IEEE Transactions on Quantum Engineering

Received 30 April 2024; revised 9 July 2024; accepted 9 July 2024; date of publication 17 July 2024; date of current version 28 August 2024.

Digital Object Identifier 10.1109/TQE.2024.3429451

BeSnake: A Routing Algorithm for Scalable Spin-Qubit Architectures

NIKIFOROS PARASKEVOPOULOS[1,2], CARMEN G. ALMUDEVER[3], AND SEBASTIAN FELD[1,2]

[1]Department of Quantum and Computer Engineering, Delft University of Technology, 2628 CD Delft, The Netherlands
[2]QuTech, Delft University of Technology, 2628 CJ Delft, The Netherlands
[3]Department of Computer Engineering, Universitat Politècnica de València, 46022 València, Spain

Corresponding author: Nikiforos Paraskevopoulos (e-mail: n.paraskevopoulos@tudelft.nl).

This work was supported in part by the Netherlands Organisation for Scientific Research through the Research Program OTP under Grant 16278. The work of Carmen G. Almudever was supported in part by the Spanish Ministry of Science, Innovation and Universities through the Beatriz Galindo Program 2020 under Grant BG20-00023 and in part by the European Regional Development Fund under Grant PID2021-123627OB-C51.

ABSTRACT As quantum computing devices increase in size with respect to the number of qubits, two-qubit interactions become more challenging, necessitating innovative and scalable qubit routing solutions. In this work, we introduce beSnake, a novel algorithm specifically designed to address the intricate qubit routing challenges in scalable spin-qubit architectures. Unlike traditional methods in superconducting architectures that solely rely on SWAP operations, beSnake also incorporates the shuttle operation to optimize the execution time and fidelity of quantum circuits and achieves fast computation times of the routing task itself. Employing a simple breadth-first search approach, beSnake effectively manages the restrictions created by diverse topologies and qubit positions acting as obstacles for up to 72% qubit density. It also has the option to adjust the level of optimization and to dynamically tackle parallelized routing tasks, all the while maintaining noise awareness. Our simulations demonstrate beSnake's advantage over an existing routing solution on random circuits and real quantum algorithms with up to 1000 qubits, showing an average improvement of up to 80% in gate overhead, 54% in depth overhead, and up to 8.33 times faster routing times.

INDEX TERMS Quantum compilation, qubit mapping, qubit routing, routing algorithm, spin qubits.

I. INTRODUCTION

The quantum software's ability to optimally transform hardware-agnostic quantum circuits such that they comply with all operational constraints of the architecture—a process known as mapping—is at the forefront of quantum compiler development right now [1], [2], [3], [4], [5], [6], [7], [8], [9], [10]. One of the tasks of the mapping process is to account for the circuit's two-qubit interactions and to deal with constraints imposed by the device architectures, notably the limited qubit connectivity. These restrictions are overcome by carefully inserting SWAP gates and thus moving quantum information around the devices—usually in nearest neighbor positions—so that the needed two-qubit gates can be performed. This particular process is also known as qubit routing, and its effectiveness is paramount to maximize the operational fidelity of noisy intermediate-scale quantum (NISQ) [11] devices, which are highly prone to different kinds of errors.

SWAP-based qubit routing algorithms have been extensively developed for superconducting quantum devices [12], [13], [14], [15], [16], [17], [18], [19] as these are the most developed qubit technology when it comes to the number of qubits and system availability in general. Spin-qubit devices, however, are not as mature, and consequently, there are no specialized routing techniques yet that can take advantage of their unique features. The fundamental differentiating aspect of a routing algorithm targeting spin-qubit architectures is the primary method of communication. Here, the *shuttling* operation is used as a means of communication within the device instead of SWAP gates used in superconducting technology. Whereas a SWAP gate exchanges the quantum state of two qubits, a shuttle operation results in the physical relocation of a qubit. Note that SWAP gates can also be implemented in spin-qubit devices; however, shuttling operations are preferred due to their superior operational fidelity and faster execution time.

© 2024 The Authors. This work is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 License.
For more information, see https://creativecommons.org/licenses/by-nc-nd/4.0/

Having said that, the shuttle operation brings unique routing challenges that have not been encountered before. A primary example is the need to have at least one, and preferably more, empty site for the qubits to be able to move, as two qubits cannot be allocated to the same place at the same time. Of course, this is not new for modular ion-trap or neutral atom devices, as they also use shuttles for communication. However, due to their fundamental architectural and topological differences with spin-qubit devices, the proposed routing techniques are not compatible. In addition, routing for spin-qubit devices deviates from the conventional notion of only enabling two-qubit interactions. It also extends to other types of operations benefiting from the shuttle operation, which necessitates innovative solutions that are able to handle different cases. One of these operations is a fast and low-error single-qubit shuttle-based Z rotation, as used in this [20] crossbar architecture proposal.

On that note, a shuttle-based SWAP (SBS) routing algorithm for a spin-qubit device has been introduced in [1]. Although it can perform qubit routing in polynomial time in terms of the number of qubits, gates, or two-qubit gate percentage, the algorithm is tightly coupled to the strict architectural constraints of a crossbar architecture [20], making it practically unusable for other architectures. To this date, no routing algorithm has been proposed that fully takes advantage of the shuttle operation, adapts to any architecture, and freely moves qubits around the topology. In this article, we close this gap by presenting beSnake, a novel routing algorithm for scalable spin-qubit architectures. In addition to the mentioned characteristics, it is also able to simultaneously handle multiple two-qubit gates and shuttle-based Z rotations, thus aiming at a low-circuit-depth overhead. beSnake's core design prioritizes execution speed, thus making it suitable for high qubit counts. In order to further reduce the computation time, it can be configured accordingly at the expense of slightly higher gate overhead. Other design considerations give beSnake the ability to choose less noisy shuttling paths and, optionally, use SWAP operations, assuming that the architecture supports them.

We have extensively evaluated beSnake by routing random generated quantum circuits that stress-test all its capabilities. Finally, comparisons with the SBS algorithm [1] on random and real quantum circuits show on average an up to 80% and 54% improvement in gate overhead and depth overhead, respectively, and up to 8.33 times faster routing time.

The main contribution of this article is beSnake, the truly first routing algorithm for scalable spin-qubit architectures. It features the following novel characteristics:

1) utilizing the full freedom of the shuttle operation to move qubits in any direction within a given topology;
2) efficiently handling complex routing scenarios involving parallelized two-qubit and shuttle-based Z gates;
3) adapting to various architecture topologies while being noise-aware and able to adjust the level of optimization;
4) significantly improving over the state of the art (the SBS routing algorithm) with a considerable decrease in gate and depth overhead for up to 1000 qubits.

The rest of this article is organized as follows. Section II presents the problem statement regarding routing for spin-qubit architectures with particular use of the shuttle operation. In Section III, we discuss related work and their limitations, including qubit routing techniques used in other quantum technologies as well as algorithms used in classical computing. We present beSnake in Section IV, explain its internal functionality with three representative examples, and discuss special routing cases in detail. Then, in Section V, we thoroughly analyze the performance of beSnake by considering several configurations related to different levels of shortest path optimizations, the SWAP replacement functionality, the behavior on more connected topologies, and various qubit densities. In the same section, we also compare beSnake with the SBS routing algorithm [1] on random and real algorithms with up to 1000 qubits and discuss the obtained improvements. Finally, Section VI concludes this article and discusses future directions.

II. PROBLEM STATEMENT

Spin-qubit realizations come with unique physical characteristics that make them a promising technology to scale up quantum computing systems. The benefits of spin-qubits include their remarkably small size—up to a thousand times smaller than other qubit technologies—paired with extensive experience in semiconductor manufacturing, prolonged coherence times (around to 20 μs) combined with short gate durations (range of 100–200 ns), and the ability to operate at higher temperatures [21], [22], [23], [24], [25], [26], [27], [28], [29], [30]. As for gate fidelities, they can achieve on average 99.9% fidelity for single-qubit gates and 99% for two-qubit gates [31], [32], [33]. The fundamental building block offering such characteristics is the quantum dot, in which a confined electron or a hole can define a physical qubit [34]. The spin qubit can then be controlled electromagnetically through several carefully fabricated gate electrodes around it. These electrodes can enable single-qubit or two-qubit operations by precise pulse sequences across various multiquantum dot arrangements. Such systems have been extensively studied in 1-D and, recently, in 2-D array formats [23], [35]. Taking, for instance, the crossbar architecture of [20] shown in Fig. 1(a), we can see the different operational lines and sites where spin qubits can be initialized. In this case, they are initialized in a checkerboard pattern. Two-qubit gates are only performed between two vertically or horizontally adjacent qubits, meaning that this architecture, or any architecture for that matter, only supports nearest neighbor interactions. Therefore, its topology can be represented by a 2-D grid, as shown in Fig. 1(b). Each node (circle) represents a site of the grid, and each edge connecting the nodes represents the possibility of interaction between the two sites. This type of abstract view is interchangeably called

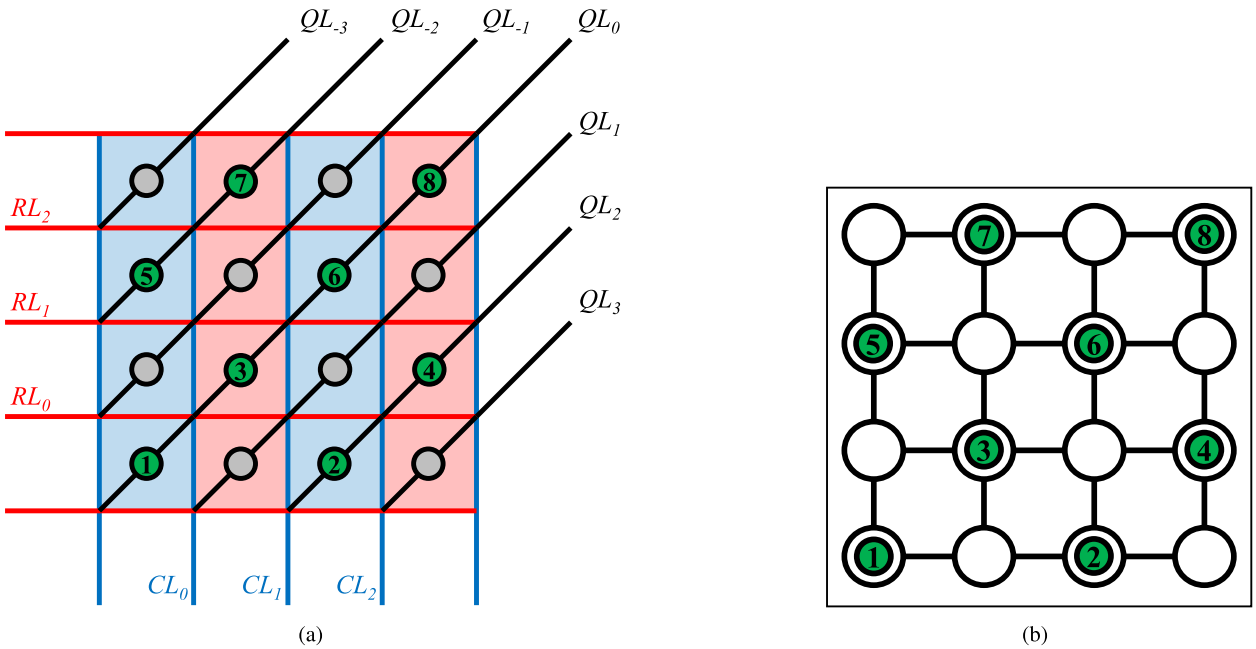

FIGURE 1. (a) Schematic overview of the crossbar architecture, taken from [1], with the various shared operational control lines: vertical (column line, CL), horizontal (row line, RL), and diagonal (qubit line, QL). These lines are used with precise pulse sequences and shared among multiple sites, 16 in this figure, to perform operations on the qubits [20], [38], [39]. Here, the qubits (green circles with numbers) are initialized in a checkerboard pattern. (b) Abstraction of the crossbar architecture representing the coupling between qubits. Each circle represents a quantum dot, and each edge represents a coupling link signifying allowed interactions.

either topology, coupling graph, or layout of the quantum processor.

The problem of bringing the qubits to neighboring positions to execute a two-qubit gate is known as qubit routing. During the NISQ era, it is crucial for the routing process to strive for gate and depth overhead minimization of quantum circuits due to devices suffering from high noise rates. As the number of quantum dots in spin-qubit processors increases in the future, operating them will be even more complex. Qubit routing then becomes a key challenge, especially in ever-increasing topology sizes.

Turning now our attention to other qubit technologies, the central distinguishing aspect of spin-qubit quantum processors is the primary communication method used for the routing process. As mentioned, in superconducting devices, qubits are "moved" employing SWAP gates, in which the quantum state of two neighboring qubits is exchanged. Furthermore, in many superconducting processors, the absence of a native SWAP operation necessitates decomposition in multiple CNOT gates for its implementation, a process that can incur significant errors. In contrast, in spin-qubit architectures, although SWAP operations are supported as well, shuttling is the preferable communication operation as it offers substantial advantages, including higher operational fidelity and quicker execution time. Moreover, certain quantum dots are deliberately left unoccupied to create free space for facilitating qubit movements, as the shuttle process entails the physical relocation of a qubit to an adjacent vacant quantum dot.

However, this introduces new challenges, especially in the context of a quantum compiler routing process, as highlighted in [1]. This is because, at any given moment, there can be only one qubit per site; therefore, it is not physically possible to shuttle a qubit in an already occupied quantum dot. Consequently, there is a need to develop new qubit routing strategies that avoid such conflicts and efficiently move obstacle qubits away. Other limitations are related to the unique constraints imposed by the classical control electronics, especially in those with shared control schemes [1], which might create conflicts in certain cases when trying to apply parallelized shuttle operations.

Broadly speaking, the main objective of a routing algorithm is to determine the most efficient path(s) to satisfy a list of, either parallelized or not, two-qubit gates while respecting the architecture's constraints and avoiding conflicts. Starting with a defined topology, initial physical and virtual-to-physical qubit placement, the algorithm receives a sequence of two-qubit gates, which, in most cases, cannot be executed directly. Therefore, the primary objective of a router is to identify and correctly integrate qubit movement operations (i.e., shuttles or SWAPS) into the existing gate sequence to ensure that all two-qubit gates can be executed. The overarching goal is to minimize the additional circuit overhead required to achieve this. In the end, the final revised sequence, which includes these movement operations, constitutes the output of our routing algorithm. This process was shown to be in complexity class NP-complete [36], [37] and thus necessitates a fast solution, as the goal is to use it for

scalable architectures that will support thousands of qubits in all sorts of topologies. To date, no algorithmic solution has been proposed to tackle this specific path-finding problem in a practically efficient way for the plethora of unique characteristics of spin-qubit architectures.

