CHAPTER 3

# Implementation

## 3.1 Topology-Aware QEC-to-QCCD Compiler

### 3.1.1 High-Level Problem

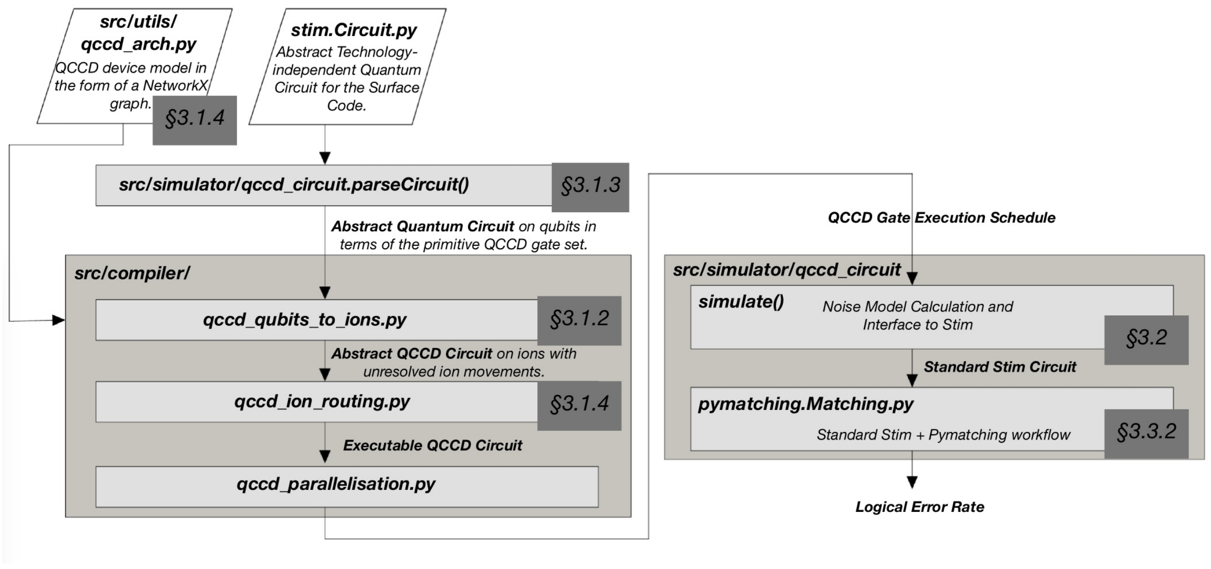
We require resource-efficient mappings of QEC cycles for the surface code onto QCCD systems with different architectures to answer the design questions. While several tool flows have been developed to map NISQ workloads on QCCD hardware, they incur large communication overheads and do not scale to high QEC code distances. This is primarily because they do not consider the local structure available in stabiliser circuits. In this chapter, I develop a surface code topology-aware compiler shown in Figure 1.1. It accepts a QEC code (stabiliser circuits, code distance) and a candidate QCCD device architecture (trap capacity, topology) as input and produces a compiled executable output, which is used for architecture evaluation.

#### Repository Structure

**src/**:

- `compiler/`: code for the QEC compiler.
  - *qccd_ion_routing* code for ion routing (§3.1.4). [222 lines]
  - *qccd_parallelisation* code for scheduling. [218 lines]
  - *qccd_qubits_to_ions* code for mapping qubits to ions (§3.1.2). [305 lines]
- `simulator/`:
  - *qccd_circuit* code for QCCD Simulator, QCCD resource estimation and logical error rate calculation (using Stim [33] and the PyMatching decoder [25] (§3.2, §3.3.2). [715 lines]
- `utils/`:
  - *qccd_arch, qccd_nodes* code for modelling QCCD hardware as a directed graph object using NetworkX, with traps and junctions as nodes (§3.1.4, §3.3.1). [996 lines]
  - *qccd_operations, qccd_operations_on_qubits* code for QCCD primitive instructions, treating them as objects with effects, associated components, operation time, and fidelity (§3.1.3). [1085 lines]

