Compiling Quantum Circuits to Realistic Hardware Architectures using Temporal Planners
Davide Venturelli\textsuperscript{1,2}, Minh Do\textsuperscript{3,4}, Eleanor Rieffel\textsuperscript{1}, and Jeremy Frank\textsuperscript{4}

\textsuperscript{1}NASA Ames Research Center, Quantum Artificial Intelligence Laboratory
\textsuperscript{2}USRA Research Institute for Advanced Computer Science (RIACS)
\textsuperscript{3}Stinger Ghaffarian Technologies (SGT Inc.)
\textsuperscript{4}NASA Ames Research Center, Planning and Scheduling Group

Abstract
To run quantum algorithms on emerging gate-model quantum hardware, quantum circuits must be compiled to take into account constraints on the hardware. For near-term hardware, with only limited means to mitigate decoherence, it is critical to minimize the duration of the circuit. 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 on compiling to superconducting hardware architectures with nearest-neighbor constraints. Our initial experiments focus on compiling Quantum Approximate Optimization Algorithm (QAOA) circuits whose high number of commuting gates allow great flexibility in the order in which the gates can be applied. That freedom makes it more challenging to find optimal compilations but also means there is a greater potential win from more optimized compilation than for less flexible circuits. We map this quantum circuit compilation problem to a temporal planning problem, and generated a test suite of compilation problems for QAOA circuits of various sizes to a realistic hardware architecture. We report compilation results from several state-of-the-art temporal planners on this test set, compile circuits of various sizes to a realistic hardware. This early empirical evaluation demonstrates 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. Currently only special purpose quantum hardware is commercially available: quantum annealers that could run one type of quantum optimization algorithm. The emerging gate-model processors, currently in prototype phase, are universal in that, once scaled up, they can run any quantum algorithm, expanding the empirical exploration of quantum algorithms beyond optimization, as well as enabling the exploration of a broader array of quantum approaches to optimization. Quantum algorithms are usually specified as idealized quantum circuits that do not take into account hardware constraints. This approach makes sense since the actual physical constraints vary from architecture to architecture. With the advent of gate-model processors, researchers have begun to explore approaches to compiling idealized quantum circuits to realistic hardware.

For example, emerging superconducting quantum processors have planar architectures with nearest-neighbor restrictions on the locations (qubits) to which the gates can be applied. Such processors include the 5-qubit processor IBM recently made publicly available the cloud (IBM 2017), recently updated to 17 qubits, and processors being fabricated by other groups, such as TU Delft (Versluis et al. 2016), UC Berkeley (Ramasesh et al. 2017), 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. Idealized circuits generally do not respect nearest neighbor constraints. For this reason, compiling idealized quantum circuits to superconducting hardware requires adding supplementary gates that move qubit states to locations where the desired gate can act on them.

Quantum computational hardware suffers from decoherence, which degrades the performance of quantum algorithms over time. Especially for near-term hardware, with only limited means to mitigate decoherence, it is critical to minimize the duration of (execution of) the circuit that carries out the quantum computation, so as to minimize the decoherence experienced by the computation. Other, more sophisticated, compilation cost functions, such as figure-of-merits taking into account fidelity of operations, could be used in the future within the temporal planning approach to compilation we explore here. Optimizing the duration of compiled circuits is a challenging problem due to the parallel execution of gates with different durations. For quantum circuits with flexibility in when the gates can be applied, or when some gates commute with each other (so 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. That freedom makes it more challenging to find optimal compilations but also means there is a greater potential win from more optimized compilation 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 2016a) (Steiger, Häner, and Troyer 2016a)
(Devitt 2016) (Barends et al. 2016), few approaches have been explored for compiling idealized quantum circuits to realistic quantum hardware (Beals et al. 2013) (Brierley 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 2005). Recent studies explore exact schemes (Wille, Lye, and Drechsler 2014), approximate tailored methods (Kole, Dutta, and Sengupta 2017) or formulations suited for off-the-shelf Mixed Integer Linear Programming (MILP) solvers such as Gurobi (Bhattacharjee and Chattopadhyay 2017). Prior work had not used a temporal planning approach, which can be applied quite generally, enabling us to address, for example, gates with variable durations, and efficiencies that can be gained when large numbers of gates commute.