III. RELATED WORK AND ITS LIMITATIONS

One can find problems in classical computer science that are similar to qubit routing in spin-qubit architectures. One example is the sliding tile puzzle (STP) [40] that aims to achieve a specified arrangement with the least possible moves within a squared grid by orthogonally sliding tiles into an empty slot. One established method for addressing this problem has been the interactive deepening A* algorithm assisted with a precomputed pattern database to increase the accuracy of the admissible heuristic [41], [42]. In more recent work, a 24-tile STP instance was solved using the Levin Tree Search with Context Models [43]. This machine learning algorithm learned from 50 000 solved instances of the puzzle and surpasses previous approaches with fewer steps. Although these two techniques and others have been very effective, they are not a good candidate for the qubit routing problem for scalable spin-qubit devices.

a) These solutions heavily rely on fine-tuned datasets of particular instances and arrangements of the puzzle. In our problem, creating such a dataset will require large computation resources as we are not aiming for one instance or one specific tile (qubit) configuration but rather a list of goals (i.e., making pairs of qubits adjacent).
b) Another factor to consider with point (a) is that with more moving directions per location (higher node degree), the tree of possible moves expands exponentially in size [44], as well. The presented STP results in the literature usually assume these datasets or the training phase and do not disclose the total computational costs. However, upon taking a closer look, it becomes obvious that as the number of qubits increases, these approaches become practically intractable. Our problem requires an algorithm that prioritizes speed before optimality (regarding the number of routing steps) to quickly satisfy multiple goals for more than 24 moving parts.
c) As previously mentioned, circuit executions involve many steps, some of which involve parallelized two-qubit gates. This contrasts with the objective of aiming for a single tile pattern in the STP problem.

Another potentially fitting formulation is the multiagent path-finding (MAPF) problem, which has been studied extensively for multirobot systems [45]. Each agent, representing an independent robotic system, navigates to a target destination through various techniques while avoiding collisions with other agents or obstacles. Agents are sequentially executing a plan consisting of specific goal destinations while trying to minimize the traveled distance within a goal. We have identified, however, the following key differences to our problem.

1) As shown in [45], classical MAPF approaches become slow and unsuccessful for large numbers of agents, especially with the existence of obstacles.
2) Our problem is not limited to a single set of goals (i.e., a set of two-qubit gates) but a plan with multiple goals in each step. Therefore, ultimately, the focus is on minimizing some cost function for the entire plan (representing a quantum circuit), as it is crucial to incur the least errors possible.
3) In our problem, qubits can be viewed as agents and as dynamic obstacles at the same time. In addition, only a subset of agents (qubits) need to complete goals at each step of the plan, and these goals are not target locations but conditions that could be satisfied in multiple locations. Currently, there is a lack of strategies specifically tailored to address this particular problem in a short amount of time, especially when dealing with a large number of agents, as highlighted in [46], [47], and [48].

In the quantum world, several qubit routing techniques have been proposed for superconducting, trapped ion, and neutral atom platforms. Routing for spin qubits resembles a similar functionality of quantum charge-coupled ion trap devices (QCCDs) [49], another promising scalable architecture. In QCCDs, trap regions are dedicated to specific tasks such as temporary storage or processing. Communication between ions of different regions is thus implemented with shuttles through a shared channel where collision avoidance optimization techniques are used [49], [50]. However, these techniques fundamentally differ because they are made specifically for the QCCD's unique regions and topology. The unique structure of a QCCD device imposes a distinct set of constraints on moving qubits and parallelizing operations, different from those found in 2-D spin-qubit topologies. Hence, these routing strategies cannot be implemented in spin-qubit platforms even though both use shuttling [51], [52].

Neutral atom quantum processors also use shuttle operations for qubit communication purposes. The distinctive ability of this platform, besides gate-based swapping [53], is the physical rearrangement of atoms' positions to arbitrary locations as they rely on optically made orthogonal lattices [54], [55], [56]. In state-of-the-art implementations, Rydberg atoms [57] can be dynamically rearranged. This process begins by transitioning from a stationary atom-trap array, which utilizes crossed spatial light modulators (SLMs), to a dynamic atom-trap array using crossed acousto-optic deflectors (AODs) that control horizontal and vertical qubit movement. Consequently, many atom qubits can be relocated simultaneously through the AODs, adhering to specific constraints [53]. These constraints regard the allowed distances between atoms in either SLM or AOD trap types, as well as atom movement dependencies;

namely, columns (or rows) of AOD-trapped atoms are only allowed to be moved horizontally (or vertically) together but without ever allowing crossing of any two columns (or rows). Having said that, different architectural designs [58], [59], [60], [61] can introduce alternative constraints to the routing problem. In general, the primary challenge in directly adopting a neutral-atom shuttle-based routing algorithm for spin-qubit devices lies in the flexibility of the topology, which can be reconfigured almost arbitrarily by the AODs. Although the physical limitations previously mentioned are platform-specific to neutral atoms, in essence, the concept of a "changing" topology does not easily fit with the static nature of quantum dot topologies. Moreover, the majority of routing solutions fail to show scalability for more than ∼200 qubits in a wide range of circuits [53], [58], [60], [61], [62], [63], despite the inherent shuttling flexibility.

In superconducting systems, routing involves consecutive state exchanges by means of SWAP gates along a chosen path until the desired qubit pair is adjacent. The most commonly used methods for qubit routing in such devices are based on heuristic search algorithms or reinforcement learning techniques [50], [51], [52], [64], [65], although recently more theoretical proposals have been introduced for bounding the minimal number of SWAPS needed [66], [67]. The only requirement to determine a viable path is the presence of at least one pair of neighboring locations within the topology where the two qubits can interact. In spin-qubit architectures, on the other hand, routing entails the physical relocation of qubits instead of state swapping. Therefore, within the notion of bringing two qubits closer to each other until adjacent, other qubit positions play an important role as well.

When routing in superconducting devices, the predominant approach is to identify the shortest paths between qubit operands, concurrently minimizing costs across several parameters. These parameters include circuit gate and depth overhead, as well as noise-aware path selection [50], [51], [52], [64], [65]. These can be optimization goals for a spin-qubit routing algorithm as well, but finding the shortest path is not enough because other qubits might block the way. The problem can become even more complex when considering routing for multiple two-qubit gates at the same time while trying to avoid conflicts and satisfy operational constraints of the architecture [1].

The SBS algorithm was conceptualized as the first routing algorithm for large-scale spin-qubit crossbar architectures [1]. In particular, it was tailored to the unique and complex constraints of this crossbar architecture [20], which necessitated the maintenance of the checkerboard pattern of physical qubits to achieve a fast compilation process. One of the main drawbacks of these constraints was the parallelization limitations imposed on shuttle operations. As a consequence, SBS can only, yet efficiently, route one two-qubit gate at a time on grid topologies for that particular crossbar architecture. As extensively discussed in [1], this severely impacts the mapping overhead for high qubit counts and high two-qubit gate percentages.

In this article, we are extending the work on spin-qubit routing to fully utilize and explore the benefits of shuttling-based movement of qubits, assuming a more generalized architecture with fewer constraints. We propose beSnake, a qubit routing algorithm for large-scale spin-qubit processors, which can handle any combination and number of parallelized shuttle-based Z and two-qubit gates in any topology. Furthermore, beSnake is designed to be more flexible, which opens opportunities for more complex routing tasks. We are highlighting their main differences in Table 1. Based on the early stage of spin-qubit technologies, beSnake is introduced as a concrete baseline routing algorithm for future scalable architectures that use shuttling.

IV. BESNAKE ROUTING ALGORITHM

The main objective of beSnake is to enable nonadjacent operand qubits of two-qubit gates to interact with each other by strategically shuttling qubits around the topology. It does so by considering multiple shortest paths between the operands and subsequently implementing an adjustable path selection heuristic to potentially minimize the added movements as much as possible. Upon a path selection, beSnake uses a breadth-first search (BFS) exploration for shuttling qubits; hence, the first part of beSnake's name comes from the first letter of BFS. This approach, with the shortest path selection, ensures the minimization of shuttle operations, viewed from the operand's perspective, significantly reducing the pool of qubits considered for movement. In terms of managing parallelized gates, each is routed in turn, while their operands are held in place once they become adjacent until all of the gates are attempted. In case of failed attempts, three strategies are implemented to overcome them, discussed in Section IV-D1, including a lookback mechanism. In the end, the gates can be simultaneously executed, thereby maintaining the circuit depth as low as possible.

In order to further elaborate on the functionalities included in beSnake, we will use three representative cases based on the squared grid topology shown in Fig. 1(b). In these examples, we will use the notion of the intermediate representation (IR) for the input circuits, as well as output routed circuits with the variable names `ir_input` and `ir_output`, respectively. Within the IR, the gates of a cycle are placed inside square brackets, and commas separate each gate. Note that the notion of a cycle refers to the basic unit of time representing one step in a sequence of gates of a quantum circuit, and each step may contain multiple gates. Moving on, the temporary `goals` variable contains a single cycle of the input circuit given to beSnake for routing, and the `goal` variable holds the current gate from `goals` to be routed. Finally, we will use the `prev_goal_qubits` variable, which contains the pairs of operand qubits that have already been satisfied from `goals`. In Fig. 2, a high-level flowchart of beSnake's main stages is shown. In the next sections, we will expand on each of these stages.

TABLE 1. Comparison of beSnake and Shuttle-Based SWAP

Routing Algorithm	*Shuttle-based SWAP*	*beSnake*
Topology	Squared grid topologies	Flexible
Architecture	Crossbar architecture	Flexible
Parallelization	Works for one two-qubit gate at a time	Works for parallelized two-qubit gates
	Cannot handle parallelized Z and two-qubit gates	Can handle parallelized Z and two-qubit gates
SWAPs	Does not support SWAPs	Supports SWAPs
Optimization	None	Shortest path optimization
Noise-aware	No	Yes

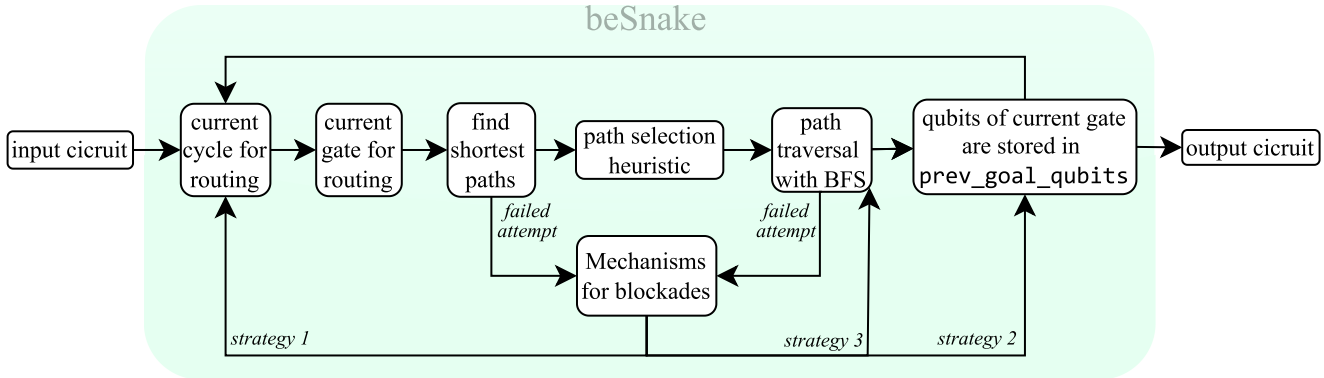

FIGURE 2. High-level flowchart of beSnake's main stages.

A. CASE 1: FOLLOWING A FREE PATH

The first example illustrates the most simple case of qubit routing. We assume a quantum circuit with only two two-qubit gates (`tqg`) in the first cycle [see Fig. 3(a)], whose interacting qubits are placed in nonneighboring positions [see Fig. 3(b)] and therefore require routing. The corresponding IR representation of the quantum circuit can be derived as `ir_input = { [ tqg [1,2], tqg [3,4] ] }`. Note that there can be more than one cycle in an IR, but we assume one for illustrative purposes. As depicted in Fig. 3(b), the beSnake algorithm takes the first gate of the cycle (variable `goals`) and assigns it to `goal`. It then calculates all possible shortest paths between its operands (1 and 2). In this case, there is only one shortest path from qubit 1 to qubit 2, comprising just two nodes. More precisely, qubit 1 will need to move upward two times.

After these movements, qubits 1 and 2 will be adjacent, as shown in Fig. 3(c), and they will remain in that position until the two-qubit gate is executed. Note that the `ir_output` list is updated accordingly, including two new shuttles. In addition, the algorithm updates `prev_goal_qubits` with qubits 1 and 2, indicating that they will remain fixed in their current positions until the rest of the gates are routed. It then proceeds to the next gate in `goals` and repeats the same process. The outcome, as shown in Fig. 3(d), is that the qubits of both gates within `goals` have been successfully routed, and now the `ir_output` encompasses all necessary shuttles to position the qubit operands adjacent to each other. After that, the two-qubit gates can be executed together as originally given from the `ir_input`. At this stage, beSnake has completed the task, and more optimization passes can further improve the output `ir_output`, such as a scheduler parallelizing even more the shuttles required for moving qubits 1 and 3.

This example showed a straightforward operation; however, as we will explore in subsequent examples, certain qubits may block the routing path and, therefore, necessitate relocation to open the way.

