Sign up

microsoft / **qio-samples** Public

**Code** Issues Pull requests 1 Actions Projects Wiki Security Insights

main

**qio-samples** / samples / job-shop-scheduling / **job-shop-sample.ipynb**

**anraman** Added plotly install command ✓

**5** contributors

```
1738 lines (1738 sloc) | 93 KB
```

# Job Shop Scheduling Sample

## Introduction

Job shop scheduling is a common and important problem in many industries. For example, in the automobile industry manufacturing a car involves many different types of operations which are performed by a number of specialized machines - optimizing the production line to minimize manufacturing time can make for significant cost savings.

The job shop scheduling problem is defined as follows: you have a set of jobs ( $J_0, J_1, J_2, \dots, J_{a-1}, \text{where } a \text{ is the total number of jobs}$), which have various processing times and need to be processed using a set of machines ( $m_0, m_1, m_2, \dots, m_{b-1}, \text{where } b \text{ is the total number of machines}$). The goal is to complete all jobs in the shortest time possible. This is called minimizing the **makespan**.

Each job consists of a set of operations, and the operations must be performed in the correct order to complete that job.

In this sample, we'll introduce the necessary concepts and tools for describing this problem in terms of a penalty model, and then solve an example problem using the Azure Quantum Optimization service.

Imagine, for example, that you have a to-do list. Each item on the list is a **job** using this new terminology.

Each job in this list consists of a set of operations, and each operation has a processing time. You also have some tools at hand that you can use to complete these jobs (the **machines**).

TODOs:

- Pay electricity bill
  1. Log in to site (*2 minutes*) - **computer**
  2. Pay bill (*1 minute*) - **computer**
  3. Print receipt (*3 minutes*) - **printer**

- Plan camping trip
  1. Pick campsite (*2 minutes*) - **computer**
  2. Pay online (*2 minutes*) - **computer**
  3. Print receipt (*3 minutes*) - **printer**

- Book dentist appointment
  1. Choose time (*1 minute*) - **computer**
  2. Pay online (*2 minutes*) - **computer**
  3. Print receipt (*3 minutes*) - **printer**
  4. Guiltily floss your teeth (*2 minutes*) - **tooth floss**

4. Quickly floss your teeth (2 *minutes*) - **tooth floss**

But there are some constraints:

1. Each of the tasks (**operations**) in a todo (**job**) must take place in order. You can't print the receipt before you have made the payment! This is called a **precedence constraint**.
2. You start an operation only once, and once started it must be completed before you do anything else. There's no time for procrastination! This is called the **operation-once constraint**.
3. Each tool (**machine**) can only do one thing at a time. You can't simultaneously print two receipts unless you invest in multiple printers. This is the **no-overlap constraint**.

## Cost functions

The rest of this sample will be spent constructing what is known as a **cost function**, which is used to represent the problem. This cost function is what will be submitted to the Azure Quantum Optimization solver.

> **NOTE**: If you have completed the Microsoft Quantum Learn Module Solve optimization problems by using quantum-inspired optimization, this concept should already be familiar. A simplified version of this job shop sample is also available here on MS Learn.

Each point on a cost function represents a different solution configuration - in this case, each configuration is a particular assignment of starting times for the operations you are looking to schedule. The goal of the optimization is to minimize the cost of the solution - in this instance the aim is to minimize the amount of time taken to complete all operations.

Before you can submit the problem to the Azure Quantum solvers, you'll need to transform it to a representation that the solvers can work with. This is done by creating an array of `Term` objects, representing the problem constraints. Positive terms penalize certain solution configurations, while negative ones support them. By adding penalties to terms that break the constraints, you increase the relative cost of those configurations and reduce the likelihood that the optimizer will settle for these suboptimal solutions.

The idea is to make these invalid solutions so expensive that the solver can easily locate valid, low-cost solutions by navigating to low points (minima) in the cost function. However, you must also ensure that these solutions are not so expensive as to create peaks in the cost function that are so high that the solver can't travel over them to discover better optima on the other side.

## Python setup

To visualize the results later on in this sample, you will need to install the `plotly.express` module, using (for example) `pip` (shown below) or `conda`.

> **NOTE**: Once you run the cell below, please restart the Jupyter kernel.

In [ ]:
```
!pip install plotly.express
```

## Azure Quantum setup

The Azure Quantum Optimization service is exposed via a Python SDK, which you will be making use of during the rest of this sample. This means that before you get started with formulating the problem, you first need to import some Python modules and set up an Azure Quantum `Workspace`.

You will need to enter your Azure Quantum workspace details in the cell below before you run it:

In [ ]:
```
# This allows you to connect to the Workspace you've previously depl
# Be sure to fill in the settings below which can be retrieved by ru
from azure.quantum import Workspace

workspace = Workspace (
    subscription_id = "",
    resource_group = "",
    name = "",
    location = ""
)
```

## Problem formulation

Now that you have set up your development environment, you can start to formulate the problem.

The first step is to take the constraints identified above and formulate them as mathematical equations that you can work with.

Let's first introduce some notation because we are lazy and also want to avoid carpal tunnel syndrome.

Let's stick with the previous example of the todo list:

- $J_0$: Pay electricity bill
  - $O_0$: Log in to site (2 *minutes*) - **computer**
  - $O_1$: Pay bill (1 *minute*) - **computer**
  - $O_2$: Print receipt (3 *minutes*) - **printer**
- $J_1$: Plan camping trip
  - $O_3$: Pick campsite (2 *minutes*) - **computer**
  - $O_4$: Pay online (2 *minutes*) - **computer**
  - $O_5$: Print receipt (3 *minutes*) - **printer**
- $J_2$: Book dentist appointment
  - $O_6$: Choose time (1 *minute*) - **computer**
  - $O_7$: Pay online (2 minutes) - computer

- $O_7$: Pay online (*2 minutes*) - **computer**
- $O_8$: Print receipt (*3 minutes*) - **printer**
- $O_9$: Guiltily floss your teeth (*2 minutes*) - **tooth floss**

Above, you can see that the jobs have been labeled as $J$ and assigned index numbers $0$, $1$ and $2$, to represent each of the three tasks you have. The operations that make up each job have also been defined, and are represented by the letter $O$.

To make it easier to code up later, all operations are identified with a continuous index number rather than, for example, starting from $0$ for each job. This allows you to keep track of operations by their ID numbers in the code and schedule them according to the constraints and machine availability. You can tie the operations back to their jobs later on using a reference.

Below, you see how these definitions combine to give us a mathematical formulation for the jobs:

$$\begin{align} J_0 &= \{O_0, O_1, O_2\} \tag{1}\\ J_1 &= \{O_3, O_4, O_5\} \tag{2}\\ J_2 &= \{O_6, O_7, O_8, O_9\} \tag{3} \end{align}$$

**More generally:**

$$J_0 = \{O_0, O_1, \ldots, O_{k_0-1}\}, \text{ where } k_0 = n_0, \text{ the number of opera}$$

$$J_1 = \{O_{k_0}, O_{k_0+1}, \ldots, O_{k_1-1}\}, \text{ where } k_1 = n_0 + n_1, \text{ the numbe}$$

$$\vdots$$

$$J_{n-1} = \{O_{k_{n-2}}, O_{k_{n-2}+1}, \ldots, O_{k_{n-1}-1}\}, \text{ where } k_{n-1} = \text{ the total nu}$$

The next piece of notation you will need is a binary variable, which will be called $x_{i,t}$.