In this paper, we apply temporal planning techniques to the problem of compiling quantum circuits to realistic gate-model quantum hardware. While our target audience is researchers with a knowledge of quantum computing, we have endeavored to make the paper accessible to others, such as those in the temporal planning community, who are willing to take on faith the planning description of the quantum circuit compilation we describe. As we will explain in detail, 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 diverse set of temporal planners to compile circuits of various sizes to an architecture inspired by those currently being built. This early empirical evaluation demonstrates that temporal planning is a viable approach to quantum circuit compilation.

In Sec. 2, we describe the problem of compiling idealized quantum circuits to specific hardware architectures in detail. Sec. 3 describes QAOA circuits, the class of circuits with many commuting gates that we target for our initial investigation. Section 4 explains our mapping of the quantum circuit compilation problem to temporal planning problem. Sec. 5 presents our results applying state-of-the-art temporal planners to this circuit compilation problem. In Sec. 6, we outline future research directions stemming from this study. Appendix A provides background information for readers who are not familiar with gate-model quantum computing.

2 Architecture-specific quantum circuit compilation problem

Quantum circuits for general quantum algorithms are often described on 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 the relevant 2-qubit quantum gates are associated to edges. In many architectures more than one 2-qubit gate may be implementable between a given number of qubits, so this structure is a multigraph (multiple edges allowed between two nodes). Gates that operate on distinct sets of qubits may be able operate concurrently, though there can be additional restrictions on which operations can be done in parallel, such as requiring the sets in operation to be non-adjacent, as in Google’s proposed architecture (Boxio 2016)). Furthermore, there are different types of quantum gates, each taking different durations, with the duration depending on the specific physical implementation in terms of primitive gates. For an succinct introduction to quantum circuits in this context, see Appendix A.

In order for the computation specified by the idealized circuit to be completed, quantum information must be moved to locations where the desired gates can be carried out. Here, we consider a particular type of 2-qubit gate, the swap gate, which exchanges the state of two qubits, though other gate choices are possible. A sequence of swap gates can be used to move the contents of two distant qubits to a location where a desired gate 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.

2.1 Formal problem statement

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 for operations (gates), each acting on a single node or set of nodes. The idealized quantum circuit also includes implicit or explicit specification of which operations commute (the order in which they are executed can be switched without affecting the computation) either individually or as blocks. A hardware architecture specification can be viewed as a weighted, labeled multigraph on a set of nodes, corresponding to physical quantum memory locations (qubits) in the hardware, where each edge represents an operation (quantum gate) that can be physically implemented directly on the qubits in the physical hardware, with the labels indicating the type of quantum gate and the weight giving its duration. The output of the compilation process is a circuit that can be used to perform a quantum computation. It does not perform the computation itself, and therefore the compilation step can be carried out on a conventional (non-quantum) computer.

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

### 2.2 Compilation examples

Fig. 1 shows a hypothetical chip design that we will use for our experiments on circuit compilation. It is inspired by the architecture proposed 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 (considering just those relevant for the algorithm), in this case swap gates and two other types of 2-qubit gates (red and blue). 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 when required, and reordering operations commuting operations when desired. The objective is to minimize the overall duration to execute all gates in the 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. 1.

**Example 1:** 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:

\[
\{ \text{SWAP}_{n_4,n_1}, \text{SWAP}_{n_2,n_3} \} \rightarrow \text{SWAP}_{n_1,n_2} \rightarrow \text{RED}_{n_2,n_3} \equiv \text{RED}(q_2,q_4)
\]

(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 \( 2 \tau_{\text{swap}} + \tau_{\text{red}} \) clock cycles where \( \tau_s \) represents the duration of the *-gate.

**Example 2:** Consider an idealized circuit that requires \( \text{BLUE}(q_1,q_2) \land \text{RED}(q_4,q_2) \), in no particular order. If \( \tau_{\text{blue}} > 3 \times \tau_{\text{swap}} \), the compiler might want to execute \( \text{BLUE}_{n_4,n_2} \) while the qubit state \( q_4 \) is swapped all the way clockwise in five SWAPS from \( n_4 \) to \( n_2 \) where \( \text{RED}_{n_2,n_3} \) can be executed. However, if \( \tau_{\text{swap}} < 3 \times \tau_{\text{blue}} \), it is preferable to wait until the end of \( \text{BLUE}_{n_1,n_2} \) and then start to execute the instruction sequence in Eq. (1).

