Weekly Progress Presentation

WEEK 1

BHAVYA

Quantum Circuits Basics - Gates

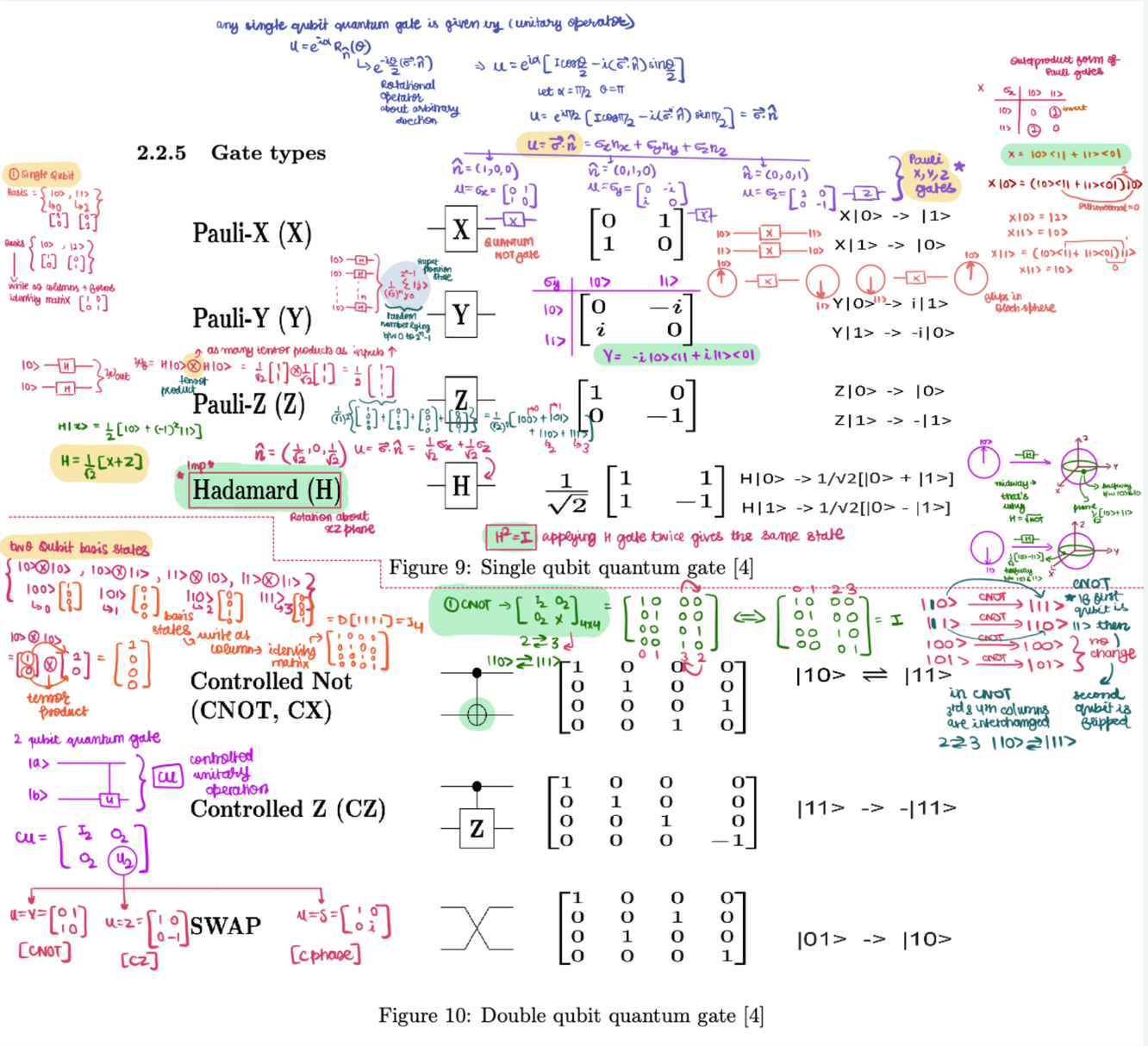
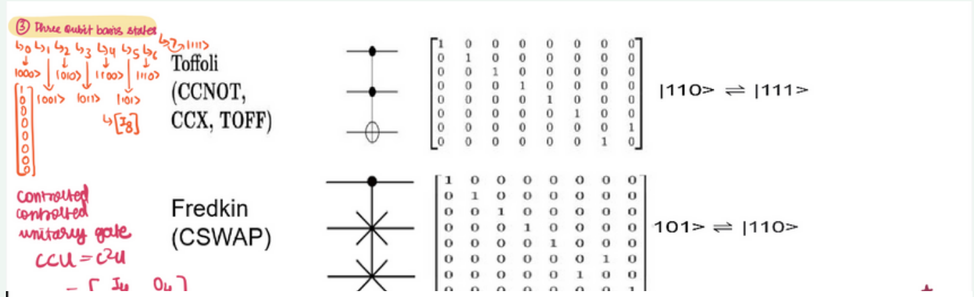


Figure 9: Single qubit quantum gate [4]

Figure 10: Double qubit quantum gate [4]



CNOT's control qubit is followed by the target qubit. The target qubit is flipped for the control qubit is in the state $|1\rangle$; else no change.

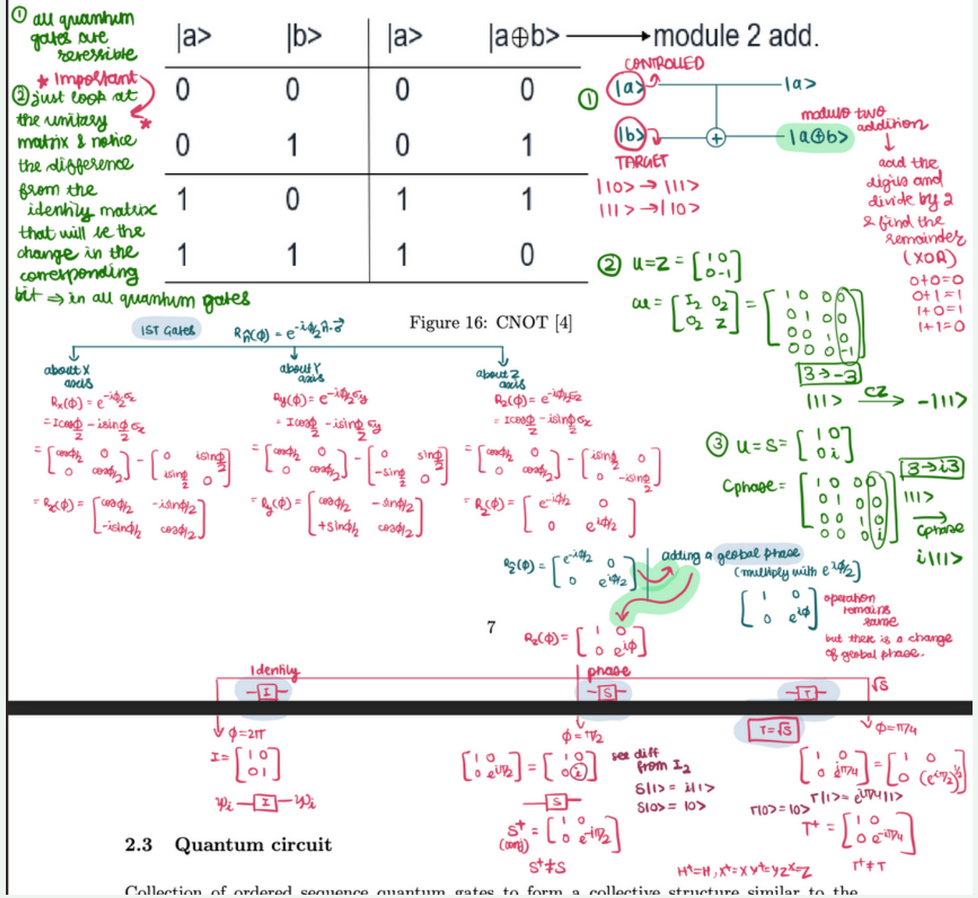


Figure 16: CNOT [4]

Research paper :

A Novel Approach for nearest neighbour realization of 2D Quantum Circuits

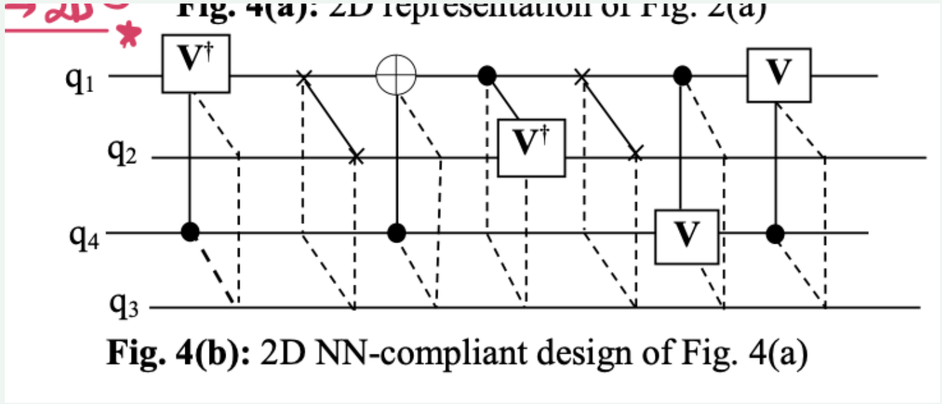


Fig. 4(b): 2D NN-compliant design of Fig. 4(a)

$$NNC_G = |c - t| - 1,$$

$$NNC_{CKT} = \sum_G NNC_G,$$

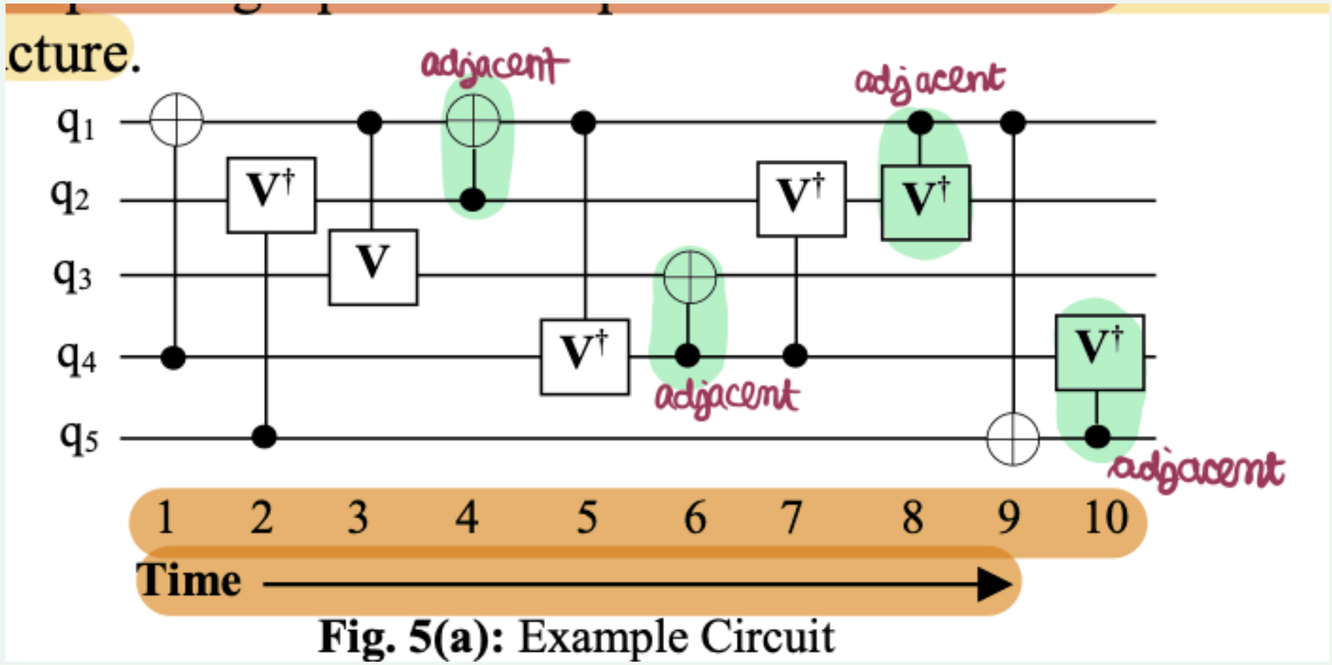


Fig. 5(a): Example Circuit

Phase 1 : Qubit selection policy

Table 2: Qubit time interaction table

Qubits	Time instants	Total interaction
q_1	1,3,4,5,8,9	30
q_2	2,4,7,8	21
q_3	3,6	9
q_4	1,5,6,7,10	29
q_5	2,9,10	21

