, qstates, and the location of the P-S and SWAP gates (i.e., edges connecting different qubits). Since qstates are associated (by means of the predicate $located\_at$, see Figure 4 for concrete example) to specific qubits, they have been modeled explicitly as planning objects, while the qubits and the gate locations (i.e., edges) are modeled implicitly. It is clear from the action definitions in Figure 4 that qubit locations are embedded explicitly within the action declaration. This approach avoids declaring qubits as part of the action parameters, significantly reducing the number of ground actions to be generated. For 2-qubit actions, the potential number of ground actions reduce from N^4 to $N^2 \times |E|$, with N the number of qubits in the chip (up to 40) and E the set of connections between qubits.

Actions: temporal planning actions are created to model: (i) 2-qubit SWAP gates, (ii) 2-qubit P-S gates, and (iii) 1-qubit MIX gates. For reference, Figure 4 shows the PDDL description of a SWAP gate between qubits 1 and 2, the MIX gate of state q_1 on qubit 1, and the P-S gates between qubits 6 and 7 at the first and second phase separation. In the action's condition list, we specify that gates are accomplished on the two qstates only if they are located on the corresponding qubits. To prevent a qstate q currently belonging to qubit X from being addressed by multiple gates at the same time (i.e. "mutex" relations in planning terminology), we assign value FALSE to the predicate $(located\_at\_Xq)$ at the starting time of all actions involving q.

The most complex constraint to model is the conditions to mix a qstate q given the requirement that *all* P-S gates involving q in the previous phase separation step have been executed. We explored several other choices to model this requirement such as: (i) use a metric variable $PScount(q)$ to model how many P-S gates involving q have been achieved at a given moment; or (ii) use ADL quantification and conditional effect constructs supported in PDDL. Ultimately, we decided to explicitly model all P-S gates that need to be achieved as conditions of the MIX(q) action. This is due to the fact that alternative options require using more expressive features of PDDL2.1 which are not supported by many effective temporal planners.[2]

Objective: we use the standard temporal planning objective

[2]For example, preliminary tests with our PDDL model using metric variables to track satisfied goals involving qstate q using several planners shows that they perform much worse than on non-metric version, comparatively. This is to be expected as currently, state-of-the-art PDDL planners still do not handle metric quantities as well as logical variables.

of minimizing the plan *makespan*. This coincides with minimizing the *circuit depth*, which is the main objective of the quantum compilation problem.

Alternative model: given that non-temporal planners can perform much better than temporal planners on problems of the same size, we have also created the non-temporal version of the domain by discretizing action durations into consecutive "time-steps" t_i, introducing additional predicates *next*(t_i, t_{i+1}) enforcing a link between consecutive time-steps. However, initial evaluation of this approach with the M/Mp SAT-based planner (Rintanen 2012) (which optimize parallel planning steps) indicated that the performance of non-temporal planners on this discretized (larger) model is much worse than the performance of existing temporal planners on the original model.

6 Empirical Evaluation

We have modeled the QAOA circuit compilation problem as described in the previous sections and tested them using various off-the-shelf PDDL 2.1 Level 4 temporal planners. The results were collected on a RedHat Linux 2.4Ghz machine with 8GB RAM.

Problem generation: We consider three problem sizes based on grids with N = 8, 21 and 40 qubits (dashed boxes in Figure 2). For each grid size, we generated two problem classes: (i) p = 1 (only one PS-mixing step) and (ii) p = 2 (two PS-mixing steps). To generate the graphs G for which a MaxCut needs to be found, for each grid size, we randomly generate 100 Erdös-Rényi graphs G (Erdös and Rényi 2005). Half (50 problems) are generated by choosing N of $N(N-1)/2$ edges over respectively 7, 18, 36 qstates randomly located on the circuit of size 8, 21, and 40 qubits (referred to herafter as 'Utilization' u=90%). The other half are generated by choosing N edges over 8, 21, and 40 qstates, respectively (referred to herafter as 'Utilization' u=100%). In total, we report tests on 600 random problems.