### 3 Compiling QAOA for the MaxCut problem

While our approach can be used to compile arbitrary quantum circuits to a wide range of architectures, in this paper we concentrate on one particular case: the class of QAOA circuits (Farhi, Goldstone, and Gutmann. 2014a) for MaxCut (defined in the later part of this section) to an architecture inspired by an architecture proposed by Rigetti Computing Inc. (Sete, Zeng, and Rigetti 2016). 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 less flexible circuits. This makes finding the optimal compilation more challenging, but also means there is potential for greater compilation optimization, compared to other less flexible classes of circuits.

QAOA circuits have 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) (Guerrreschi 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). Recently Google Inc. proposed an alternative quantum approximate optimization approach in fixed nearest-neighbor architectures explicitly to avoid the compilation step that is the subject of our work (Farhi et al. 2017). Their numerical results on MaxCut instances show a small hit in performance. Furthermore, unlike the QAOA circuits we consider here, in which the number of parameters is independent of the number of qubits, their alternative approach has many more parameters, which increase with the number of qubits, and these parameters must be optimized separately for each architecture. While their approach makes good use of near-term hardware with numbers of qubits for which the parameter optimization is
tractable, ultimately one wants a scalable algorithm that can be compiled to arbitrary architectures. Thus, while interesting, especially in the very near-term, rather than obviating the need, their work serves to underscore the need for efficient approaches to optimize compilation.

We chose MaxCut as the target problem of reference, as it is becoming one of the de facto benchmark standards for quantum optimization of all types and it is considered a primary target for experimentation in the architecture of (Sete, Zeng, and Rigetti 2016).

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.

A quadratic boolean objective function for MaxCut is:

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

where \( s_i \) are binary variables, one for each vertex \( v_i \), with values +1 or -1 indicating to which partition the vertex \( v_i \) is assigned.

Idealized QAOA circuits alternate between a phase separation step (PS), based on the objective function, and a mixing step. The phase-separation step for QAOA for MaxCut 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. Specifically, the idealized QAOA circuit for MaxCut requires a 2-qubit gate for each quadratic term in the objective function of Eq. (2), as well as 1-qubit gates for each vertex for the mixing step (Farhi, Goldstone, and Gutmann. 2014a). In Fig. 2 a 6-vertex graph is shown, providing an illustrative instance that will be used to describe the compilation procedure. 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 without changing the computation. In the mixing phase, a set of 1-qubit operations are applied, one to each qubit\(^1\) 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 steps are repeated \( p \) times. We consider \( p = 1 \) and \( p = 2 \) in our experiments (detailed in Section 5).

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\((q_i, q_j)\). We ignore the final mixing step since it is trivial to compile by just applying the 1-qubit mixing gate to each qubit as the last operation.

We chose the architecture proposed by Rigetti Computing in (Sete, Zeng, and Rigetti 2016) (see Fig. 1) for our initial exploration because it offers a particularly interesting compilation, and therefore planning, problem, due to the existence of two different kinds of nearest neighbor relation in the proposed hardware. After the synthesis of the QAOA MaxCut gates (see Appendix A), these two different relations become two different durations of two-qubit gates, which corresponds to the red and blue edges as described above. In our problem specification, while there are two flavors of p-s gates (red, blue), corresponding to two different durations of execution, the compilation goals (see figure 2) do not care on which of these two types of gates carries out the required steps. For the purpose of this proof-of-concept work, these durations we assign to the gates are not derived from actual designs of ongoing experiments, but are realistic and serve to illustrate possible future designs.

The constraints on the compilation problem can be understood, with reference to Fig. 1, as:

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

4 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 approaching the problem of compiling quantum circuits as a temporal planning problem.

\(^1\)This is another simple feature of MaxCut, and it is due to the fact that all possible \( 2^N \) single variable assignments are defining a valid cut. If this wasn’t the case, then the mixing phase would also likely require the application of 2-qubit gates, further complicating the scheduling problem.
Classical planning problems are expressed in terms of binary state variables and actions. Examples of state variables for our problem are “The quantum state $\Psi$ is assigned to qubit number $X$” and “The quantum state $\Phi$ has been transformed by the application of gate $G$ present on qubits $X$ and $Y$,” which may be True or False. Actions consist of two lists, a set of preconditions and a set of effects. The effects of an action consists of a subset of state variables with the values they take on if the action is carried out. For example, the action “State $\Psi$ is now moved from qubit $X$ to qubit $Y$” has one precondition, “State $\Psi$ is assigned to $X = True$” and has two effects “State $\Psi$ is assigned to $X = False$” and “State $\Psi$ is assigned to $Y = True$.”