B. CASE 2: FOLLOWING A BLOCKED PATH

In our second example, we address a similar situation, but this time, qubits are blocking the shortest paths, as shown in Fig. 4(a). We can identify four potential shortest paths for addressing the first two-qubit gate between qubits 1 and 2, with each path comprising three steps. For simplicity, we focus only on two paths, labeled `a` and `b`, as illustrated.

beSnake employs a two-tiered filtering criterion to optimize the path selection. First, it will keep the path(s) with the least obstacle qubits, and out of the remaining paths, it will select one with the highest accumulated degree of traversed nodes. This is to mitigate the likelihood of congestion in the less connected areas, such as the corners of the current topology. This heuristic aims to minimize obstacle

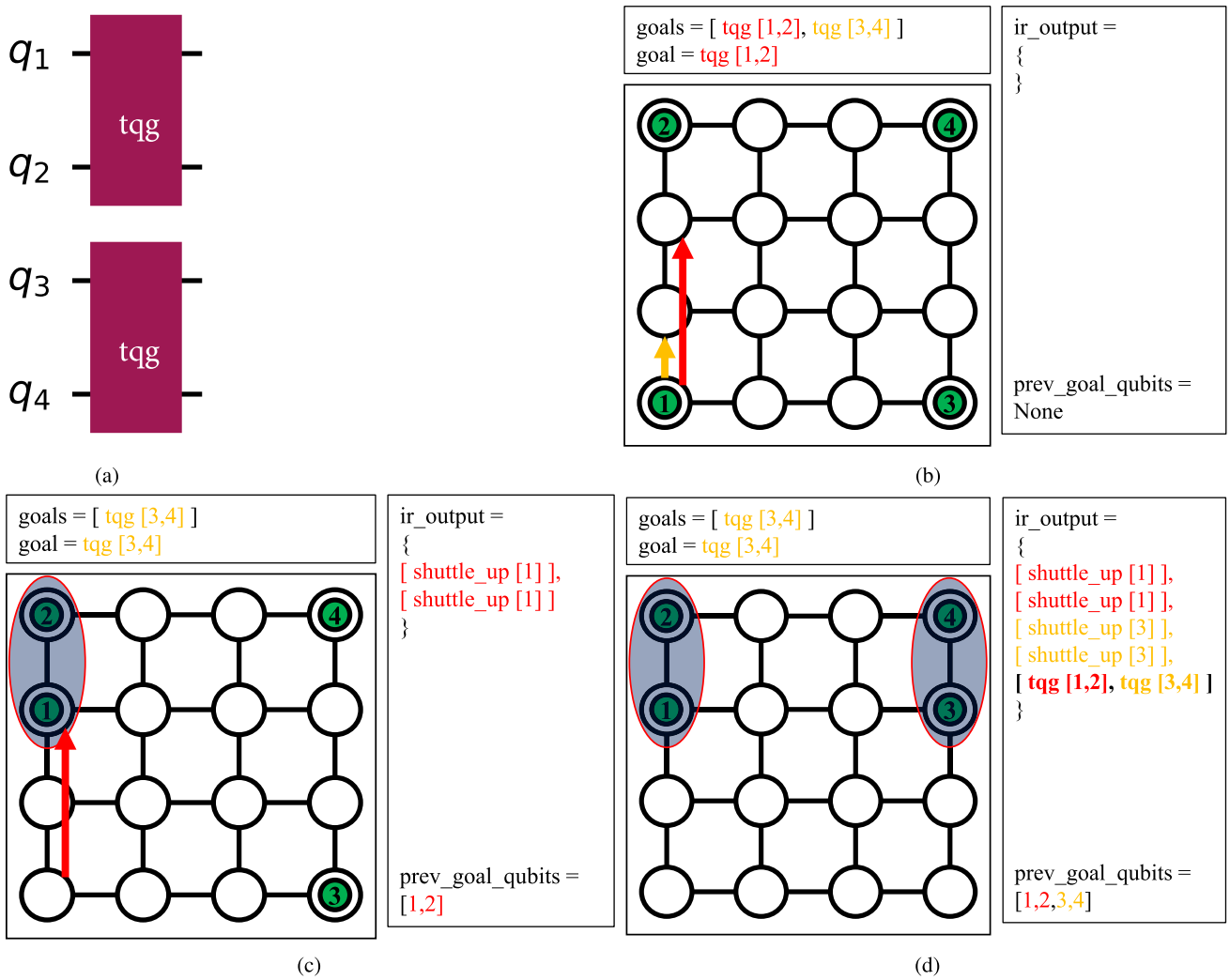

FIGURE 3. Example showcasing how beSnake follows the shortest path to enable two-qubit gate interactions. (a) Diagram circuit representation given to beSnake for routing with the following IR input `ir_input = { [ tqg [1,2], tqg [3,4] ] }`. (b) First step is to consider the shortest paths of the first gate to be routed. In this case, there is only one, shown with the red arrow. The yellow arrow represents the first step of qubit 1 following the shortest path. (c) Completion of the first `goal` with the two qubits becoming adjacent and fixed until the rest of the gates in variable `goals` are tried. (d) Completion of routing for all gates in `goals` and the final IR output.

encounters and to reduce the overhead associated with qubit movement.

Upon comparison, both paths `a` and `b` involve a single obstacle qubit; however, path `b` has a notably higher accumulated degree of the traversed nodes, satisfying our selection criteria. When initiating the first step of the preferred shortest path `b`, as indicated by an orange arrow in Fig. 4(b), we encounter a blockage due to qubit 6. Then, an exploration of potential movements for qubit 6 to clear the path is conducted, and three possible directions are identified: moving left toward qubit 5, moving up along the shortest path, and moving to the right. Of these possible movements, only one is practical—moving to the right. By doing so, we avoid the additional overhead that would otherwise arise later by moving it again away from the shortest path.

Fig. 4(c) showcases qubit 1 having reached qubit 2, updating `prev_goal_qubits` and removing the gate from `goals`. The progression from this point on is similar to the one presented in Fig. 3(d).

C. CASE 3: SATISFYING MULTIPLE TWO-QUBIT GATES

In the complex scenario depicted in Fig. 5(a), beSnake is tasked with satisfying four gates within a cycle amid a denser qubit topology. By considering routing for these four gates simultaneously, we can achieve a lower depth overhead compared to routing for each separately. We focus on the initial gate listed in `goals`. As always, we look for the shortest path with the least number of qubit obstacles and with the largest number of adjacent edges. Among the six conceivable paths, each share an identical qubit count; however, only two paths have the most connections to other vertices. The algorithm

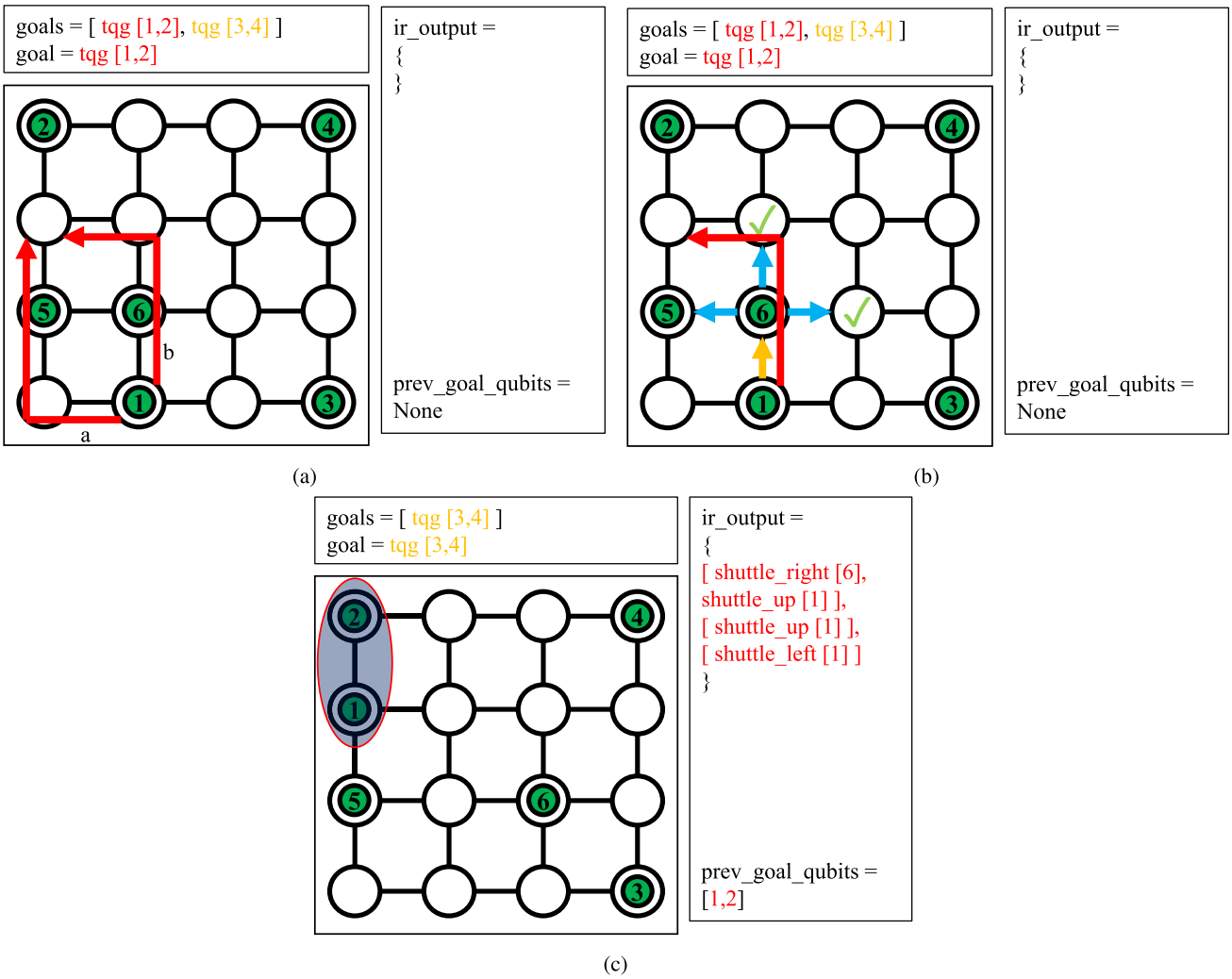

FIGURE 4. Example showcasing how beSnake resolves routing scenarios with blockading qubits within the shortest path for a circuit with `ir_input = { [ tqg [1,2], tqg [3,4] ] }`. (a) First step is to consider the shortest paths and select one based on two heuristic criteria. For illustration purposes, two paths are drawn with an equal number of qubit obstacles. However, path `b` traverses a path with higher accumulated degrees of the traversed nodes, hence its selection. (b) BFS exploration of beSnake to move obstacle qubit 6 with the least possible number of shuttles. Each layer of exploration is depicted with different colored arrows until an empty position can be found. (c) State of the qubit positions after completing the first `goal`. The example can continue from this point on similarly to the example in Fig. 3.

resolves ties by randomly selecting a path, path `b` for this example.

The subsequent phase involves determining the sequence of shuttle movements required to reposition qubit 1 one step further in the chosen shortest path. Fig. 5(b) illustrates the exploratory moves beginning from qubit 1. Intuitively, we can envision this as a BFS style of exploration where qubits sequentially push each other until the first vacancy is encountered. Expressly, qubit 1 is set to push qubit 2 (indicated by an orange arrow), while qubit 2 tries to push qubits 4 and 6 (blue arrows), and qubits 4 and 6 try to push qubits 3, 5, and 7 (purple arrows). At this point, a solution has been identified, rendering additional exploration unnecessary, as it would certainly increase the number of moves.

Fig. 5(c) updates `ir_output` with the first sequence of moves from the first step of qubit 1 within a single cycle. This figure also displays the exploration of qubit 1's next move, with each expansion layer marked by arrows colored orange, blue, and purple, respectively. At this point, we see two possible solutions, moving qubits 3 and 5. Both solutions need the same number of shuttles. BeSnake then becomes noise-aware by selecting the one with the highest accumulated shuttle fidelity based on predefined fidelity attributes for each coupling connection. The accumulated fidelity is calculated by multiplying the fidelities of each shuttle on the specific locations. This mechanism is used in case different fidelities are associated with each coupling link, thus maximizing the routed circuit's fidelity. As a result, the movement of qubit 3 is chosen for this example, and in Fig. 5(d), the sequence of shuttles is displayed in the second cycle of `ir_output`. In the same figure, we have fast-forwarded to the final step of the shortest path and

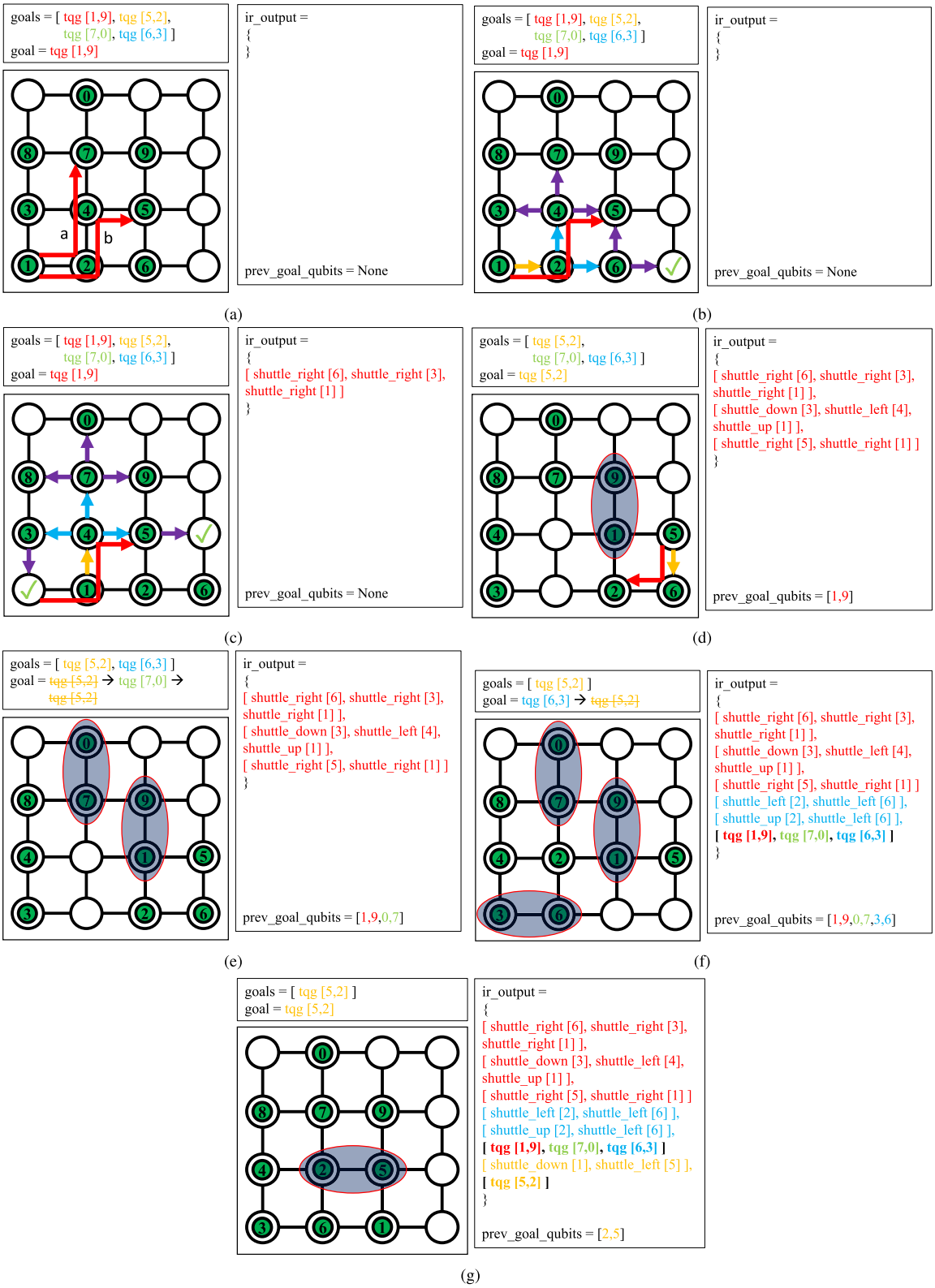

FIGURE 5. **Example showcasing how beSnake resolves more complex routing scenarios with multiple blockading qubits for a circuit with** `ir_input = { [ tqg [1,9], tqg [5,2], tqg [7,0], tqg [6,3] ] }`**. (a) For illustration purposes, we present only two shortest paths that equally satisfy our two heuristic criteria. Path b is selected for this example. (b) BFS exploration of beSnake to move obstacle qubit 2 away with the least number of shuttles. (c) Next BFS exploration for moving obstacle qubit 4 away with the least number of shuttles. Two solutions are found in this case, and beSnake selects one based on the least occurred error rate. (d) Completion of the first gate in** `goals` **and the exploration of shortest paths for the next, which cannot be resolved at this stage. (e) Resolution of the next available gate in** `goals` **and the revisiting of the previous failed one. (f) Completion of the next available gate in** `goals` **and the scheduling of all so-far successful** `goals` **in** `ir_output`**. (g) Previously fixed positions of the scheduled two-qubit gates are freed, and now the two-qubit gate** `tqg [5,2]` **can finally be routed.**