Table 3: Qubit time costing table

Qubits	Time instants	Total interaction	Time instants	Total costing
q_1	1,3,4,5,8,9	30	1,3,5,9	18
q_2	2,4,7,8	21	2,7	9
q_3	3,6	9	3	3
q_4	1,5,6,7,10	29	1,5,7	13
q_5	2,9,10	21	2,9	11

preference index = total costing / total interaction (≤1)* for each qubit

Table 4: Qubit preference table

Qubits	Total interaction	Total costing	Preference index
q_1	30	18	18/30=0.6
q_2	21	9	9/21=0.42
q_3	9	3	3/9=0.33
q_4	29	13	13/29=0.44
q_5	21	11	11/21=0.52

Table 5: Sorted qubit preference table

Qubits	Total interaction	Total costing	Preference index
q_1	30	18	0.6
q_5	21	11	0.523
q_4	29	13	0.448
q_2	21	9	0.428
q_3	9	3	3/9=0.33

Phase 2 : Qubit Placement policy

1. Find an empty location on the grid adjacent to maximum number of empty cells
2. In case of multiple occurences, place the qubits in locations in order left , up right , down of the last placed qubit as per the availibility of space until such a distinct location is found

in Fig. 5(d).

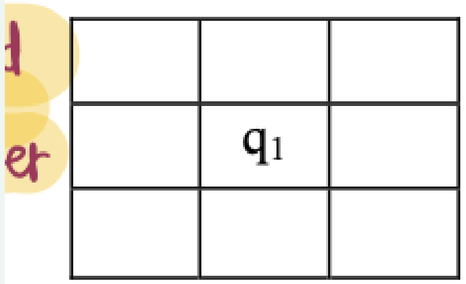


Fig. 5(b): q_1 inserted in 3×3grid

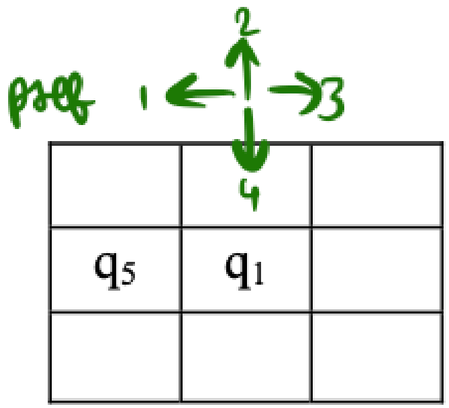


Fig. 5(c): q_5 inserted to the left of q_1

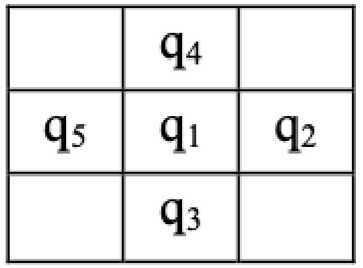


Fig. 5(d): Qubits q_4 , q_2, q_3 placed around q_1

Phase 3 : SWAP gate insertion

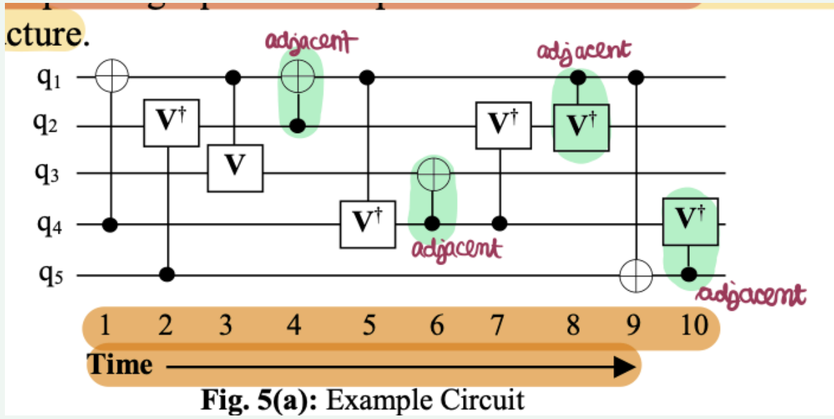
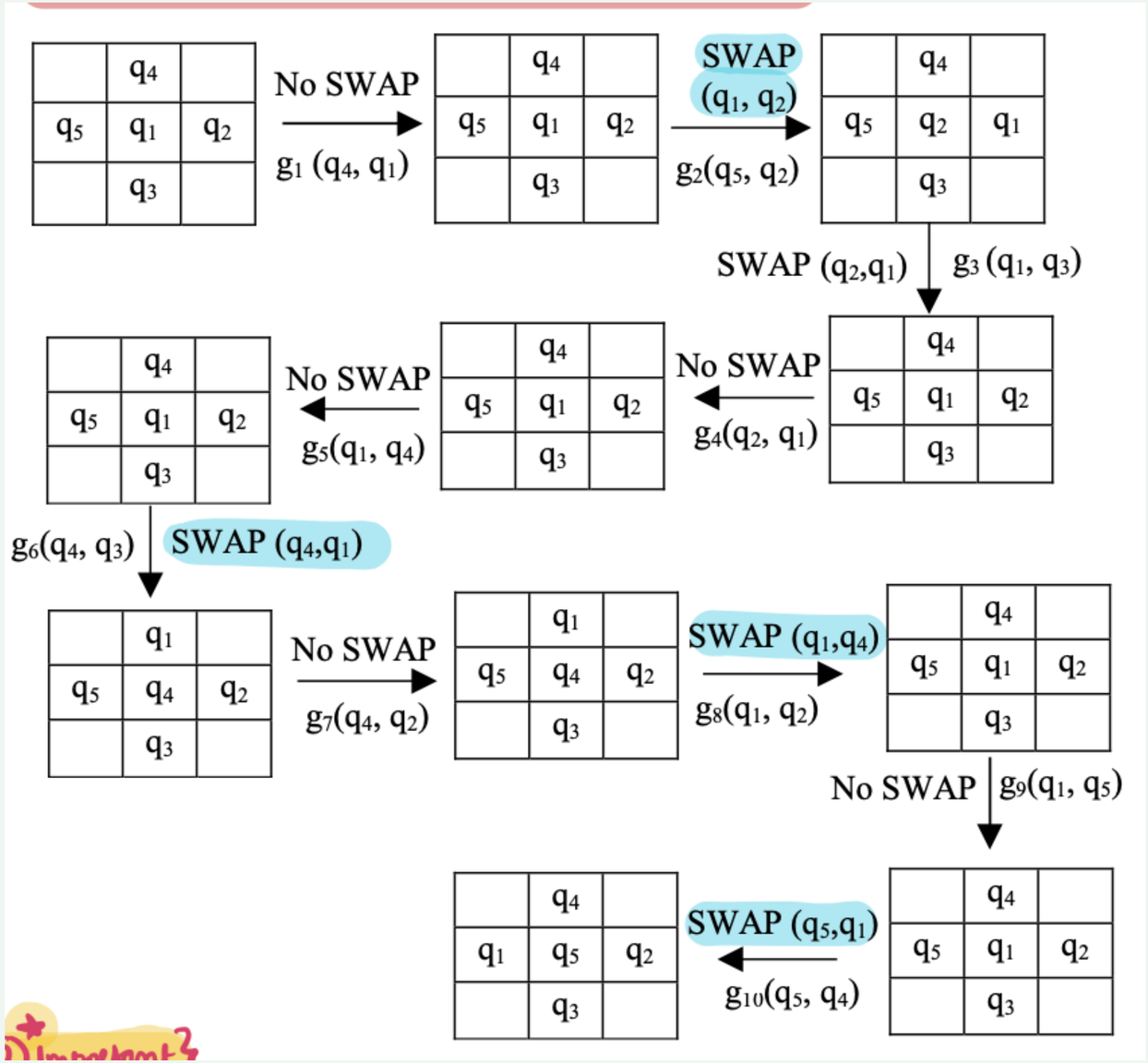


Fig. 5(a): Example Circuit



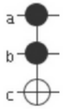
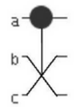
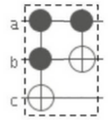
Benchmarks	No. of qubits	Gate count	Results in 1D from [15]		Results in 2D from[15]			Results from [18]			Proposed work results		
			SWAP Count	Quantum Cost	Grid Size	SWAP Count	Quantum Cost	Grid Size	SWAP Count	Quantum Cost	Grid Size	SWAP Count	Quantum Cost
3_17_13	3	14	4	17	2×2	6	20	2×2	3	17	2×2	5	19
4_49_17	4	32	12	45	2×2	13	45	-	-	-	2×2	10	42
aj-e11_165	4	60	36	96	2×3	24	84	3×2	22	82	2×3	18	78
decod24-v3_46	4	9	3	12	3×2	3	12	-	-	-	2×2	2	11
hwb4_52	4	23	10	33	2×2	9	32	2×2	9	32	2×2	7	30
rd32-v0_67	4	8	2	10	2×3	2	10	-	-	-	2×3	2	10
4gt11_84	5	7	1	8	2×3	2	9	2×3	2	9	2×3	2	9
4gt10-v1_81	5	36	20	56	3×2	16	52	3×2	15	51	2×3	14	50
4gt12-v1_89	5	53	35	88	3×2	19	72	2×4	18	71	3×2	20	73
4mod5-v1_23	5	24	9	33	2×3	11	35	3×2	7	31	2×3	8	32
4gt5_75	5	22	12	34	3×3	8	30	2×5	10	32	3×3	9	31
4gt4-v0_80	5	44	34	78	2×3	17	61	4×4	15	59	2×3	16	60
4mod7-v0_95	5	40	21	61	3×3	13	53	2×5	14	54	3×3	10	50
alu-v4_36	5	32	18	50	2×3	10	42	2×5	11	43	2×3	11	43
hwb5_55	5	109	63	172	3×2	45	154	2×7	49	158	3×2	38	147
QFT5	5	10	6	16	3×2	5	15	4×2	5	15	3×2	5	15
hwb6_58	6	146	118	264	2×3	79	225	2×3	76	222	2×3	63	209
mod5adder_128	6	87	51	138	3×2	41	128	2×3	36	123	3×2	28	115
mod8-10_177	6	109	72	181	3×3	45	154	4×3	43	152	3×3	39	148
QFT6	6	15	12	27	2×3	6	21	3×2	7	22	2×3	5	20
rd53_135	7	78	66	144	5×2	39	117	2×7	40	118	3×3	29	107
ham7_104	7	87	68	155	3×3	48	135	2×7	45	132	3×3	38	125
QFT7	7	21	26	47	2×4	18	39	6×2	14	35	2×4	14	35
QFT8	8	28	33	61	4×2	18	46	4×2	23	51	4×2	18	46
QFT9	9	36	54	90	3×3	34	70	5×2	36	72	3×3	24	60
rd73_140	10	76	56	132	4×3	37	113	3×6	43	119	4×3	28	104
sys6-v0_144	10	62	59	121	4×4	31	93	-	-	-	4×3	30	92
QFT10	10	45	70	115	5×3	53	98	4×3	51	96	4×3	33	78
rd84_142	15	112	148	260	5×3	54	166	4×5	62	174	5×3	48	160
cnt3-5_180	16	125	127	252	3×6	69	194	4×4	84	209	3×6	54	179
Total			**971**	**2796**		**775**	**2325**		**740**	**2179**		**628**	**2178**