- **tests/**: code for unit and integration tests that validate the compiler's functionality using the *pytest* framework. [225 lines]
- **experiments/**: Jupyter notebooks for QCCD architectural exploration. [3397 lines]
- **results/**: outputs of simulations.
- *Makefile*: code for build automation, allowing execution in virtual environments.
- *config.yaml*: configuration file to specify inputs for experiments.



src/utils/ qccd_arch.py
QCCD device model in the form of a NetworkX graph.
§3.1.4
stim.Circuit.py
Abstract Technology-independent Quantum Circuit for the Surface Code.
src/simulator/qccd_circuit.parseCircuit()
§3.1.3
Abstract Quantum Circuit on qubits in terms of the primitive QCCD gate set.
src/compiler/
qccd_qubits_to_ions.py
§3.1.2
Abstract QCCD Circuit on ions with unresolved ion movements.
qccd_ion_routing.py
§3.1.4
Executable QCCD Circuit
qccd_parallelisation.py
QCCD Gate Execution Schedule
src/simulator/qccd_circuit
simulate()
Noise Model Calculation and Interface to Stim
§3.2
Standard Stim Circuit
pymatching.Matching.py
Standard Stim + Pymatching workflow
§3.3.2
Logical Error Rate

**Figure 3.1:** Overview of the core QCCD compilation and simulation workflow used to calculate logical error rates. Each block represents a script within the codebase, grouped by functionality into compiler, simulator, and utility modules. Arrows indicate data flow and transformation between intermediate representations, shown in bold. Section references indicate where each component is described in detail.

### 3.1.2 Mapping Qubits to Ions

The first pass in Figure 1.1 assigns each physical qubit in the QEC code to a unique QCCD device ion. Figure 3.2 frames the mapping problem as a minimum weight maximum bipartite graph matching.

Given a specific trap capacity in the architecture, a mapping that fills traps well below capacity will increase the number of ion movements. Similarly, filling traps to maximum capacity is generally inefficient , as incoming ion movement would require displacing another ion, leading to heating and increased noise. I adopt a design where traps are filled to $\mathit{capacity} - 1$, leaving one ion position free for communication. To determine the mapping, 1) I cluster the QEC qubits into balanced partitions and 2) map the clusters to traps using a matching algorithm.

**Qubit clustering:** partition QEC qubits (into clusters of size $\mathit{capacity} - 1$), by solving a balanced graph partitioning problem. Given a graph $G = (V, E)$, where nodes $V$ represent qubits and edges $E$ represent pairs of data and ancilla qubits undergoing entanglement operations, with edge weight proportional to the priority of the operation,

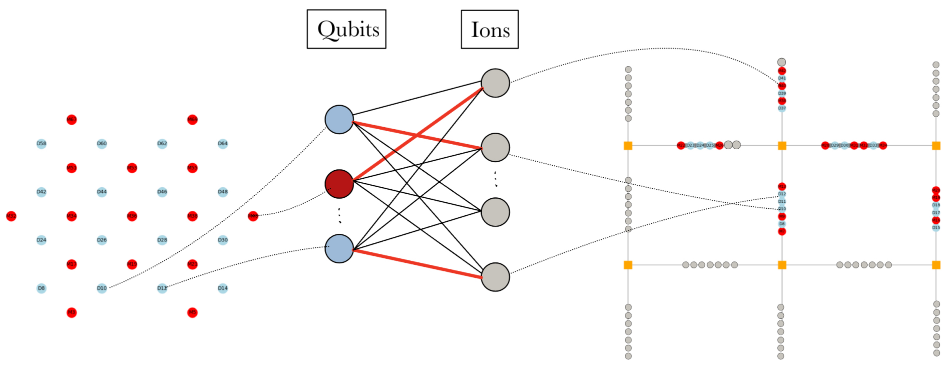


**Figure 3.2:** The problem of mapping qubits to ions in the QCCD hardware can be viewed as a bipartite graph minimum weight maximum matching problem, where the matching size equals the number of qubits. In the QCCD hardware on the right, each QCCD trap has a capacity of 7, represented by 7 circles, where grey circles represent unused ions, while red and blue circles represent ancilla and data qubits, respectively.

the objective is to divide $V$ into equal-sized clusters $C_1, \ldots, C_k$ such that the total weight of cut edges is minimised. Here, $k$ will equal the number of traps used by the logical qubit in the QCCD hardware. The number of ion movement operations is minimised by minimising the number of high-priority entanglement operations cut.