Planner setup: Since larger N and/or p lead to more complex setting with more predicates, ground actions, and requires planners to find longer plans, the allocated cutoff time for different setting are as follow: (i) 10 minutes for $N = 8$, (ii) 30 minutes for $P = 1, N = 21$; (iii) 60 minutes for other cases. We select planners that performed well in the temporal planning track of previous IPCs, while at the same time representing a diverse set of planning technologies: (i) *LPG*: which is based on local search with restarts over action graphs (Gerevini, Saetti, and Serina 2003); (ii) *Temporal FastDownward (TFD):* a heuristic forward state-space search planner with post-processing to reduce makespan (Eyerich, Mattmüller, and Röger 2009), and (iii) *SPGlan:* partition the planning problem into subproblems that can be solved separately, while resolving the inconsistencies between partial plans using extended saddle-point condition (Wah and Chen 2004) (Chen and Wah 2006).

	P1						P2			
	N8		N21		N40		N8		N21	
Util	0.9	1.0	0.9	1.0	0.9	1.0	0.9	1.0	0.9	1.0
SGPlan	50	50	50	50	50	50	50	50	-	-
TFD	50	50	50	50	-	-	50	50	50	50
LPG	50	50	50	50	10	14	50	50	-	6

Table 1: Summary of the solving capability of selected planners. Numbers indicate how many random problems out of 50 have been solved.

	p=1, N8		p=1, N21		p=2, N8	
Utilization	0.9	1.0	0.9	1.0	0.9	1.0
SGPlan	0.74	0.76	0.68	0.68	0.76	0.80
TFD	**0.96**	**0.98**	**0.96**	**0.95**	**1.0**	**0.99**
LPG	0.82	0.83	0.83	0.81	0.53	0.51

Table 2: Plan quality comparison between different planners using IPC formula (higher value indicates better plan quality).

We ran SGPlan (Ver 5.22) and TFD (Ver IPC2014) with their default parameters while for LPG (Ver TD 1.0) we ran all three available options (i) *-speed* that uses heuristic geared toward finding a valid plan quickly, (ii) *-quality* that uses heuristic balancing plan quality and search steps, and (iii) *-n 10* ($k = 10$) that will try to find within the time limit up to *10* plans of gradually better quality by using the makespan of previously found plan as upper-bound when searching for a new plan. Since LPG ($k = 10$) option always dominates both LPG-quality and LPG-speed by solving more problems with better overall quality for all setting, we will exclude results for LPG-quality and LPG-speed from our evaluation discussion. For the rest of this section, LPG result is represented by LPG ($k = 10$).

Evaluation Result Summary: Table 1 shows the overall performance on the ability to find a plan of different planners. SGPlan stops after finding one valid plan while TFD and LPG exhaust the allocated time limit and try to find gradually improving quality plans. Since no planner was able to find a single solution for N = 40 and p = 2, we omit the result for this case from Table 1. Overall, SGPlan and TFD were able to solve the highest number of problems, followed by LPG. SGPlan can find a solution very quickly, compared to the time it takes other two planners to find the first solution. It is the only planner that can scale up to $N = 40$ for p = 1 (finding plans with 150-220 actions). Unfortunately, SGPlan stopped with an internal error for N = 21 and p = 2. TFD generally spent a lot of time on preprocessing for $p = 1, N = 21$ (around 15 minutes) and $p = 2, N = 21$ (around 30 minutes) but when it's done with the pre-processing phase it can find a solution very quickly and also can improve the solution quality very quickly. TFD spent all of the 60 minutes time limit on pre-processing for $N = 40$ problems. LPG can generally find the first solution quicker than TFD (still much slower than SGPlan) but does not improve the solution quality as