Table12: Results of large size benchmarks

Benchmarks	Total lines (1D)	Gate count	Initial QC	Grid size	No. Of SWAPs	QC in NN design
rev_17	17	136	136	6×3	214	443
rev_18	18	153	153	5×4	221	374
rev_19	19	171	171	4×5	256	427
ac_21_1	21	130	130	6×4	116	246
ac_21_2	21	67	67	6×4	53	120
ac_21_3	22	42	42	6×4	51	93
hm_20	20	73	73	5×4	69	142
hm_21	21	79	79	6×4	102	181
hm_22	22	85	85	6×4	94	179

Revlib & Benchmarks
File Format and Gates used

```
# 1-bit adder (1bitadder.real)
.version 2.0
.numvars 4
.variables x1 x2 x3 x4
.inputs a b cin 0
.outputs g g cout sum
.constants ---0
.garbage 11--
.state ff1 x1
.state ff2 x2 x3 2 # init. value
.begin
t3 x2 x1 x4
t2 x1 x2
t3 x3 x2 x4
t2 x2 x3
.end
```

Gate	Specification
Toffoli 	A (multiple control) Toffoli gate is signified by the character `t` and an integer indicating the size of the gate followed by a list of identifiers for the respective lines such that the target line is at the end of the list. Note that Toffoli gates include (controlled) NOT gates as well. Example: `t3 a b c`
Fredkin 	A (multiple control) Fredkin gate is signified by the character `f` and an integer indicating the size of the gate followed by a list of identifiers for the respective lines such that the targets of the gates are at the end of the list. Example: `f3 a b c`
Peres 	A Peres gate is signified by the character `p` and an integer indicating the size of the gate followed by a list of identifiers for the respective lines such that the targets of the gates are at the end of the list. Example: `p3 a b c`
V gate 	A V gate is signified by the character `v` and an integer indicating the size of the gate (i.e. 2) followed by a list of identifiers for the respective lines such that the target line is at the end of the list. Example: `v a b`
V+ gate 	A V+ gate is signified by the characters `v+` and an integer indicating the size of the gate (i.e. 2) followed by a list of identifiers for the respective lines such that the target line is at the end of the list. Example: `v+ a b`

Python code to convert .real text file to Qiskit format

```
i = open("/Users/bhavya/Desktop/real1.txt", "r")
o = open("/Users/bhavya/Desktop/qiskit1.txt", "w")
o.write("import numpy as np\n")
o.write("from qiskit import QuantumCircuit\n")
num = 4
o.write("circ = QuantumCircuit(" + str(num) + ")\n")
for line in i :
    st = line.split()
    if (st[0][0] == 't'):
        if (st[0][1] == '1'):
            o.write("circ.x(" + str((ord(st[1])-97)) + ")\n")
        elif (st[0][1] == '2'):
            o.write("circ.cx(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + ")\n")
        elif (st[0][1] == '3'):
            o.write("circ.ccx(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + "," + str((ord(st[3])-97)) + ")\n")
        else:
            control = []
            n = len(st)-1
            for i in range(1,n):
                control.append(ord(st[i])-97)
            o.write("circ.mct(" + str(control) + "," + str((ord(st[-1])-97)) + ")\n")

    elif (st[0][0] == 'p'):
        o.write("circ.barrier()\n")
        o.write("circ.ccx(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + "," + str((ord(st[3])-97)) + ")\n")
        o.write("circ.cx(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + ")\n")
        o.write("circ.barrier()\n")

    elif (st[0][0] == 'v'):
        if (st[0][1] == '+'):
            o.write("csxdg_gate = SXdgGate().control()\n")
            o.write("circ.append(csxdg_gate, [" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + ")\n")
        else :
            o.write("circ.csx(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + ")\n")

    elif (st[0][0] == 'f'):
        if (st[0][1] == '2'):
            o.write("circ.swap(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + ")\n")
        elif (st[0][1] == '3'):
            o.write("circ.cswap(" + str((ord(st[1])-97)) + "," + str((ord(st[2])-97)) + "," + str((ord(st[3])-97)) + ")\n")
```

Qiskit & IBM Quantum Experience

4mod7-v0_95

In [1]:
```
import numpy as np
from qiskit import QuantumCircuit
```

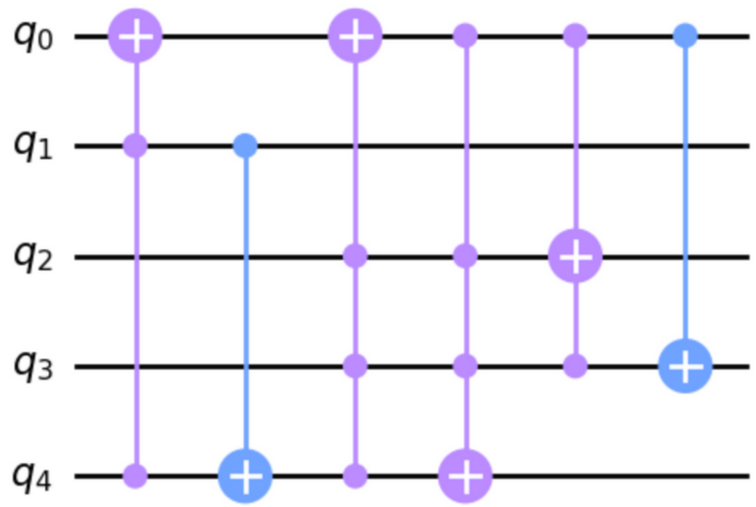
In [2]:
```
circ = QuantumCircuit(5)
```

In [3]:
```
circ.ccx(4,1,0)
circ.cx(1,4)
circ.mct([4,3,2],0)
circ.mct([3,2,0],4)
circ.ccx(0,3,2)
circ.cx(0,3)
```

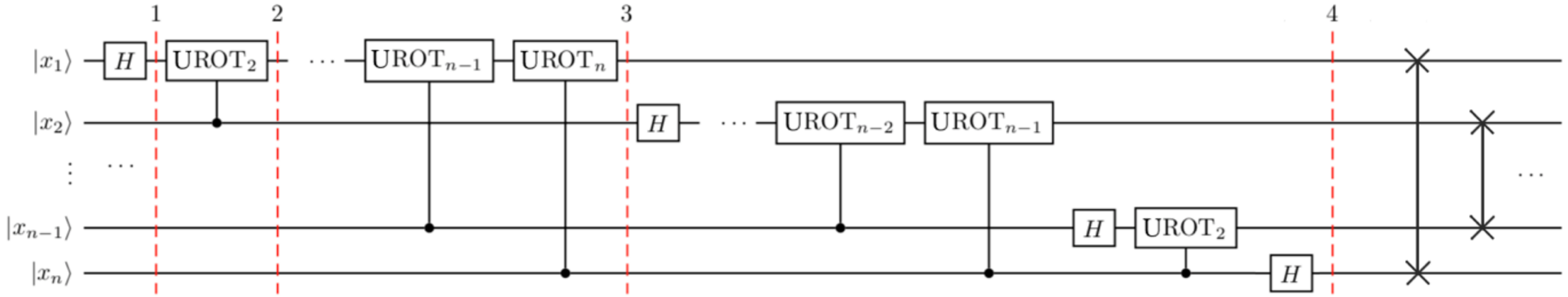
Out[3]:
```
<qiskit.circuit.instructionset.InstructionSet at 0x1265e33a0>
```

In [4]:
```
circ.draw('mpl')
```

Out[4]:



n qubit Fourier Transform in Qiskit



Controlled phase rotation gate

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & e^{i\theta} \end{bmatrix}$$

$$\theta = 2\pi/2^k = \pi/2^{k-1}$$

```
def qft_rotations(circuit, n):
    if n == 0:
        return circuit
    n = n - 1
    circuit.h(n)
    for qubit in range(n):
        circuit.cp(pi/2**(n-qubit), qubit, n)
    qft_rotations(circuit, n)
def qft_swaps(circuit,n):
        for qubit in range(int(n/2)):
            circuit.swap(qubit,(n-qubit-1))
qc = QuantumCircuit(9)
qft_rotations(qc,9)
qc.barrier()
qft_swaps(qc,9)
qc.draw('mpl')
```

WEEK 2 & 3

```python
def nnc(qubits) :
    k = len(qubits)
    if(k==1):
        return 0
    elif(k==2):
        return abs(qubits[1]-qubits[0])-1
    else:
        qubits.sort()
        gate_cost = 0
        for i in range(1,k):
            gate_cost += (qubits[i]-qubits[i-1]-1)
        return gate_cost
i = 1
num_qubits = circ.num_qubits
time_instants = []
total_interaction = []
time_costings = []
total_costings = []
preference_index = []
for q in range(num_qubits) :
    time_instants.append([])
    time_costings.append([])
for instruction, qargs, _ in circ:
    qubits = [circ.qubits.index(qarg) for qarg in qargs]
    for q in qubits :
        time_instants[q].append(i)
    if nnc(qubits) > 0 :
        for q in qubits :
            time_costings[q].append(i)
    i = i+1
for q in range(num_qubits):
    total_interaction.append(sum(time_instants[q]))
for q in range(num_qubits):
    total_costings.append(sum(time_costings[q]))
print(time_instants)
print(total_interaction)
print(time_costings)
print(total_costings)
```

```
for q in range(num_qubits):
        k = total_costings[q]
        l = total_interaction[q]
        preference_index.append([k/l,q])
print(preference_index)
preference_index = (sorted(preference_index))
preference_index.reverse()
print(preference_index)
grid_size = [2,2]
grid = []
for p in range(qubits) :
    if p = 0 :
        grid[grid_size[0]/2][grid_size[1]/2] = p
    else :
```

```
[[2, 4, 5], [3, 4, 5, 6], [1, 2, 3, 4, 5, 6]]
[11, 18, 21]
[[2], [], [2]]
[2, 0, 2]
[[0.18181818181818182, 0], [0.0, 1], [0.09523809523809523, 2]]
[[0.18181818181818182, 0], [0.09523809523809523, 2], [0.0, 1]]
```

TYPES OF QUANTUM (QUBIT) TECHNOLOGY

TRAPPED ION QUANTUM COMPUTERS Gate-based ion trap processors	Trapped ion quantum computers **implement qubits using electronic states of charged atoms called ions.** The ions are **confined and suspended above the microfabricated trap using electromagnetic fields**. Trapped-ion based systems **apply quantum gates using lasers to manipulate the electronic state of the ion**. Trapped ion qubits use atoms that come from nature, rather than manufacturing the qubits synthetically.
SUPERCONDUCTING QUANTUM COMPUTERS Gate-based superconducting processors	Superconducting quantum computing is an **implementation** of a quantum computer i**n superconducting electronic circuits**. Superconducting qubits are built with superconducting electric circuits that **operate at cryogenic temperatures.**
Photonic processors	A quantum photonic processor is a device that manipulates light for computations. Photonic quantum computers use **quantum light sources that emit squeezed-light pulses**, with **qubit equivalents** that **correspond to modes of a continuous operator, such as position or momentum.**

Neutral atom processors

Neutral atom qubit technology is **similar to trapped ion technology. However, it uses light instead of electromagnetic forces to trap the qubit and hold it in position.** The atoms are not charged and the circuits can **operate at room temperatures**

Rydberg atom processors

A Rydberg atom is an **excited atom with one or more electrons that are further away from the nucleus, on average**. Rydberg atoms have a number of peculiar properties including an exaggerated response to electric and magnetic fields, and long life. When used as qubits, they offer **strong and controllable atomic interactions** that you can tune by selecting different states.