The balanced graph partitioning problem, particularly for $k = 2$ *(known as the Minimum Bisection problem)*, is NP-complete [34]. Even worse, no polytime approximation algorithm can guarantee a finite approximation factor when partitions are precisely equal in size [35]. At the same time, other polytime approximation algorithms do not guarantee partitions of size $\leq$ trap capacity, which violates compiler correctness. In general, a heuristic-based search does not help either because evaluating partition quality is challenging, as the sum of edge weights across partitions depends heavily on the input parameters, such as trap capacity, code topology, and code distance. Therefore, other compilers [1–3] that are designed for NISQ circuits are unsuitable for large QEC code distances.

The compiler computes a balanced partitioning by exploiting the regular qubit topology in QEC circuits. We use a top-down regular partitioning of the QEC code topology, as depicted in Figure 3.3. This minimises ancilla movement between clusters because qubit neighbourhoods are preserved during ion routing. For example, to map Figure 2.4 on a device with capacity 9, first partition into $ceil(N_{qubits}/(capacity - 1)) = ceil(31/8) = 4$ clusters by top-down regular partitioning of the primal lattice, resulting in each cluster containing one weight-4 Z plaquette of data qubits. Thereafter, the dual lattice is partitioned such that the ancilla qubit clusters are of size $\leq floor(capacity - N_{qubits}/(capacity - 1))$. The primal and dual clusters are combined, resulting in three clusters of size 8 and one of size 7.

**Mapping clusters to traps:** Clusters are then mapped to traps by solving a minimum edge-weight, maximum cardinality matching problem.

Edge weights are defined for a topology-based mapping, such that, for every qubit, the neighbourhood of qubits in the code topology is the same in hardware. Since ancilla qubits only interact with their nearest-neighbours in topological QEC codes, the ion movement will be minimised if the nearest neighbours are co-located in the same trap. To determine edge weights, map both the QEC and ion coordinate systems onto a unit square. The weight of an edge between a qubit cluster $Q$ and a trap $t$ is then $\sum_{q\in Q}\sum_{n\in KNN(q)}((x_t - x_n)^2 + (y_t - y_n)^2)$,

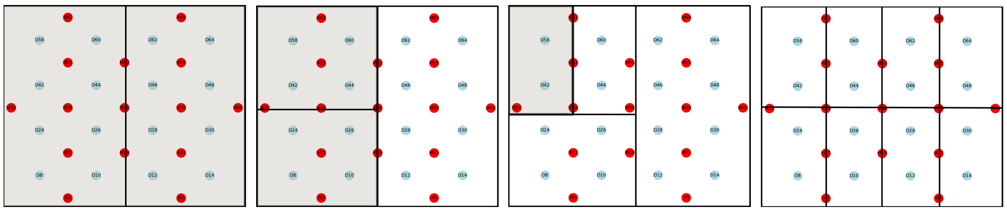


**Figure 3.3:** Top-down regular partitioning of data qubits, denoted in blue, in the distance-4 rotated surface code. The grey-shaded regions indicate recursive partitioning of the code topology. The final diagram shows a balanced partitioning of data qubits into partitions of size 2. Note that ancilla qubits, denoted in red, are ignored in this phase but grouped into partitions afterwards using a greedy-based strategy.

where $KNN(q)$ represents the $k$-nearest neighbours (including $q$ itself) to the qubit $q$. Therefore, a minimum-weight matching minimises the relative Euclidean distance between all qubit neighbourhoods of size $k$ and their corresponding ion neighbourhoods in the hardware, where the choice of $k$ is dependent on the weight of the stabilisers in the QEC code.

