Symbolic Algorithms for Token Swapping

Bruno Schmitt[1] Mathias Soeken[2] Giovanni De Micheli[1]
[1]EPFL, Lausanne, Switzerland, [2]Microsoft, Switzerland

***Abstract*—We study different symbolic algorithms to solve two related reconfiguration problems on graphs: the *token swapping problem* and the *permutation routing via matchings* problem. Input to both problems is a connected graph with labeled vertices and a token in each vertex. The goal is to move each token to its destination vertex using swap operations. In the token swapping problem, the goal is to find a solution with a minimum number of swaps. In the permutation routing via matchings problem, the goal is to find a solution with a minimum number of steps, where a step is a set of disjoint swaps which can be performed in parallel. First, we present an A* search algorithm. This algorithm can find optimal solutions if used with an admissible heuristic. We also evaluate the use of non-admissible heuristics. In this case, we prove that the result will deviate from an optimum result by at most an even number of swaps. We also present an algorithm based on Boolean satisfiability. We evaluate our methods on a large set of practical benchmarks.**

I. Introduction

In this work, we study several symbolic algorithms to solve two related problems on graphs: the *token swapping problem*, introduced by Yamanaka et al. [1], and the *permutation routing via matchings* problem, proposed by Alon et al. [2]. In both problems, we are given a connected graph with n vertices, a set of n tokens and an initial bijective assignment between tokens and vertices, i.e., a permutation. We are also given a target permutation. The goal is to move each token to its destination vertex in the target permutation by applying a sequence of token swaps among adjacent vertices.

The difference between the two problems lies in the optimization goal. In the token swapping problem the aim is to minimize the total number of swaps. In the permutation routing via matchings the goal is to minimize the total number of steps by picking matchings, i.e., a disjoint collection of edges, and swapping the tokens of all vertices connected by an edge on the matching at each step—some authors have referred to this as the parallel token swapping problem. It is known that solutions to both problems always exist [1], [2]. The length of a swap sequence to solve the token swapping problem is $O(n^2)$ [1].

The token swapping problem in its full generality was introduced only recently [1]. Its decision version, where we are interested if a target permutation can be reached in at most k swaps, is NP-complete and APX-complete [3]. The existence of an exact algorithm on general graphs which solves the problem in time $2^{O(n)}$ would refute the exponential time hypothesis [4], [3]. Works studying this problem from a theoretical point of view exist [5], [3]; but only a few practical algorithms to solve the problems have very lately been proposed. For example, the author in [6] adapted frameworks used for solving the multi-agent pathfinding problem to solve the token swapping problem and some of its variations.

Determining the minimum sequence of necessary matchings to move all the tokens for a given permutation is a special case of the minimum generator sequence problem for groups [7]. In this problem we are given a permutation group $\mathcal{G}$ and a set of generators. Given a permutation $\pi \in \mathcal{G}$ the task is to determine if there exists a generator sequence of length at most k that generates π from the identity permutation. This problem is NP-hard [7].

In this paper, we present the results of research directed towards the development and analysis of symbolic algorithms for solving both problems:

- *A*-based algorithm*. We study the use of A* search algorithm [8] to solve both problems using admissible heuristics, which guarantee optimality and non-admissible heuristics.
- When trying to optimize for the number of swaps, we prove that any non-admissible heuristic will obtain a result that deviates from an optimal result by at most an even number of swaps.
- *SAT-based algorithm*. Inspired by [6], we introduced a new encoding which relies only on SAT for solving both problems.

II. Preliminaries

A. Graphs

A *graph* is an ordered pair of sets $G = (V, E)$, where V is a finite set of vertices and E is a finite set of edges or arcs. In an *undirected* graph, the edges are unordered pairs. In this work, we write $u \text{ — } v$ instead of $\{u, v\}$ to denote the undirected edge between vertices u and v. In a *directed* graph, we use the term arc in a set $E \subseteq V \times V$ to denote a link between two vertices and write $u \longrightarrow v$ instead of (u, v) to denote an arc from u to v.

For any edge $u \text{ — } v$ in an undirected graph, we call u a *neighbor* of v, and vice versa. We denote $\delta(v)$ the set of neighbors of v and the *degree* of a vertex is $|\delta(v)|$, i.e., its number of neighbors. In directed graphs, we distinguish two kinds of neighbors. For any directed edge $u \longrightarrow v$, we call u a predecessor of v and v a successor of u. Accordingly, we use $\delta^-(v)$ and $\delta^+(v)$ to denote the set of predecessors and the set of successors of a vertex v respectively.