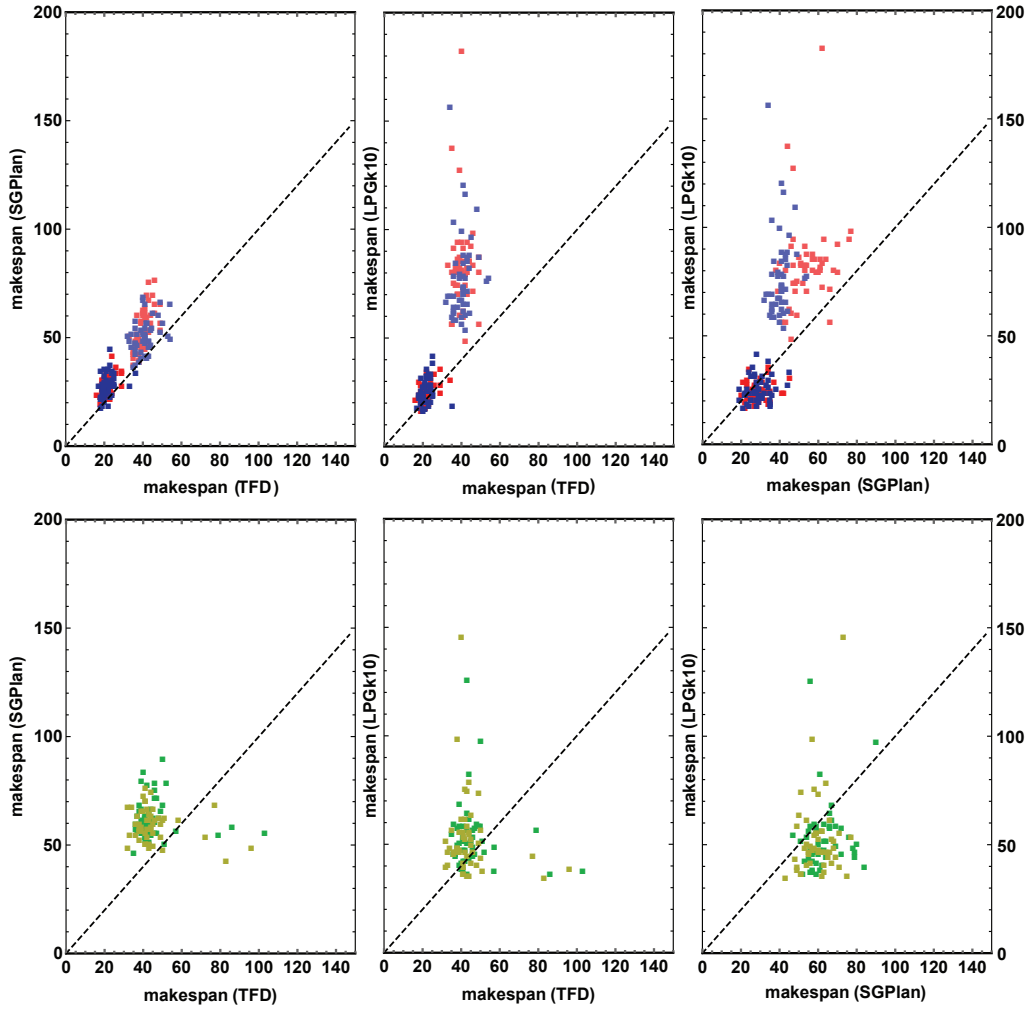

Figure 5: Instance-by-instance comparison of *SGPlan*, *TFD* and *LPG*. Top panel refers to results for N=8: Red dots indicate instances with u=90% while blue dots are for u=100%. Darker data points (lower makespans) refer to p=1 while lighter points (higher makespans) refer to p=2 (see Table 1). Bottom panel refers to results for N=21: Green indicates u=90% and yellow u=100%.

quickly as TFD over the allocated timelimit.

Plan quality comparison: to compare the plan quality across planners, we use the formula employed by the IPCs to grade planners in the temporal planning track since IPC6 (Helmert, Do, and Refanidis 2008): for each planning instance i, if the best-known makespan is produced by a plan P_i, then for a given planner X that returns a plan P_X^i for i, the score of P_X^i is calculated as: $makespan(P_i)$ divided by $makespan(P_X^i)$. A comparative value closer to 1.0 indicates that planner X produces better quality plan for instance i. We use this formula and average the score for our three tested planners over the instance ensembles that are completely solved by the time cutoff. Table 2 shows the performance of different planners with regard to plan quality. For N = 8 and p = 1, TFD found the best or close to the best quality plans. LPG is about 15% worse while SGPlan, which unlike TFD and LPG only find a single solution, produce lower quality plans. The comparison results for $N = 21$ and $p = 1$ is similar. For N = 8 and p = 2, TFD again nearly always produce the best quality plan. However, for this more complex case, SGPlan produces overall better quality plans compared to LPG, even though LPG returns multiple plans for each instance.

