1. Mapping problem to BQM (binary quadric model)

A schedule has to satisfy 3 constraints:

1. Machine qualification: a job cannot be allocated to a machine *j* that is not qualified for this job.
2. Non-splitting allocation: a given job cannot be split and executed on two different machines.
3. Non-preemption: a job cannot be preempted. When started, the job must be executed until its completion.

This can be formulated as the following:

$i : jobs \ \ (i \in I)$

j : machines $(j \in M)$

$I : the\ set\ of\ jobs$

$M : a\ set\ of\ \ machines$

$Mi : a\ set\ of\ alternative\ machines\ on\ which\ job\ i\ can\ be\ processed$

$p_{ij}: the\ processing\ time\ of\ job\ i\ on\ machine\ j$

$w_j: the\ welcome\ rate\ for\ machine\ j$

Define a binary variable $q_{i,j}$, taking value 1 if job *i* is processed on machine *j*, otherwise 0

$$0 = \left(\sum_{j \in \mathrm{M}i} q_{ij} - 1 \right)^2 \qquad (1)$$

$$\text{cost}\ (j) = \sum_i q_{ij} \times p_{ij} \qquad (2)$$

$$C_{max} \geqslant \text{cost}\ (j) \qquad (3)$$

Constraint (1) ensure each job can be performed only on one machine. Constraint (2) determines the total processing time on machine *j*. Constraint (3) determines the make-span.

After written all of the components (objective and constraints) as BQM expressions, defined the final BQM by adding all of the components together.

$$BQM = min(C_{max} + \lambda \sum_i \left(\sum_j q_{ij} - 1 \right)^2) \qquad (4)$$

2. Building QUBO matrix in Python

Now the cost function can be formulated as a quadratic, upper-triangular matrix, as required for the QUBO problem. We keep a mapping of binary variable $q_{i,j}$ to index in the QUBO matrix Q, given by $I(i,j)$. These indices are the diagonals of the QUBO matrix.

Firstly, we define a matrix Q with binary variable $q_{i,j}$, add the constraint to enforce that each job can be performed only on one machine, as per Constraint (1):

1. For every job i with alternative machine j, add($-\lambda$) to the diagonal of Q given by index $I\,(i, j)$.
2. For every cross-term arising from Constraint (1), add (2λ) to the corresponding off-diagonal.

The full code for Constraint (1) is shown below:

```python
# Machine qualification: job cannot be allocated to a machine j that is not qualified for this job
lam = job_num*machine_number
for i in range(0, job_num):
    machine_list_for_single_job = [m[0] for m in problem[i]]
    for j in range(0, machine_number):
        if j not in machine_list_for_single_job:
            Q.update({(i * machine_number + j, i * machine_number + j): lam * 2})

# Constraint (1): each job can be performed only on one machine
for i in range(job_num):
    for j in range(machine_number):
        # For every cross-term arising from Constraint (1), add (2λ) to the corresponding off-diagonal
        for k in range(j + 1, machine_number):
            Q.update({(i * machine_number + j, i * machine_number + k): lam * 2})
        # For every job i with alternative machine j, add(-λ) to the diagonal of Q given by index I (i, j).
        Q.update({(i * machine_number + j, i * machine_number + j): (
                    Q[(i * machine_number + j, i * machine_number + j)] - lam)})
```

Secondly, we then add (w_j*0.2+$p_{i,j}$) at diagonal index $I\,(i, j)$ for every job proposed with machine j. The code is shown below:

```python
#  Equation (4)  # add (w_j*0.2+p_{i,j})at diagonal index I (i, j) for every job proposed with machine j
machine_welcome_factor = cal_welcome_factor(problem, machine_number)
for i in range(0, job_num):
    machine_list_for_single_job = [m[0] for m in problem[i]]
    for j in range(0, machine_number):
        if j in machine_list_for_single_job:
            processing_time = find_job_processing_time(problem,i,j)
            Q.update({(i * machine_number + j, i * machine_number + j):
                          Q[(i * machine_number + j, i * machine_number + j)]+
                          machine_welcome_factor[j]*0.2+processing_time})
```

After defined the Q matrix, we try to solve Qubo problem on D-wave's advanced 4000 QPU. The matrix below here is the result for example problem (10 job*10 machine).

```
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 0, 0, 1, 0, 0, 0, 0, 0]
[1, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 1, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 1, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 1]
```