see the three cycles within `ir_output` and updates to `prev_goal_qubits`. The `goal` now includes the next gate from `goals`, and the previous gate has been removed. A single shortest path is discovered since it is not allowed to push fixed qubits, yet qubit 6 is immobilized. This is because the operand qubits are prohibited from being pushed by any other qubit as well. Consequently, beSnake omits this gate and proceeds to the subsequent one. This mechanism is one of three mechanisms implemented (see Section IV-D1) to deal with unsuccessful attempts when routing certain two-qubit gates. In this particular case, we will move on to route the next available gate in `goals`, the `tqg [0,7]`, which might potentially move qubits around the topology and free the blockage shown in Fig. 5(d).

Fig. 5(e) shows that the next goal is immediately satisfied, and the algorithm reattempts the routing of the previously unsuccessful gate, the `tqg [5,2]`. This is the lookback mechanism used to iterate over the remaining gates in `goals`. Once `tqg [7,0]` is accomplished, it is removed from `goals`, and the algorithm revisits the `goals` set, starting with any gates that previously failed. However, the `tqg [5,2]` remains unsuccessful due to the same reasons. Fig. 5(f) demonstrates the resolution of the following gate, `tqg [6,3]`, the next gate in `goals`, at which stage all gates have been satisfied except for one. A repeated attempt yields no solution, as no shortest path is viable anymore between qubits 5 and 2. Due to this fact, we update `ir_output` by scheduling only the satisfied gates, thus splitting the original cycle.

Subsequently, all qubits in `prev_goal_qubits` are released, and the algorithm reattempts the unsolved gate. This time, a solution is possible, and Fig. 5(g) depicts the successful outcome.

D. SPECIAL CASES

This section introduces the mechanisms employed by beSnake to address special routing cases. We outline three primary mechanisms utilized to circumvent blockades, showcasing the algorithm's potential to adapt to conflicts.

1) MECHANISMS FOR DEALING WITH BLOCKADES

An obstacle qubit might disallow the movement of any operand qubit within the shortest path. For example, in Fig. 5(c), obstacle qubit 6 is blocking the very first step in the shortest path. However, there can be cases where a similar situation is reached at any step in the path due to qubits being pushed in areas of the topology with fewer connections, such as the corners of squared topologies. Then, the following mechanisms will be tried in this order.

1) Revisit the same gate (`goal`) after satisfying other ones. This lookback mechanism was used in Fig. 5(e).
2) Splitting the original cycle and trying the remaining gate(s) without the previously fixed qubits. This mechanism was used in Fig. 5(f) and (g).

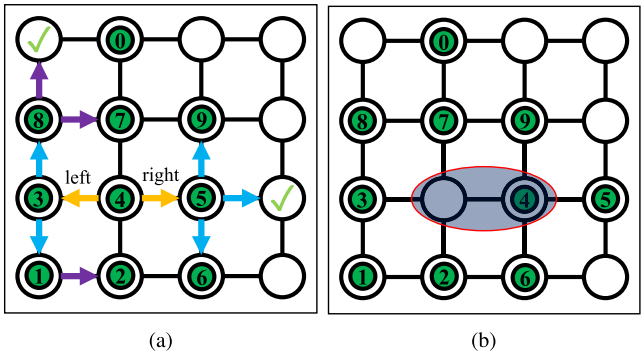

FIGURE 6. Example of how beSnake optimizes the number of shuttles for shuttle-based Z gates. (a) beSnake will explore both possibilities to shuttle qubit 4 left and right and select the one with the least moves. In this case, moving qubit 5 to the right requires fewer shuttles than trying to move qubit 3. (b) Qubit positions after the shuttles and the fixation of positions to ensure qubit 4 can return in time.

3) Execute a forced SWAP between the obstacle and the operand qubit.

Since the first two mechanisms have been demonstrated in Section IV-C, we are now focusing on the third one: If we isolate the scenario in Fig. 5(d) and suppose that the first two mechanisms failed, then we will insert a forced SWAP between qubits 5 and 2 to overcome this obstacle. This mechanism was implemented to efficiently resolve such rare scenarios where qubit obstacle(s) are located in between both operands while having no available edges to be pushed away. This can be triggered in topologies that may have only two available coupling links at one or more nodes (quantum dots), as shown in case 3 of Section IV-C. In the case of a square grid, this place is at the corners. Alternatively, this can be addressed by taking another path, not necessarily a shortest path, that avoids such restricted regions. However, this does not guarantee a solution and most likely will result in more overhead. In Section VI, more alternatives are discussed.

2) HANDLING SHUTTLE-BASED Z ROTATIONS

Shuttle-based Z rotation gates are a unique high-fidelity implementation of a Z rotation with two time-sensitive qubit shuttles to and from a neighboring column in either direction [1], [20], [38], [39]. This kind of single-qubit manipulation technique, otherwise called "hopping spins" [68], can, in fact, achieve an arbitrary rotation axis, but in this work, we assume the proposal in [20] for simplicity. BeSnake can determine which direction (left or right) will result in the least sequence of shuttles. Then, it will fix the positions (one empty and one containing the qubit operand of the Z gate) so that the return shuttle can be inserted immediately. To demonstrate that, we suppose a shuttle-based Z rotation on qubit 4 in the example of Fig. 5(a). This will result in the example shown in Fig. 6(a), where moving to the right takes fewer moves than moving left. In Fig. 6(b), the original (now empty) location of qubit 4 is not allowed to change

until the return shuttle is executed. More specifically, if the empty location gets occupied before the return shuttle, qubit 4 cannot return in time. In case there are parallel Z gates to be satisfied in a cycle, beSnake will follow the same logic as described in Fig. 5(e) and (f) and iteratively try all of them. An exception is applied here where succeeding Z gates can override fixed empty locations from preceding Z gates. Since implementing Z gates involves a second shuttle back to the original location, multiple of these shuttles can take place simultaneously to complete the cycle even if they previously occupied fixed empty locations of other Z gates. One example of this exception is when a shuttle-based Z gate for qubit 3 is scheduled together with the Z gate on qubit 4 in Fig. 6(b). Qubit 3 will shuttle to the right since this is the only available direction, overriding the fixed position created by the Z gate of qubit 4. Then, supposing that the cycle finishes here, both qubits can return to their original locations with the well-timed return shuttles, completing their Z rotations. This exception provides better parallelization as extra room is provided for more Z gates while maintaining the same gate overhead (as we would have without this exception).

Finally, in case there are Z and two-qubit gates in a cycle, the two-qubit gates will be sorted at the beginning of `goals` to be tried first. This is because routing for two-qubit gates will more likely require multiple steps to complete, and therefore, in the beginning, it is more important to have as few fixed positions as possible (i.e., more free space to push/move qubits). In Section VI, more ideas are discussed related to sorting gates in cycles.

E. OPTIONAL FUNCTIONALITIES

beSnake has two optimization settings that can potentially improve the performance of both the routed circuit and the speed of the algorithm itself.

1) *SWAP replacement:* beSnake can become even more noise-aware by replacing a sequence of shuttles, facilitating a step within the shortest path through a SWAP operation. One example is the sequence of shuttles inserted in Fig. 5(c). It will do so when the accumulated fidelity of those shuttles exceeds that of the SWAP, particularly in these locations.
2) *Shortest path optimizations:* As described before, beSnake will assess multiple shortest paths and heuristically pick one that will most likely produce the least amount of gate and depth overhead. However, this ability is by far the most time-consuming task of beSnake as the discovery of all shortest paths, and subsequently, their filtration is iterated several times. To mitigate that, there is an option to place a time limit on the discovery of the shortest paths, thus substantially reducing the time this process takes. In addition, there is an option to take a single shortest path, which significantly speeds up the process. Later, in Section V-B1, we will deliberately test these options and determine the routing time benefits at the expense of increased circuit overhead.

F. SUMMARY OF MAIN LIMITATIONS

In this section, we summarize the main limitations of beSnake based on the previous illustrative examples.

1) *Operand qubits:* beSnake relies on the operand qubits following the selected shortest path only. This means that these two qubits cannot be pushed outside of the designated path as obstacle qubits can, which can restrict some qubit movements, as demonstrated in Section IV-C. Consequently, the mechanisms for blockades discussed in Section IV-D1 will be triggered.
2) *Path selection:* beSnake considers only the shortest paths and tries to heuristically select one in an effort to minimize qubit movements in the future. However, this particular two-tiered path filtration does not always guarantee the least qubit movements in subsequent steps as it is a heuristic solution. Therefore, another more optimal path could reduce the gate overhead produced and execution time. While Dijkstra's algorithm is one method to implement this approach, it is important to account for the scalability of any implementation for a wide range of topology graphs.
3) *High qubit density:* Since beSnake predominately relies on shuttling, it is expected that with fewer empty sites (quantum dots), computation time will increase. In fact, later in Section V-B4, we test a range of qubit densities, and we find that after 72% ratio, the computation time becomes exponential.
4) *Intermediate scheduling:* As illustrated in the preceding three examples, each routing step may generate multiple shuttle operations. For instance, Fig. 3(d) displays two pairs of shuttle operations inside `ir_output`, each associated with a two-qubit gate (`tqg`) that are color coded in red and yellow. It is apparent that qubits 1 and 3 can be simultaneously moved as pairs, thereby reducing the `ir_output` by two cycles, but beSnake does not have the ability to intermediately schedule them.
5) *Parallelization implementation:* As shown in the third example of Section IV-C, the parallelization of two-qubit gates is satisfied by routing one qubit at a time, fixing the qubit operands once adjacent and repeating the same for the subsequent two-qubit gates. However, fixing the qubits' position gradually restricts the topology, as demonstrated in Fig. 5(d). If those qubits were allowed to be moved without breaking their adjacency (given this is implemented efficiently), it could increase the performance of beSnake. More considerations are discussed in Section VI.

V. SIMULATIONS AND EVALUATIONS

With the following simulations, our goal is to test beSnake's different functionalities and create valuable performance insights under various use cases. More specifically, we test beSnake's capabilities with different levels of shortest path optimizations and SWAP replacement options, as well as its

behavior on more connected topologies and ranges of qubit densities. Based on these insights, we move on to thoroughly compare it with the SBS algorithm on synthetic and real quantum circuits with up to 1000 qubits.

A. EXPERIMENTAL SETUP

The beSnake and SBS routing algorithms have been incorporated into the SpinQ compilation framework [1] and executed single-threaded on a 2.4-GHz and 768-GB memory server running Python 3.6.8.

Regarding the metrics used to determine their performance, we have used the total computation time of routing for each circuit, which we simply named *routing time*. In the case of routing time comparisons, we provide the ratio (*rTime*) of the two results as ($results_a/results_b$). It should be noted that routing times can vary depending on the server's total load and dynamic job scheduling at the time of submission. We have taken measures to maintain good time consistency between our simulations by submitting jobs in dedicated resources with no influence from other jobs.

As for overhead metrics, we have used the same gate and depth overhead definitions as in [1]. *Gate overhead* is calculated as the percentage relation of additional gates incorporated by the router divided by the number of gates (after decomposition). *Depth overhead* is expressed as the percentage of extra depth generated by the router in relation to the initial ideal circuit depth (after decomposition). Whenever there are overhead comparisons, we use a relative performance metric. Specifically, the relative gate overhead (*rGO*) and relative depth overhead (*rDO*) are calculated as ($results_a - results_b$)/$results_a$.

Furthermore, the physical initialization of qubits follows a checkerboard pattern and with one-to-one virtual-to-physical qubit allocation, similar to the ones used in [1]. Although the checkerboard pattern has a qubit density of 50%, in Section V-B4 particularly, we use a different physical placement to achieve a range of qubit densities.

Finally, both algorithms are benchmarked with randomly generated circuits, as well as 11 well-known quantum algorithms. Regarding the randomly generated quantum circuits, they consist of two-qubit and shuttle-based Z gates in three different ratios (25, 50, and 75 two-qubit gate percent) and are sampled ten times for each data point. With this choice of random algorithms, which contain many combinations of the two gate types parallelized in different ways, we aim to stress-test both routing algorithms. As for the number of gates, we fixed it at 3000 gates since its impact on the gate and depth overhead is negligible, according to [1]. The 11 real quantum algorithms were scaled up to 1000 qubits to observe their performance in Section V-C2. These algorithms were taken from the Qlib [69], Revlib [70], MQT Bench [71], and qbench [72] libraries.

B. BENCHMARKING BESNAKE

This section focuses on testing beSnake's various functionalities. Specifically, we used four time limits dictating the shortest path optimization level and compared them with taking only one shortest path in Section V-B1. In Section V-B2, we test the behavior of the SWAP replacement option and observe the scaling on three SWAP fidelities while maintaining the shuttle fidelity constant. Then, in Section V-B3, we are investigating beSnake's time performance on more connected topologies, whereas, in Section V-B4, we are fixing the topology sizes but vary the qubit density.