A specific planning problem specifies an initial state, with values specified for all state variables, and a goal, specified values for one or more state variables. As for preconditions, goals are conventionally positive, so the goal value for the goal variables is True. Generally, the goal specifies values for only a small subset of the state variables. A plan is a sequence of actions.

A valid plan, or a solution to the planning problem, is a sequence of actions $A_1, \ldots, A_L$ such that the state at time step $t_i-1$ meets the preconditions for action $A_i$, the effects of action $A_i$ are reflected in the state at time step $t_i$, and the state at the end has all of the goal variables set to True. For an introduction on Automated Planning and Scheduling, see (Ghallab, Nau, and Traverso 2004).

**Planners:** A planner is software implementing a collection of algorithms; it takes as input a specification of domain and a problem description 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.

**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 facto standard for 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 is a flexible language that offers multiple alternatives for modeling a planning problem. These modeling choices greatly affect the performance of existing PDDL planners. For instance, many planners pre-process the original domain description before building plans; this process 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.

**Modeling Quantum Gate Compilation in PDDL 2.1:** To apply a temporal planner to the circuit compilation problem, we must represent the allowed gates as actions and the desired circuit as a set of goal variables. We describe how to do so for the QAOA circuit compilation problem exemplified in Fig. 2, ensuring that for a plan to be valid, the required P-S or MIX gates are scheduled for each step of the algorithm. At the high-level, in this domain, we need to model: (i) how actions representing P-S, SWAP, and MIX gates affect qubits and qubit states (qstate); (ii) the actual qubits and qstates involved in a particular compilation problem, with their initial locations and final goal requirements, (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 Fig. 1), 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 in the multigraph of Fig. 1 connecting different qubits). Since qstates are associated (by means of the predicate located_at, see Fig. 3 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 Fig. 3 that qubit locations are embedded explicitly within the action declaration. This approach avoids declaring qubits as part of the action.
Figure 3: PDDL model of actions representing some exemplary SWAP, MIX, and P-S gates. The first line indicates that this compilation problem involves 8 qubit states. For each action, the duration indicates how long the action takes. The first action, a swap at qubits 1 and 2, has as parameters the two qstate to swap. The condition checks that these states are indeed located at the qubits on which the swap will occur. The effect makes sure the states have been swapped. The second action mixes qstate q1 at qubit 1, with conditions that state q5 is indeed at qubit 1, and both the phase separation gates involving qstate q5 (see Fig. 2) have been carried out. The effects include setting mixed q5 to TRUE The third and fourth actions are phase separation actions in a p = 2 circuit corresponding to the first and second levels of the algorithm.

### 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, Fig. 3 shows the PDDL description of a SWAP gate between qubits 1 and 2, the MIX gate of state q5 on qubit 1, and the P-S gate 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 X q) at the starting time of all actions involving q.

The most complex constraint to model is the condition 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 accomplished 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 accomplished 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.

### Objective: For a level p QAOA circuit, the goal is to have all of the (GOAL P-S(q1,q2)) predicates, for any q1 and q2 connected in the graph, for 1 ≤ i ≤ p set to TRUE and all mixed(qj) set to true for 1 ≤ j ≤ N and 1 ≤ i ≤ p − 1 (since the final mixing step can be added by hand at the end). Since we only consider p = 1 and p = 2, so only have a mixing step in the p = 2 case, we have simplified the

---

2The full set of PDDL model for all our tested problems is available at: https://ti.arc.nasa.gov/m/groups/asr/planning-and-scheduling/VentCirComp17_data.zip

3Only one of six planners in the Temporal track of the latest IPC (2014) supports numeric variables and also only one of six supports quantified conditions. 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.
notation to simply $\text{mixed}e\text{d}j$. Further, we use the standard temporal planning objective of minimizing the plan $\text{makespan}$. Minimizing the makespan coincides with minimizing the circuit depth, which is the main objective of the compilation problem.