You will use this variable to represent whether an operation starts at time $t$ or not:

$$\begin{align} &\text{If } x_{i,t} = 1,\ O_i \text{ starts at time } t \tag{11}\\ &\text{If } x_{i,t} = 0,\ O_i \text{ does not start at time } t \tag{12} \end{align}$$

Because $x_{i,t}$ can take the value of either $0$ or $1$, this is known as a binary optimization problem. More generally, this is called a polynomial unconstrained binary optimization (or PUBO) problem. You may also see these PUBO problems referred to as Higher Order Binomial Optimization (HOBO) problems - these terms both refer to the same thing.

$t$ is used to represent the time. It goes from time $0$ to $T - 1$ in integer steps. $T$ is the latest time an operation can be scheduled:

$$0 \leq t < T$$

Lastly, $p_i$ is defined to be the processing time for operation $i$ - the amount of time it takes for operation $i$ ($O_i$) to complete:

$$\text{If } O_i \text{ starts at time } t\text{, it will finish at time } t + p_i$$

$$\text{If } O_{i+1} \text{ starts at time } s\text{, it will finish at time } s + p_{i+1}$$

Now that the terms have been defined, you can move on to formulating the problem.

The first step is to represent the constraints mathematically. This will be done using a penalty model - every time the optimizer explores a solution that violates one or more constraints, you need to give that solution a penalty:

| Constraint | Penalty condition |
|---|---|
| **Precedence constraint** Operations in a job must take place in order. | Assign penalty every time $O_{i+1}$ starts before $O_i$ has finished (they start out of order). |
| **Operation-once constraint** Each operation is started once and only once. | Assign penalty if an operation isn't scheduled within the allowed time. **Assumption:** if an operation starts, it runs to completion. |
| **No-overlap constraint** Machines can only do one thing at a time. | Assign penalty every time two operations on a single machine are scheduled to run at the same time. |

You will also need to define an objective function, which will minimize the time taken to complete all operations (the **makespan**).

## Expressing a cost function using the Azure Quantum Optimization SDK

As you will see during the exploration of the cost function and its constituent penalty terms below, the overall cost function is quadratic (because the highest order polynomial term you have is squared). This makes this problem a **Quadratic Unconstrained Binary Optimization (QUBO)** problem, which is a specific subset of **Polynomial Unconstrained Binary Optimization (PUBO)** problems (which allow for higher-order polynomial terms than quadratic). Fortunately, the Azure Quantum Optimization service is set up to accept PUBO (and Ising) problems, which means you don't need to modify the above representation to fit the solver.

As introduced above, the binary variables over which you are optimizing are the operation starting times $x_{i,t}$. Instead of using two separate indices as in the mathematical formulation, you will need to define a singly-indexed binary variable $x_{i \cdot T+t}$. Given time steps $t \in [0, T-1]$, every operation $i$ contributes $T$ indices. The operation starts at the value of $t$ for which $x_{i \cdot T+t}$ equals 1.

In order to submit a problem to the Azure Quantum services, you will first be creating a `Problem` instance. This is a Python object that stores all the required information, such as the cost function details and what kind of problem we are modeling.

To represent cost functions, we'll make use of a formulation using `Term` objects. Ultimately, any polynomial cost function can be written as a simple sum of products.

That is, the function can be rewritten to have the following form, where $p_k$ indicates a product over the problem variables $x_0, x_1, \ldots$:

$$H(x) = \sum_k \alpha_k \cdot p_k(x_0, x_1, \ldots)$$

$$\text{e.g. } H(x) = 5 \cdot (x_0) + 2 \cdot (x_1 \cdot x_2) - 3 \cdot ({x_3}^2)$$

In this form, every term in the sum has a coefficient $\alpha_k$ and a product $p_k$. In the `Problem` instance, each term in the sum is represented by a `Term` object, with parameters `c` - corresponding to the coefficient, and `indices` - corresponding to the product. Specifically, the `indices` parameter is populated with the indices of all variables appearing in the term. For instance, the term $2 \cdot (x_1 \cdot x_2)$ translates to the following object: `Term(c=2, indices=[1,2])`.

More generally, `Term` objects take on the following form:

```
Term(c: float, indices: []) # Constant terms like +1
Term(c: float, indices: [int]) # Linear terms like x
Term(c: float, indices: [int, int]) # Quadratic terms like
x^2 or xy
```

If there were higher order terms (cubed, for example), you would just add more elements to the indices array, like so:

```
Term(c: float, indices: [int, int, int, ...])
```

## Defining problem parameters in code

Now that you've defined the problem parameters mathematically, you can transform this information to code. The following two code snippets show how this is done.

First, the helper function `process_config` is defined:

In [ ]:

```
def process_config(jobs_ops_map: dict, machines_ops_map: dict, proce
    """
    Process & validate problem parameters (config) and generate inve

    Keyword arguments:

    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    machines_ops_map(dict): Mapping of operations to machines, e.g.:
        machines_ops_map = {
            0: [0,1],          # Operations 0 & 1 assigned to machin
            1: [2,3]           # Operations 2 & 3 assigned to machin
        }
    processing_time (dict): Operation processing times
    T (int): Allowed time (jobs can only be scheduled below this lim
    """

    # Problem cannot take longer to complete than all operations exe
    ## Sum all operation processing times to calculate the maximum m
    T = min(sum(processing_time.values()), T)

    # Ensure operation assignments to machines are sorted in ascendi
    for m, ops in machines_ops_map.items():
        machines_ops_map[m] = sorted(ops)
    ops_jobs_map = {}

    for job, ops in jobs_ops_map.items():
        # Fail if operation IDs within a job are out of order
        assert (ops == sorted(ops)), f"Operation IDs within a job mu

        for op in ops:
            # Fail if there are duplicate operation IDs
            assert (op not in ops_jobs_map.keys()), f"Operation IDs
            ops_jobs_map[op] = job

    return ops_jobs_map, T
```

Below, you can see the code representation of the problem parameters: the maximum allowed time `T`, the operation processing times `processing_time`, the mapping of operations to jobs (`jobs_ops_map` and `ops_jobs_map`), and the assignment of operations to machines (`machines_ops_map`).

In [ ]:

```
# Set problem parameters
## Allowed time (jobs can only be scheduled below this limit)
T = 21

## Processing time for each operation
processing_time = {0: 2, 1: 1, 2: 3, 3: 2, 4: 2, 5: 3, 6: 1, 7: 2, 8

## Assignment of operations to jobs (job ID: [operation IDs])
### Operation IDs within a job must be in ascending order
jobs_ops_map = {
    0: [0, 1, 2],    # Pay electricity bill
    1: [3, 4, 5],    # Plan camping trip
    2: [6, 7, 8, 9] # Book dentist appointment
}

## Assignment of operations to machines
### Ten jobs, three machines
machines_ops_map = {
    0: [0, 1, 3, 4, 6, 7],  # Operations 0, 1, 3, 4, 6 and 7 are ass
    1: [2, 5, 8],           # Operations 2, 5 and 8 are assigned to
    2: [9]                  # Operation 9 is assigned to machine 2 (
}

## Inverse mapping of jobs to operations
ops_jobs_map, T = process_config(jobs_ops_map, machines_ops_map, pro
```

In the next sections, you will construct mathematical representations of the penalty terms and use these to build the cost function, which will be of the format:

$$H(x) = \alpha \cdot f(x) + \beta \cdot g(x) + \gamma \cdot h(x) + \delta \cdot k(x)$$

where:

$$f(x), g(x) \text{ and } h(x) \text{ represent the penalty functions.}$$

$$k(x) \text{ represents the objective function.}$$

$$\alpha, \beta, \gamma \text{ and } \delta \text{ represent the different weights assigned to the penalties.}$$

The weights represent how important each penalty function is, relative to all the others. In the following units, you will learn how to build these penalty and objective functions, combine them to form the cost function $H(x)$, and solve the problem using Azure Quantum. Over the rest of this sample, you will learn how to build these penalty and objective functions, combine them to form the cost function $H(x)$, and solve the problem using Azure Quantum.