Figure 5 shows in further detail the head-to-head makespan comparison between different pairs of planners, specifically pairwise comparisons between TFD, SGPLan, and LPG: TFD always dominates SGPlan, TFD dominates LPG majority of the times, and SGPlan dominates LPG on bigger problems, but is slightly worse on smaller problems.

Planning time comparison: Both TFD and LPG use "anytime" search algorithms and use all of their allocated time to try finding better gradually better quality plans. In

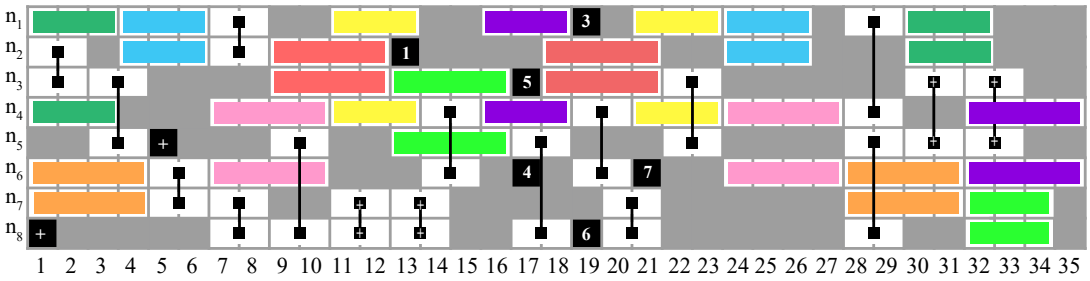

Figure 6: Compilation of p = 2 QAOA performed by TFD for the MaxCut problem depicted in Fig. 3 on the N=8 processor in Fig. 2; with time on the x-axis and qubit locations on the y-axis. Each row indicates what gate operates on each qubit at a given time during the plan. With reference to Fig. 1, colored blocks represents p-s gates and White blocks are swap gates (both synchronized in pair, since they are 2-qubit gates, same color indicate same logical gate); black blocks with numbers are mix gates acting on the corresponding state. Gates marked with a + indicate superfluous gates that were inserted in the plan by TFD, that could be detected and eliminated in postprocessing.

contrast, LPG-quality and SGPlan return a single solution and thus generally take a very short amount of time with the median solving time for SGPlan in p=1$|N_8$, p=1$|N_{21}$, P=1$|N_{40}$ and P=2$|N_8$ are 0.02, 1, 25, and 0.05 seconds[3].

Other planners: We have also conducted tests on: *VHPOP*, *HSP**, and *CPT* and POPF. While LPG, SGPlan, and TFD were selected for their ability to solve large planning problems, we hoped that HSP*, CPT, and VHPOP would return optimal plans to provide a baseline for plan quality estimation. Unfortunately, HSP*, CPT, and VHPOP failed to find a single plan even for our smallest problems for various reason: CPT underwent internal errors after a quick search time, VHPOP ran out of memory quickly, while HSP* couldn't find any plan for a cutoff time of 2 hours. POPF, which does not guarantee finding optimal plans, but produced good quality plans for other temporal planning domain, also does not find any solution.

Discussion: our preliminary empirical evaluation shows that the test planners provide a range of tradeoff between scalability and plan quality. At one end, we have SGPlan that can scale up to large problems and solve them in a short amount of time while providing reasonably good quality plans (compared to the best known solutions). At the other end, we have TFD, which utilizes all of the allocated time to find the best quality solutions but in general is the slowest by far to come up with some valid solution. LPG balances between the two: it can either find one solution quickly like SGPlan or can utilize the whole cutoff time to find better quality solutions. What's missing from our analysis is the assessment on how good the quality of the best found plans compared to the optimal solutions. At the moment, there is no published work on finding optimal solution for this problem. Moreover, as outlined in the previous paragraph, our current effort in getting the existing optimal-makespan planners to find solutions have not been fruitful. This is one important future research direction for us.