**Alternative models:** Given that non-temporal planners can perform much better than temporal planners on problems of the same size, we also created the non-temporal version of the domain by discretizing action durations into consecutive “time-steps” $t_i$, introducing additional predicates $\text{next}(t_i, t_{i+1})$ 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.

Another option is to totally ignore the temporal aspect and encode it as a “classical” planning problem where actions are instantaneous. A post-processing step is then introduced to inject back the temporal constraints and schedule actions in the found classical plans. While we do not believe this approach would produce good quality plans, it’s another promising option to scale up to larger problems in this domain.

## 5 Empirical Evaluation

We 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 Fig. 1). The utilized grids are representative of devices to come in the next 2 years. A gate-model 8-qubit chip with the grid we used should be available shortly from Rigetti.

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 Erdős-Rényi graphs $G$ (Erdős and Rényi 1960). 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 hereafter as ‘Utilization’ $u=90\%$). The other half are generated by choosing $N$ edges over 8, 21, and 40 qstates, respectively (referred to hereafter as ‘Utilization’ $u=100\%$). In total, we report tests on 600 random planning problems with size ranging from $\{1024,192\}$ to $\{232000,8080\}$ in terms of number of grounded actions and predicates.

**Planner setup:** Since larger $N$ and $p$ lead to more complex setting with more predicates, ground actions, requiring 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:

- **LPG:** which is based on local search with restarts over action graphs (Gerevini, Saetti, and Serina 2003). Specifically, LPG incrementally builds a multi-level graph structure with each layer is represented by a single action and each graph edge represents a supporting connection between one action’s effect with a condition of another action appearing in a later layer. The graph leaf nodes represent action conditions that have not been supported (i.e., “connected”) by other action effects. At the beginning of the search process, LPG starts with a two-layer graph consisting of two newly created actions: (i) $A_{\text{init}}$: which occupies the first layer of the graph, has an empty condition list, and has an effect list represents state variables that are true in the initial states; (ii) $A_{\text{goal}}$: which occupies the last layer, has an empty effect list, and has a condition list represents all goals. At each search step, LPG generates the local search neighborhood by considering all decisions of either: (i) establishing a new edge connecting an existing action’s effect with an open condition of another action appears in a later layer (without conflicting with negative effects of other actions), (ii) removing an edge from the existing graph; (iii) adding another action to the graph; (iv) removing an action from the graph. Each resulting candidate partial (i.e., incomplete) plan in the local search neighborhood is evaluated by a heuristic function balancing between how close that candidate is from being a complete plan (i.e., fewer unsatisfied conditions) and how good the quality of the likely complete plan starting from that candidate partial plan based on the user’s defined objective function (e.g., minimizing the plan makespan). LPG then selects the best candidate partial plan from the search neighborhood and starts a new search episode. This process is repeated until a complete plan is found. When LPG is ran in the “anytime” mode, it does not stop when the first complete plan is found but will restart its planning process with the found plan(s) used as the baseline quality comparison on the subsequent trials.

- **Temporal FastDownward (TFD):** a heuristic forward state-space (FSS) search planner with post-processing to reduce makespan (Eyerich, Mattmüller, and Röger 2009). In the FSS framework, the planner starts from the initial state $I$ with an empty plan $P$ and tries to extends $P$ until the state resulted from applying $P$ satisfies all goalsootnote{This is in contrast to the backward state-space (BSS) planners, which build the plan “backward” starting from the goals until it reaches the initial state}. In each search step, FSS planners will generate new search nodes by taking a state $S_P = \text{Apply}(P, I)$, reached from applying $P$ to the initial state $I$, and considers all actions $A$ applicable in $S_P$ (i.e., all conditions of $A$ are satisfied by $S_P$). All newly generated states $S' = \text{Apply}(A, S_P)$ are put in the search queue,
ordered by the heuristically evaluated “quality” of $S'$. The heuristic value evaluating a given state $S$ generally depends on two factors: (i) the quality of the partial plan leading from $I$ to $S$, and (ii) the estimation on the quality of the remaining plan leading from $S$ to the goals. In TFD, the second part is estimated through analyzing a set of special structure called the domain-transition graphs (DTG) that are statically built for each planning problem. After a valid plan $P$ is found, TFD also tries to improve the final plan makespan by rescheduling actions in $P$, pushing them to start as early as possible without violating the various logical and temporal constraints between different actions in $P$ such as causal supports and potential conflicts caused by actions’ negative effects. This post-processing step is done greedily and takes little time compared to the planning process.