Given two undirected graphs $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$, we say that a *subgraph isomorphism* from G_1 to G_2 is an injective function $f : V_1 \to V_2$ such that $v_i \text{ — } v_j$ implies $f(v_i) \text{ — } f(v_j)$ for all $v_i, v_j \in V_1$. A *matching* M of a graph G is subgraph where every vertex has degree 1, i.e., no two edges on M have a common vertex.

In a directed graph, a *directed walk* is a sequence of vertices $v_1 \longrightarrow v_2 \longrightarrow \cdots \longrightarrow v_l$ such that $v_{i-1} \longrightarrow v_i \in E$ for every

index i. A directed walk is called a *directed path* if it visits each vertex at most once.

B. Permutation

A permutation is a bijective function $f : X \to X$ where X is a finite set $\{1, 2, \ldots, n\}$. A permutation is represented as a tuple $\pi = (a_1, a_2, \ldots, a_n)$ in which each item k moves to a_k. For example, $\pi = (3, 1, 2, 4)$ implies $1 \mapsto 3$, $2 \mapsto 1$ and $3 \mapsto 2$—in this example item 4 is already on its place. We write S_n to denote the symmetric group on X, i.e., the set of all permutations, and use π_e to denote the identity permutation $(1, 2, \ldots, n)$. A transposition $\tau_{(i,j)} \in S_n$ is an elementary permutation which interchanges two elements and leaves all others unchanged. Any permutation can be decomposed into a sequence of transpositions which can be made canonical. We can represent the previous example permutation as $\pi = \tau_{(3,1)} \circ \tau_{(2,1)}$. Note that we apply transpositions from right to left.

An *inversion* of π is a pair of indices $i < j$ such that $\pi(i) > \pi(j)$, where $i, j \in [n]$ and $\pi(i)$ denotes the i^{th} element. The number of inversions of π is denoted $\mathrm{inv}(\pi)$. The *sign* of a π is $\mathrm{sgn}(\pi) = (-1)^{\mathrm{inv}(\pi)}$. For any transposition $\mathrm{sgn}(\tau_{(\mathrm{i,j})}) = -1$. For all $\pi_i, \pi_j \in S_n$, $\mathrm{sgn}(\pi_i \circ \pi_j) = \mathrm{sgn}(\pi_i) \cdot \mathrm{sgn}(\pi_j)$.

Theorem 1: Let $\pi = \tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)} \circ \tau_{(a_1,b_1)}$ be any decomposition of $\pi \in S_n$ into a sequence of transpositions. Then

$$\mathrm{sgn}(\pi) = (-1)^k.$$

Proof: We can rewrite π as $\pi_2 \circ \pi_1$ where $\pi_1 = \tau_{(a_1,b_1)}$ and $\pi_2 = \tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)}$, then $\mathrm{sgn}(\pi) = \mathrm{sgn}(\pi_1) \cdot \mathrm{sgn}(\pi_2)$. Next we can do the same for π_2 and iteratively for all subsequent π_k such that each π_k is a transposition$\tau_{(a_k,b_k)}$. Clearly, $\mathrm{sgn}(\pi) = \prod_{i=1}^{k} sgn(\pi_i)$. The conclusion now follows by recalling that $\mathrm{sgn}(\tau_{(\mathrm{i,j})}) = -1$. ∎

C. Reconfiguration problems

Definition 1 (Token assignment): Given an undirected graph $G = (V, E)$, with $V = \{v_1, \ldots, v_n\}$, a *token assignment* is a bijective mapping $\Pi : V \to \{1, \ldots, n\}$. By assuming an order on the vertices $v_1 < \cdots < v_n$, a token assignment can equally be described by a permutation $\pi \in S_n$, where $\pi(i) = \Pi(v_i)$.

Given a connected undirected graph $G = (V, E)$, an initial token assignment $\pi_{\mathrm{init}} = \pi_0$ and a target token assignment π_{target}. We define the two problem of interest as follows.

Problem 1 (token swapping, [1]): Find a sequence of transpositions $\tau_{(a_1,b_1)}, \ldots, \tau_{(a_k,b_k)}$ with $v_{a_i} \text{—} v_{b_i}$ such that

$$\pi_i = \pi_{i-1} \circ \tau_{(a_i,b_i)} \qquad \text{for all } 1 \le i \le k, \tag{1}$$

and $\pi_k = \pi_{\mathrm{target}}$, where k is minimum.