To do this, you will explore how to formulate each of these constraints mathematically, and how this translates to code.

## Precedence constraint

The precedence constraint is defined as follows:

| Constraint | Penalty condition |
|---|---|
| **Precedence constraint** Operations in a job must take place in order. | Assign penalty every time $O_{i+1}$ starts before $O_i$ has finished (they start out of order). |

### Worked Example

Let's take job 1 ($J_1$) as an example:

- $J_1$: Plan camping trip
    - $O_3$: Pick campsite (*2 minutes*)
    - $O_4$: Pay online (*2 minutes*)
    - $O_5$: Print receipt (*3 minutes*)

Let's formulate the penalty conditions for $O_3$ and $O_4$: you want to add a penalty if $O_4$ starts before $O_3$ finishes. First, you'll define our terms and set some of their values:

$$\text{Total simulation time } T = 4$$

$$O_3 \text{ processing time: } p_3 = 2$$

$$O_3 \text{ starts at time } t \text{, and finishes at time } t + p_3$$

$$O_3 \text{ starts at any time } 0 \leq t < T$$

$$O_4 \text{ can start at time } s \geq t + p_3$$

$O_3$'s finishing time is given by adding its processing time $p_3$ (which we've set to be 2) to its start time $t$. You can see the start and end times for $O_3$ in the table below:

| $t$ | $t = p_3$ |
|---|---|
| 0 | 2 |
| 1 | 3 |
| 2 | 4 |

To avoid violating this constraint, the start time of $O_4$ (denoted by $s$) must be greater than or equal to the end time of $O_3$, like we see in the next column:

| $t$ | $t = p_3$ | $s \geq t + p_3$ |
|---|---|---|
| 0 | 2 | 2, 3, 4 |
| 1 | 3 | 3, 4 |
| 2 | 4 | 4 |
| | **Valid configuration?** | ✔ |

The ✔ means that any $s$ value in this column is valid, as it doesn't violate the precedence constraint.

Conversely, if $s$ is less than $t + p_3$ (meaning $O_4$ starts before $O_3$ finishes), you need to add a penalty. Invalid $s$ values for this example are shown in the rightmost column:

| $t$ | $t = p_3$ | $s \geq t + p_3$ | $s < t + p_3$ |
|---|---|---|---|
| 0 | 2 | 2, 3, 4 | 0, 1 |
| 1 | 3 | 3, 4 | 0, 1, 2 |
| 2 | 4 | 4 | 0, 1, 2, 3 |
| | **Valid configuration?** | ✔ | ✘ |

In the table above, ✘ has been used to denote that any $s$ value in the last column is invalid, as it violates the precedence constraint.

## Penalty Formulation

This is formulated as a penalty by counting every time consecutive operations $O_i$ and $O_{i+1}$ in a job take place out of order.

As you saw above: for an operation $O_i$, if the start time of $O_{i+1}$ (denoted by $s$) is less than the start time of $O_i$ (denoted by $t$) plus its processing time $p_i$, then that counts as a penalty. Mathematically, this penalty condition looks like: $s < t + p_i$.

You sum that penalty over all the operations of a job ($J_n$) for all the jobs:

$$f(x) = \sum_{k_{n-1} \leq i < k_n, s < t + p_i} x_{i,t} \cdot x_{i+1,s} \text{ for each job } n.$$

Let's break that down:

- $k_{n-1} \leq i < k_n$

  This means you sum over all operations for a single job.

- $s < t + p_i$

  This is the penalty condition - any operation that satisfies this condition is in violation of the precedence constraint.

- $x_{i,t} \cdot x_{i+1,s}$

  This represents the table you saw in the example above, where $t$ is allowed to vary from $0 \to T-1$ and you assign a penalty whenever the constraint is violated (when $s < t + p_i$).

This translates to a nested `for` loop: the outer loop has limits $0 \leq t < T$ and the inner loop has limits $0 \leq s < t + p_i$

## Code

Using the mathematical formulation and the breakdown above, you can now translate this constraint function to code. You will see the `weight` argument included in this code snippet - this will be assigned a value later on when you call the function:

In [ ]:
```python
from azure.quantum.optimization import Term

"""
# Reminder of the relevant parameters
## Time to allow for all jobs to complete
T = 21

## Processing time for each operation
processing_time = {0: 2, 1: 1, 2: 3, 3: 2, 4: 2, 5: 3, 6: 1, 7: 2, 8

## Assignment of operations to jobs (job ID: [operation IDs])
### Operation IDs within a job must be in ascending order
jobs_ops_map = {
    0: [0, 1, 2],   # Pay electricity bill
    1: [3, 4, 5],   # Plan camping trip
    2: [6, 7, 8, 9] # Book dentist appointment
}
"""

def precedence_constraint(jobs_ops_map:dict, T:int, processing_time:
    """
    Construct penalty terms for the precedence constraint.

    Keyword arguments:

    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    T (int): Allowed time (jobs can only be scheduled below this lim
    processing_time (dict): Operation processing times
    weight (float): Relative importance of this constraint
    """

    terms = []

    # Loop through all jobs:
    for ops in jobs_ops_map.values():
        # Loop through all operations in this job:
        for i in range(len(ops) - 1):
            for t in range(0, T):
                # Loop over times that would violate the constraint:
                for s in range(0, min(t + processing_time[ops[i]], T
                    # Assign penalty
                    terms.append(Term(c=weight, indices=[ops[i]*T+t,

    return terms
```

> **NOTE**: This nested loop structure is probably not the most efficient

> way to do this but it is the most direct comparison to the mathematical formulation.

## Operation-once constraint

The operation-once constraint is defined as follows:

| Constraint | Penalty condition |
|---|---|
| **Operation-once constraint** Each operation is started once and only once. | Assign penalty if an operation isn't scheduled within the allowed time. **Assumption:** if an operation starts, it runs to completion. |

### Worked Example

We will again take job 1 ($J_1$) as an example:

- $J_1$: Plan camping trip
  - $O_3$: Pick campsite (*2 minutes*)
  - $O_4$: Pay online (*2 minutes*)
  - $O_5$: Print receipt (*3 minutes*)

Recall the variable $x_{i,t}$:

$$\text{If } x_{i,t} = 1,\ O_i \text{ starts at time } t \tag{13}$$
$$\text{If } x_{i,t} = 0,\ O_i \text{ does not start at time } t \tag{14}$$

According to this constraint, $x_{i,t}$ for a specific operation should equal 1 **once and only once** from $t = 0 \rightarrow T - 1$ (because it should start once and only once during the allowed time).

So in this case, you need to assign a penalty if the sum of $x_{i,t}$ for each operation across all allowed times doesn't equal exactly 1.

Let's take $O_3$ as an example again:

| $t$ | $x_{3,t}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 0 |
| $\sum_t x_{3,t} =$ | 1 |
| **Valid configuration?** | ✔ |

In the right hand column, you see that $O_3$ starts at time 1 and no other time ( $x_{3,t} = 1$ at time $t = 1$ and is $0$ otherwise). The sum of $x_{i,t}$ values over all $t$ for this example is therefore 1, which is what is expected! This is therefore a valid solution.