Quantum annealers

Quantum annealing uses a **physical process to place a quantum system's qubits in an absolute energy minimum**. From there, the hardware gently alters the system's configuration so that its energy landscape reflects the problem that needs to be solved. The advantage of quantum annealers is that the number of qubits can be much larger than those available in a gate-based system. However, their use is limited to specific cases only.

This method of quantum computing uses **superconducting circuits to create and manipulate qubits.** These circuits are made from materials that can conduct electricity with zero resistance at very low temperatures, “down to the milli-kelvin,” says Professor Young. This makes them **ideal for creating and controlling the fragile quantum states needed for successful quantum computing processes.**

Currently, this method is one of the leading technologies for building practical quantum computers, with companies like **Google and IBM** developing their own working prototypes with dozens of qubits

Maintaining fragile quantum states for long enough to perform calculations or the need to reduce errors caused by environmental factors such as temperature fluctuations.

The qubits in trapped ion quantum computing are highly stable, with very low error rates, and can be manipulated with high precision. This makes trapped ion quantum computers promising for certain types of quantum algorithms, such as those used in cryptography and simulation.

Whilst the methodology of trapped ion quantum computing is great, it is increasingly difficult to trap and manipulate large numbers of ions without causing errors.

Quantum point defects are defects in the crystalline structure of certain materials that can be used as qubits in quantum computing.

In these structures, certain atoms are missing or replaced by other types of atoms, creating a clear ‘defect’ in the crystal lattice. **These defects can then be used to trap electrons and create ‘spin’ states, which can be used as qubits. Spin qubits can be manipulated by applying electromagnetic fields, allowing them to perform quantum operations.**

However, quantum point defects are still in-development, and there are many challenges involved in the consistent creation and manipulation of these defects to a high level of precision.

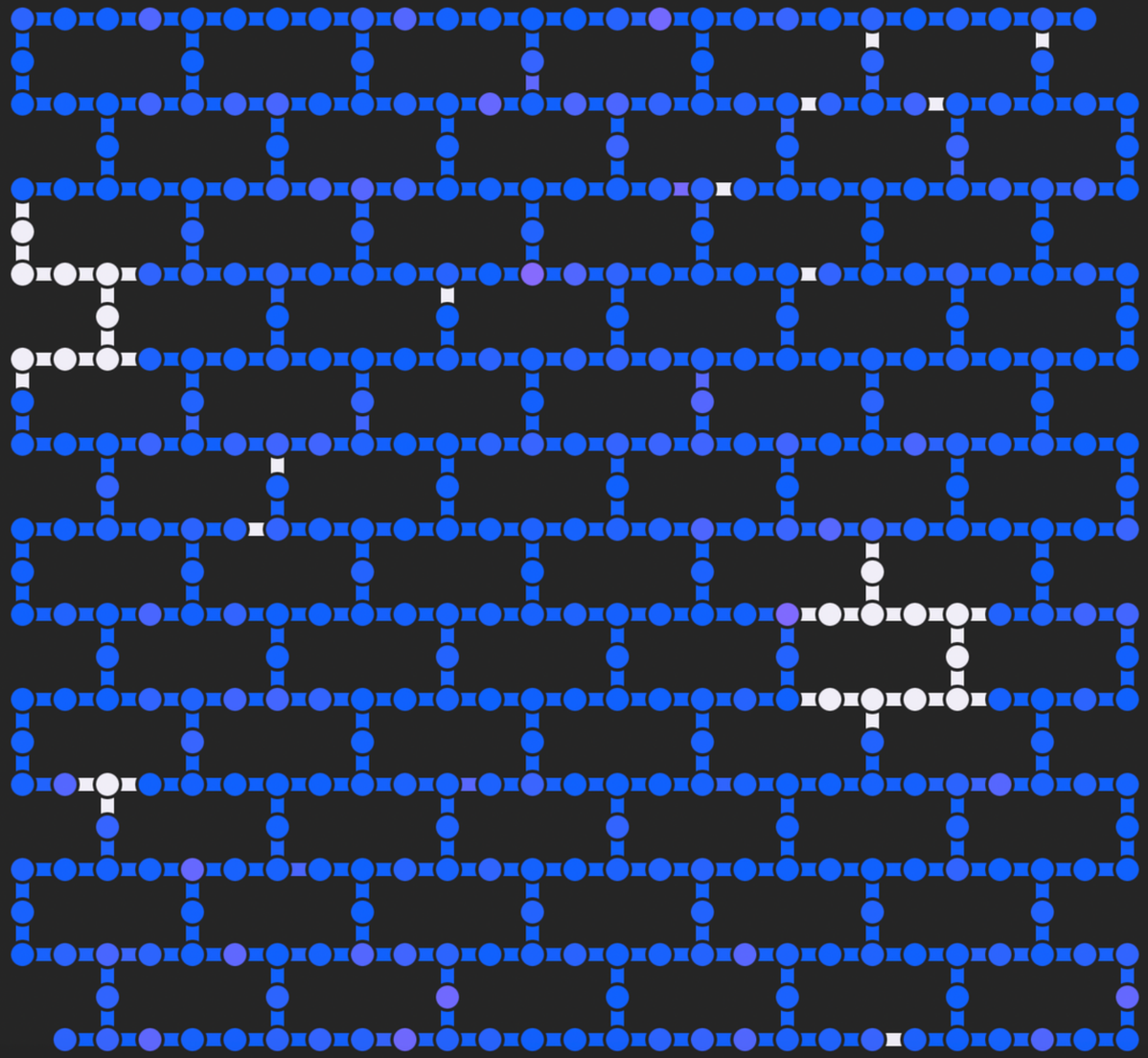
All quantum systems deployed by IBM Quantum are based on superconducting qubit technology, as the control and scalability of this technology pave a clear path to achieving quantum advantage with these systems. These systems are denoted by names that start with ibmq_* (older systems) or ibm_* (newer systems).

Name	Qubits ↓	QV	CLOPS	Status	Total pending jobs	Processor type	Plan	Features
ibm_seattle Exploratory	433	-	-	Online	45	Osprey r1	premium	
ibm_sherbrooke	127	32	904	Online	53	Eagle r3	premium	OpenQASM 3
ibm_kyiv Exploratory	127	-	-	Online - Queue paused	35	Eagle r3	premium	
ibm_brisbane	127	-	-	Online	3	Eagle r3	premium	
ibm_nazca	127	-	-	Online - Queue paused	120	Eagle r3	premium	
ibm_ithaca Exploratory	65	-	-	Online	0	Hummingbird r3	premium	
ibm_prague Exploratory	33	-	-	Offline	0	Egret r1	premium	
ibm_algiers	27	128	2.2K	Online	79	Falcon r5.11	premium	OpenQASM 3
ibmq_kolkata	27	128	2K	Online	1289	Falcon r5.11	premium	OpenQASM 3
ibmq_mumbai	27	128	1.8K	Online	482	Falcon r5.10	premium	OpenQASM 3

Items per page: 10 | 1-10 of 23 items | 1 of 3 pages

ibm_seattle

Exploratory



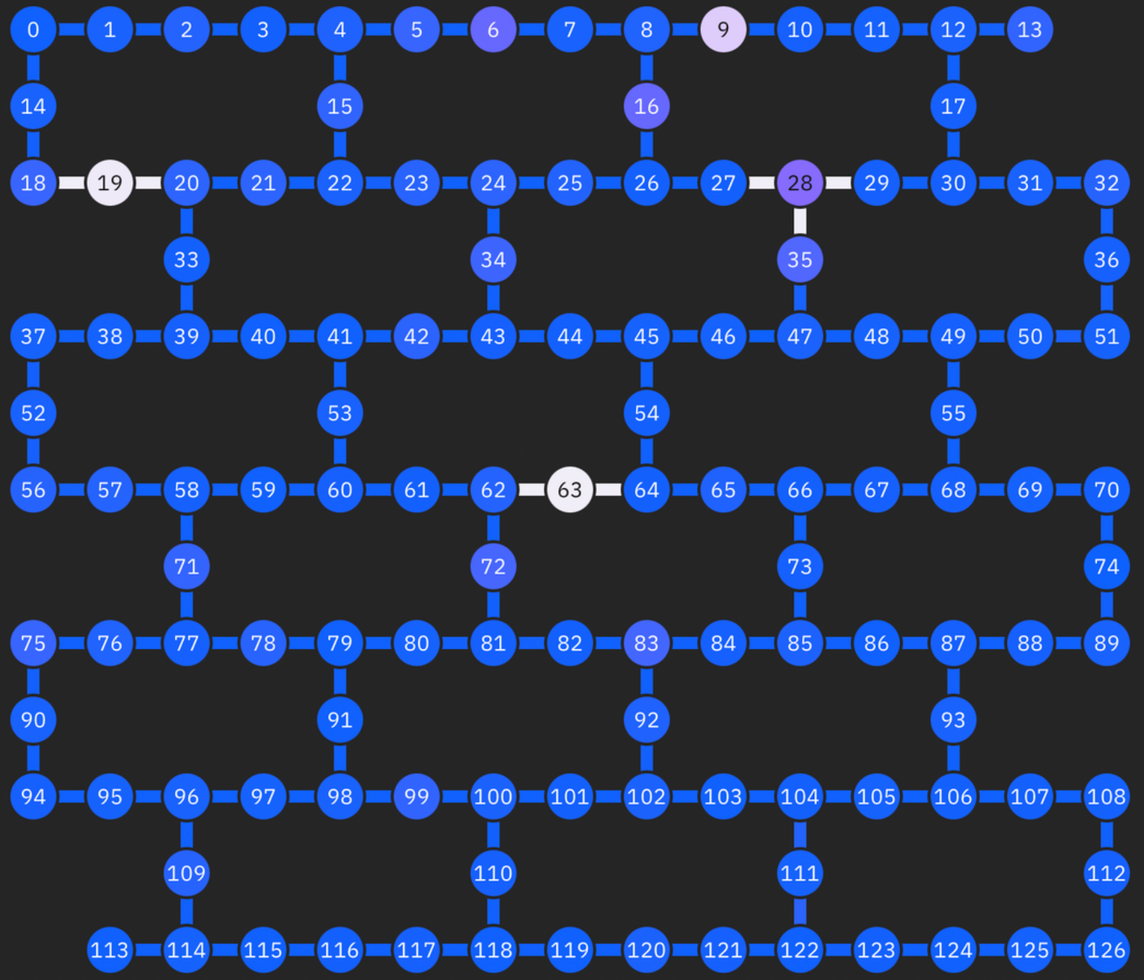
433
Qubits

Status:	Online
Total pending jobs:	49 jobs
Processor type ⓘ:	Osprey r1
Version:	1.0.0
Basis gates:	ECR, ID, RZ, SX, X
Your usage:	--