Problem 2 (permutation routing via matchings, [2]): Find a sequence of matchings $M_1, M_2, \ldots, M_\ell \subseteq E$ on G such that

$$\pi_i = \pi_{i-1} \circ \prod_{\{v_{a_i}, v_{b_i}\} \in M_i} \tau_{(a_i,b_i)} \qquad \text{for all } 1 \le i \le \ell, \tag{2}$$

and $\pi_\ell = \pi_{\mathrm{target}}$, where ℓ is minimum.

Note that it is always possible to relabel the graph such that π_0 is the identity permutation, $\pi_0 = \pi_e$. We assume such an initial token assignment in the remainder of the paper, without loss of generality.

D. State space

A state space is a 6-tuple $\mathcal{S} = \langle S, A, \mathrm{cost}, T, s_0, S_\star \rangle$ where S is a finite set of states, A is a finite set of actions, $\mathrm{cost} : A \to \mathbb{R}_0^+$, $T \subseteq S \times A \times S$ is a transition relation (deterministic in $\langle s, a \rangle$), $s_0 \in S$ is the initial state, and $S_\star \subseteq S$ is the set of goal states. State spaces are often represented as directed graphs. Given a state space $\mathcal{S}$, we can construct a directed graph $D_s = (S, E)$, where the set of vertices V consists of all possible states, and an arc $v \longrightarrow w$ only if there exists an action $a \in A$ that transforms the state in v to the state in w, denoted $v \underset{a}{\to} w$.

E. Boolean Satisfiability

The Boolean satisfiability problem (SAT) asks whether a given propositional formula representing an n-variable Boolean function f is satisfiable or not, i.e., whether there exists an assignment of variables $x \in \mathbb{B}^n$ such that $f(x) = 1$. When such an assignment does not exist, the formula is said to be unsatisfiable (UNSAT).

The SAT problem is NP-complete [9]. Still, several instances of practical interest are efficiently solvable using state-of-the-art SAT solvers [10]. Most modern SAT solvers require a conjunctive normal form (CNF) encoding of the problem. In such encoding, the presence (absence) of a given property is represented by a positive (negative) literal of a variable. The combined literals form clauses—i.e., a disjunction of literals. The conjunction of clauses forms the CNF. The encoding of a problem is crucial and can significantly impact the run-time of an SAT solver.

III. A*-based Algorithm

Both problems 1 and 2 can be abstracted to the mathematical problem of finding a minimal cost path from a start vertex to a goal vertex in a directed graph $D_s = (S, E)$ which represents a state space. Each vertex $s \in S$ represents a state, i.e., one of the possible token assignments. The set of actions A, which transforms states, changes depending on the problem. In the token swapping problem, A corresponds to the set of edges in G, while in the permutation routing via matchings A corresponds to the set of matchings $\mathcal{M}$ in G.

The graph D_s is not explicitly specified as the number of vertices and edges is too large. Instead, the graph is implicitly represented by means of a set of source vertices $S' \subset S$ and a successor operator $\Gamma : S' \to (S \times \mathrm{cost})^{|A|}$, i.e., an operator that, given a vertex s_i, generates the set of tuples $(s_k, \mathrm{cost}(a))$, where $s_k \in \delta^+(s_i)$, for all $a \in A$.

We employ an informed search algorithm, namely A* [8], to solve the problem of finding a lowest-cost path in D_s. Given a start vertex s_0, the algorithm iteratively generates parts of a subgraph of D_s by applying the successor operator $\Gamma(s_i)$. We say that a vertex has been expanded when the successor operator is applied to it. For each expanded vertex $s_i \in D_s$ a weight is calculated for all its successors s_k, $s_i \underset{a}{\to} s_k$, using

$$\mathrm{weight}(s_k) = g(s_k) + h(s_k), \tag{3}$$

where $g(s_k) = g(s_i) + \mathrm{cost}(a)$ is the cost to reach vertex s_k and $h(s_k)$ is a heuristic function that estimates the cost

from s_k to the target vertex. In other words, $\mathrm{weight}(s_k)$ gives an estimate of the total cost of a path using that vertex. At each iteration, the vertex with the lowest cost is chosen to be expanded. The algorithm expands the vertex with the lowest cost first, some parts of the search space (those that lead to expensive solutions) are never explored. Hence use of a good heuristic is important in determining the performance of A*. Further, if $h(s_k)$ is admissible, that is, it never overestimates the cost of the cheapest path from s_k to a target vertex, then A* guarantees finding an optimal solution.

A. Token swapping

We describe two heuristics for solving the token swapping problem. The first is admissible and follows from the following simple lemma.