In the example below, you see an instance where $O_3$ is scheduled more than once ( $x_{3,t} = 1$ more than once), in violation of the constraint:

| $t$ | $x_{3,t}$ |
|---|---|
| 0 | 0 |
| 1 | 1 |
| 2 | 1 |
| $\sum_t x_{3,t} =$ | 2 |
| **Valid configuration?** | ✘ |

You can see from the above that $O_3$ has been scheduled to start at both time 1 and time 2, so the sum of $x_{i,t}$ values over all $t$ is now greater than 1. This violates the constraint and thus you must apply a penalty.

In the last example, you see an instance where $O_3$ has not been scheduled at all:

| $t$ | $x_{3,t}$ |
|---|---|
| 0 | 0 |
| 1 | 0 |
| 2 | 0 |
| $\sum_t x_{3,t} =$ | 0 |
| **Valid configuration?** | ✘ |

In this example, none of the $x_{3,t}$ values equal 1 for any time in the simulation, meaning the operation is never scheduled. This means that the sum of $x_{3,t}$ values over all $t$ is 0 - the constraint is once again violated and you must allocate a penalty.

In summary:

| $t$ | $x_{3,t}$ | $x_{3,t}$ | $x_{3,t}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 0 |
| 2 | 0 | 1 | 0 |
| $\sum_t x_{3,t} =$ | 1 | 2 | 0 |
| **Valid configuration?** | ✔ | ✘ | ✘ |

Now that you understand when to assign penalties, let's formulate the constraint mathematically.

## Penalty Formulation

As seen previously, you want to assign a penalty whenever the sum of $x_{i,t}$ values across all possible $t$ values is not equal to 1. This is how you represent that mathematically:

$$g(x) = \sum_i \left( \left( \sum_{0 \le t < T} x_{i,t} \right) - 1 \right)^2 .$$

Let's break that down:

- $\left(\sum_{0\le t<T} x_{i,t}\right) - 1$

  As you saw in the sum row of the tables in the worked example, $\sum_{0\le t<T} x_{i,t}$ should always equal exactly 1 (meaning that an operation must be scheduled **once and only once** during the allowed time). This means that $\left(\sum_{0\le t<T} x_{i,t}\right) - 1$ should always give 0. This means there is no penalty assigned when the constraint is not violated.

  In the case where $\sum_{0\le t<T} x_{i,t} > 1$ (meaning an operation is scheduled to start more than once, like in the second example above), you now have a positive, non-zero penalty term as $\left(\sum_{0\le t<T} x_{i,t}\right) - 1 > 0$.

  In the case where $\sum_{0\le t<T} x_{i,t} = 0$ (meaning an operation is never scheduled to start, like in the last example above), you now have a $-1$ penalty term as $\left(\sum_{0\le t<T} x_{i,t}\right) - 1 = 0 - 1 = -1$.

- $\left(\sum \ldots\right)^2$

  Because the penalty terms must always be positive (otherwise you would be *reducing* the penalty when an operation isn't scheduled), you must square the result of $\left(\sum_{0\le t<T} x_{i,t}\right) - 1$.

  This ensures that the penalty term is always positive (as $(-1)^2 = 1$).

- $\sum_i \left((\ldots)^2\right)$

  Lastly, you must sum all penalties accumulated across all operations $O_i$ from all jobs.

To translate this constraint to code form, you are going to need to expand the quadratic equation in the sum.

To do this, Let's once again take $O_3$ as an example. Let's set $T = 2$ so the $t$ values will be 0 and 1. The first step will be to substitute in these values:

$$\sum_i \left(\left(\sum_{0\le t<T} x_{i,t}\right) - 1\right)^2 = (x_{3,0} + x_{3,1} - 1)^2 \tag{15}$$

For simplicity, the $x_{3,t}$ variables will be renamed as follows:

$$x_{3,0} = x \tag{16}$$
$$x_{3,1} = y \tag{17}$$

Substituting these values in, you now have the following:

$$\sum_i \left(\left(\sum_{0\le t<T} x_{i,t}\right) - 1\right)^2 = (x_{3,0} + x_{3,1} - 1)^2 \tag{18}$$

$$= (x + y - 1)^2 \quad (19)$$

Next, you need to expand out the bracket and multiply each term in the first bracket with all terms in the other bracket:

$$\begin{aligned}
\sum_i \left( \left( \sum_{0 \le t < T} x_{i,t} \right) - 1 \right)^2 &= (x_{3,0} + x_{3,1} - 1)^2 && (20)\\
&= (x + y - 1)^2 && (21)\\
&= (x + y - 1) \cdot (x + y - 1) && (22)\\
&= x^2 + y^2 + 2xy - 2x - 2y + 1 && (23)
\end{aligned}$$

The final step simplifies things a little. Because this is a binary optimization problem, $x$ and $y$ can only take the values of $0$ or $1$. Because of this, the following holds true:

$$x^2 = x$$

$$y^2 = y,$$

as

$$0^2 = 0$$

and

$$1^2 = 1$$

This means that the quadratic terms in the penalty function can combine with the two linear terms, giving the following formulation of the penalty function:

$$\begin{aligned}
\sum_i \left( \left( \sum_{0 \le t < T} x_{i,t} \right) - 1 \right)^2 &= x^2 + y^2 + 2xy - 2x - 2y + 1 && (24)\\
&= x + y + 2xy - 2x - 2y + 1 && (25)\\
&= 2xy - x - y + 1 && (26)
\end{aligned}$$

If $T$ was larger, you would have more terms ($z$ and so on, for example).

## Code

You can now use this expanded version of the penalty function to build the penalty terms in code. Again, the `weight` argument is included, to be assigned a value later on:

In [ ]:

```
"""
# Reminder of the relevant parameters
## Allowed time (jobs can only be scheduled below this limit)
T = 21

## Assignment of operations to jobs (operation ID: job ID)
ops_jobs_map = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2
"""
```

```
def operation_once_constraint(ops_jobs_map:dict, T:int, weight:float
    """
    Construct penalty terms for the operation once constraint.
    Penalty function is of form: 2xy - x - y + 1

    Keyword arguments:

    ops_jobs_map (dict): Map of operations to jobs {op: job}
    T (int): Allowed time (jobs can only be scheduled below this lim
    weight (float): Relative importance of this constraint
    """

    terms = []

    # 2xy - x - y parts of the constraint function
    # Loop through all operations
    for op in ops_jobs_map.keys():
        for t in range(T):
            # - x - y terms
            terms.append(Term(c=weight*-1, indices=[op*T+t]))

            # + 2xy term
            # Loop through all other start times for the same job
            # to get the cross terms
            for s in range(t+1, T):
                terms.append(Term(c=weight*2, indices=[op*T+t, op*T+

    # + 1 term
    terms.append(Term(c=weight*1, indices=[]))

    return terms
```

## No-overlap constraint

The no-overlap constraint is defined as follows:

| Constraint | Penalty condition |
|---|---|
| **No-overlap constraint** Machines can only do one thing at a time. | Assign penalty every time two operations on a single machine are scheduled to run at the same time. |

### Worked Example

For this final constraint, $J_1$ will once again be used as an example:

- $J_1$: Plan camping trip
  - $O_3$: Pick campsite (2 *minutes*) - **computer**
  - $O_4$: Pay online (2 *minutes*) - **computer**
  - $O_5$: Print receipt (3 *minutes*) - **printer**

Recall once more the variable $x_{i,t}$:

$$\text{If } x_{i,t} = 1,\ O_i \text{ starts at time } t \quad (27)$$
$$\text{If } x_{i,t} = 0,\ O_i \text{ does not start at time } t \quad (28)$$

As you can see from the above, $O_3$ and $O_4$ must be completed using the same machine (the computer). You can't do two things at the same time using the same machine, so to avoid violating the no-overlap constraint, you must ensure that $O_3$ and $O_4$ begin at different times: $x_{3,t}$ and $x_{4,t}$ must not equal 1 at the same time. You must also make sure that the operations don't overlap, just like you saw in the precedence constraint. This means that if $O_3$ starts at time $t$, $O_4$ must not start at times where $t \leq s < t + p_3$ (after $O_3$ has started but before it has been completed using the machine).