- **SPGPlan**: 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). Specifically, SPG tries to use sub-goal partitioning strategy in which a high-level planning problem is divided into smaller planning problems, each one targets a smaller subset of goals. Furthermore, if a “landmark” (i.e., a given state or condition that needs to be visited by all plans when solving a given problem) is found through landmark-analysis technique in a planning problem for a subset of goal, that landmark can be used to further partition a sub-planning problem into a subset of secondary sub-problems. Thus, the original planning problem can be partitioned into a hierarchy of multi-level interconnected smaller sub-problems, each with its own initial state and set of goals. Each sub-problem can be solved by any off-the-shelf planner. In particular, SGPlan uses a slightly modified Metric-FF, a forward state-space planner, and an earlier version of LPG to solve sub-planning problems.

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 $p = 2$ within the 60 minute cutoff, 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 pre-processing for $p = 1, N = 21$ (around 15 minutes) and $p = 2, N = 21$ (around 30 minutes) but when it is finished with the pre-processing phase\(^5\) it can find a solution quickly and also can improve the solution quality quickly. TFD spent all of the 60 minutes time limit on pre-processing for $N = 40$ problems. LPG can generally find the first solution more quickly than TFD (still much more slowly than SGPlan) but does not improve the solution quality as quickly as TFD over the allocated timelimit. We also tested YAHSP3-mt (Vidal 2014), another recent award winning temporal planner, but it did not return any solution.

|       | 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 50 50| 50 50 50 50|            |
| TFD   | 50 50 50 50| 50 50 - -  | 50 50 50 10| 14 10 50 50| 50 50 - 6  |
| LPG   | 50 50 50 50| 50 50 50 50| 10 14 50 50| 50 50 50 50|            |

Table 1: Summary of the solving capability of selected planners. Numbers indicate how many random problems out of 50 have been solved. **DV**: I am still convinced that this table is useless, its information can be very easily conveyed in the main text and we can save some space for the IJCAI version.

**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 (Helmer, 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 $P'_X$ for $i$, the score of $P'_X$ is calculated as: $makespan(P'_X)$

\(^5\)The two most time-consuming parts in TFD’s pre-processing routine are “processing axioms” and “invariant analysis”. While “processing axioms” are always consistently time-consuming, “invariant analysis” is heuristically done and sometime can be quick while some other times can be very time consuming.
Figure 4: 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 $u=100\%$. 
Figure 5: Compilation of \( p = 2 \) QAOA performed by TFD for the MaxCut problem depicted in Fig. 2 on the N=8 processor in Fig. 1; 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. Colored blocks represent \( p \)-s gates (of duration 3 or 4 depending on whether they are SHORT or LONG) and White blocks are swap gates (both types of 2-qubit gates are shown as a synchronized pair, since as 2-qubit gates they must act on two qubits at the same time. The 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.

For comparison purpose, LPG-quality, which also try to “anytime” search algorithms and use all of their allocated time to find better gradually better quality plans. In 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\(^6\).

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 tradeoffs between scalability and plan quality. At one end, SGPlan can scale up to large problem sizes and solve them in a short amount of time, providing reasonably good quality plans (compared to the best known solutions). At the other end, TFD utilizes all of the allocated time to find the best quality solutions but in general is the slowest by far to obtain a valid solution. LPG balances between the two: it can either find one solution quickly like SGPlan or can utilize the whole time prior to cutoff to find better quality solutions. Compared to TFD, LPG’s solutions are not as good, but it can scale up to larger problems; compared to SGPlan, LPG cannot solve as many problems, but returns better quality solutions. Overall, LPG is dominated by TFD on simpler problems and dominated by SGPlan on more complex problems.

Since planning is exponentially hard with regard to the problem size (i.e., number of state variables and actions), being able to partition it into sub-problems of smaller sizes definitely helps SGPlan to be find a valid solution quickly. However, there are several reasons that TFD and LPG can find overall better quality solutions: (i) their anytime algorithms allow them to gradually find better quality plans, using the previously found plans as baseline for pruning unpromising search directions; (ii) SGP’s partitioning algorithm is based on logical relationship between state variables.