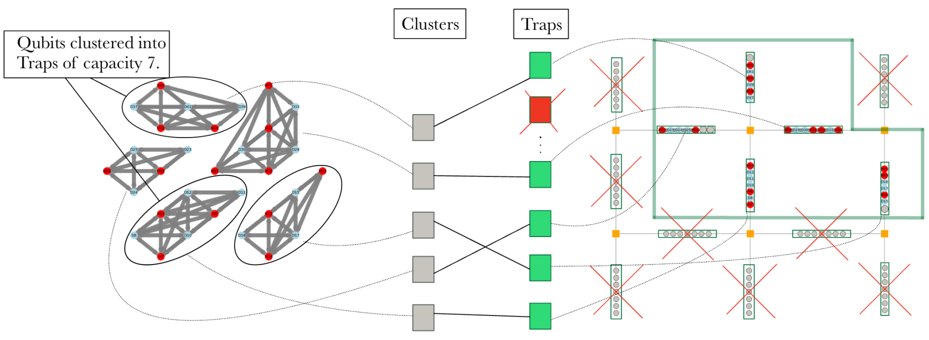


**Figure 3.4:** Translation of the problem of mapping clusters of qubits to traps into a maximum perfect matching problem. The figure shows clusters of qubits, each of which needs to be mapped to a trap in a pre-determined selection of traps with equal cardinality to the number of qubit clusters.

The matching problem is solved by considering all subsets of traps with cardinality equal to the number of clusters. For each subset, I use the Hungarian algorithm [32] (which is best explained using the slides from [36]) to compute the minimum perfect matching in polynomial time, and the subset with the lowest total cost is selected. Figure 3.4 depicts the translation of the mapping problem into a maximum perfect matching problem solvable using the Hungarian algorithm. Among all possible subsets of traps with the same cardinality as the number of qubit clusters, the subset with the maximum total weight for its perfect matching must be selected. Therefore, the Hungarian algorithm efficiently computes the maximum perfect matching for relevant subsets, and then the subset that achieves the highest total weight compared to all other subsets is chosen.

For general quantum circuits, the search space can be restricted by focusing solely on canonical subsets of traps, for which the neighbours of inner traps are in the subset, and the central trap is within a 2-hop distance around the centre of the hardware topology (as depicted in §A.2).

Let $k^2 \leq n < (k+1)^2$, where $n$ is the number of qubit clusters, and $(k+1)^2$ is the total number of traps. The benefits of this constrained search strategy are supported by asymptotic analysis. In a compact neighbourhood around a centroid trap, the number of candidate traps grows quadratically with the side length of the expanding region. The search stops before exceeding $n$, so the number of guaranteed traps $g = k^2$, and the number of additional candidate traps $m = (k+2)^2 - k^2 = 4k + 4$.

For each centroid trap, the number of compact subsets to explore is given by the binomial coefficient $\binom{m}{n-g}$. Substituting $m = 4k+4$ and $g = k^2$ and $n < (k+1)^2$, we note that the binomial coefficient becomes:

$$\binom{m}{n-g} \sim \binom{4k}{2k} = \frac{(4k)!}{(2k)!(2k)!} \sim \frac{4^{4k}k^{4k}}{(2^{2k}k^{2k})^2} = \frac{4^{4k}k^{4k}}{4^{2k}k^{4k}} = 4^{2k}$$

Using Stirling's Formula: $(n)! \sim \sqrt{2\pi(n)}\left(\frac{n}{e}\right)^n (1 + O(\frac{1}{n}))$. The total cost for computing the perfect matching for each subset using the Hungarian algorithm is proportional to:

$$O\left(\binom{m}{n-g} \cdot T_H\right) = O(2^k \cdot k^3) = O(2^k)$$

Where $T_H = O(k^3)$ is the complexity of the Hungarian algorithm for a bipartite graph with $k$ nodes on each side.