One example of a valid configuration is shown below:

| $t$ | $x_{3,t}$ | $x_{4,t}$ | $x_{3,t} \cdot x_{4,t}$ |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 0 | 0 | 0 |
| 2 | 0 | 1 | 0 |
| | | $\sum_t x_{3,t} \cdot x_{4,t} =$ | 0 |
| | | **Valid configuration?** | ✔ |

As you can see, when you compare $x_{i,t}$ values pairwise at each time in the simulation, their product always equals 0. Further to this, you can see that $O_4$ starts two time steps after $O_3$, which means that there is no overlap.

Below, we see a configuration that violates the constraint:

| $t$ | $x_{3,t}$ | $x_{4,t}$ | $x_{3,t} \cdot x_{4,t}$ |
|---|---|---|---|
| 0 | 0 | 0 | 0 |
| 1 | 1 | 1 | 1 |
| 2 | 0 | 0 | 0 |
| | | $\sum_t x_{3,t} \cdot x_{4,t} =$ | 1 |
| | | **Valid configuration?** | ✘ |

In this instance, $O_3$ and $O_4$ are both scheduled to start at $t = 1$ and given they require the same machine, this means that the constraint has been violated. The pairwise product of $x_{i,t}$ values is therefore no longer always equal to 0, as for $t = 1$ we have: $x_{3,1} \cdot x_{4,1} = 1$

Another example of an invalid configuration is demonstrated below:

| $t$ | $x_{3,t}$ | $x_{4,t}$ | $x_{3,t} \cdot x_{4,t}$ |
|---|---|---|---|
| 0 | 1 | 0 | 0 |
| 1 | 0 | 1 | 0 |
| 2 | 0 | 0 | 0 |
| | | $\sum_t x_{3,t} \cdot x_{4,t} =$ | 0 |
| | | **Valid configuration?** | ✘ |

In the above scenario, the two operations' running times have overlapped ( $t \leq s < t + p_3$), and therefore this configuration is not valid.

You can now use this knowledge to mathematically formulate the constraint

## Penalty Formulation

As you saw from the tables in the worked example, for the configuration to be valid, the sum of pairwise products of $x_{i,t}$ values for a machine $m$ at any time $t$ must equal 0. This gives you the penalty function:

$$h(x) = \sum_{i,t,k,s} x_{i,t} \cdot x_{k,s} = 0 \text{ for each machine } m$$

Let's break that down:

- $\sum_{i,t,k,s}$

  For operation $i$ starting at time $t$, and operation $k$ starting at time $s$, you need to sum over all possible start times $0 \leq t < T$ and $0 \leq s < T$. This indicates the need for another nested `for` loop, like you saw for the precedence constraint.

  For this summation, $i \neq k$ (you should always be scheduling two different operations).

  For two operations happening on a single machine, $t \neq s$ or the constraint has been violated. If $t = s$ for the operations, they have been scheduled to start on the same machine at the same time, which isn't possible.

- $x_{i,t} \cdot x_{k,s}$

  This is the product you saw explicitly calculated in the rightmost columns of the tables from the worked example. If two different operations $i$ and $k$ start at the same time ($t = s$), this product will equal 1. Otherwise, it will equal 0.

- $\sum(\ldots) = 0 \text{ for each machine } m$

  This sum is performed for each machine $m$ independently.

  If all $x_{i,t} \cdot x_{k,s}$ products in the summation equal 0, the total sum comes to 0. This means no operations have been scheduled to start at the same time on this machine and thus the constraint has not been violated. You can see an example of this in the bottom row of the first table from the worked example, above.

  If any of the $x_{i,t} \cdot x_{k,s}$ products in the summation equal 1, this means that $t = s$ for those operations and therefore two operations have been scheduled to start at the same time on the same machine. The sum now returns a value greater than 1, which gives us a penalty every time the constraint is violated. You can see an example of this in the bottom row of the second table from the worked example.

## Code

Using the above, you can transform the final penalty function into code that will generate the terms needed by the solver. As with the previous two penalty functions, the `weight` is included in the definition of the `Term` objects:

In [ ]:
```
"""
# Reminder of the relevant parameters
## Allowed time (jobs can only be scheduled below this limit)
T = 21

## Processing time for each operation
processing_time = {0: 2, 1: 1, 2: 3, 3: 2, 4: 2, 5: 3, 6: 1, 7: 2, 8

## Assignment of operations to jobs (operation ID: job ID)
ops_jobs_map = {0: 0, 1: 0, 2: 0, 3: 1, 4: 1, 5: 1, 6: 2, 7: 2, 8: 2

## Assignment of operations to machines
### Ten jobs, three machines
machines_ops_map = {
    0: [0, 1, 3, 4, 6, 7],  # Operations 0, 1, 3, 4, 6 and 7 are ass
    1: [2, 5, 8],           # Operations 2, 5 and 8 are assigned to
    2: [9]                  # Operation 9 is assigned to machine 2 (
}
"""

def no_overlap_constraint(T:int, processing_time:dict, ops_jobs_map:
    """
    Construct penalty terms for the no overlap constraint.

    Keyword arguments:

    T (int): Allowed time (jobs can only be scheduled below this lim
    processing_time (dict): Operation processing times
    weight (float): Relative importance of this constraint
    ops_jobs_map (dict): Map of operations to jobs {op: job}
    machines_ops_map(dict): Mapping of operations to machines, e.g.:
        machines_ops_map = {
            0: [0,1],          # Operations 0 & 1 assigned to machin
            1: [2,3]           # Operations 2 & 3 assigned to machin
        }
    """

    terms = []

    # For each machine
    for ops in machines_ops_map.values():
        # Loop over each operation i requiring this machine
        for i in ops:
            # Loop over each operation k requiring this machine
            for k in ops:
                # Loop over simulation time
                for t in range(T):
                    # When i != k (when scheduling two different ope
                    if i != k:
                        # t = s meaning two operations are schedulec
                        terms.append(Term(c=weight*1, indices=[i*T+t

                        # Add penalty when operation runtimes overla
                        for s in range(t, min(t + processing_time[i]
                            terms.append(Term(c=weight*1, indices=[i

                        # If operations are in the same job, penaliz
                        if ops_jobs_map[i] == ops_jobs_map[k]:
                            for s in range(0, t):
                                if i < k:
```

```
                                            terms.append(Term(c=weight*1, in
                                        if i > k:
                                            terms.append(Term(c=weight*1, in

    return terms
```

## Minimize the makespan

So far you've learned how to represent constraints of your optimization problem with a penalty model, which allows you to obtain *valid* solutions to your problem from the optimizer. Remember however that your end goal is to obtain an *optimal* (or close to optimal) solution. In this case, you're looking for the schedule with the fastest completion time of all jobs.

The makespan $M$ is defined as the total time required to run all jobs, or alternatively the finishing time of the last job, which is what you want to minimize. To this end, you need to add a fourth component to the cost function that adds larger penalties for solutions with larger makespans:

$$H(x) = \alpha \cdot f(x) + \beta \cdot g(x) + \gamma \cdot h(x) + \delta \cdot \mathbf{k(x)}$$

Let's come up with terms that increase the value of the cost function the further out the last job is completed. Remember that the completion time of a job depends solely on the completion time of its final operation. However, since you have no way of knowing in advance what the last job will be, or at which time the last operation will finish, you'll need to include a term for each operation and time step. These terms need to scale with the time parameter $t$, and consider the operation processing time, in order to penalize large makespans over smaller ones.