Median ECR Error:	2.133e-2
Median SX Error:	6.481e-4
Median Readout Error:	5.667e-2
Median T1:	85.86 us
Median T2:	59.88 us
Instances with access:	--

ibm_sherbrooke

OpenQASM 3



127
Qubits

32
QV

904
CLOPS

Status:	Online
Total pending jobs:	22 jobs
Processor type:	Eagle r3
Version:	1.4.4
Basis gates:	ECR, ID, RZ, SX, X
Your usage:	--

Median ECR Error:	8.014e-3
Median SX Error:	2.266e-4
Median Readout Error:	9.700e-3
Median T1:	297.02 us
Median T2:	162.06 us
Instances with access:	--

Qubit:
Readout assignment error

Median 9.700e-3
min 2.500e-3 max 2.828e-1

Connection:
ECR error

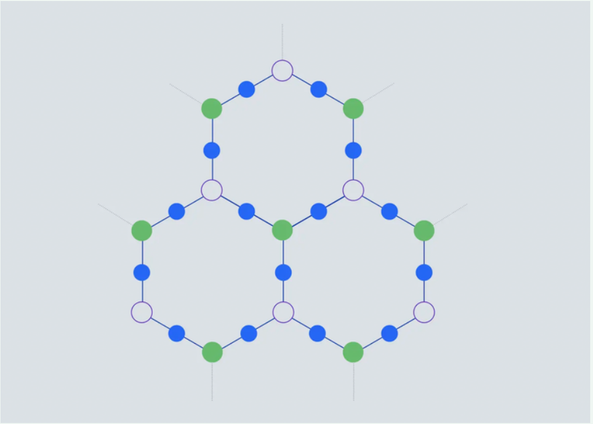
Median 8.014e-3
min 3.298e-3 max 1.000e+0

Quantum Processors by IBM

As quantum processors scale up, **each additional qubit doubles the amount of space complexity** — the amount of memory space required to execute algorithms — for a classical computer to reliably simulate quantum circuits. We hope to see quantum computers bring real-world benefits across fields as **this increase in space complexity moves us into a realm beyond the abilities of classical computers.**

Quantum mechanics-based computers could outperform classical computers at simulating nature. However, constructing one of these devices is an enormous challenge. **Qubits can decohere** — **or forget their quantum information** — **with even the slightest nudge from the outside world**.

Eagle is based on the **heavy-hexagonal qubit layout** as debuted in the Falcon processor, where qubits connect with either two or three neighbors as if sitting upon the edges and corners of tessellated hexagons. **This particular connectivity decreased the potential for errors caused by interactions between neighboring qubits** — **providing significant boosts in yielding functional processors.**



Osprey

Osprey is nearly quadruple the size of Eagle at **433 qubits**. The larger chip sizes have required further enhancements to **device packaging, as well as custom flex cabling in the cryostat to fit the greater I/O requirements within the same wiring footprint**.

Eagle

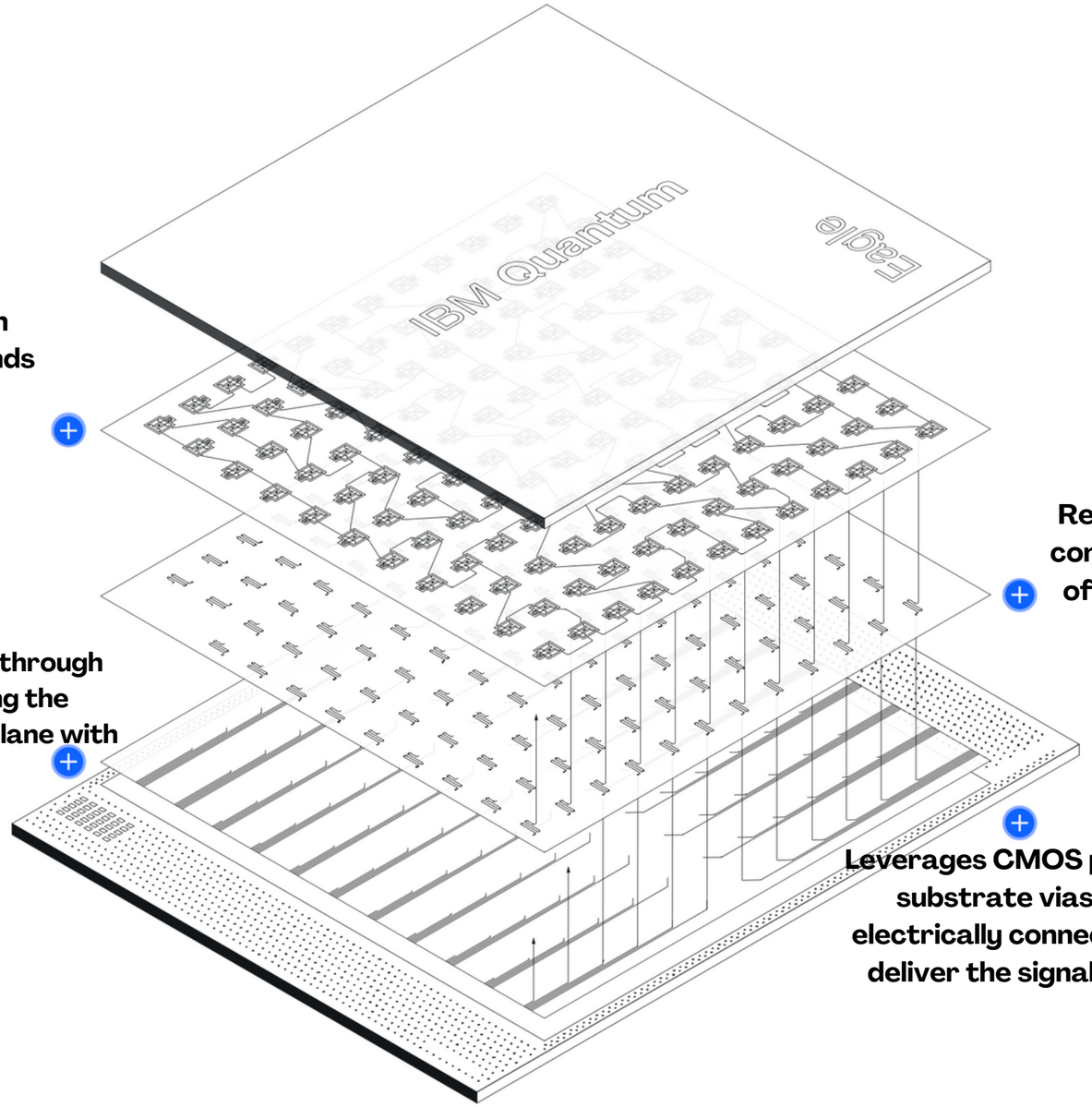
At **127 qubits,** the Eagle processor family incorporates **more scalable packaging technologies** than previous generations. In particular, signals pass through multiple chip layers so as to allow for **high-density I/O without sacrificing performance**

Hummingbird

Using a **heavy-hexagonal qubit layout,** the Hummingbird family allows up to **65 qubits.**

Qubit plane
Transmon qubits attached to an interposer chip through bump bonds offer hardware simplicity in a scalable architecture with controllable features.

Wiring plane
Buried wiring layer connects to the other planes through superconducting thru-substrate vias, providing the flexibility to efficiently route signals to the qubit plane with low crosstalk.



Resonator plane
Resonators for qubit readout wired through connectors. Measured shifts in the frequency of the resonator depend on the state of the qubit.

Interposer
Leverages CMOS packaging techniques, including thru-substrate vias, to exploit the third dimension to electrically connect the qubits to the other planes and deliver the signals while protecting their coherence.

Egret

Egret brings the innovations of tunable couplers onto a **33-qubit** platform, resulting in **faster and higher-fidelity two-qubit gates.**

Falcon

The Falcon family of devices offers a valuable platform for **medium-scale circuits,** and also serves as a valuable platform for demonstrating **performance and scalability improvements before they're pushed onto the larger devices.**

Canary

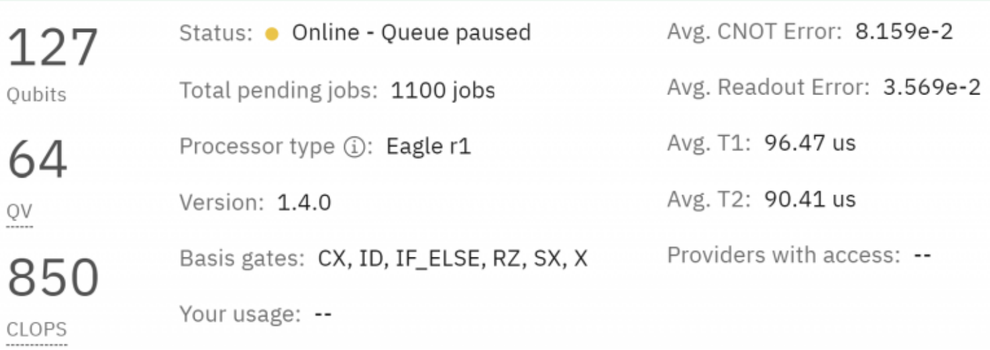
The Canary family comprises small designs containing anywhere from **5 to 16 qubits.**

Optimized 2D lattice: All of the qubits and readout resonators are on the same layer

CHALLENGES FACED in QUANTUM COMPUTER DEVELOPMENT

Current quantum hardware is subject to different sources of noise, the most well-known being qubit decoherence, individual gate errors, and measurement errors. These errors limit the depth of the quantum circuit that we can implement. However, even for shallow circuits, noise can lead to faulty estimates.

First Eagle 127-qubit generation -
Two-qubit gate error rate is 8% and readout errors are 3.7%. . It means that after just 10 entangling quantum gates, you have less than 50% chance to get a good result.



"Quantum chips have to operate at **very low temperatures** in order to maintain the quantum information,"

While the refrigeration system can bring temperatures down to extremes, it can't remove heat very quickly - so if you have a chip in there that's creating a lot of heat, you're going to have a problem.