Considering $C$, the number of centroid traps, is a constant ( $C \leq 9$ for typical 2D QCCD topologies), the total complexity for the mapping is: $O\left(C \cdot 2^k\right) = O(2^k)$.

For the unconstrained search problem, without compacting subsets at the centroid, the problem involves exhaustively checking all subsets of the $(k+1)^2$ traps that are of size equal to the number of clusters $n$:

$$\binom{(k+1)^2}{n} \sim \binom{(k+1)^2}{k^2} = \frac{((k+1)^2)!}{(k^2)! \cdot ((k+1)^2 - k^2)!} = O(k^{2k^2}).$$

Using Stirling's formula, the problem is super-exponential concerning $k$.

Therefore, by ensuring the search space is restricted such that the problem has exponential growth rather than hyper-exponential growth, the mapping is more feasible to compute in a reasonable time for smaller code distances. However, exponential growth soon becomes unmanageable for small $k$. For example, take $n = 100$ qubit clusters, we have $k = \sqrt{n} = 10$ and the unconstrained search $\binom{(k+1)^2}{k^2}$ exploded to roughly $k^{2k^2} = 10^{2 \cdot 10^2} = 10^{200}$, an astronomically large number, while $\binom{4k}{2k}$ grows as roughly $4^{2 \cdot 10} = 2^{40}$, which still takes too long to compute. Nevertheless, the constrained exhaustive exponential search is useful for validating local search approaches for smaller distance codes, which would not be possible for the unconstrained search. For distance $d = 7$ and trap capacity $c = 2$, we have $n \leq 49$ qubit clusters and $k = 7$, so while the unconstrained search $\binom{(k+1)^2}{k^2}$ exploded to roughly $k^{2k^2} = 7^{2 \cdot 7^2} 10^{49}$, while $\binom{4k}{2k}$ grows to $7^{2 \cdot 7} 10^{14}$, which takes about an hour to complete on my laptop.

Fortunately, the mapping can place more constraints by assuming regularity at the boundary of the input code topology, which reduces the $\binom{m}{n-g}$ subsets of the candidate traps of $m$ of size $n - g$ to check to polynomial in $k$. The compiler exploits the regularity at the boundary of the input code topology through a local hill-climbing search (§1) of the $\binom{m}{n-g}$ subsets. This is because the search starts near an optimal solution when the code topology has a regular boundary pattern.

**Algorithm 1** Hill-Climbing Algorithm for Subset Searching

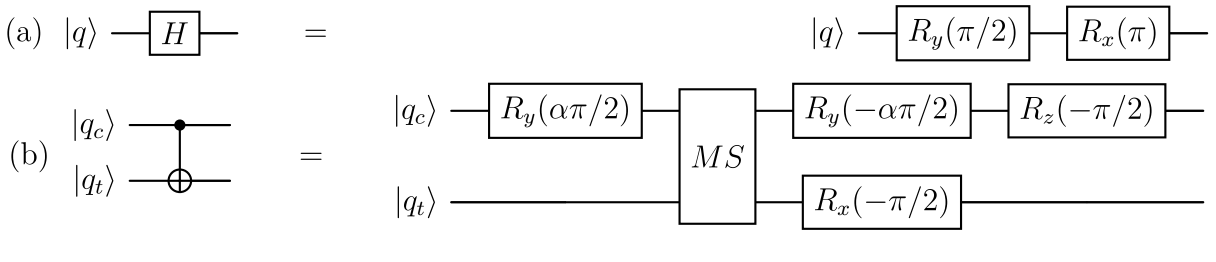
```
 1: Initialize $g$ as the list of guaranteed ions in the subset
 2: Initialize $s$ as the list of an ion path around the outer edge of the expanding square with regular
    spacing.
 3: Initialize subset $B_{\text{in}} = s + g$
 4: Compute initial score $f_{current} \leftarrow f(B_{in})$ using the Hungarian algorithm
 5: while true do
 6:     Initialize $B_{next} = []$, $f_{\text{next}} \leftarrow \infty$
 7:     for $i \in \{1, \ldots, |B_{\text{in}}|\}$ do
 8:         for $b \in W \setminus B_{\text{in}}$ do
 9:             Replace $B_{\text{in}}[i]$ with $b$ to form subset $B_{candidate}$
10:             Compute new cost $f_{\text{candidate}} \leftarrow f(B_{candidate})$ using the Hungarian algorithm
11:             if $f_{\text{candidate}} < f_{\text{next}}$ then
12:                 $B_{next} \leftarrow B_{candidate}$
13:                 $f_{\text{next}} \leftarrow f_{\text{candidate}}$
14:             end if
15:         end for
16:     end for
17:     if $(f(s) \leq f_{\text{next}})$ or maximum iterations exceeded then
18:         return $B_{in}$
19:     end if
20:     Update $B_{in} \leftarrow B_{\text{next}}$, $f_{current} \leftarrow f_{\text{next}}$
21: end while
```