Lemma 1: Let $d(\Pi(v_i))$ be the distance of a token $\Pi(v_i)$ to its target vertex v_t. Let K be the sum of distances of all tokens to their target vertices $K = \sum_{i=1}^{n} d(\Pi(v_i))$, then

$$h(s_k) = \frac{K}{2} \tag{4}$$

is an admissible heuristic for solving the token swapping problem.

Proof: Any solution would need a least $\frac{K}{2}$ swap operations as every swap reduces K by at most 2. ∎

Experiments in Section V demonstrate that such admissible heuristic can still take a long time to find optimal solutions for problems with up to 20 vertices. Given the inherent complexity of the problem, this is a good indication that employing an admissible heuristic might be prohibitive, especially if the problem instances grow. Hence, we explored different non-admissible heuristics which aggressively prune the search space. We report results for when using $h(s_k) = K$. Furthermore, we prove the following corollary to Theorem 1:

Corollary 1: Any non-optimal solution to the token swapping problem differs from an optimal solution by an even number of swaps.

Proof: Let π_{target} be any permutation and $\tau_{(a_k,b_k)} \circ \cdots \circ \tau_{(a_2,b_2)} \circ \tau_{(a_1,b_1)}$ be any sequence of swaps which transforms the identity permutation π_e into π_{target}. If $\mathrm{sgn}(\pi_{\mathrm{target}})$ is 1 (-1), then such sequence must have an even (odd) number of swaps, since $\mathrm{sgn}(\pi_{\mathrm{target}}) = (-1)^k$. The conclusion now follows because this is true for both optimal and non-optimal sequences. ∎

B. Permutation routing via matchings

Solving this variant of the problem requires knowing all the matchings of the graph, i.e., computing the set of all combinations of edges in which no two edges have a common vertex. We use ZDDs to represent the set of all matchings—further details on this can be found in [11]. Given a graph $G(E, V)$ the ZDD is defined over the $|E|$ variables $e \in E$. The ZDD with all matchings $\mathcal{M}$ is described by

$$\mathcal{M} = \wp \searrow \bigcup_{v \in V} \binom{\delta(v)}{2},$$

where $\wp$ refers to the ZDD that represents the universal family of all subsets of E.

We also employ one admissible and one non-admissible heuristic to solve this problem. The admissible heuristic follows from the following simple lemma.

Lemma 2: Let $d(\Pi(v_i))$ be the distance of a token $\Pi(v_i)$ to the target v_i. Let K be the maximum distance of all tokens to their target vertices $L = \max d(\Pi(v_i))$, then

$$h(s_k) = L \tag{5}$$

is an admissible heuristic to solve the permutation routing via matchings problem.

Proof: Any solution needs at least L steps as every step can only reduce L by at most 1. ∎

In summary, we explored different non-admissible heuristics which aggressively prune the search space. We report results for when using $h(s_k) = L + K$, where $K = \sum_{i=1}^{n} d(\Pi(v_i))$ as in Lemma 1.

IV. SAT-based Algorithm

Using the state space representation, we formulate the token swapping and permutation routing via matching problems as instances of the Boolean satisfiability problem. In our encoding, one variable is used to indicate whether a token is placed in a vertex at a given step and one variable is used to indicate whether a swap between two vertices took place at a given step. Our encoding uses four different types of clauses to constrain the problem such that the solution corresponds to a valid placement of swaps:

- **C1.** At each level, each token must be assigned to exactly one vertex and each vertex must be assigned to exactly one token.
- **C2.** If at steps l and $l+1$ a vertex is assigned the same token, then no swapping involving that vertex occurred at l.
- **C3.** If at levels l and $l+1$ a vertex is assigned to different tokens, then a swap involving that vertex occurred and must have occurred with its adjacent vertex that at level l is assigned the token.

The last necessary constraint changes depending on the problem we are solving. The constraint 4a is necessary when searching for an optimal number of swaps, while constraint 4b is necessary when searching for an optimal number of matchings.

- **C4a.** At each level at most one swap can occur.
- **C4b.** At each level, each vertex can only be involved in at most one swap.