1) ADJUSTING THE SHORTEST PATH OPTIMIZATION

In the following simulations, we explore the effect various time limits have on gate overhead, as outlined in Section IV-E. Therefore, in Fig. 7, we compared the gate overhead (in blue) and total routing time (in red) for 0.05-, 0.45-, 0.85-, and 1.05-s time limits imposed to `NetworkX.all_shortest_paths()` function against just taking one shortest path with `NetworkX.shortest_path()`. The random circuits' gates have been fully (ideally) parallelized based only on their dependences, and beSnake is configured without the ability of SWAP replacements (see Section IV-E), such that no other factor can influence the results. The underlying question to answer with this simulation is as follows.

- Which time limit offers the best relation between gate overhead and routing time, and how does it compare with just taking one shortest path?

In Fig. 7(a)–(c), we can observe the gate overhead for 25, 50, and 75 two-qubit gate percentages, respectively. We observe that in all cases, the increased time limits yield a benefit in terms of gate overhead compared to taking only one shortest path, but the benefits are negligent among the time limits. As expected, higher two-qubit gate percentages result in higher gate overhead and routing time. Thus, the gate overhead gap between the one shortest path and the time limit simulations is more prominent in higher percentages. However, the overall routing time is severely increasing for each time limit. In fact, finding only one shortest path appears polynomial in time over the number of qubits with a small rGO increase of 11.98%, 11.87%, and 11.76% on average for 25, 50, and 75 two-qubit gate percentages, respectively. To solidify this point, in Fig. 7(d), we isolated the one shortest path option, clearly showing a nearly linear behavior. A 0.05-s time limit is thus sufficient; increasing it beyond this limit does not yield significant gate overhead benefits, and further increases negatively impact the total routing time.

2) SWAP REPLACEMENT

In this simulation, the behavior of the SWAP replacement option on all three metrics is investigated. To test the effect of different fidelity scales, we have fixed the shuttle to 99.98%, but tested SWAP gates with 99.97%, 99.95%, and 99.94% fidelities. As explained in Section IV-E, the SWAP replacement is triggered when the accumulated

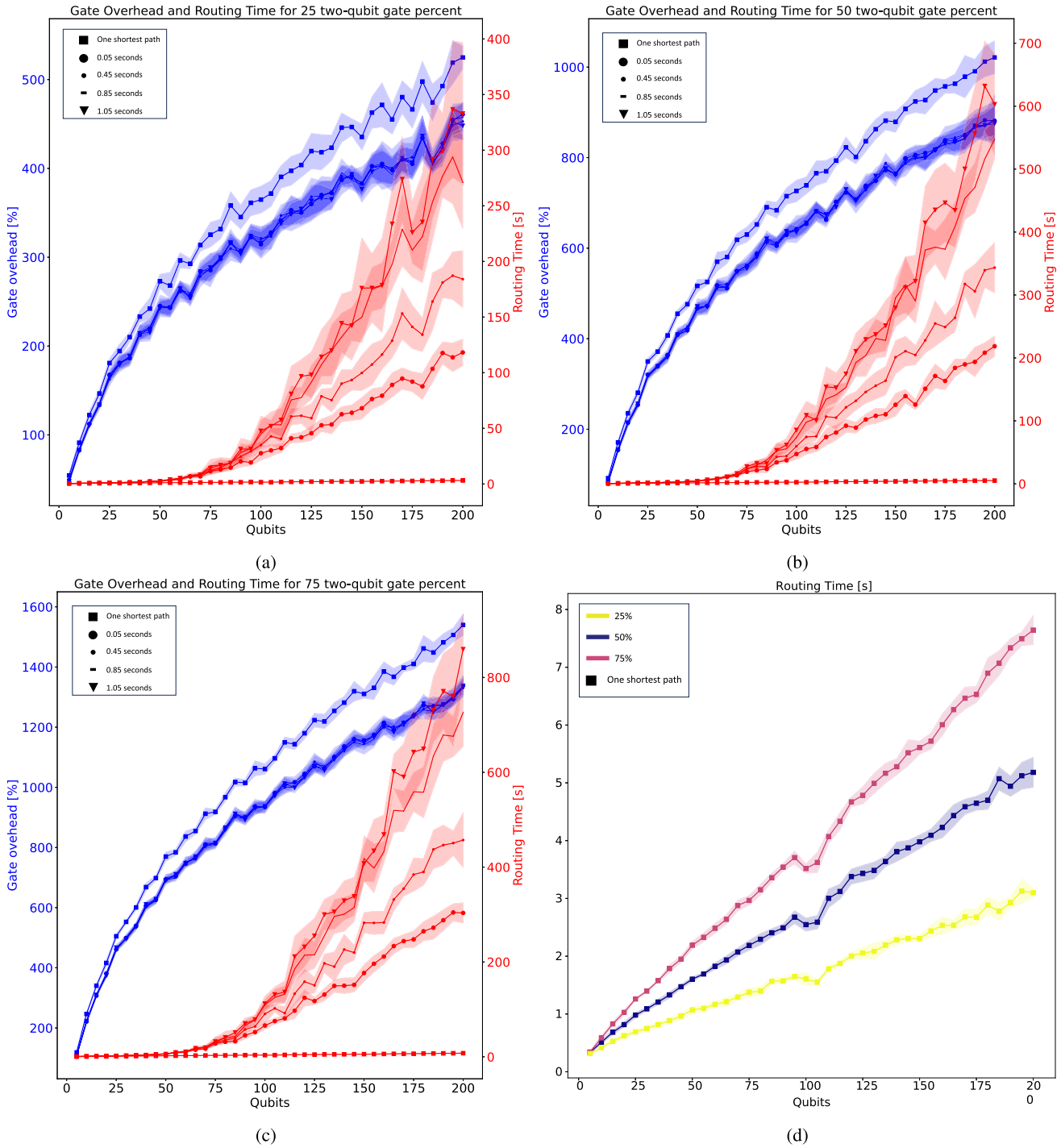

FIGURE 7. **Routing time and gate overhead of beSnake for different shortest path optimizations. Each optimization is dictated by the time limit on finding the shortest paths, except for one option that considers just one shortest path. (a)–(c) show the results of 25, 50, and 70 two-qubit gate percentages. (d) focuses only on one shortest path.**

shuttle fidelity is lower than the SWAP fidelity. Intuitively, we expect replacements to be triggered more often with higher SWAP fidelities. It should be noted that these fidelity values are chosen for the sole purpose of demonstrating the progressive behavior of the SWAP replacement functionality of beSnake and not to represent realistic properties of spin-qubit devices. Finally, the gates in the random circuits are serialized in order to pronounce the effect in both overheads (when replacing multiple shuttles at once), and the shortest path optimization is fixed with a 0.05-s time limit. The underlying question to answer with this simulation is as follows.

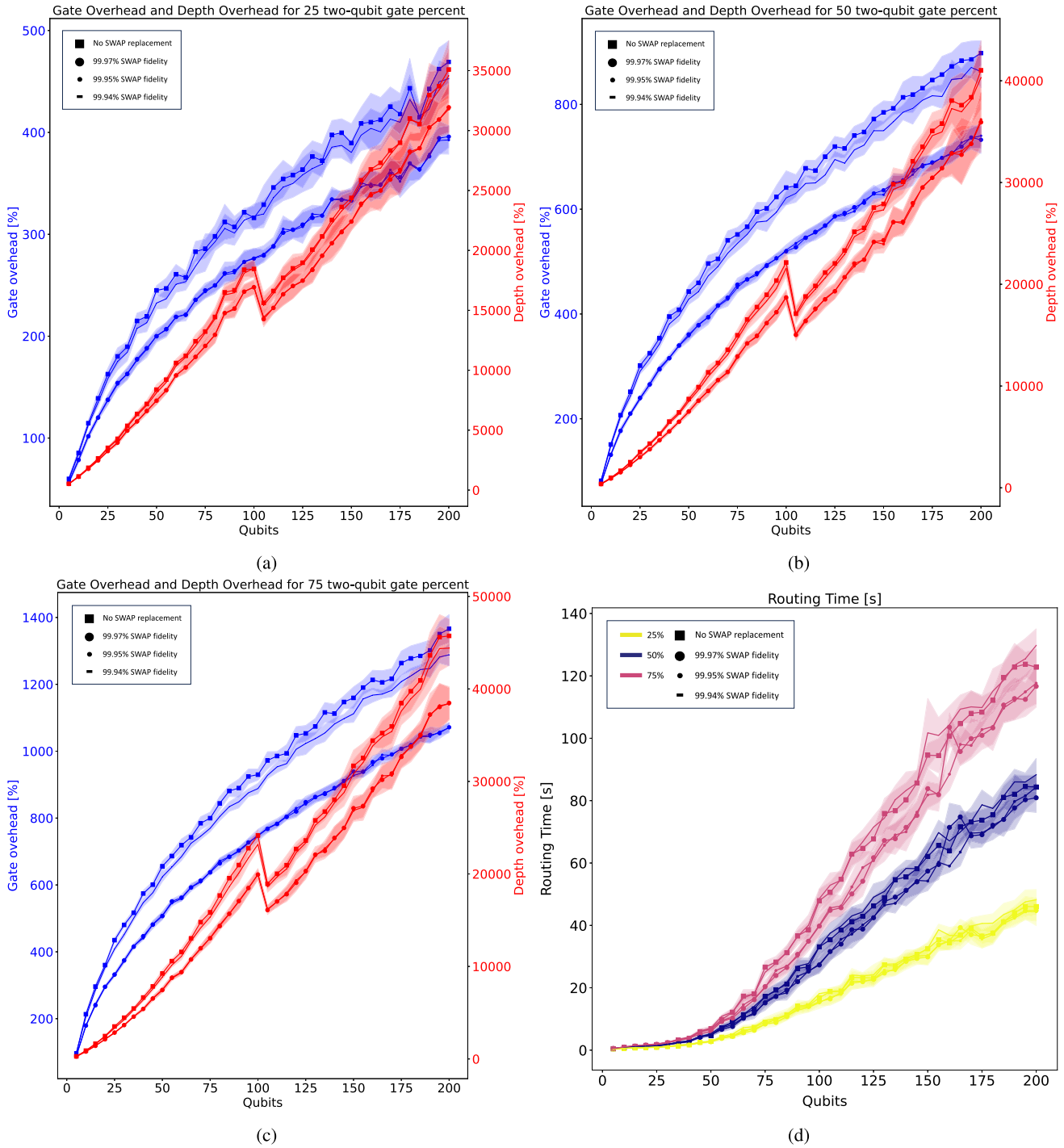

FIGURE 8. **Gate overhead, depth overhead, and routing time of beSnake while using the SWAP replacement option with varying SWAP fidelities. (a)–(c) show the results of 25, 50, and 70 two-qubit gate percentages, whereas (d) focuses only on routing time.**

- Is the potential increase in routing time with beSnake worth it for architectures that support SWAP gates?

In Fig. 8, SWAP replacements improved all overheads for every two-qubit gate percentage. As expected, the SWAP replacement was utilized more often with higher SWAP fidelities and higher two-qubit gate percentages. However, there is no significant overhead difference shown in the figures for higher than 99.95% (see the clustered lines), potentially due to the maximum accumulated shuttle fidelity being somewhere between 99.94% and 99.95%. In particular, three shuttles are needed to obtain an accumulated fidelity

between 99.94% and 99.95%. This indicates that the BFS exploration never reached more than three layers, which is equivalent to three consecutive shuttles [similar to the example in Fig. 5(b)]. Unexpectedly, we observe that the routing time has not worsened but mostly improved for higher than 99.94% fidelity, possibly due to less time taken to update the IR data structure of the circuit with just one gate (a SWAP) instead of multiple shuttles. Therefore, the SWAP replacement functionality can be used without an execution time penalty in most cases. It should be noted that, given a higher qubit density on the topology, there can be more consecutive shuttles at one time, which will lower the trigger requirement and allow for even lower SWAP fidelities.

3) BEHAVIOR ON MORE CONNECTED TOPOLOGIES

In the following experiments, we are interested in seeing how beSnake performs on topologies with higher connectivity. Here, the comparison is made between a squared grid topology (i.e., each qubit is coupled with four neighboring qubits and with two at the corners) and a squared grid topology in which diagonal edges in every direction have been added (i.e., each qubit is coupled with eight neighboring qubits, and with three at the corners). Of course, it is expected that there will be an improvement in both overheads on the topologies with more connections. However, the increased graph size and solution space might negatively impact beSnake's speed in exploring solutions. The random circuits are given the same way as in Section V-B1, as well as the configuration of beSnake, but only with one time limit of 0.05 s. The underlying question to answer with this simulation is as follows.

- Does a highly connected topology adversely affect the efficiency of beSnake in quickly finding solutions?

As can clearly be seen in Fig. 9, beSnake can obtain significantly better gate and depth overhead as was expected, and surprisingly, it can also achieve it in less time. In particular, beSnake is, on average, 1.17, 1.24, and 1.27 faster (rTime) for 25, 50, and 75 two-qubit gate percentages, respectively. The benefit is more pronounced with higher two-qubit gate percentages, which is expected. One explanation for the faster times can be attributed to the shorter traversed paths, hence fewer steps to take and fewer BFS exploration layers.

4) QUBIT DENSITY

Unlike the other simulations using a checkerboard physical initial placement pattern, we will try a different placement to vary the qubit densities here. The topology sizes are fixed, and qubits are placed next to each other in a left-to-right bottom-up approach until a particular density is reached. As for the executed circuits and beSnake's configuration, they are similar to Section V-B1 with a 0.05-s time limit. The underlying question to answer with this simulation is as follows.

- Up to which qubit density does beSnake perform in a scalable manner based on all three metrics?

In Fig. 10, we have simulated seven levels of densities in each subfigure for four different topology sizes: 25-, 49-, 64-, and 100-site topologies. On the y-axis, we can observe the normalized mean values between 0.1 and 1 for all three metrics. Focusing on the 25-site topology, routing time (in green) increases exponentially after 72% density, which is expected primarily due to the additional time needed to finish each BFS exploration. This becomes more severe at 88% density, where there are only three empty locations on the topology. Observing the routing time for the larger topology sizes, we see a steeper upward trend after 72%. However, the routing time behavior until 72% appears linear for all sizes. Notably, it becomes flatter for larger sizes, indicating that beSnake can perform better on larger topologies. This shows that beSnake's qubit density sweet spot is around 72%.