Some care is required in determining the penalty values, or *coefficients*, of these terms. Recall that you are given a set of operations $\{O_i\}$, which each take processing time $p_i$ to complete. An operation scheduled at time $t$ will then *complete* at time $t + p_i$. Let's define the coefficient $w_t$ as the penalty applied to the cost function for an operation to finish at time $t$. As operations can be scheduled in parallel, you don't know how many might complete at any given time, but you do know that this number is at most equal to the number of available machines $m$. The sum of all penalty values for operations completed at time $t$ are thus in the range $[0,\ m \cdot w_t]$. You want to avoid situations were completing a single operation at time $t + 1$ is less expensive than m operations at time $t$. Thus, the penalty values cannot follow a simple linear function of time.

Precisely, you want your coefficients to satisfy:

$$w_{t+1} > m \cdot w_t$$

For a suitable parameter $\epsilon > 0$, you can then solve the following recurrence relation:

$$w_{t+1} = m \cdot w_t + \epsilon$$

The simplest solution is given by the function:

The simplest solution is given by the function:

$$w_t = \epsilon \cdot \frac{m^t - 1}{m - 1}$$

## Limiting the number of terms

Great! You now have a formula for the coefficients of the makespan penalty terms that increase with time while taking into account that operations can be scheduled in parallel. Before implementing the new terms, let's try to limit the amount of new terms you're adding as much as possible. To illustrate, recall the job shop example you've been working on:

$$J_0 = \{O_0, O_1, O_2\} \tag{29}$$
$$J_1 = \{O_3, O_4, O_5\} \tag{30}$$
$$J_2 = \{O_6, O_7, O_8, O_9\} \tag{31}$$

First, consider that you only need the last operation in every job, as the precedence constraint guarantees that all other operations are completed before it. Given $n$ jobs, you thus consider only the operations $\{O_{k_0-1}, O_{k_1-1}, \ldots, O_{k_{n-1}-1}\}$, where the indices $k_j$ denotes the number of operations up to and including job $j$. In this example, you only add terms for the following operations:

$$\{O_2, O_5, O_9\}$$

$$\text{with } k_0 = 3, k_1 = 6, k_2 = 10$$

Next, you can find a lower bound for the makespan and only penalize makespans that are greater than this minimum. A simple lower bound is given by the longest job, as each operation within a job must execute sequentially. You can express this lower bound as follows:

$$M_{lb} = \max_{0 \leq j < n} \{ \sum_{i=k_j}^{k_{j+1}-1} p_i \} \leq M$$

For the processing times given in this example, you get:

$$J_0: \quad p_0 + p_1 + p_2 = 2 + 1 + 3 = 6 \tag{32}$$
$$J_1: \quad p_3 + p_4 + p_5 = 2 + 2 + 3 = 7 \tag{33}$$
$$J_2: \quad p_6 + p_7 + p_8 + p_9 = 1 + 2 + 3 + 2 = 8 \tag{34}$$
$$\tag{35}$$
$$\Rightarrow M_{lb} = 8 \tag{36}$$

Finally, the makespan is upper-bounded by the sequential execution time of all jobs, 6 + 7 + 8 = 21 in this case. The simulation time T should never exceed this upper bound. Regardless of whether this is the case or not, you need to include penalties for all time steps up to T, or else larger time steps without a penalty will be favored over smaller ones!

To summarize:

To summarize:

- Makespan penalty terms are only added for the last operation in every job $\{O_{k_0-1}, O_{k_1-1}, \ldots, O_{k_{n-1}-1}\}$
- The makespan is lower-bounded by the longest job $\Rightarrow$ only include terms for time steps $M_{lb} < t < T$

## Implementing the penalty terms

You are now ready to add the makespan terms to the cost function. Recall that all terms contain a coefficient and one (or multiple) binary decision variables $x_{i,t}$. Contrary to the coefficients $w_t$ defined above, where $t$ refers to the completion time of an operation, the variables $x_{i,t}$ determine if an operation $i$ is *scheduled* at time t. To account for this difference, you'll have to shift the variable index by the operation's processing time $p_i$. All makespan terms can then be expressed as follows:

$$k(x) = \sum_{i \in \{k_0-1, \ldots, k_{n-1}-1\}} \left( \sum_{M_{lb} < t < T + p_i} w_t \cdot x_{i,\, t-p_i} \right)$$

Lastly, you need to make a small modification to the coefficient function so that the first value $w_{M_{lb}+1}$ always equals one. With $\epsilon = 1$ and $t_0 = M_{lb}$ you get:

$$w_t = \frac{m^{t-t_0} - 1}{m - 1}$$

## Code

The code below implements the ideas discussed above by generating the necessary `Term` objects required by the solver.

In [ ]:
```
"""
# Reminder of the relevant parameters
## Allowed time (jobs can only be scheduled below this limit)
T = 21

## Processing time for each operation
processing_time = {0: 2, 1: 1, 2: 3, 3: 2, 4: 2, 5: 3, 6: 1, 7: 2, 8

## Assignment of operations to jobs (job ID: [operation IDs])
jobs_ops_map = {
    0: [0, 1, 2],    # Pay electricity bill
    1: [3, 4, 5],    # Plan camping trip
    2: [6, 7, 8, 9] # Book dentist appointment
}
"""

def calc_penalty(t:int, m_count:int, t0:int):
    assert m_count > 1                           # Ensure you don't
    return (m_count**(t - t0) - 1)/float(m_count - 1)

def makespan_objective(T:int, processing_time:dict, jobs_ops_map:dic
    """
    Construct makespan minimization terms.

    Keyword arguments:
```

```
    Keyword arguments:

    T (int): Allowed time (jobs can only be scheduled below this lim
    processing_time (dict): Operation processing times
    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    m_count (int): Number of machines
    weight (float): Relative importance of this constraint
    """

    terms = []

    lower_bound = max([sum([processing_time[i] for i in job]) for jo
    upper_bound = T

    # Loop through the final operation of each job
    for job in jobs_ops_map.values():
        i = job[-1]
        # Loop through each time step the operation could be complet
        for t in range(lower_bound + 1, T + processing_time[i]):
            terms.append(Term(c=weight*(calc_penalty(t, m_count, low

    return terms
```

## Putting it all together

As a reminder, here are the penalty terms:

| Constraint | Penalty condition |
|---|---|
| **Precedence constraint** Operations in a job must take place in order. | Assign penalty every time $O_{i+1}$ starts before $O_i$ has finished (they start out of order). |
| **Operation-once constraint** Each operation is started once and only once. | Assign penalty if an operation isn't scheduled within the allowed time. **Assumption:** if an operation starts, it runs to completion. |
| **No-overlap constraint** Machines can only do one thing at a time. | Assign penalty every time two operations on a single machine are scheduled to run at the same time. |

- **Precedence constraint**:

$$f(x) = \sum_{k_{n-1} \le i < k_n, s < t + p_i} x_{i,t} \cdot x_{i+1,s} \text{ for each job } n$$

- **Operation-once constraint**:

$$g(x) = \sum_{i} \left( \left( \sum_{0 \le t < T} x_{i,t} \right) - 1 \right)^2$$

- **No-overlap constraint**:

$$h(x) = \sum_{i,t,k,s} x_{i,t} \cdot x_{k,s} = 0 \text{ for each machine } m$$

- **Makespan minimization**:

$$k(x) = \sum_{i\in\{k_0-1,\ldots,k_{n-1}-1\}} \left( \sum_{M_{lb}<t<T+p_i} w_t \cdot x_{i,\,t-p_i} \right)$$

As you saw earlier, combining the penalty functions is straightforward - all you need to do is assign each term a weight and add all the weighted terms together, like so:

$$H(x) = \alpha \cdot f(x) + \beta \cdot g(x) + \gamma \cdot h(x) + \delta \cdot k(x)$$

where $\alpha, \beta, \gamma$ and $\delta$ represent the different weights assigned to the pena

The weights represent how important each penalty function is, relative to all the others.