Let x_{tv}^{l} be a Boolean variable which indicates whether a token t is assigned to the vertex v at the level l. Constraint C1 can be split into two parts: one guarantees that each token is assigned to exactly one vertex, expressed as

$$\forall l \; \forall t, \sum_{v} x_{tv}^{l} = 1, \tag{6}$$

and the other ensures that each vertex is assigned to one token

$$\forall l \; \forall v, \sum_{t} x_{tv}^{l} = 1. \tag{7}$$

Let s^l_{vw} be a Boolean variable representing whether a swap between vertices v and w occurs at step l. The constraint C2

$$x^l_{tv} \wedge x^{l+1}_{tv} \Rightarrow \bigwedge_{w \in \delta(v)} \overline{s}^l_{vw}, \tag{8}$$

where $\delta(v)$ is the set of vertices adjacent to v. Similarly, constraint C3

$$\overline{x}^l_{tv} \wedge x^{l+1}_{tv} \Rightarrow \bigvee_{w \in \delta(v)} (s^l_{vw} \wedge x^l_{tw}). \tag{9}$$

Constraint C4a can be expressed as

$$\forall l \sum_{\{v,w\} \in E} s^l_{vw} \leq 1. \tag{10}$$

Constraint C4b allows multiple non-conflicting swaps per step, i.e., a set of swaps that act on different vertices, and is described by

$$\forall l \; \forall v, \sum_{w \in \delta(v)} s^l_{vw} \leq 1. \tag{11}$$

As the SAT-based approach always answers a given formula in a satisfiable/unsatisfiable (yes/no) manner we need to translate the minimization of the number of swaps (or steps) into series of queries to the SAT solver. We build a formula that encodes a question of whether there is a solution to the problem using a specified number of swaps (or steps) using the above constraints. In practice, for both optimizations goals, this corresponds to the number of steps l we encoded.

Our implementation solves the problem incrementally by adding a new step whenever the SAT formula is unsatisfiable. We do not always start with $l = 1$ as it is possible to analyze each problem instance to get a lower bound on the necessary number of swaps (or steps) required. When optimizing the number of swaps, the lower bound corresponds to the sum of shortest paths connecting the start and target vertices of each token divided by 2 (see Lemma 1), while the lower bound on the number of steps corresponds to the longest shortest path (see Lemma 2).

The encoding can have an enormous effect on the speed with which a SAT solver can find an answer to our problem. Thus it is important to consider means to improve it. In this case, we can reduce the number of variables and simplify some clauses by noting that when encoding the possibility of a token been at a certain vertex at a given step l, we only need to consider those vertices that reachable by the token in l steps from its initial position. Further, when optimizing for the number of swaps, we can reduce the number of queries to the solver by encoding two new steps, instead of one, whenever the solver returns unsatisfiable—this can be done because of Lemma 1.

V. Experimental results

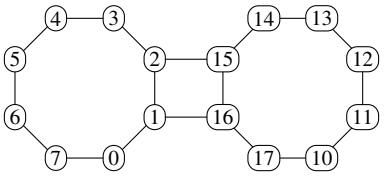
We implemented our methods in C++ into the quantum compilation framework *tweedledum*.[1] We evaluate the different approaches with a set of benchmarks on the latest reported hardware model based on the superconducting circuit technology. All experiments were run on an Intel(R) Xeon(R) CPU E5-2680 v3 @ 2.50GHz. As SAT solver, we used a variation of MapleSAT [12].

[1] github.com/boschmitt/tweedledum

A. Benchmarks

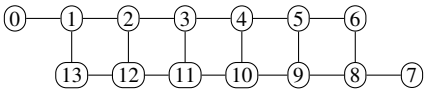
As benchmarks, we use a set of 8338 instances of the token swapping problem which were encountered while mapping quantum circuits from previous works on quantum mapping [13]. The quantum circuits in these benchmarks contain a variety of functions taken from RevLib [14] as well as quantum algorithms written in Quipper [15] or the Scaffold language [16] (and pre-compiled by the ScaffoldCC compiler [17]). The set of circuits was chosen for evaluation because they are functions commonly seen in quantum compilation literature and are publicly available for use.

We partition the quantum circuits as described in [18]. We use three real device topologies. The first is Rigetti's 16-qubit quantum computer where the topology consists of two octagons connected by a square:



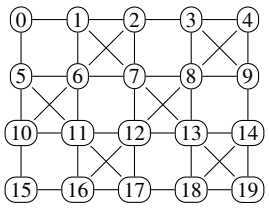
. (12)

We also use the IBM Q14 Melbourne, a 14-qubit quantum computer with the following coupling constraints—we use an undirected graph for this device.



, (13)

and IBM Q20 Tokyo, a 20-qubit quantum computer:



. (14)

B. Methodology

We try to solve each problem instance for two optimization goals (minimum number of swaps and minimum number of steps) by employing three methods: A*-based with admissible heuristic, A*-based with non-admissible heuristic and SAT-based. Thus, we run our program a total of 50028 times. Each time we set a timeout of 3 hours.