Figure 6 shows a visualization of a plan in a 'Gantt chart' format. Based on the "eye-test" and manual analysis, the best plans returned are usually of good quality but not without defects. The example plan shown in Figure 6, generated by TFD, has a very short makespan, but contains some unnecessary gates. Examples are the repeated swaps at time 11 and 30, and the mixing of the un-utilized logical states q_2 and q_8 at times 1,5. These spurious gates/actions do not affect the makespan, and they can be identified and eliminated by known plan post-processing techniques (Do and Kambhampati 2003). We also believe a tighter PDDL model can also help eliminate some of the extra gates.

7 Conclusion and Future Work

In this paper we presented a novel approach to the problem of compiling idealized quantum circuits to specific quantum hardware, focusing our experiments on QAOA circuits. Three well-established temporal planners were able to compile the QAOA circuits with reasonable efficiency, demonstrating the viability of this approach. A virtue of the planning approach is that the framework is very flexible with respect to features of the hardware graph, including irregular structures. This work paves the way for potentially impactful future work on the use of artificial intelligence methods for quantum computing. In future work, we plan to further tune the performance of the planners, including choosing an initial assignment of qstates to qubits favorable for compilation. In order to scale reliably to larger plan sizes we will develop as well decomposition approaches where $p > 1$ could be divided into multiple p = 1 problems to be solved independently and matched in a postprocessing phase. We will also compare with other approaches to this compilation problem such as sorting networks (Beals et al. 2013) (Brierly 2015) (Bremner, Montanaro, and Shepherd. 2016), and we will look at parameters values for the durations that are fitting existing hardware, in collaboration with experimental physicists. Moreover, we will include in

[3]For comparison purpose, LPG-quality, which also try to returns a single solution of good quality, produces the median solving time for P=1$|N_8$ and P=2$|N_8$ are 0.9 and 70 seconds respectively.

the PDDL modeling additional features that are characteristics of quantum computer architectures, such as the crosstalk effects of 2-qubit gates and the ability to *quantum teleport* qstates across the chip.

References

[Barends et al. 2016] Barends, R.; Shabani, A.; Lamata, L.; Kelly, J.; Mezzacapo, A.; Heras, U. L.; Babbush, R.; Fowler, A. G.; Campbell, B.; Chen, Y.; et al. 2016. Digitized adiabatic quantum computing with a superconducting circuit. *Nature* 534(7606):222–226.

[Beals et al. 2013] Beals, R.; Brierley, S.; Gray, O.; Harrow, A. W.; Kutin, S.; Linden, N.; Shepherd, D.; and Stather, M. 2013. Efficient distributed quantum computing. *Proceedings of the Royal Society A.* 469(2153):767 – 790.

[Boxio 2016] Boxio, S. 2016. Characterizing quantum supremacy in near-term devices. In *arXiv preprint arXiv:1608.0026*.

[Bremner, Montanaro, and Shepherd. 2016] Bremner, M. J.; Montanaro, A.; and Shepherd., D. J. 2016. Achieving quantum supremacy with sparse and noisy commuting quantum computations. In *arXiv preprint arXiv:1610.01808*.

[Brierly 2015] Brierly, S. 2015. Efficient implementation of quantum circuits with limited qubit interactions.". In *arXiv preprint arXiv:1507.04263*.

[Chen and Wah 2006] Chen, Y., and Wah, B. 2006. Temporal planning using subgoal partitioning and resolution in sg-plan. *Journal of Artificial Intelligence Research* 26:323 – 369.

[Devitt 2016] Devitt, S. J. 2016. Performing quantum computing experiments in the cloud. *Physical Review A* 94(3):222–226.

[Do and Kambhampati 2003] Do, M. B., and Kambhampati, S. 2003. Improving the temporal flexibility of position constrained metric temporal plans. In *Proceedings of the* 13th *International Conference on Artificial Intelligence Planning and Scheduling (ICAPS)*.

[Erdös and Rényi 2005] Erdös, P., and Rényi, A. 2005. A faster optimal register allocator. *The Journal of Instruction-Level Parallelism* 7:1 – 31.

[Eyerich, Mattmüller, and Röger 2009] Eyerich, P.; Mattmüller, R.; and Röger, G. 2009. Using the context-enhanced additive heuristic for temporal and numeric planning. In *Proceedings of the* 19th *International Conference on Automated Planning and Scheduling*, 318 – 325.