> **NOTE:** Along with modifying your cost function (how you represent the penalties), tuning these weights will define how much success you will have solving your optimization problem. There are many ways to represent each optimization problem's penalty functions and many ways to manipulate their relative weights, so this may require some experimentation before you see success. The end of this sample dives a little deeper into parameter tuning.

## Code

As a reminder, below you again see the code representation of the problem parameters: the maximum allowed time `T`, the operation processing times `processing_time`, the mapping of operations to jobs (`jobs_ops_map` and `ops_jobs_map`), the assignment of operations to machines (`machines_ops_map`), and the helper function `process_config`.

In [ ]:

```
def process_config(jobs_ops_map:dict, machines_ops_map:dict, process
    """
    Process & validate problem parameters (config) and generate inve

    Keyword arguments:

    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    machines_ops_map(dict): Mapping of operations to machines, e.g.:
        machines_ops_map = {
            0: [0,1],          # Operations 0 & 1 assigned to machin
            1: [2,3]           # Operations 2 & 3 assigned to machin
        }
    processing_time (dict): Operation processing times
    T (int): Allowed time (jobs can only be scheduled below this lim
    """

    # Problem cannot take longer to complete than all operations exe
    ## Sum all operation processing times to calculate the maximum m
    T = min(sum(processing_time.values()), T)

    # Ensure operation assignments to machines are sorted in ascendi
    for m, ops in machines_ops_map.items():
        machines_ops_map[m] = sorted(ops)
    ops_jobs_map = {}

    for job, ops in jobs_ops_map.items():
        # Fail if operation IDs within a job are out of order
        # Fail if operation IDs within a job are out of order
        assert (ops == sorted(ops)), f"Operation IDs within a job mu

        for op in ops:
            # Fail if there are duplicate operation IDs
            assert (op not in ops_jobs_map.keys()), f"Operation IDs
            ops_jobs_map[op] = job

    return ops_jobs_map, T

# Set problem parameters
## Allowed time (jobs can only be scheduled below this limit)
T = 21

## Processing time for each operation
processing_time = {0: 2, 1: 1, 2: 3, 3: 2, 4: 2, 5: 3, 6: 1, 7: 2, 8

## Assignment of operations to jobs (job ID: [operation IDs])
### Operation IDs within a job must be in ascending order
jobs_ops_map = {
    0: [0, 1, 2],
    1: [3, 4, 5],
    2: [6, 7, 8, 9]
}

## Assignment of operations to machines
### Three jobs, two machines
machines_ops_map = {
    0: [0, 1, 3, 4, 6, 7],  # Operations 0, 1, 3, 4, 6 and 7 are ass
    1: [2, 5, 8],           # Operations 2, 5 and 8 are assigned to
    2: [9]                  # Operation 9 is assigned to machine 2 (
}

## Inverse mapping of jobs to operations
ops_jobs_map, T = process_config(jobs_ops_map, machines_ops_map, pro
```

The following code snippet shows how you assign weight values and assemble the penalty terms by summing the output of the penalty and objective functions, as was demonstrated mathematically earlier in this sample. These terms represent the cost function and they are what you will submit to the solver.

In [ ]:

```
# Generate terms to submit to solver using functions defined previou
## Assign penalty term weights:
alpha = 1  # Precedence constraint
beta = 1   # Operation once constraint
gamma = 1  # No overlap constraint
delta = 0.00000005  # Makespan minimization (objective function)

## Build terms
### Constraints:
c1 = precedence_constraint(jobs_ops_map, T, processing_time, alpha)
c2 = operation_once_constraint(ops_jobs_map, T, beta)
c3 = no_overlap_constraint(T, processing_time, ops_jobs_map, machine

### Objective function
c4 = makespan_objective(T, processing_time, jobs_ops_map, len(machin

### Combine terms:
terms = []
terms = c1 + c2 + c3 + c4
```

> **NOTE**: You can find the full Python script for this sample here

## Submit problem to Azure Quantum

This code submits the terms to the Azure Quantum `SimulatedAnnealing` solver. You could also have used the same problem definition with any of the other Azure Quantum Optimization solvers available (for example, `ParallelTempering`). You can find further information on the various solvers available through the Azure Quantum Optimization service here.

The job is run synchronously in this instance, however this could also be submitted asynchronously as shown in the next subsection.

In [ ]:

```
from azure.quantum.optimization import Problem, ProblemType
from azure.quantum.optimization import SimulatedAnnealing # Change t

# Problem type is PUBO in this instance. You could also have chosen 
problem = Problem(name="Job shop sample", problem_type=ProblemType.p

# Provide details of your workspace, created at the beginning of thi
# Provide the name of the solver you wish to use for this problem (a
solver = SimulatedAnnealing(workspace, timeout = 100) # Timeout in s

# Run job synchronously
result = solver.optimize(problem)
config = result['configuration']

print()
print(config)
```

## Run job asynchronously

Alternatively, a job can be run asynchronously, as shown below:

```
# Submit problem to solver
job = solver.submit(problem)
print(job.id)

# Get job status
job.refresh()
print(job.details.status)

# Get results
result = job.get_results()
config = result['configuration']
print(config)
```

## Map variables to operations

This code snippet contains several helper functions which are used to parse the results returned from the solver and print them to screen in a user-friendly format.

In [ ]:

```
from typing import List
```

```
def create_op_array(config: dict):
    """
    Create array from returned config dict.

    Keyword arguments:
    config (dictionary): config returned from solver
    """

    variables = []
    for key, val in config.items():
        variables.insert(int(key), val)
    return variables

def print_problem_details(ops_jobs_map:dict, processing_time:dict, m
    """

    Print problem details e.g. operation runtimes and machine assign

    Keyword arguments:
    ops_jobs_map (dict): Map of operations to jobs {operation: job}
    processing_time (dict): Operation processing times
    machines_ops_map(dict): Mapping of machines to operations
    """

    machines = [None] * len(ops_jobs_map)

    for m, ops in machines_ops_map.items():
        for op in ops:
          machines[op] = m

    print(f"           Job ID: {list(ops_jobs_map.values())}")
    print(f"     Operation ID: {list(ops_jobs_map.keys())}")
    print(f"Operation runtime: {list(processing_time.values())}")
    print(f" Assigned machine: {machines}")
    print()

def split_array(T:int, array:List[int]):
    """
    Split array into rows representing the rows of our operation mat

    Keyword arguments:
    T (int): Time allowed to complete all operations
    array (List[int]): array of x_i,t values generated from config r
    """

    ops = []
    i = 0
    while i < len(array):
        x = array[i:i+T]
        ops.append(x)
        i = i + T
    return ops

def print_matrix(T:int, matrix:List[List[int]]):
    """
    Print final output matrix.

    Keyword arguments:
    T (int): Time allowed to complete all operations
    matrix (List[List[int]]): Matrix of x_i,t values
    """

    labels = "    t:"
    for t in range(0, T):
        labels += f" {t}"
    print(labels)

    idx = 0
    for row in matrix:
        print("x_" + str(idx) + ",t: ", end="")
        print(' '.join(map(str,row)))
        idx += 1
    print()

def extract_start_times(jobs_ops_map:dict, matrix:List[List[int]]):
    """
    Extract operation start times & group them into jobs.

    Keyword arguments:
    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    matrix (List[List[int]]): Matrix of x_i,t values
    """
    #jobs = {}
    jobs = [None] * len(jobs_ops_map)
    op_start_times = []
    for job, ops in jobs_ops_map.items():
        x = [None] * len(ops)
        for i in range(len(ops)):
            try :
                x[i] = matrix[ops[i]].index(1)
                op_start_times.append(matrix[ops[i]].index(1))
            except ValueError:
                x[i] = -1
                op_start_times.append(-1)
        jobs[job] = x

    return jobs, op_start_times
```

## Results

Finally, you take the config returned by the solver and read out the results.