It should be noted, however, that for another experimental setup in regard to the hardware used to run beSnake, we might obtain a different trend and possibly an improved one if beSnake becomes multithreaded, for instance. In addition, changing device characteristics can change beSnake's performance. For example, increasing the connectivity will favor the run time based on the simulations in Section V-B3. Having said that, the total run time might be reduced equivalently for all densities, and therefore, the same behavior at 72% could still appear. Whether this sweet spot (or around that region) is universally found for any given setup is likely but left open for future work.

Focusing now on both overheads, we observe overall that beSnake favors gate overhead (in blue) over depth overhead (in orange), which is a direct outcome of the core routing strategy described in Section IV. Then, depth overhead follows a linear trend for all sizes. Surprisingly, gate overhead creates a slight curve gravitating toward 72% qubit density. Therefore, this reinforces the previous conclusion that beSnake can operate well for more than 50% qubit density and particularly 72% for our particular setup. Finally, the simulations show that in terms of gate and depth circuit overhead, there are no exponential penalties for higher than 72% densities.

C. COMPARISON BETWEEN BESNAKE AND SBS

In this section, we will compare the performance of both routers for random and real circuits for up to 1000 qubits. To facilitate a fair comparison (later named as *beSnake fair*), the circuits are given with their gates as parallelized as possible based on SBS's maximum capability [1]. The SWAP replacement of beSnake has been deactivated as well because SBS does not support it. In addition, beSnake's shortest path optimizations are disabled, and we only use one shortest path (see Section V-B1) to obtain the quickest routing times at the expense of overhead (see Section V-B1). As noted before, beSnake can handle more complex routing tasks with ideally scheduled circuits based purely on their gate's dependencies,

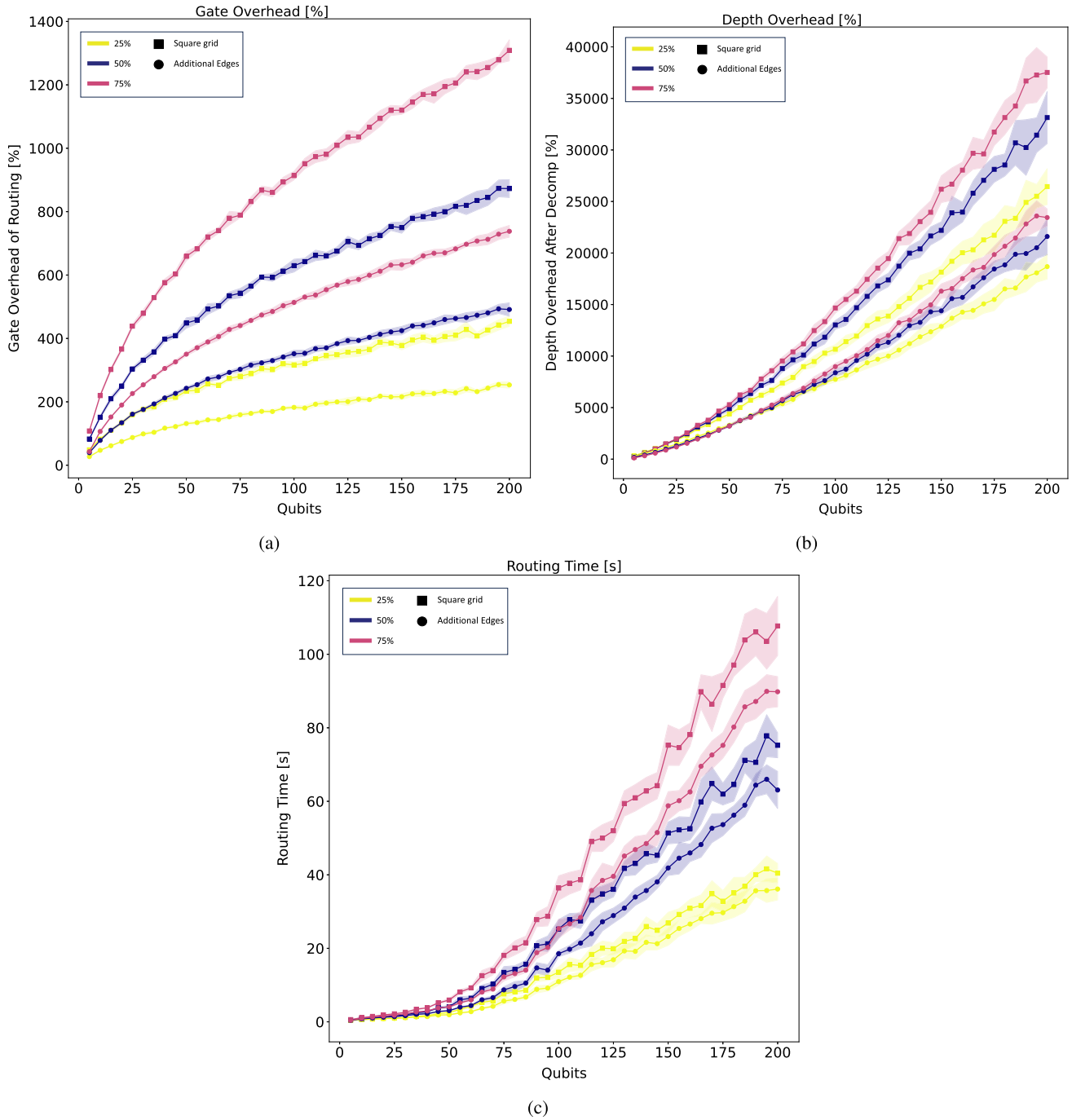

FIGURE 9. **(a) Gate overhead, (b) depth overhead, and (c) routing time of beSnake obtained on two different topologies: one square grid and one with additional diagonal edges. Each subfigure is dedicated to each of the three performance metrics.**

and, for comparison purposes, we will provide such results in Section V-C2, named as *beSnake full*.

1) RANDOMLY GENERATED CIRCUITS WITH UP TO 1000 QUBITS

The underlying question to answer with this simulation is as follows.

- Which routing algorithm will more likely produce lower gate and circuit depth overhead for scalable architectures that go up to 1000 qubits, and at what relative time cost?

In Fig. 11(a) and (b), the gate overhead and depth overhead of both algorithms (marked as *SBS* and *beSnake*) are shown for the different two-qubit gate percentages (in yellow, blue,

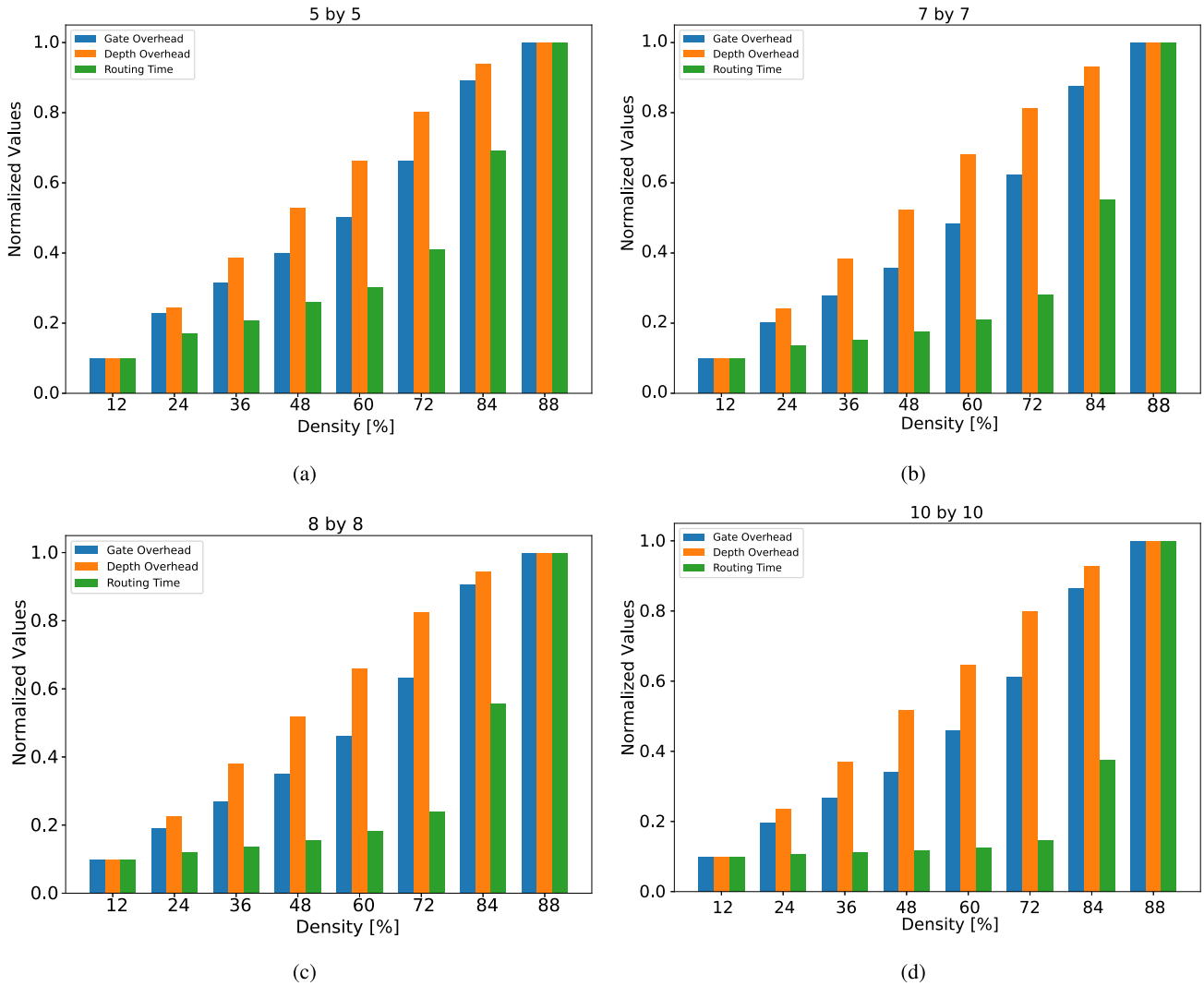

FIGURE 10. **(a)–(d) BeSnake's normalized performance on four fixed topology sizes in each subfigure while varying the qubit density.**

and pink). We can observe in beSnake (circle markers) a relative improvement in gate overhead (rGO) of 32.2% and in depth overhead (rDO) of 30.3% on average compared to SBS (square markers). In addition, we observe beSnake's relative benefit in both overheads increasing with higher qubit counts. Looking at the three two-qubit gate percentages, we see an increased gate overhead improvement with higher percentages (see the distance between curves of the same color). More specifically, the average rGO is at 36.61% for 25%, 32.64% for 50%, and 30.36% for 75%, and the average rDO is calculated at 24.05% for 25%, 30.5% for 50%, and 33.19% for 75%. BeSnake will only be, on average, 0.78 slower (rTime) than SBS at 25%, 0.88 at 50%, and 0.87 at 75%. Notably, both algorithms' routing time starts getting closer around and after 900 qubits, indicating that beSnake takes almost the same amount of time at 1000 qubits, especially for more complex routing tasks [i.e., 75 two-qubit gate percentage shown in Fig. 11(c)]. Considering the performance gains, beSnake is a more attractive solution overall for large-scale architectures. Finally, as discussed before, in these comparison simulations, beSnake is not utilized fully; hence, we expect even better performance .

2) REAL ALGORITHMS WITH UP TO 1000 QUBITS

In the following simulations, we compare the performance of SBS over beSnake for real algorithms with up to 1000 qubits. Previously, the random circuit set was produced to stress-test the capabilities of both routing algorithms; however, the results are not representative of routing for real algorithms, which might exhibit different circuit characteristics. Similarly to Section V-C1, we will compare both routers fairly to see their relative performance. However, we will also assume ideally parallelized circuits and enable the SWAP replacement option (at 99.95% SWAP fidelity) in order to test beSnake at

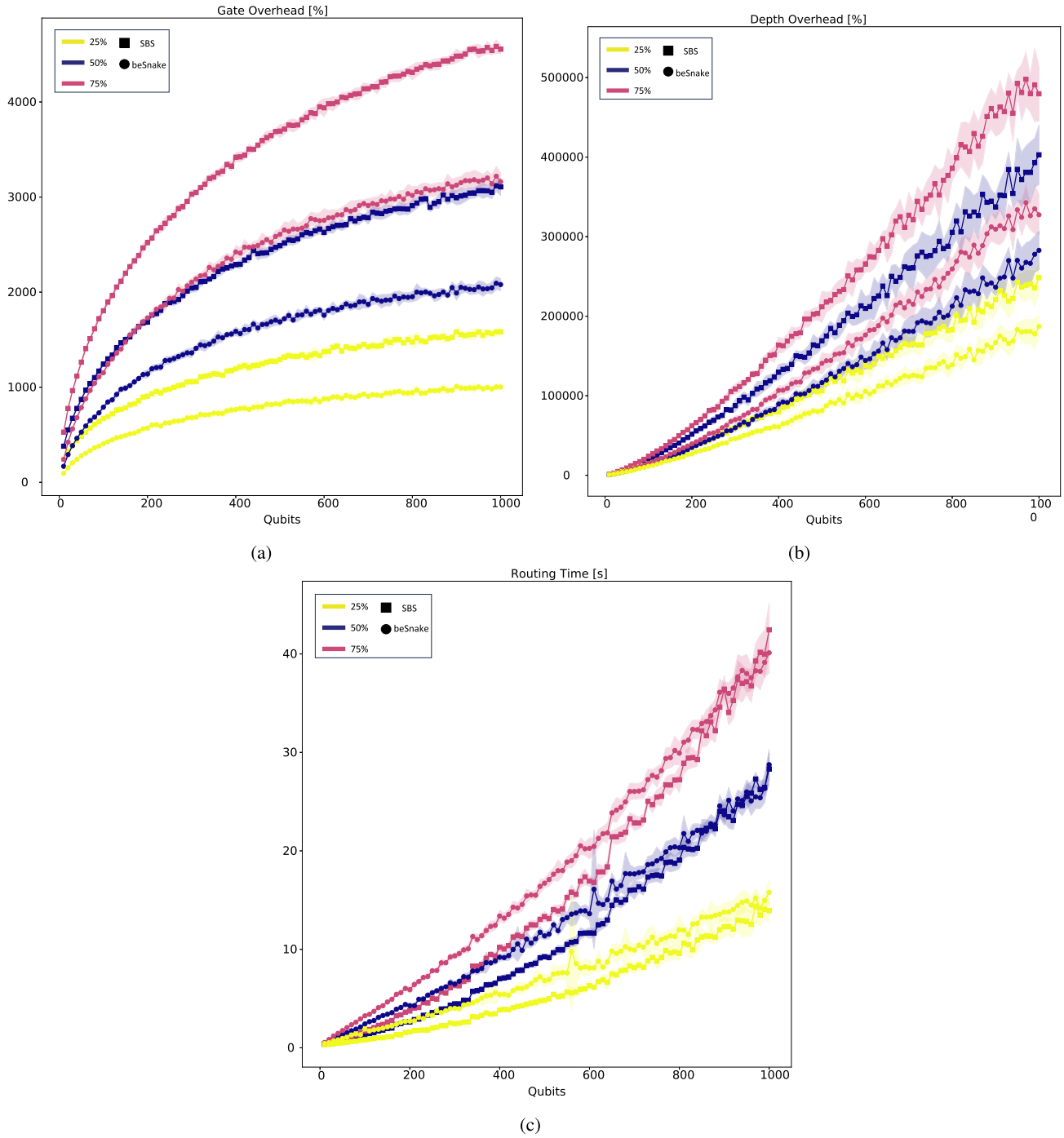

FIGURE 11. **(a)–(c) Performance comparison between beSnake (circle markers) and SBS (square markers) on random algorithms with up to 1000 qubits.**

its full capacity. The underlying question to answer with this simulation is as follows.