Fig. 1 shows the run-time results for solving instances with different optimization goals and using A*-based methods with different heuristics—one admissible and one non-admissible. In the top row the goal it to minimize number of swaps, while on the bottom row the goal is to minimize number of steps. As expected, the results show that the non-admissible heuristic has better run-time for both goals. We note that the non-admissible heuristic is a clear winner when optimizing the number of swaps; when optimizing number of steps, we see that the

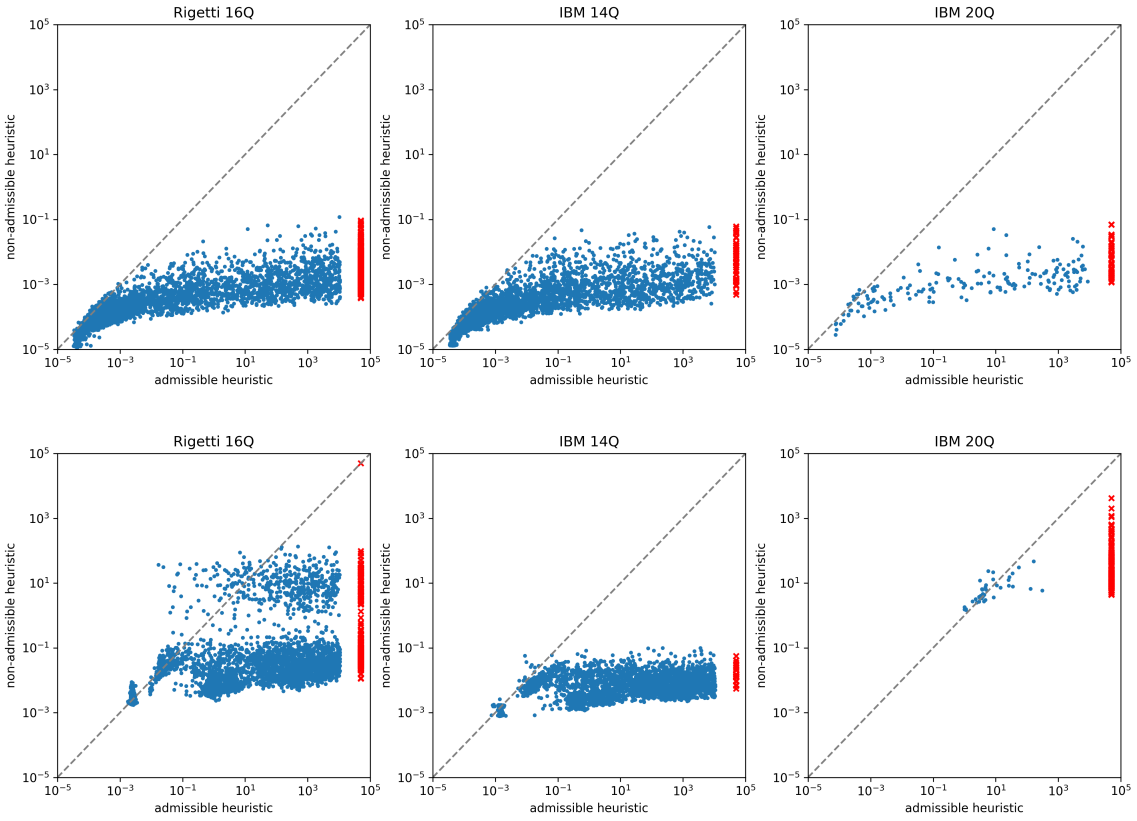


Fig. 1. Run-time comparison between A*-based algorithm with an admissible heuristic and A*-based algorithm with a non-admissible heuristic. In the top row, the optimization goal is the number of swaps. In the bottom row, the optimization goal is the number of steps.

non-admissable heuristic is still a winner, but in many cases performs similar to the admissable heuristic.

Next, we analyzed the quality of results obtained by employing A*-based method with a non-admissible heuristic. We calculate the difference in the number of swaps and steps between the obtained result and a known optimal result for each problem instance. As we proved in Section III, the difference will always be an even number. When solving the token swapping problem the non-admissible heuristic found an optimal solution in 85.2% and 70% of the benchmarks for Rigetti's 16Q and 20Q, respectively. For IBM's 14Q it found solutions using two extra swaps in 87.8% of the cases. In the worst cases, the heuristic would use 8 extra swaps, but this only happens in less than 1% of the cases. When optimizing for the number of steps, the non-admissible heuristic does not perform as good. Indeed, for this case, optimum results are only reached 55.5%, 55.8%, and 13.0% of the time for Rigetti's 16Q, IBM's 14Q and 20Q, respectively. Further, the number of extra steps might be as high as 13.