\(^6\)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.
variables and actions and ignores all temporal aspects. Thus, combining plans for sub-problems using logical global constraints can lead to plans of lower quality for time-sensitive objective function such as minimizing the plan makespan.

What’s missing from our analysis is the assessment on how good the quality of the best plans found compared to the optimal solutions. At the moment, there is no published work on finding optimal solution for this problem and, as outlined in the previous paragraph, our current effort to get existing optimal-makespan planners to find solutions has not been successful. This is one important future research direction.

Based on “eye-test” and manual analysis, the best plans returned are usually of good quality but not without defects. Fig. 5 shows a visualization of a plan in a ‘Gantt chart’ format. Consider the qstate $q_1$ initially located at time $t_1$. The first gate it is involved in is the phase separation gate shown in green between qstates $q_1$ and $q_4$. The second gate is a phase separation gate between qubits 1 and 2 which contain qstates $q_1$ and $q_3$ respectively, because states $q_2$ and $q_3$ were swapped in the previous step. State $q_1$ is then swapped with the state in qubit 2, prior to being involved in another phase separation gate, between the contents of qubits 2 and 3, this time with state $q_5$ that was swapped into qubit 3 during time steps $3 - 4$. It is then mixed while still located at qubit 2. Continuing to read through the diagram in this way, we see that qstate $q_1$ undergoes the following sequence of actions:

$$P\cdot S_3(q_1, q_4) \rightarrow P\cdot S_3(q_1, q_3) \rightarrow P\cdot S_4(q_1, q_5) \rightarrow \text{MIX}(q_1)$$

$$\rightarrow \text{WAIT}(4) \rightarrow P\cdot S_4(q_1, q_5) \rightarrow \text{WAIT}(2)$$

$$\rightarrow P\cdot S_3(q_1, q_3) \rightarrow \text{WAIT}(3) \rightarrow P\cdot S_3(q_1, q_4)$$

where we denote the duration of the P-S gates in subscript and we introduced a WAIT gate to indicate inaction times. The second mixing phase is trivially scheduled at the end of the last tasks for each qstate. The example plan shown in Fig. 5, pictures the compilation of the problem instance in Fig. 2. The displayed output, generated by TFD, has a 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 $q_4$ 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 will help eliminate extra gates.

6 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. Our presentation and tests have been focused on the pedagogical and practically relevant example of MaxCut, but the approach is sufficiently general to be applied to QAOA circuits for any discrete optimization problem, such as max E3LIN2 (Farhi, Goldstone, and Gutmann. 2014b), and to arbitrary quantum circuits more generally. Three well-established temporal planners were able to compile the QAOA circuits with reasonable efficiency, demonstrating the viability of this approach. The data used in our tests as well as the PDDL models is available online at (NASA 2017).

This work paves the way for future work on the use of artificial intelligence methods for quantum circuit compilation and design. 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 QAOA circuits with more levels and therefore larger plan sizes, we will develop decomposition approaches in which $p > 1$ 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) (Brierley 2015) (Bremner, Montanaro, and Shepherd 2016), and develop more advanced compilation methods building on the various strengths of the temporal planning approaches and other approaches. A virtue of the planning approach is that the temporal planning framework is very flexible with respect to features of the hardware, including irregular graph structures and diverse gate durations. As hardware graphs and gate durations for processors build by experimental groups become available, we will apply this temporal planning approach with these hardware parameters as input. In the future, we can include in the PDDL modeling additional features that are characteristics of quantum computer architectures, such as the crosstalk effects of 2-qubit gates or the ability to quantum teleport quantum states across the chip (Copsey et al. 2003), and features of broad classes of quantum algorithms including measurement and feedback, error correction, and fault tolerant gate implementations. We will also consider other families of quantum circuits and more sophisticated measures against which to optimize the compilation beyond simply the duration of the circuit. This temporal planning approach to quantum circuit compilation should be of great interest to the community developing low-level quantum compilers for generic architectures (Steiger, Häner, and Troyer 2016b) (Häner et al. 2016) and to designers of machine-instructions languages for quantum computing (Smith, Curtis, and Zeng 2016b) (Bishop 2017).