- How much better can beSnake perform over SBS on real quantum algorithms, and is it worth the potentially increased routing time?

Table 2 summarizes the relative performance of beSnake over SBS (used as a baseline) in two configurations: one fair comparison, *beSnake fair*, and one scenario with beSnake at its full capacity, *beSnake full*. Due to algorithm availability, we have specified each algorithm's qubit range and incremented qubit steps on the second column. In the next three columns, we have provided the average two-qubit and Z gates together with their ratio. However, it should be noted that the number or percentage of gates is not the only performance indicator or predictor. To get a full picture of their features,

TABLE 2. Relative Performance of beSnake's Two Configurations, *beSnake Fair* and *beSnake Full*, Over SBS for Real Algorithms Scaled up to 1000 Qubits

Name	Qubit range	Average two-qubit gates	Average Z gates	Ratio [%]	beSnake fair			beSnake full		
					rGO [%]	rDO [%]	rTime	rGO [%]	rDO [%]	rTime
Grover's	*10–1000, 10*	*12036*	*19057*	*38.71*	**16**	**29**	**0.58**	**64**	**37**	**0.64**
Shor's	*3–999, 10*	*566078*	*949078*	*37.36*	**72**	**30**	**8.33**	**73**	**39**	**7.69**
Deutsch–Jozsa	*10–1000, 10*	*1000*	*1000*	*50*	**5**	**44**	**0.44**	**66**	**43**	**0.09**
Bernstein–Vazirani	*3–50, 1*	*2*	*2*	*50*	**57**	**4**	**0.24**	**58**	**4**	**0.24**
QFT	*10–1000, 10*	*77672*	*135047*	*36.51*	**56**	**37**	**3.57**	**80**	**54**	**2.94**
Vbe adder	*5–148, 3*	*1372*	*1960*	*41.18*	**63**	**29**	**0.65**	**69**	**32**	**0.52**
Cuccaro Adder	*5–100, 2*	*802*	*1102*	*42.12*	**67**	**31**	**0.55**	**72**	**31**	**0.62**
Cuccaro Multiplier	*6–246, 5*	*70304*	*93550*	*42.91*	**52**	**29**	**2.04**	**63**	**33**	**2**
Amplitude Estimation	*10 – 60, 10*	*2825*	*4876*	*36.68*	**59**	**28**	**0.77**	**67**	**41**	**0.6**
GHZ State	*10–1000 10*	*1008*	*1008*	*50*	**67**	**45**	**1.39**	**68**	**46**	**1.41**
W-State	*10–1000 10*	*2016*	*2520*	*44.44*	**51**	**45**	**3.7**	**71**	**51**	**1.02**

Ratio: (Average of two-qubit gates + Average of Z gates) / Average of two-qubit gates. **beSnake fair**: beSnake is configured such that it matches the maximum capabilities of the shuttle-based SWAP (SBS) algorithm. **beSnake full**: beSnake is used at its full capacity. **rGO [%]** and **rDO [%]** refer to the relative gate and depth overhead, respectively: (SBS average results – beSnake average results) / SBS average results. **rTime**: SBS average time / beSnake average time.

a deeper analysis has to be conducted with the help of the algorithm's internal interaction and dependence graph [1], [72]. In this simulation, we are focusing on the performance comparison of the two routers rather than on specific quantum algorithm performance or how these algorithms scale.

Moving on to the rest of the columns, we provide the relative performance of beSnake over SBS in bold numbers for gate overhead, depth overhead, and routing time, as defined in Section V-A. For example, beSnake obtained a 16% gate overhead and 29% depth overhead improvement over SBS under a fair comparison for Grover's algorithm. As for routing time, beSnake is 42% slower ($rTime = 0.58$) than SBS.

Looking only at the fair comparisons overall, we observe an improvement in both rGO and rDO across all algorithms, with up to a 67% improvement in rGO and up to 45% improvement in rDO. On the one hand, rTime results for Deutsch–Jozsa (up to 1000 qubits) and for Bernstein–Vazirani (up to 50 qubits) seem relatively high for their low-average two-qubit and Z gate values. Evidently, these algorithms do not require a lot of routing; therefore, the relative time increase possibly comes from beSnake's code initialization overhead (i.e., loading libraries, allocating memory, initializing variables, and setting up data structures) and not the actual routing process. On the other hand, Cuccaro Multiplier and quantum Fourier transform (QFT), two highly connected circuits [1], obtained more than 50% rGO and rDO improvement with more than 2 and 3.5 times faster routing time.

Focusing on *beSnake full* columns, we see the biggest improvement in rGO, and not in rDO, compared to *beSnake fair*, and in a few algorithms, this was achieved with less routing time. This indicates that beSnake can handle highly parallelized circuits faster than less parallelized ones despite imposing more demanding routing tasks, and it can always do so with less gate overhead. This shows that, on average, holding in place qubits to satisfy two-qubit gates, as seen in the third example in Section IV-C, does not negatively impact the gate overhead. This can be attributed to increased available space in larger topologies, which offers more alternative shortest paths around the fixed positions. Another contributing factor to this gate overhead benefit is the SWAP replacement option. Nonetheless, *beSnake full* can reduce both overheads more compared to *beSnake fair* for up to 1000 qubits while taking less time in some real algorithms.

VI. CONCLUSION

This article presents beSnake, a novel routing algorithm designed explicitly for scalable spin-qubit architectures. It addresses the unique challenges posed by spin-qubit systems, such as the incorporation of the shuttle operation over the traditional SWAP gate used in other qubit technologies. We presented beSnake's ability to dynamically adapt to different routing challenges and efficiently handle complex scenarios involving multiple parallel gates. We stress-tested beSnake in various configuration options to provide valuable performance insights. In particular, we showed that adjusting the heuristic of the shortest path selection can offer a significant speed improvement with a relatively small overhead cost. Then, beSnake was tested on its SWAP replacement option

and its capacity to handle a highly connected topology faster than a less connected one. Furthermore, we gave an insight into the optimal qubit density, up to 72%, that beSnake can route in a scalable manner. In the second simulation phase, we conducted an extensive comparison between the SBS routing algorithm acting on random and real algorithms with up to 1000 qubits, in which beSnake demonstrated significant improvements, with up to 80% and 54% on average in gate overhead and depth overhead, respectively, and up to 8.33 times faster routing time.

While performing our research, several avenues for future work have emerged, which warrant further exploration and investigation. Regarding the mechanisms for dealing with blockages discussed in Section IV-D1, the list of alternative approaches can be further expanded for such edge cases. This should be done with scalability in mind in terms of the extra computation time and whether it will be worth it based on their utility on the particular topological patterns and size of the coupling graph. One example is moving qubits for two or more gates concurrently or allowing the operand qubits to be pushed by other qubits. This will possibly improve the output circuit's overhead but also make it slower for beSnake to find a solution. In other cases, it can increase both overheads because of deeper BFS explorations due to path conflicts within the different layers.

As for ways to sort parallelized Z and two-qubit gates contained in a cycle, besides the one discussed in Section IV-D2, more nuanced techniques will most likely increase the algorithm's complexity but improve both overheads. One possible direction is to create a heuristic upon which the order of all the different gate types is sorted based on their predicted routing paths. On a similar note, the shortest path selection heuristic can be expanded by adding a third filtration stage in cases with more than one candidate path remaining. More precisely, it could make a prediction of subsequent routing steps and select the current path, which would result in the least overall overhead in the future. Nonetheless, such strategies should be explored and compared in targeted architectures that will properly utilize them.

Last but not least, as mentioned, beSnake utilizes NetworkX [73]. However, similar libraries are known to be more time and memory efficient, especially for large graphs with complex structures [74], [75]. This is attributed to NetworkX's pure Python implementation compared to other libraries written in C/C++. NetworkX was chosen for its user-friendly application programming interface while offering flexibility for future adaptations. As quantum devices increase in size beyond 1000 qubits, better performance might be gained by using other libraries, such as igraph [76], graph-tool [77], or rustworkx [78].

REFERENCES

[1] N. Paraskevopoulos, F. Sebastiano, C. G. Almudever, and S. Feld, "SpinQ: Compilation strategies for scalable spin-qubit architectures," *ACM Trans. Quantum Comput.*, vol. 5, no. 1, pp. 1–36, Dec. 2023, doi: 10.1145/3624484.

[2] S. Sivarajah, S. Dilkes, A. Cowtan, W. Simmons, A. Edgington, and R. Duncan, "t|ket >: A retargetable compiler for NISQ devices," *Quantum Sci. Technol.*, vol. 6, no. 1, 2020, Art. no. 014003, doi: 10.1088/2058-9565/ab8e92.

[3] J.-A. Ali et al., "Quantum computing with qiskit," 2024, *arXiv:2405.08810*, doi: 10.48550/arXiv.2405.08810.

[4] N. Khammassi et al., "OpenQL: A portable quantum programming framework for quantum accelerators," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 18, no. 1, pp. 1–24, 2021, doi: 10.1145/3474222.

[5] M. Salm, J. Barzen, F. Leymann, B. Weder, and K. Wild, "Automating the comparison of quantum compilers for quantum circuits," in *Proc. Symp. Summer Sch. Serv.-Oriented Comput.*, 2021, pp. 64–80, doi: 10.1007/978-3-030-87568-8_4.

[6] C. Developers, "Cirq," Zenodo, 2023. [Online]. Available: https://zenodo.org/records/11398048

[7] R. Computing, "Pyquil documentation," pp. 64–65, 2019. [Online]. Available: http://pyquil.readthedocs.io/en/latest

[8] A. JavadiAbhari et al., "ScaffCC: A framework for compilation and analysis of quantum computing programs," in *Proc. 11th ACM Conf. Comput. Front.*, 2014, pp. 1–10, doi: 10.1145/2597917.2597939.

[9] F. T. Chong, D. Franklin, and M. Martonosi, "Programming languages and compiler design for realistic quantum hardware," *Nature*, vol. 549, no. 7671, pp. 180–187, 2017, doi: 10.1038/nature23459.

[10] X.-C. Wu et al., "Intel quantum SDK version 1.0: Extended C++ compiler, runtime and quantum hardware simulators for hybrid quantum-classical applications," in *APS Mar. Meeting Abstr.*, vol. 2023, pp. RR08–005, 2023. [Online]. Available: https://meetings.aps.org/Meeting/MAR23/Session/RR08.5.

[11] J. Preskill, "Quantum computing in the NISQ era and beyond," *Quantum*, vol. 2, 2018, Art. no. 79, doi: 10.22331/q-2018-08-06-79.

[12] S. S. Tannu and M. K. Qureshi, "Not all qubits are created equal: A case for variability-aware policies for NISQ-era quantum computers," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 987–999, doi: 10.1145/3297858.3304007.

[13] G. Li, Y. Ding, and Y. Xie, "Tackling the qubit mapping problem for NISQ-era quantum devices," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 1001–1014, doi: 10.1145/3297858.3304023.

[14] A. A. Saki, M. Alam, J. Li, and S. Ghosh, "Error-tolerant mapping for quantum computing," in *Emerging Computing: From Devices to Systems: Looking Beyond Moore and Von Neumann*. Singapore: Springer, pp. 371–403, 2022, doi: 10.1007/978-981-16-7487-7_12.

[15] P. Murali, J. M. Baker, A. Javadi-Abhari, F. T. Chong, and M. Martonosi, "Noise-adaptive compiler mappings for noisy intermediate-scale quantum computers," in *Proc. 24th Int. Conf. Archit. Support Program. Lang. Oper. Syst.*, 2019, pp. 1015–1029, doi: 10.1145/3297858.3304075.

[16] S. Niu, A. Suau, G. Staffelbach, and A. Todri-Sanial, "A hardware-aware heuristic for the qubit mapping problem in the NISQ era," *IEEE Trans. Quantum Eng.*, vol. 1, 2020, Art. no. 3101614, doi: 10.1109/TQE.2020.3026544.

[17] A. Sinha, U. Azad, and H. Singh, "Qubit routing using graph neural network aided Monte Carlo tree search," in *Proc. AAAI Conf. Artif. Intell.*, 2022, vol. 36, no. 9, pp. 9935–9943, doi: 10.1609/aaai.v36i9.21231.

[18] J. Liu, E. Younis, M. Weiden, P. Hovland, J. Kubiatowicz, and C. Iancu, "Tackling the qubit mapping problem with permutation-aware synthesis," in *Proc. IEEE Int. Conf. Quantum Comput. Eng.*, 2023, vol. 1, pp. 745–756, doi: 10.1109/QCE57702.2023.00090.

[19] L. Lao, H. van Someren, I. Ashraf, and C. G. Almudever, "Timing and resource-aware mapping of quantum circuits to superconducting processors," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. 41, no. 2, pp. 359–371, Feb. 2021, doi: 10.1109/TCAD.2021.3057583.

[20] R. Li et al., "A crossbar network for silicon quantum dot qubits," *Sci. Adv.*, vol. 4, no. 7, 2018, Art. no. eaar3960, doi: 10.1126/sciadv.aar3960.