[Farhi and Harrow 2016] Farhi, E., and Harrow, A. W. 2016. Quantum supremacy through the quantum approximate optimization algorithm. In *arXiv preprint arXiv:1602.07674*.

[Farhi, Goldstone, and Gutmann. 2014a] Farhi, E.; Goldstone, J.; and Gutmann., S. 2014a. A quantum approximate optimization algorithm. In *arXiv preprint arXiv:1411.4028*.

[Farhi, Goldstone, and Gutmann. 2014b] Farhi, E.; Goldstone, J.; and Gutmann., S. 2014b. A Quantum Approximate Optimization Algorithm Applied to a Bounded Occurrence Constraint Problem. In *arXiv preprint axXiv:1412.6062*.

[Fu, Wilken, and Goodwin 1960] Fu, C.; Wilken, K.; and Goodwin, D. 1960. On the evolution of random graphs. *Publications of the Mathematical Institute of the Hungarian Academy of Sciences* 5:569–573.

[Gerevini, Saetti, and Serina 2003] Gerevini, A.; Saetti, L.; and Serina, I. 2003. Planning through stochastic local search and temporal action graphs. *Journal of Artificial Intelligence Research* 20:239 – 290.

[Guerreschi and Smelyanski 2017] Guerreschi, G., and Smelyanski, M. 2017. Practical optimization for hybrid quantum-classical algorithms. In *arXiv preprint arXiv:1701.01450*.

[Helmert, Do, and Refanidis 2008] Helmert, M.; Do, M.; and Refanidis, I. 2008. The 2008 international planning competition: Deterministic track. http://icaps-conference.org/ipc2008/deterministic/. 2008-02-19.

[IBM 2017] IBM. 2017. The ibm quantum experience. http://www.research.ibm.com/quantum/. 2017-02-19.

[Jiang, Rieffel, and Wang 2017] Jiang, Z.; Rieffel, E.; and Wang, Z. 2017. A qaoa-inspired circuit for grover's unstructured search using a transverse field. In *arXiv preprint arXiv:1702.0257*.

[Rieffel and Polak 2011] Rieffel, E. G., and Polak, W. H. 2011. *Quantum computing: A gentle introduction*. MIT Press.

[Rintanen 2012] Rintanen, J. 2012. Planning as satisfiability: Heuristics. *Artificial Intelligence* 193:45 – 86.

[Sete, Zeng, and Rigetti 2016] Sete, E. A.; Zeng, W. J.; and Rigetti, C. T. 2016. A functional architecture for scalable quantum computing. In *, IEEE International Conference on Rebooting Computing (ICRC)*.

[Smith, Curtis, and Zeng 2016] Smith, R. S.; Curtis, M. J.; and Zeng, W. J. 2016. A practical quantum instruction set architecture. In *arXiv preprint arXiv:1608.03355*.

[Steiger, Häner, and Troyer 2016] Steiger, D. S.; Häner, T.; and Troyer, M. 2016. Projectq: An open source software framework for quantum computing. In *arXiv preprint arXiv:1612.08091*.

[Versluis et al. 2016] Versluis, R.; Poletto, S.; Khammassi, N.; Haider, N.; Michalak, D.; Bruno, A.; Bertels, K.; and DiCarlo, L. 2016. Scalable quantum circuit and control for a superconducting surface code. *arXiv preprint arXiv:1612.08208*.

[Wah and Chen 2004] Wah, B., and Chen, Y. 2004. Subgoal partitioning and global search for solving temporal planning problems in mixed space. *International Journal on Artificial Intelligence Tools* 13(4):767 – 790.

[Wecker and Svore 2014] Wecker, D., and Svore, K. M. 2014. Liqui | >: A software design architecture and domain-

specific language for quantum computing. In *arXiv preprint arXiv:1402.4467*.

[Wecker, Hastings, and Troyer 2016] Wecker, D.; Hastings, M. B.; and Troyer, M. 2016. Training a quantum optimizer. In *arXiv preprint arXiv:1605.05370*.

[Yang et al. 2016] Yang, Z. C.; Rahmani, A.; Shabani, A.; Neven, H.; and Chamon, C. 2016. Optimizing variational quantum algorithms using pontryagin's minimum principle. In *arXiv preprint arXiv:1607.06473*.