Fig. 2 shows the run-time results for solving instances with different optimization goals and using the two methods which guarantee optimal results, namely A*-based with admissible heuristic and SAT-based. The top row shows the results for obtaining an optimal number of swaps, while the bottom row shows results for obtaining an optimal number of steps. We find that the SAT-based algorithm is better suited for solving the permutation routing via matchings problem, while the A*-based algorithm does better when solving the token swapping problem—although for many instances this difference is negligible or non-existent.

VI. Conclusion

We presented two symbolic algorithms for solving both the token swapping and permutation routing via matchings problems. We evaluated the algorithms using a set of practical benchmarks obtained while mapping quantum circuits into different device architectures. The results demonstrate that the A*-based algorithm with an admissible heuristic performs better than the SAT-based method when optimally solving the token swapping problem. The SAT-based method, on the other hand, outperforms A*-based algorithm when optimally solving the permutation routing via matchings problems.

We also evaluated the use of non-admissible heuristics when using the A*-based algorithm. For the token swapping problem, we proved that any result from using a non-admissible heuristic, if not optimal, will deviate from an optimal result by at most an even number of swaps. Further, our non-admissible heuristic obtained optimal solutions for the vast majority of the benchmarks. When dealing with the permutation routing

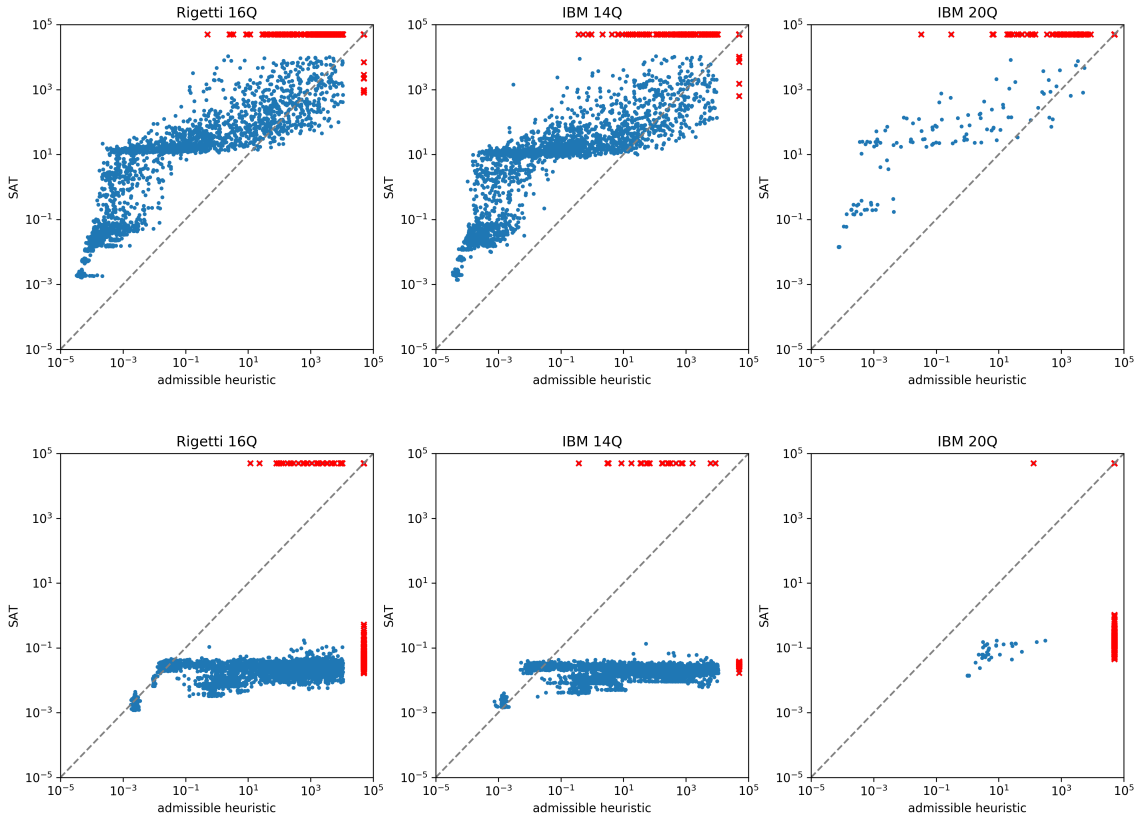


Fig. 2. Run-time comparison between A*-based algorithm with an admissible heuristic and SAT-based algorithm. In the top row, the optimization goal is the number of swaps. In the bottom row, the optimization goal is the number of steps.

via matchings problem, the use of a non-admissible heuristic is not as advantageous.