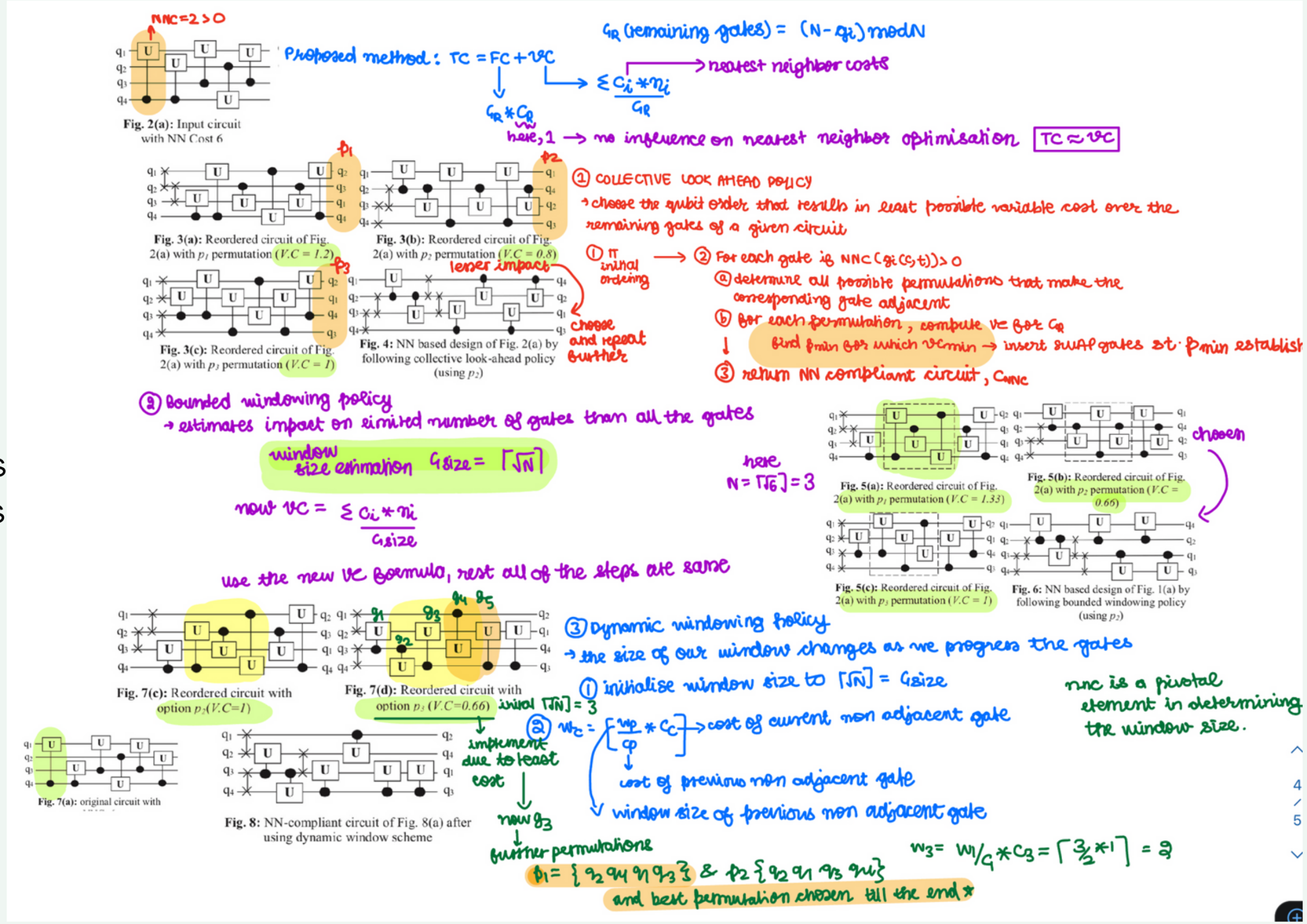
Decoherence errors are a major challenge for building practical quantum computers as they can cause errors in calculations and make it incredibly hard to maintain quantum information for long enough to perform demanding computational tasks. Reducing decoherence errors remains a major challenge for scaling up quantum computing to larger and more complex systems.

Research Papers

- **Not All SWAPs Have the Same Cost**: A Case for Optimization-Aware Qubit Routing
- Improved **Look-ahead Approaches for Nearest Neighbor Synthesis of 1D** Quantum Circuits
- Look-ahead Schemes for **Nearest Neighbor Optimization of 1D and 2D** Quantum Circuits
- Linear Nearest Neighbor Synthesis of **Reversible Circuits by Graph Partitioning**
- Efficient **Mapping** of Quantum Circuits to the **IBM QX Architectures**

Improved Look-ahead Approaches for Nearest Neighbor Synthesis of 1D Quantum Circuits

We introduce a heuristic look-ahead design methodology consisting of **three distinct cost estimation models** to obtain a linear NN based representations. Although the notion of look-ahead scheme has already been discussed in the earlier works, the way it has been implemented there differs from the one used here which have shown significant improvements over the reported works.



In [29]:

```
gate_count = circ.count_ops()
total_gates = sum(gate_count.values())
N = total_gates
def gr(i):
    return (N-i)%N
def fixed_cost() :
    cost = 0
    for i in range(1,N+1):
        cost += gr(i)
    return cost
def nnc(qubits) :
    k = len(qubits)
    if(k==1):
        return 0
    elif(k==2):
        return abs(qubits[1]-qubits[0])-1
    else:
        qubits.sort()
        gate_cost = 0
        for i in range(1,k):
            gate_cost += (qubits[i]-qubits[i-1]-1)
        return gate_cost
i = 1
# i describes the gate number (it starts from 1)
variable_cost = 0
for instruction, qargs, _ in circ:
    qubits = [circ.qubits.index(qarg) for qarg in qargs]
#    print(nnc(qubits))
#    if(nnc(qubits) > 0) :
    print("Qubits:", qubits)
#    permutation_set = [[0,2,1],[1,0,2]]
#    for p in permutation_set :
    variable_cost = 0
    variable_cost += nnc(qubits)
    if(gr(i) != 0):
        variable_cost /= gr(i)
    print(variable_cost)
    i = i+1
    print()
total_cost = variable_cost
num_qubits = circ.num_qubits
print(num_qubits)
print(N)
print(fixed_cost())
```

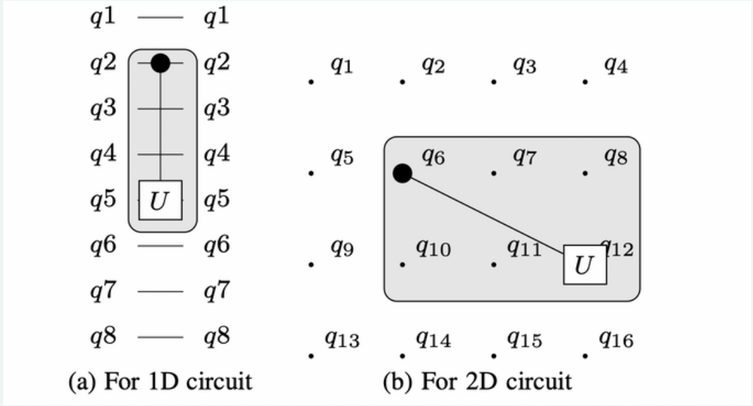
Look-ahead Schemes for Nearest Neighbor Optimization of 1D and 2D Quantum Circuits

This work introduces a heuristic which aims for tackling that problem by **additionally incorporating a look-ahead methodology**. The general idea is to determine all possible options on how to "move" qubits together in order to make methodology them adjacent. Then, the **one is chosen which has the best impact to the following gates**

In the following, we call the possible nearest neighbor- compliant positions SWAP regions. In either case, the precise "movement" influences the qubit/target positions of all follow- ing gates and, by this, has a significant effect to the overall costs (i.e. the number of SWAP gates) obtained by nearest neighbor optimization.

For each non-adjacent gate which is considered, a decision has to be made how exactly the corresponding control/target qubits are "moved" together. In fact, several options exists for this. More precisely:

- In **1D architectures**, the control qubit can be "moved" to the target qubit, the target qubit can be moved to control qubit, or both qubits can be moved together. .
- In **2D architectures,** the control qubit and the target qubit can be "moved" to any adjacent position within a rectangle spanned by the original positions of these qubits.



nearest neighbor costs

$nnc(g) = \begin{cases} 0 & g \text{ unary} \\ \sum_{i=1}^{m} |t_i - c_i| - 1 & \text{otherwise} \end{cases}$

$nnc_\pi(g) = \left(\sum_{i=1}^{m} |\pi(t_i) - \pi(c_i)| - 1 \right)$

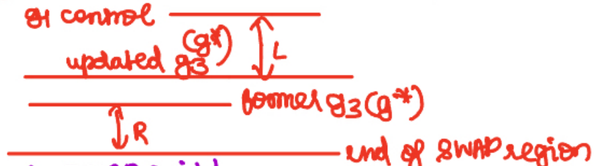
$nnc(C) = \sum_{g \in C} nnc(g)$

① Joint consideration scheme - iterate over all the leftover gates to find the permutation with least NNC

② Iterative scheme → don't consider the gates all at once - consider them one by one (iteratively)

↓

For each gate check whether qubits of g_j motivate a particular option



↳ if no such SWAP → j+1

determine a SWAP gate g* within the SWAP region that reduces costs of g_j

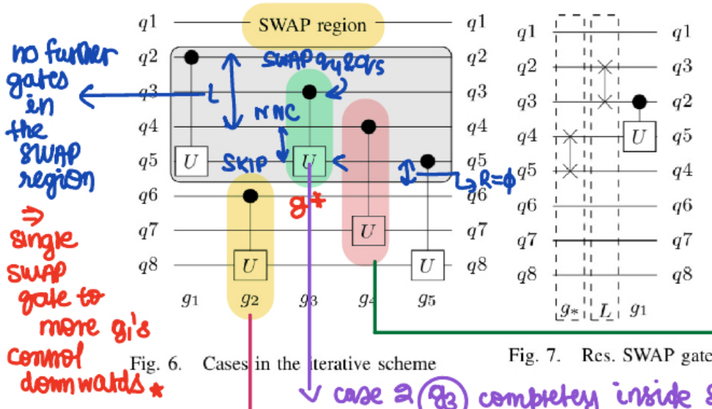
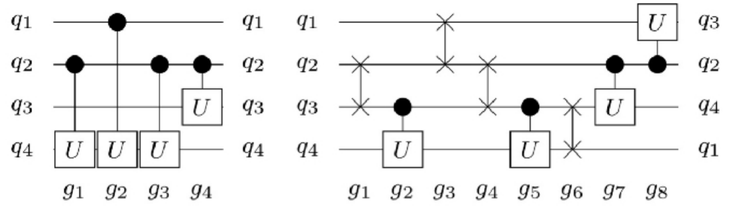


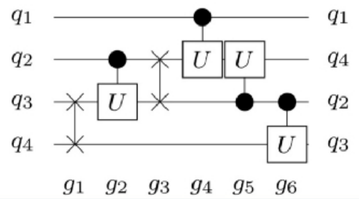
Fig. 6. Cases in the iterative scheme

Fig. 7. Res. SWAP gates

(if end of circuit is reached, choose direct permutation from source to target and return corresponding SWAP gates)



(a) Given circuit (b) Following the scheme from Section IV-A



(c) Following the same scheme with a window of size 2

Fig. 8. Limitations of the look-ahead methodology

LIMITATION: in large circuits we don't need very farther gates to influence the decision of SWAP gates ↑ Restricted circuit tool * ↓ CIRCUIT WINDOW

Linear Nearest Neighbor Synthesis of Reversible Circuits by Graph Partitioning

The algorithm determines the reordering of indices of the qubit line(s) for **both single control and multiple controlled gates.** Experimental results for **placing the target qubits of Multiple Controlled Toffoli (MCT) library of benchmark circuits show a significant reduction in gate count and quantum gate cost** compared to those of related research works.