Figure 1 result matrix for 10job*10machine problem

3. Result visualization

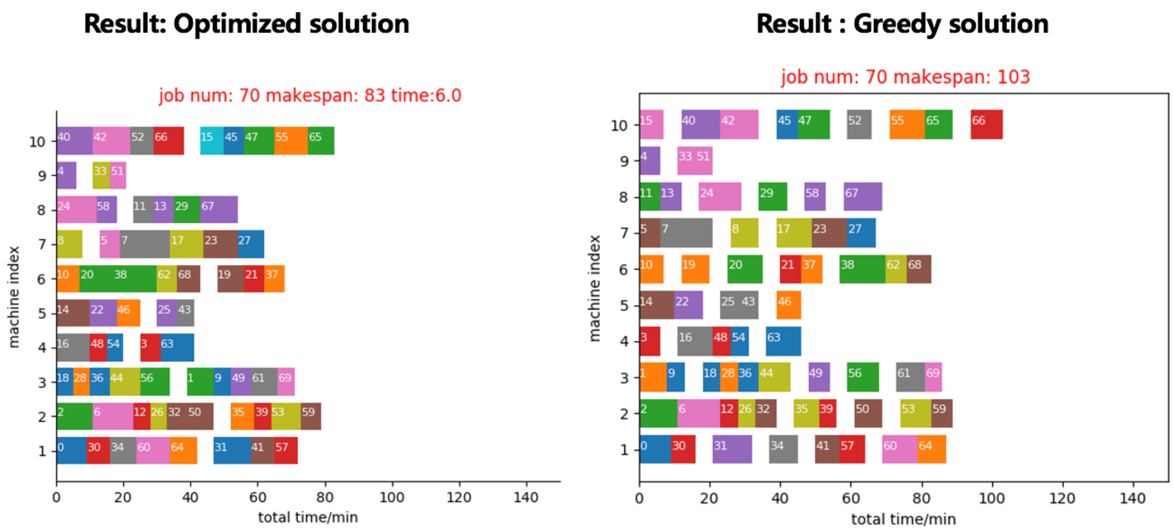

Figure 2 optimized solution vs greedy solution

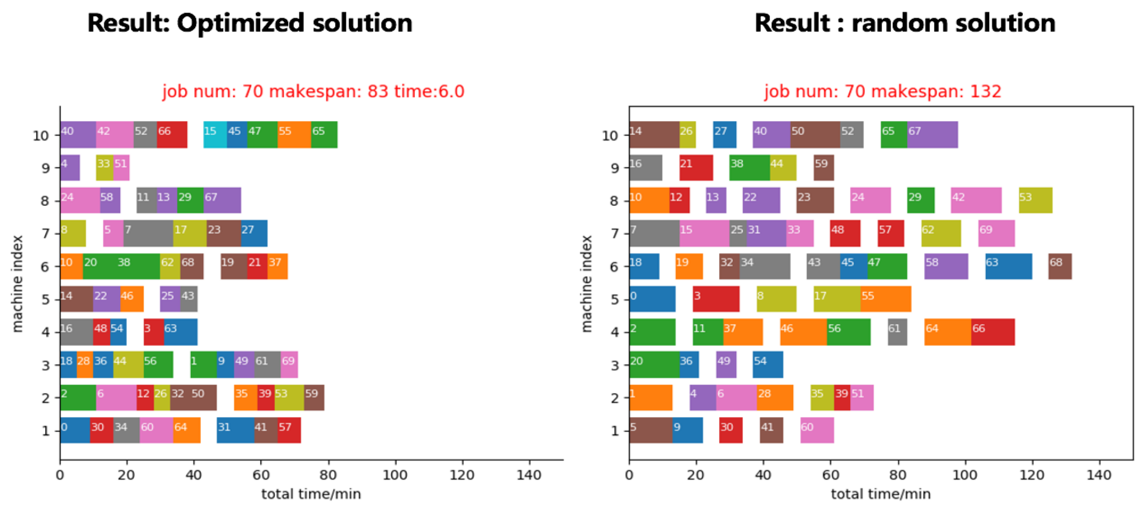

Figure 3optismized solution vs random solution