Acknowledgments: This research was supported by the European Research Council in the project H2020-ERC-2014-ADG 669354 CyberCare.

References

[1] K. Yamanaka, E. D. Demaine, T. Ito, J. Kawahara, M. Kiyomi, Y. Okamoto, T. Saitoh, A. Suzuki, K. Uchizawa, and T. Uno, "Swapping labeled tokens on graphs," *Theoretical Computer Science*, vol. 586, pp. 81–94, 2015.

[2] N. Alon, F. R. Chung, and R. L. Graham, "Routing permutations on graphs via matchings," *SIAM journal on discrete mathematics*, vol. 7, no. 3, pp. 513–530, 1994.

[3] T. Miltzow, L. Narins, Y. Okamoto, G. Rote, A. Thomas, and T. Uno, "Approximation and Hardness of Token Swapping," in *24th Annual European Symposium on Algorithms (ESA 2016)*, ser. Leibniz International Proceedings in Informatics (LIPIcs), P. Sankowski and C. Zaroliagis, Eds., vol. 57. Dagstuhl, Germany: Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, 2016, pp. 66:1–66:15.

[4] R. Impagliazzo and R. Paturi, "The complexity of k-SAT," in *Proceedings of the Fourteenth Annual IEEE Conference on Computational Complexity*. IEEE Computer Society, 1999, p. 237.

[5] É. Bonnet, T. Miltzow, and P. Rzążewski, "Complexity of token swapping and its variants," *Algorithmica*, vol. 80, no. 9, pp. 2656–2682, 2018.

[6] P. Surynek, "Finding optimal solutions to token swapping by conflict-based search and reduction to SAT," in *International Conference on Tools with Artificial Intelligence (ICTAI)*. IEEE, 2018, pp. 592–599.

[7] S. Even and O. Golderich, "The minimum legth generator sequence problem is NP-hard," Computer Science Department, Technion, Tech. Rep., 1981.

[8] P. E. Hart, N. J. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE transactions on Systems Science and Cybernetics*, vol. 4, no. 2, pp. 100–107, 1968.

[9] S. A. Cook, "The complexity of theorem-proving procedures," in *Symposium on Theory of computing*. ACM, 1971, pp. 151–158.

[10] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 6: Satisfiability*, 1st ed. Addison-Wesley Professional, 2015.

[11] K. Smith, M. Thornton, M. Soeken, B. Schmitt, and G. De Micheli, "Using ZDDs in the mapping of quantum circuits."

[12] V. Ryvchin and A. Nadel, "Maple_LCM_Dist_ChronoBT: Featuring chronological backtracking," *Proceedings of SAT Competition 2018*, p. 29, 2018.

[13] A. Zulehner, A. Paler, and R. Wille, "An efficient methodology for mapping quantum circuits to the IBM QX architectures," *IEEE Trans. on CAD of Integrated Circuits and Systems*, 2018.

[14] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Int'l Symp. on Multiple-Valued Logic*. IEEE, 2008, pp. 220–225.

[15] A. S. Green, P. L. Lumsdaine, N. J. Ross, P. Selinger, and B. Valiron, "Quipper: a scalable quantum programming language," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 333–342.

[16] A. J. Abhari, A. Faruque, M. J. Dousti, L. Svec, O. Catu, A. Chakrabati, C.-F. Chiang, S. Vanderwilt, J. Black, and F. Chong, "Scaffold: Quantum programming language," Princeton University, NJ, Dept of Computer Science, Tech. Rep., 2012.

[17] A. J. Abhari, S. Patil, D. Kudrow, J. Heckey, A. Lvov, F. T. Chong, and M. Martonosi, "ScaffCC: a framework for compilation and analysis of quantum computing programs," in *Proceedings of the 11th ACM Conference on Computing Frontiers*. ACM, 2014, p. 1.

[18] W. Hattori and S. Yamashita, "Quantum circuit optimization by changing the gate order for 2D nearest neighbor architectures," in *Int'l Conf. on Reversible Computation*. Springer, 2018, pp. 228–243.