The work is based on reordering the qubit lines so that the interacting qubits are adjacent to each other, i.e., **the distance between target and control lines are minimized** (this is termed as Nearest Neighbor Cost (NNC)

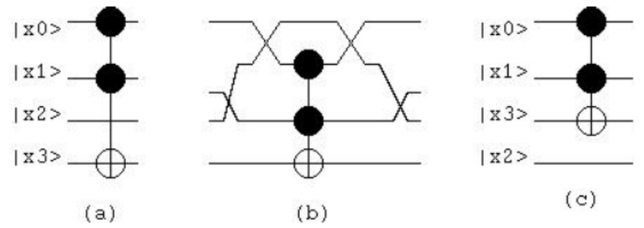
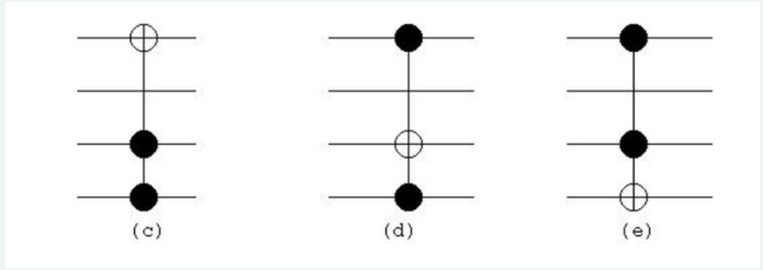


Fig. 1. LNN synthesis of TOFFOLI (a) a single TOFFOLI gate, (b) introduction of SWAP gates resulting 4 SWAPs. (c) is the reordered circuit resulting zero SWAP gate requirement

we further sub-divide the LNN synthesis problem for QBCs as one for NOT, CNOT and TOFFOLI (NCT) and another for multiple controlled TOFFOLI (MCT) gates

A. LNN Synthesis for NCT gate library

1. Straightforward for CNOT gate - abs(t-c)-1
2. TOFFOLI gate - three cases possible



Rules for counting SWAP pairs

Case 1: $target < ctrl_1 < ctrl_2$
if $ctrl_1 - target > 1$, then required SWAP pair is $s_1 + s_2$ where $s_1 = ctrl_1 - target - 1$ and $s_2 = ctrl_2 - target - 2$
else if $ctrl_2 - ctrl_1 > 1$, then required SWAP pair is s where $s = ctrl_2 - ctrl_1 - 1$

Case 2: $ctrl_1 < target < ctrl_2$
if $target - ctrl_1 > 1$, then required SWAP pair is s where $s = target - ctrl_1 - 1$, otherwise no SWAP pair is needed
if $ctrl_2 - target > 1$, then required SWAP pair is s where $s = ctrl_2 - target - 1$, otherwise no SWAP pair is needed

Case 3: $ctrl_1 < ctrl_2 < target$
if $target - ctrl_2 > 1$, then required SWAP pair is $s_1 + s_2$ where $s_1 = target - ctrl2 - 1$ and $s_2 = target - ctrl_1 - 2$
else if $ctrl_2 - ctrl_1 > 1$, then required SWAP pair is s where $s = ctrl_2 - ctrl_1 - 1$

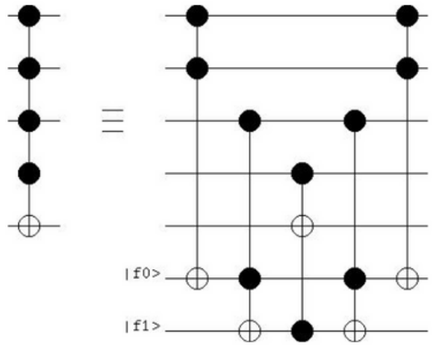


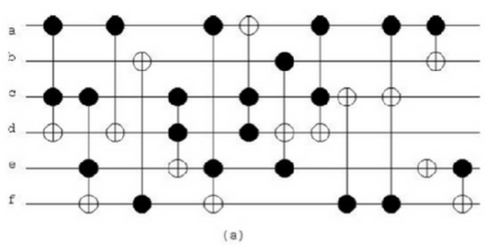
Fig. 3. Decomposition of MCT gate: Replacement of a C^4NOT gate by equivalent TOFFOLI with two ancillary qubits f_0,f_1

B. LNN Synthesis for MCT gate library

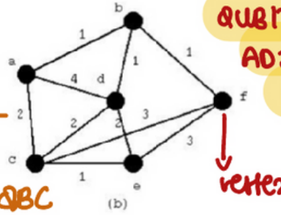
Decompose each MCT gate in the reversible circuit into an equivalent NCT network. After then the nearest neighbor synthesis is to be done for the entire circuit

It can be shown that for decomposition of a single C^k NOT gate, the number of TOFFOLI required is 2(k – 2) + 1 and number of ancillary qubits required is k – 2 . The resultant circuit contains only NOT, CNOT and TOFFOLI gates.

Balanced graph partioning approach → Reordering can be achieved with less overhead



weight = number of times the pair of qubit lines appear as the target and control qubit of any quantum gate in QBC



QUBIT LINE ADJACENCY GRAPH

vertex: input qubit line

Fig. 4. Graph formation: (a) is the TOFFOLI network having NOT, CNOT & TOFFOLI gates, (b) is the weighted graph formed from this circuit

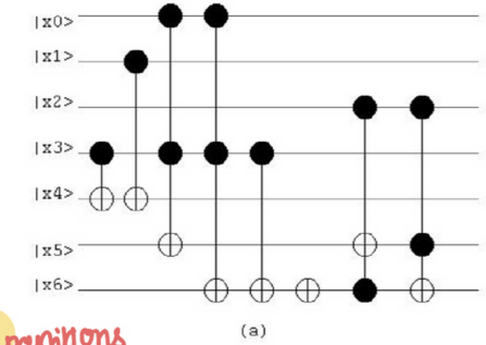
problem of reordering input qubit lines in QBC → reduced to → finding linear ordering $\beta: V \to \{1, 2, \ldots, |V|\}$ such that $\sum_{e=(u,v)\in E} |\beta(u) - \beta(v)|$ is minimum

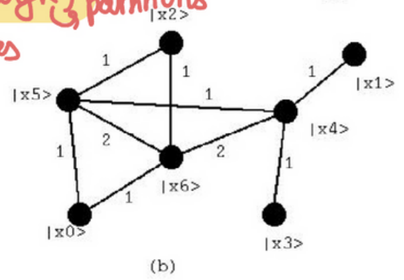
✓ heuristic for getting linear ordering by listing the leaves of a binary tree representing recursive balanced bi-partitioning of graph

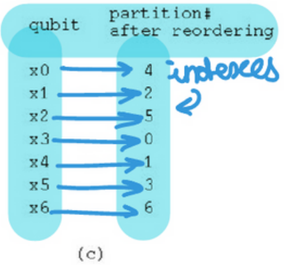
linear ordering for partitions ↑ output

pmetis → graph partitioning

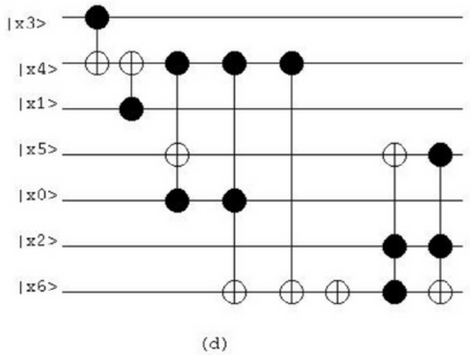
$O((m+n) * \log k)$ → partitions, edges, nodes







indexes



new circuit only requires 10 SWAP pairs

Fig. 5. Synthesis of the circuit *4mod5-bdd_287*. (a) the original circuit which needs 15 SWAP pairs for LNN architecture, (b) the graph formed from reordered circuit in (d), (c) the ordering of qubit lines after partitioning, (d) is the reconstructed circuit from new ordering, with only 10 SWAP pairs for LNN architecture

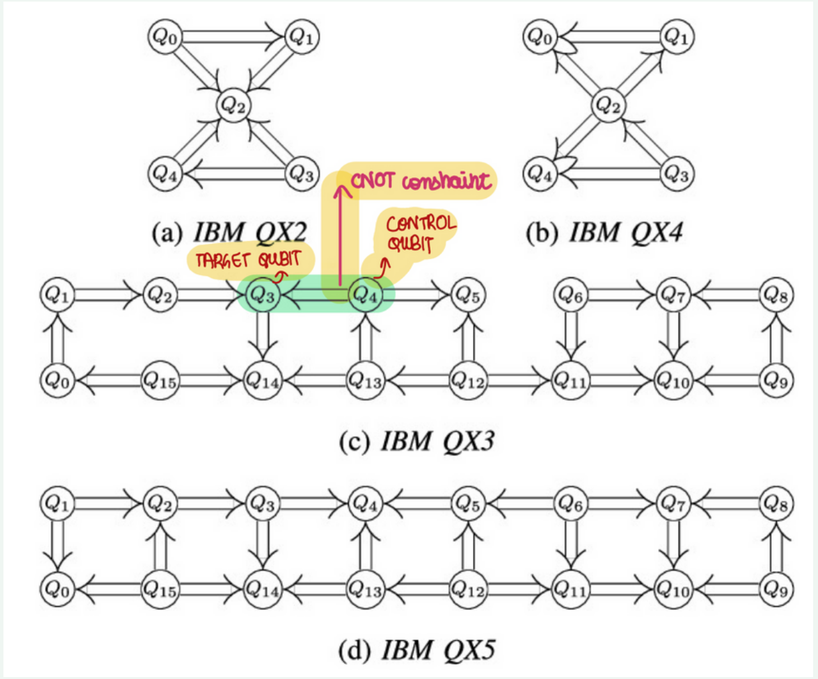
Efficient Mapping of Quantum Circuits to the IBM QX Architectures

An approach which satisfies all the underlying physical constraints given by the architecture and, at the same time, aims to keep the overhead in terms of additionally required quantum gates minimal.

Physical limitations, namely that certain quantum operations can be applied to selected physical qubits of the IBM QX architectures only. Consequently, the **logical qubits of a quantum circuit have to be mapped to the physical qubits of the quantum computer** such that all operations can be conducted

The IBM QX architectures support the elementary single qubit operation $\mathbf{U(\theta,\varphi,\lambda) = Rz(\varphi)Ry(\theta)Rz(\lambda)}$ that is composed by two rotations around the z-axis and one rotation around the y-axis, as well as the **CNOT** operation.

Besides **decomposing all non-elementary quantum operations** (e.g. Toffoli gate or SWAP gate) to the elementary operations U(θ,φ,λ) and CNOT, further constraints have to be satisfied. These **restrictions apply to CNOT gates and are given by the so-called coupling-map**



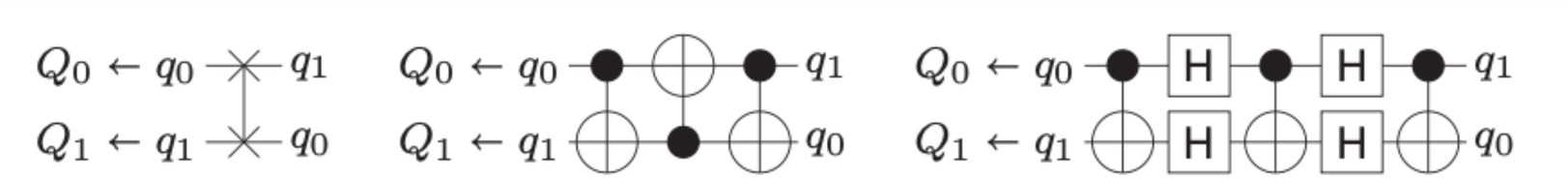
STEP 1.