[21] J. Yoneda et al., "A quantum-dot spin qubit with coherence limited by charge noise and fidelity higher than 99.9%," *Nature Nanotechnol.*, vol. 13, no. 2, pp. 102–106, 2018, doi: 10.1038/s41565-017-0014-x.

[22] L. C. Camenzind, S. Geyer, A. Fuhrer, R. J. Warburton, D. M. Zumbühl, and A. V. Kuhlmann, "A hole spin qubit in a fin field-effect transistor above 4 kelvin," *Nature Electron.*, vol. 5, no. 3, pp. 178–183, 2022, doi: 10.1038/s41928-022-00722-0.

[23] N. W. Hendrickx et al., "A four-qubit germanium quantum processor," *Nature*, vol. 591, no. 7851, pp. 580–585, 2021, doi: 10.1038/s41586-021-03332-6.

[24] A. Chatterjee, P. Stevenson, S. De Franceschi, A. Morello, N. P. de Leon, and F. Kuemmeth, "Semiconductor qubits in practice," *Nature Rev. Phys.*, vol. 3, no. 3, pp. 157–177, 2021, doi: 10.1038/s42254-021-00283-9.

[25] F. A. Zwanenburg et al., "Silicon quantum electronics," *Rev. Mod. Phys.*, vol. 85, pp. 961–1019, Jul. 2013, doi: 10.1103/RevModPhys.85.961.

[26] D. Loss and D. P. DiVincenzo, "Quantum computation with quantum dots," *Phys. Rev. A*, vol. 57, pp. 120–126, Jan. 1998, doi: 10.1103/PhysRevA.57.120.

[27] L. Vandersypen et al., "Interfacing spin qubits in quantum dots and donors—Hot, dense, and coherent," *npj Quantum Inf.*, vol. 3, no. 1, pp. 1–10, 2017, doi: 10.1038/s41534-017-0038-y.

[28] M. Veldhorst et al., "A two-qubit logic gate in silicon," *Nature*, vol. 526, no. 7573, pp. 410–414, 2015, doi: 10.1038/nature15263.

[29] D. Zajac, T. Hazard, X. Mi, K. Wang, and J. R. Petta, "A reconfigurable gate architecture for Si/SiGe quantum dots," *Appl. Phys. Lett.*, vol. 106, no. 22, 2015, Art. no. 223507, doi: 10.1063/1.4922249.

[30] T. Watson et al., "A programmable two-qubit quantum processor in silicon," *Nature*, vol. 555, no. 7698, pp. 633–637, 2018, doi: 10.1038/nature25766.

[31] S. G. Philips et al., "Universal control of a six-qubit quantum processor in silicon," *Nature*, vol. 609, no. 7929, pp. 919–924, 2022, doi: 10.1038/s41586-022-05117-x.

[32] X. Xue et al., "Quantum logic with spin qubits crossing the surface code threshold," *Nature*, vol. 601, no. 7893, pp. 343–347, 2022, doi: 10.1038/s41586-021-04273-w.

[33] A. Noiri et al., "Fast universal quantum gate above the fault-tolerance threshold in silicon," *Nature*, vol. 601, no. 7893, pp. 338–342, 2022, doi: 10.1038/s41586-021-04182-y.

[34] R. Hanson, L. P. Kouwenhoven, J. R. Petta, S. Tarucha, and L. M. Vandersypen, "Spins in few-electron quantum dots," *Rev. Modern Phys.*, vol. 79, no. 4, 2007, Art. no. 1217, doi: 10.1103/RevModPhys.79.1217.

[35] F. Borsoi et al., "Shared control of a 16 semiconductor quantum dot crossbar array," *Nature Nanotechnol.*, vol. 19, no. 1, pp. 21–27, 2024, doi: 10.1038/s41565-023-01491-3.

[36] M. Y. Siraichi, V. F. dos Santos, C. Collange, and F. M. Q. Pereira, "Qubit allocation," in *Proc. Int. Symp. Code Gener. Optim.*, 2018, pp. 113–125, doi: 10.1145/3168822.

[37] A. Botea, A. Kishimoto, and R. Marinescu, "On the complexity of quantum circuit compilation," in *Proc. Int. Symp. Combinatorial Search*, 2018, vol. 9, no. 1, pp. 138–142, doi: 10.1609/socs.v9i1.18463.

[38] J. Helsen, M. Steudtner, M. Veldhorst, and S. Wehner, "Quantum error correction in crossbar architectures," *Quantum Sci. Technol.*, vol. 3, no. 3, 2018, Art. no. 035005, doi: 10.1088/2058-9565/aab8b0.

[39] A. M. Tejerina, "Mapping quantum algorithms in a crossbar architecture," Master's thesis, Dept. Quantum Comput. Eng., Delft Univ. Technol., The Netherlands, 2019. [Online]. Available: https://repository.tudelft.nl/record/uuid:cb7194fb-7b99-4ea9-a5ad-8e3617d1f9f5.

[40] I. Dirgová Luptáková and J. Pospíchal, "Transition graph analysis of sliding tile puzzle heuristics," in *Recent Advances in Soft Computing and Cybernetics*. Cham, Switzerland: Springer, 2021, pp. 149–156.

[41] R. E. Korf and A. Felner, "Disjoint pattern database heuristics," *Artif. Intell.*, vol. 134, nos. 1/2, pp. 9–22, 2002, doi: 10.1016/S0004-3702(01)00165-5.

[42] A. Felner, R. E. Korf, and S. Hanan, "Additive pattern database heuristics," *J. Artif. Intell. Res.*, vol. 22, pp. 279–318, 2004, doi: 10.1613/jair.1480.

[43] L. Orseau, M. Hutter, and L. H. Leli, "Levin tree search with context models," 2023, *arXiv:2305.16945*, doi: 10.48550/arXiv.2305.16945.

[44] M. Gozon and J. Yu, "On computing makespan-optimal solutions for generalized sliding-tile puzzles," 2023, *arXiv:2312.10887*, doi: 10.48550/arXiv.2312.10887.

[45] J. Gao, Y. Li, X. Li, K. Yan, K. Lin, and X. Wu, "A review of graph-based multi-agent pathfinding solvers: From classical to beyond classical," *Knowl.-Based Syst.*, vol. 283, 2023, Art. no. 111121, doi: 10.1016/j.knosys.2023.111121.

[46] P. Raja and S. Pugazhenthi, "Optimal path planning of mobile robots: A review," *Int. J. Phys. Sci.*, vol. 7, no. 9, pp. 1314–1320, 2012, doi: 10.5897/IJPS11.1745.

[47] H. Ma, "Graph-based multi-robot path finding and planning," *Curr. Robot. Rep.*, vol. 3, no. 3, pp. 77–84, 2022, doi: 10.1007/s43154-022-00083-8.

[48] S. Tjiharjadi, S. Razali, and H. A. Sulaiman, "A systematic literature review of multi-agent pathfinding for maze research," *J. Adv. Inf. Technol.*, vol. 13, no. 4, pp. 358–367, 2022, doi: 10.12720/jait.13.4.358-367.

[49] C. D. Bruzewicz, J. Chiaverini, R. McConnell, and J. M. Sage, "Trapped-ion quantum computing: Progress and challenges," *Appl. Phys. Rev.*, vol. 6, no. 2, 2019, Art. no. 021314, doi: 10.1063/1.5088164.

[50] P. Murali, D. M. Debroy, K. R. Brown, and M. Martonosi, "Architecting noisy intermediate-scale trapped ion quantum computers," in *Proc. ACM/IEEE 47th Annu. Int. Symp. Comput. Archit.*, 2020, pp. 529–542, doi: 10.1109/ISCA45697.2020.00051.

[51] A. A. Saki, R. O. Topaloglu, and S. Ghosh, "Muzzle the shuttle: Efficient compilation for multi-trap trapped-ion quantum computers," in *Proc. IEEE Des., Autom. Test Eur. Conf. Exhib.*, 2022, pp. 322–327, doi: 10.23919/DATE54114.2022.9774619.

[52] D. Schoenberger, S. Hillmich, M. Brandl, and R. Wille, "Using Boolean satisfiability for exact shuttling in trapped-ion quantum computers," 2023, *arXiv:2311.03454*, doi: 10.48550/arXiv.2311.03454.

[53] L. Schmid et al., "Computational capabilities and compiler development for neutral atom quantum processors—Connecting tool developers and hardware experts," *Quantum Sci. Technol.*, vol. 9, no. 3, 2024, Art. no. 0033001, doi: 10.1088/2058-9565/ad33ac.

[54] R. Grimm, M. Weidemüller, and Y. B. Ovchinnikov, "Optical dipole traps for neutral atoms," *Adv. Atomic Mol. Opt. Phys.*, vol. 42, pp. 95–170, doi: 10.1016/S1049-250X(08)60186-X.

[55] P. Jessen and I. Deutsch, "Optical lattices," *Adv. Atomic, Molecular, Opt. Phys.*, vol. 37, pp. 95–138, 1996, doi: 10.1016/S1049-250X(08)60099-3.

[56] A. M. Kaufman and K.-K. Ni, "Quantum science with optical Tweezer arrays of ultracold atoms and molecules," *Nature Phys.*, vol. 17, no. 12, pp. 1324–1333, 2021, doi: 10.1038/s41567-021-01357-2.

[57] M. Saffman, "Quantum computing with atomic qubits and Rydberg interactions: Progress and challenges," *J. Phys. B: Atomic, Mol. Opt. Phys.*, vol. 49, no. 20, 2016, Art. no. 202001, doi: 10.1088/0953-4075/49/20/202001.

[58] H. Wang et al., "Q-pilot: Field programmable qubit array compilation with flying ancillas," in *Proc. 61st ACM/IEEE Des. Autom. Conf.*, 2024, pp. 1–6.

[59] Y. Stade, L. Schmid, L. Burgholzer, and R. Wille, "An abstract model and efficient routing for logical entangling gates on zoned neutral atom architectures," 2024, *arXiv:2405.08068*, doi: 10.48550/arXiv.2405.08068.

[60] D. B. Tan, D. Bluvstein, M. D. Lukin, and J. Cong, "Compiling quantum circuits for dynamically field-programmable neutral atoms array processors," *Quantum*, vol. 8, 2024, Art. no. 1281, doi: 10.22331/q-2024-03-14-1281.

[61] N. Nottingham, M. A. Perlin, R. White, H. Bernien, F. T. Chong, and J. M. Baker, "Decomposing and routing quantum circuits under constraints for neutral atom architectures," 2023, *arXiv:2307.14996*, doi: 10.48550/arXiv.2307.14996.

[62] S. Brandhofer, I. Polian, and H. P. Büchler, "Optimal mapping for near-term quantum architectures based on Rydberg atoms," in *Proc. IEEE/ACM Int. Conf. Comput. Aided Des.*, 2021, pp. 1–7, doi: 10.1109/ICCAD51958.2021.9643490.

[63] L. Schmid, S. Park, S. Kang, and R. Wille, "Hybrid circuit mapping: Leveraging the full spectrum of computational capabilities of neutral atom quantum computers," 2023, *arXiv:2311.14164*, doi: 10.48550/arXiv.2311.14164.

[64] T. Schmale et al., "Backend compiler phases for trapped-ion quantum computers," 2022, *arXiv:2206.00544*, doi: 10.48550/arXiv.2206.00544.

[65] M. G. Pozzi, S. J. Herbert, A. Sengupta, and R. D. Mullins, "Using reinforcement learning to perform qubit routing in quantum compilers," *ACM Trans. Quantum Comput.*, vol. 3, no. 2, pp. 1–25, 2022, doi: 10.1145/3520434.

[66] M. Steinberg, M. Bandic, S. Szkudlarek, C. G. Almudever, A. Sarkar, and S. Feld, "Resource bounds for quantum circuit mapping via quantum circuit complexity," 2024, *arXiv:2402.00478*, doi: 10.48550/arXiv.2402.00478.

[67] P. Escofet et al., "Revisiting the mapping of quantum circuits: Entering the multi-core era," *ACM Trans. Quantum Comput.*, early access, 2024, doi: 10.1145/3655029.

[68] C.-A. Wang et al., "Operating semiconductor quantum processors with hopping spins," 2024, *arXiv:2402.18382*, doi: 10.48550/arXiv.2402.18382.

[69] C.-C. Lin, A. Chakrabarti, and N. K. Jha, "QLib: Quantum module library," *ACM J. Emerg. Technol. Comput. Syst.*, vol. 11, no. 1, pp. 1–20, 2014, doi: 10.1145/2629430.

[70] R. Wille, D. Große, L. Teuber, G. W. Dueck, and R. Drechsler, "RevLib: An online resource for reversible functions and reversible circuits," in *Proc. IEEE 38th Int. Symp. Mult. Valued Log.*, 2008, pp. 220–225, doi: 10.1109/ISMVL.2008.43.

[71] N. Quetschlich, L. Burgholzer, and R. Wille, "MQT bench: Benchmarking software and design automation tools for quantum computing," *Quantum*, vol. 7, 2023, Art. no. 1062, doi: 10.22331/q-2023-07-20-1062.

[72] M. Bandic, C. G. Almudever, and S. Feld, "Interaction graph-based characterization of quantum benchmarks for improving quantum circuit mapping techniques," *Quantum Mach. Intell.*, vol. 5, no. 2, 2023, Art. no. 40, doi: 10.1007/s42484-023-00124-1 .

[73] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring network structure, dynamics, and function using networkX," in *Proc. 7th Python Sci. Conf.*, 2008, pp. 11–15, doi: https://www.osti.gov/biblio/960616.

[74] J. Leskovec and R. Sosič, "SNAP: A general-purpose network analysis and graph-mining library," *ACM Trans. Intell. Syst. Technol.*, vol. 8, no. 1, pp. 1–20, 2016, doi: 10.1145/2898361.

[75] C. L. Staudt, A. Sazonovs, and H. Meyerhenke, "NetworKit: A tool suite for large-scale complex network analysis," *Netw. Sci.*, vol. 4, no. 4, pp. 508–530, 2016, doi: 10.1017/nws.2016.20 .

[76] G. Csardi and T. Nepusz, "The igraph software," *Complex Syst.*, vol. 1695, pp. 1–9, 2006, doi: 10.3389/fimmu.2022.862049.

[77] T. P. Peixoto, "The graph-tool Python library," figshare, 2014. [Online]. Available: http://figshare.com/articles/graph_tool/1164194

[78] M. Treinish, I. Carvalho, G. Tsilimigkounakis, and N. Sá, "rustworkx: A high-performance graph library for Python," 2021, *arXiv:2110.15221*, doi: 10.48550/arXiv.2110.15221.