7 Acknowledgements

The authors acknowledge useful discussions with Will Zeng, Robert Smith, and Bryan O’Gorman. The authors appreciate support from the NASA Advanced Exploration Systems program and NASA Ames Research Center (Sponsor Award No. NNX12AK33A). The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the U.S. Government. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purpose notwithstanding any copyright annotation thereon.
A brief overview of gate-model quantum circuit basics and its abstraction

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. 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. 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. 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. A quantum circuit consists of an ordered sequence of gates, with gates on non-overlapping qubits potentially carried out in parallel. (See Fig. 6 and Fig. 5.)

Gate-model architectures operate as digital computers with a clock, and times could be expressed in terms of clock cycles. Elementary or primitive gates are the gates that are hardcoded in the hardware and could be reasonably considered to be the fastest possible operations, taking a single clock cycle. However, from a sequence of primitive gates, composite arbitrary gates can be synthesized, making it possible to describe an hardware chip in terms of the relevant gates for the algorithm, as long as we take into account the clock cycles that are required to perform the wanted gate. This origin of the "duration" abstraction for gates is exemplified in Figure 6 for the SWAP gate. Beside the timings dictated by gate synthesis, different choices of synchronization and time-scales of executions are possible, leading to different possible durations.

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 measuring, which probabilistically projects each qubit’s quantum state to up or down (0 or 1).

The objective of such quantum algorithms is to obtain, with high probability, a state that when measured, yields a classical bit string that solves the problem of interest.

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.

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

[Bishop 2017] Bishop, L. S. 2017. Qasm 2.0: A quantum circuit intermediate representation. Bulletin of the American Physical Society 62.

[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. arXiv preprint arXiv:1610.01808.

[Brierley 2015] Brierley, S. 2015. Efficient implementation of quantum circuits with limited qubit interactions. 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.

[Copsey et al. 2003] Copsey, D.; Oskin, M.; Impens, F.; Metodiev, T.; Cross, A.; Chong, F. T.; Chuang, I. L.; and Kubiatowicz, J. 2003. Toward a scalable, silicon-based quantum computing architecture. IEEE Journal of selected topics in quantum electronics 9(6):1552–1569.

[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).

[Erdős and Rényi 1960] Erdős, P., and Rényi, A. 1960. On the evolution of random graphs. Publications of the Mathematical Institute of the Hungarian Academy of Sciences 5:569–573.

[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 et al. 2017] Farhi, E.; Goldstone, J.; Gutmann, S.; and Neven, H. 2017. Quantum algorithms for fixed qubit architectures. arXiv preprint arXiv:1703.06199.

[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 arXiv:1412.6062.

[Fu, Wilken, and Goodwin 2005] Fu, C.; Wilken, K.; and Goodwin, D. 2005. A faster optimal register allocator. The Journal of Instruction-Level Parallelism 7:1 – 31.

[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.

[Ghallab, Nau, and Traverso 2004] Ghallab, M.; Nau, D.; and Traverso, P. 2004. Automated Planning: theory and practice. Elsevier.

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

[Häner et al. 2016] Häner, T.; Steiger, D. S.; Svore, K.; and Troyer, M. 2016. A software methodology for compiling quantum programs. arXiv preprint arXiv:1604.01401.

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

[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.

[Kole, Datta, and Sengupta 2017] Kole, A.; Datta, K.; and Sengupta, I. 2017. A new heuristic for n-dimensional nearest neighbor realization of a quantum circuit. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems.

[NASA 2017] NASA. 2017. Repository with complementary materials for the reproduction of the tests of the paper (pddl files and instances). https://ti.arc.nasa.gov/m/groups/asr/planning-and-scheduling/IJCAI17PublicRelease.zip.

[Ramasesh et al. 2017] Ramasesh, V.; O’Brien, K.; Dove, A.; Kreikebaum, J. M.; Colless, J.; and Siddiqi, I. 2017. Design and characterization of a multi-qubit circuit for quantum simulations. In March Meeting 2017. American Physical Society.

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

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

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

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

[Vidal 2014] Vidal, V. 2014. Yahsp3 and yahsp3-mt in the 8th international planning competition. In *Proceedings of the 8th International Planning Competition (IPC14).*

[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.*

[Wille, Lye, and Drechsler 2014] Wille, R.; Lye, A.; and Drechsler, R. 2014. Exact reordering of circuit lines for nearest neighbor quantum architectures. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 33(12):1818–1831.

[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.*