All gates of the given quantum circuit to be mapped have to be **decomposed to elementary operations supported by the hardware**, i.e. CNOTs and parameterized U gates. This has already intensely been considered in the past

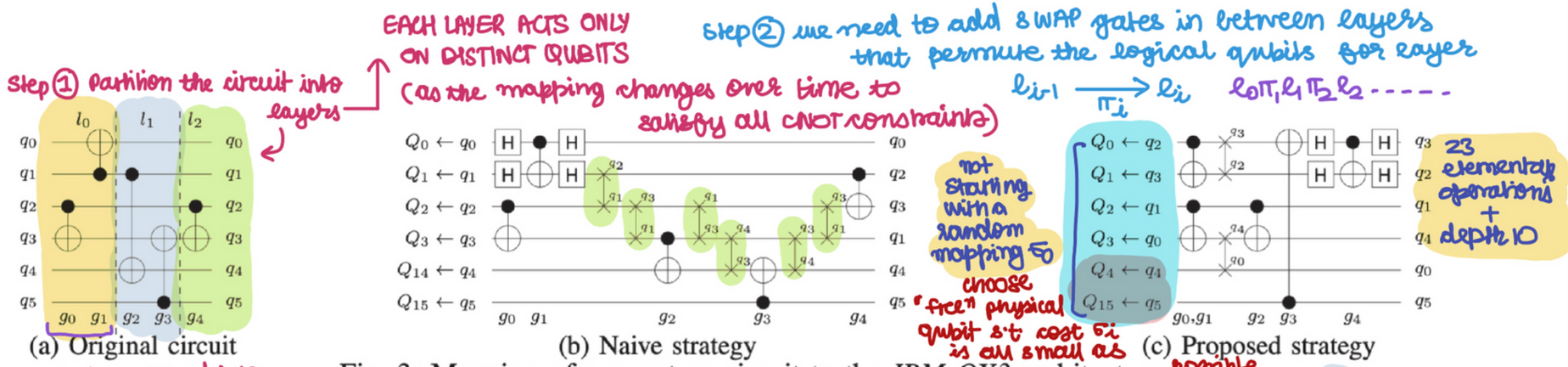
STEP 2.

The **n logical qubits** q0, q1, . . . qn−1 of that quantum circuit have to be mapped to the **m physical qubits** Q0, Q1,...Qm−1 (m = 5 for QX2 and m = 16 for QX3) of the IBM QX architecture such that **all CNOT constraints are satisfied.**

Previous works are not applicable here since they consider more simplistic architectures where a two-qubit gate can be applied to any adjacent qubits. The **CNOT constraints to be satisfied for the IBM QX architectures are much stricter with respect to that and also what physical qubit may act as control and as target qubit**.

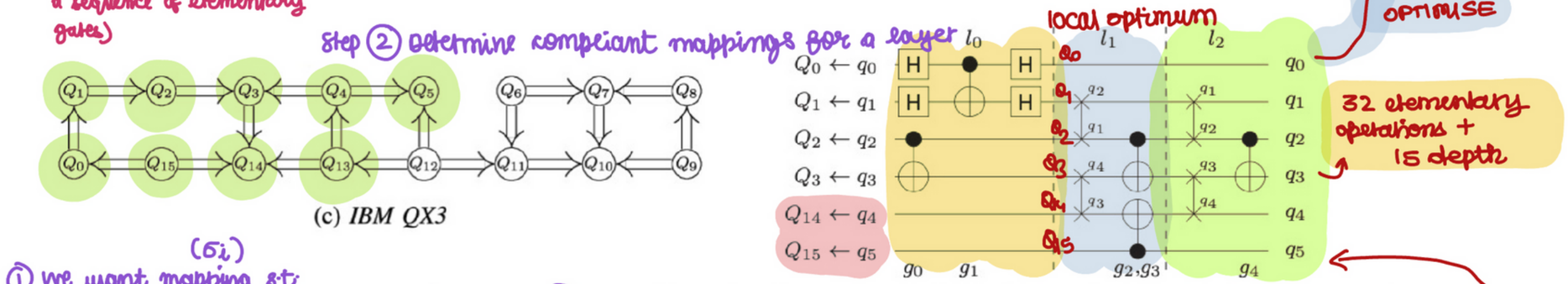


Experimental evaluations show that the proposed approach clearly outperforms IBM's own mapping solution with respect to runtime as well as resulting costs.



(a) Original circuit (b) Naive strategy (c) Proposed strategy

Fig. 3: Mapping of a quantum circuit to the *IBM QX3* architecture



(c) *IBM QX3*

Fig. 4: Circuit resulting from locally optimal mappings

(σi)
① we want mapping s.t
① CNOT constraints satisfied
② it should have min distance from previous mapping σi-1

πi should have fewest elementary operations

worst case → m!/(m-n)! possibilities (exponential complexity)

cope

A* search algorithm → optimal solution
guided by a heuristic (which considers real distance b/w qubits through coupling map)
start with σi-1
SWAP gates → σi
to satisfy CNOT constraints

Global optimum → look ahead scheme

incorporate the cost of πi+1 to the cost function of A* search → not desirable

we want locally sub optimal solutions

OUTPERFORMS IBM's own mapping solution!

Not All SWAPs Have the Same Cost: A Case for Optimization-Aware Qubit Routing

1. **Qubit connectivity limitation remains to be a critical challenge.**

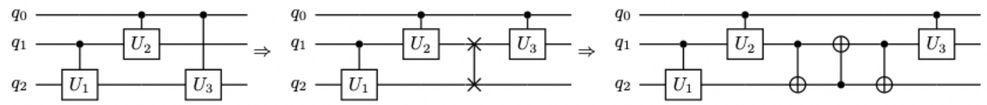
2. During the **qubit routing step,** the compiler inserts SWAP gates and performs circuit transformations. Given the connectivity topology of the target hardware, there are typically **multiple qubit routing candidates. The state-of-the-art compilers use a cost function to evaluate the number of SWAP gates for different routes and then select the one with the minimum number of SWAP gates.**
After qubit routing, the quantum compiler **performs gate optimizations** upon the circuit with the newly inserted SWAP gates.

We find that with the consideration of gate optimizations, not all of the SWAP gates have the same basis-gate cost.

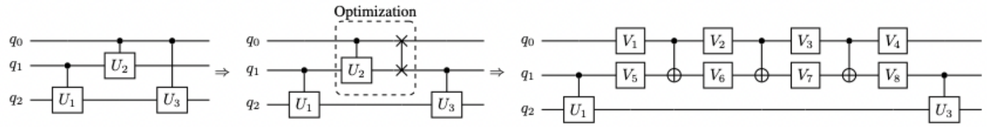
NASSC (**Not All Swaps have the Same Cost**)**.** NASSC is the first algorithm that **considers the subsequent optimizations during the routing step**. Our optimization-aware qubit routing leads to better routing decisions and benefits subsequent optimizations. We also propose a new **optimization-aware decomposition for the inserted SWAP gates.**

To perform the CNOT gate between q0 and q2, we need to insert SWAP gates and there are two options: either insert a SWAP gate between (q0, q1) or between (q1, q2). If we only consider the SWAP- gate count at the routing step, both routing options have the same SWAP gate cost as one SWAP. Therefore, the **compiler may randomly select between these two designs**.

However, **if we consider the subsequent optimizations that would re-synthesize the consecutive two-qubit gates, these two routes actually have different costs in the number of CNOT gates.**



(a) SWAP insertion resulting in three extra CNOT gates



(b) SWAP insertion resulting in one extra CNOT gate

The following self-inverse gates are considered: H, X, Y, Z, CX, CY, and CZ.

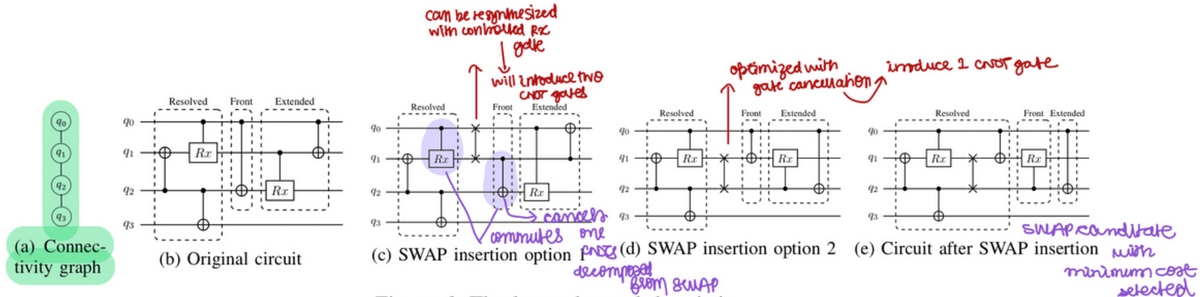


Figure 6: The layered search heuristic.

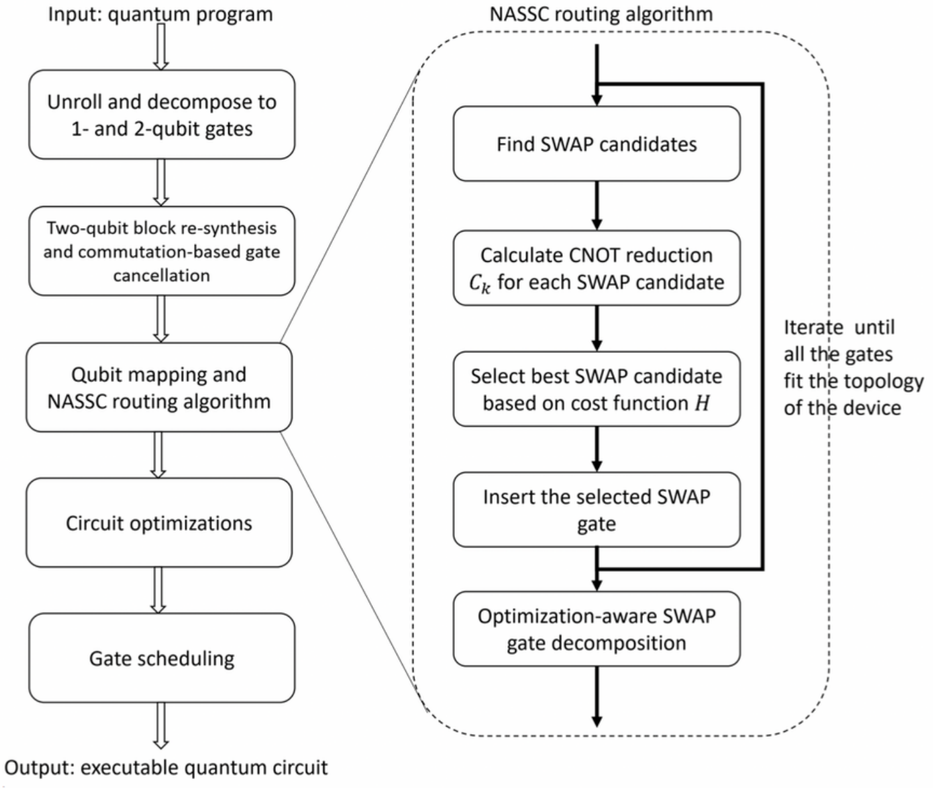


Figure 5: The compilation flow of NASSC integrated with Qiskit

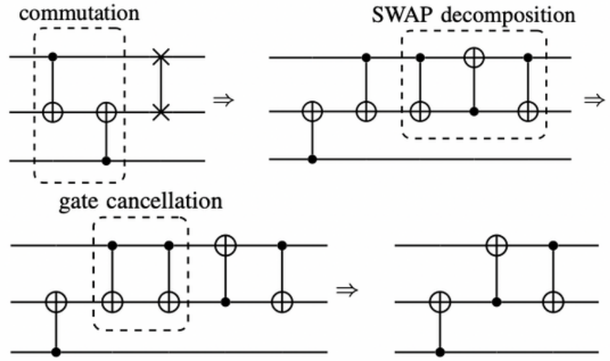


Figure 4: SWAP gate optimization with gate commutation and cancellation.