### 3.1.3 Mapping QEC Instructions to QCCD Instructions

Stabiliser circuits are expressed in terms of Hadamard, CNOT and measurement operations. These operations are converted into sequences of MS operations and single-qubit rotations from the QCCD toolbox (§2.3.4) using known gate identities [37]. This is a straightforward IR transformation. However, to be able to execute an entanglement operation between ions located in different traps, the compiler must determine the appropriate sequence of ion movement operations to ensure that both ions co-exist in the same trap.

Figure 3.5 illustrates the decomposition of the Hadamard gate and the CNOT gate into QCCD primitive operations.

**Figure 3.5:** Decomposition of quantum gates: (a) Decomposition of the Hadamard gate $H$ using $R_z(\pi)$, $R_y\left(\frac{\pi}{2}\right)$, and $R_x(\pi)$ rotations; (b) CNOT gate implementation using $XX(\chi), R_x(\theta), R_y(\theta),$ and $R_z(\theta)$ gates [37].



### 3.1.4 Ion Routing Algorithm

The ion routing algorithm computes a path for each ancilla qubit to reach its corresponding data qubit's trap while satisfying QCCD hardware constraints:

- **Trap capacity:** Each trap has a fixed maximum ion count at any time [2, 13].
- **Junction exclusivity:** Only one ion may occupy a junction at any time [16].
- **Segment exclusivity:** Only one ion may occupy a shuttling segment at any time [17, 18].

The QCCD architecture is modelled as a directed graph where nodes represent traps and junctions, while edges are labelled with the sequence of movement primitives required to transfer an ion between nodes. For each ancilla qubit, the shortest path from the source to the destination trap is determined in the directed graph, and then edge labels along this route are concatenated for the sequence of primitives needed to move the qubit. The ancilla ions are re-routed after executing the two-qubit gate to preserve the QCCD constraints. Since entanglement operations must execute in parallel, the routing schedule minimises serialisation using a dynamic forwarding strategy augmented with a global reservation system. Barriers are inserted in the schedule using the happens-before relation in the QEC stabiliser circuit to maximise the number of qubit gates executed in sync.

The algorithm processes the sequence in multiple passes, moving operation primitives into the output schedule until none remain. At the start and end of each pass, each trap is at most one ion below its capacity while no junction or segment contains an ion. This invariants not only ensures that the trap capacity constraint is upheld during execution, but also ensures the termination (§3.1.5) . Each pass of the algorithm is as follows:

**Algorithm 2** Ion Routing Algorithm for a Single Pass

```
 1: Identify and schedule gates that do not require ion movement.
 2: while ∃ gates between qubits in different traps do
 3:     Determine the destination trap for each ancilla qubit that needs routing (Fig. A.2, left).
 4:     for each ancilla do
 5:         Determine the shortest routing path to the destination trap (Fig. A.2, right)
 6:         for each component excepth the source trap on the path do
 7:             Allocate an additional qubit to the component (Fig. A.3 (left)).
 8:             if a component reaches its capacity then
 9:                 Remove the component from the QCCD graph (Fig. A.3 (right)).
10:             end if
11:         end for
12:     end for
13:     Schedule the movement of ancillas along their reserved paths (Fig. A.6).
14:     Schedule the gates that required routing
15:     Resolve the invariant (1) by re-routing ancillas (Fig. A.5).
16: end while
17: Insert a barrier to the schedule (Fig. A.7).
18: Reset the QCCD graph.
```

A barrier is added at the end of each pass to guarantee these conditions for the next pass. The full description of each pass of the ion routing algorithm, with examples, can be found in the Appendix (§2).

### 3.1.5 Proof of Correctness of Ion Routing

First, prove two invariants:

1. At the start and end of each pass, every trap is at most capacity($T$) $-$ 1, and no junctions or segments contain ions.

   - **Proof by induction:** the invariant holds initially by assumption on the input, so it is sufficient to show the inductive step. Assume the invariant at the start of a pass, by induction hypothesis. For a trap $T_i$ at the end of the entanglement operation $i$, let the number of ions in $T_i$ be $n_i$. If $n_i >$ capacity$(T_i) - 1$, the algorithm dynamically re-routes ions back along their reserved paths until $n_i \leq$ capacity$(T_i) - 1$. If the ion has to be re-routed back to its source trap $S$ because no trap was free on the reserved path, then we need to prove numberOfIons$(S) \leq$ capacity$(S) - 1$. However, if numberOfIons$(S) =$ capacity$(S)$, by assumption, a different ancilla must have moved into trap $S$ during the pass and so the ancilla will be re-routed back along its routing path because $S$ is at capacity.

2. During a pass, no two routing paths overlap on any junction or segment.

   - **Proof:** the algorithm allocates routing paths sequentially, ensuring that for each chosen path, all junctions and segments it traverses are marked as unavailable for subsequent routing paths.

Proof of of correctness then has three parts:

**(1) No deadlock:** Let the set of routing paths in a pass be $\{P_1, P_2, \ldots, P_k\}$, where each path $P_i = (e_1, e_2, \ldots, e_m)$ is a sequence of edges (segments or junctions). Since paths are allocated sequentially, and edges are removed from the graph once included in a path, we have:

$$P_i \cap P_j = \emptyset \quad \forall i \neq j.$$

This ensures that ions in $P_i$ are never obstructed by ions in $P_j$.

Also, at any time $t$, consider an ion $q$ at junction $J$ in $P_i$. The next segment $e \in P_i$ is guaranteed to be available because no other ion can reserve $e$ (invariant (2)). Thus, $q$ can move forward. By induction over all segments in $P_i$, $q$ can always reach its destination, proving that no deadlock occurs.

**(2) Termination:** Let the set of unresolved operations at the start of pass $i$ be $O_i$, and let $R_i \subseteq O_i$ denote the subset of operations requiring routing. It is sufficient to prove that if $R_i \neq \emptyset$, then $|R_{i+1}| < |R_i| \quad \text{and} \quad |O_{i+1}| < |O_i|$. In each pass, we have three cases:

1. Case 1: $|O_i| = 0$. There are no unresolved operations, so the algorithm terminates.

2. Case 2: $|O_i| > 0, |R_i| = 0$. Non-routing operations $N_i = O_i \setminus R_i$ are added directly to the schedule because no routing operations are blocking them from taking place, reducing the number of unresolved operations by $|O_i| > 0$.

3. Case 3: $|O_i| > 0, |R_i| > 0$. The ancilla $q \in R_i$ with the highest priority is guaranteed to be routed to its corresponding data qubit from invariant (1); every trap can gain at least one ancilla ion during the pass. Therefore, the number of unresolved operations is reduced by at least one. If $R_i \neq \emptyset$, then:

$$|R_{i+1}| < |R_i| \quad \text{and} \quad |O_{i+1}| < |O_i|.$$