In [ ]:

```
# Produce 1D array of x_i,t = 0, 1 representing when each operation
op_array = create_op_array(config)

# Print config details:
print(f"Config dict:\n{config}\n")
print(f"Config array:\n{op_array}\n")

# Print problem setup
print_problem_details(ops_jobs_map, processing_time, machines_ops_ma

# Print final operation matrix, using the returned config
print("Operation matrix:")
matrix = split_array(T, op_array)
print_matrix(T, matrix)

# Find where each operation starts (when x_i,t = 1) and return the s
print("Operation start times (grouped into jobs):")
jobs, op_start_times = extract_start_times(jobs_ops_map, matrix)
print(jobs)

# Calculate makespan (time taken to complete all operations - the ob
op_end_times = [op_start_times[i] + processing_time[i] for i in rang
makespan = max(op_end_times)
```

```
print(f"\nMakespan (time taken to complete all operations): {makespa
```

## Visualize results

The cell below shows how you can visualize the schedule generated by the service using Plotly Express.

In [ ]:
```
import plotly.express as px
import pandas as pd

# Graphics
machines = [None] * len(ops_jobs_map)
for m, ops in machines_ops_map.items():
    for op in ops:
      machines[op] = m

ops = list(ops_jobs_map.keys())

# Create data frame
d = {'Operation': ops, 'Machine': machines, 'Start time': op_start_t
df = pd.DataFrame(data=d)
df['delta'] = processing_time.values()

# Produce plot
fig = px.timeline(df, x_start='Start time', x_end='Finish time', y='

fig.update_yaxes(autorange="reversed")
fig.update_layout(
    font_family="Segoe UI",
    title_font_family="Segoe UI",
    width=800,
    height=800,
)
fig.layout.xaxis.type = 'linear'
fig.data[0].x = df.delta.tolist()
fig.update_layout(
    xaxis_title="Time step",
)
fig.update_yaxes(tick0=0, dtick=1)
fig.update_xaxes(tick0=0, dtick=1)
fig.show()
```

For this small problem instance, the solver quickly returned a solution. For bigger, more complex problems you may need to run the job asynchronously, as shown earlier in this sample.

## Validate the solution

In this instance, it is possible to visually verify that the solution does not validate any constraints:

- Operations belonging to the same job happen in order
- Operations are started once and only once
- Each machine only has one operation running at a time

In this particular instance, you can also tell that the solver scheduled the repair tasks

In this particular instance, you can also tell that the solver scheduled the repair tasks in such a way that the **total time to complete them all (the makespan) was minimized** - both machines are continuously in operation, with no time gaps between scheduled operations. This is the solution with the lowest possible cost, also known as the global minimum for the cost function. However, you must remember that these solvers are heuristics and are therefore not guaranteed to find the best solution possible, particularly when the problem definition becomes more complex.

Depending on how well the cost function is defined and the weights are tuned, the solver will have varying degrees of success. This reinforces the importance of verifying and evaluating returned solutions, to enable tuning of the problem definition and parameters (such as weights/coefficients) in order to improve solution quality.

For larger or more complex problems, it will not always be possible to verify the solution by eye. It is therefore common practice to implement some code to verify that solutions returned from the optimizer are valid, as well as evaluating how good the solutions are (at least relative to solutions returned previously). This capability is also useful when it comes to tuning weights and penalty functions.

You can perform this validation using the following code snippet, which checks the solution against all three constraints before declaring the solution valid or not. If any of the constraints are violated, the solution will be marked as invalid. An example of an invalid solution has also been included, for comparison.

In [ ]:

```python
def check_precedence(processing_time, jobs):
    """
    Check if the solution violates the precedence constraint.
    Returns True if the constraint is violated.

    Keyword arguments:
    processing_time (dict): Operation processing times
    jobs (List[List[int]]): List of operation start times, grouped i
    """

    op_id = 0
    for job in jobs:
        for i in range(len(job) - 1):
            if job[i+1] - job[i] < processing_time[op_id]:
                return True
            op_id += 1
        op_id += 1
    return False

def check_operation_once(matrix):
    """
    Check if the solution violates the operation once constraint.
    Returns True if the constraint is violated.

    Keyword arguments:
    matrix (List[List[int]]): Matrix of x_i,t values
    """
    for x_it_vals in matrix:
        if sum(x_it_vals) != 1:
            return True
    return False
```

```
def check_no_overlap(op_start_times:list, machines_ops_map:dict, pro
    """
    Check if the solution violates the no overlap constraint.
    Returns True if the constraint is violated.

    Keyword arguments:
    op_start_times (list): Start times for the operations
    machines_ops_map(dict): Mapping of machines to operations
    processing_time (dict): Operation processing times
    """
    pvals = list(processing_time.values())

    # For each machine
    for ops in machines_ops_map.values():
        machine_start_times = [op_start_times[i] for i in ops]
        machine_pvals = [pvals[i] for i in ops]

        # Two operations start at the same time on the same machine
        if len(machine_start_times) != len(set(machine_start_times))
            return True

        # There is overlap in the runtimes of two operations assigne
        machine_start_times, machine_pvals = zip(*sorted(zip(machine
        for i in range(len(machine_pvals) - 1):
            if machine_start_times[i] + machine_pvals[i] > machine_s
                return True

    return False

def validate_solution(matrix:dict, machines_ops_map:dict, processing
    """
    Check that solution has not violated any constraints.
    Returns True if the solution is valid.

    Keyword arguments:
    matrix (List[List[int]]): Matrix of x_i,t values
    machines_ops_map(dict): Mapping of machines to operations
    processing_time (dict): Operation processing times
    jobs_ops_map (dict): Map of jobs to operations {job: [operations
    """

    jobs, op_start_times = extract_start_times(jobs_ops_map, matrix)

    # Check if constraints are violated
    precedence_violated = check_precedence(processing_time, jobs)
    operation_once_violated = check_operation_once(matrix)
    no_overlap_violated = check_no_overlap(op_start_times, machines_

    if not precedence_violated and not operation_once_violated and n
        print("Solution is valid.\n")
    else:
        print("Solution not valid. Details:")
        print(f"\tPrecedence constraint violated: {precedence_violat
        print(f"\tOperation once constraint violated: {operation_onc
        print(f"\tNo overlap constraint violated: {no_overlap_violat

print_problem_details(ops_jobs_map, processing_time, machines_ops_ma

print("Azure Quantum solution:")
print_matrix(T, matrix)

print("Operation start times (grouped into jobs):")
print(jobs)
print()
```

```
validate_solution(matrix, machines_ops_map, processing_time, jobs_op
```

As you can see, the result returned by the Azure Quantum solver has been confirmed as valid (it does not violate any of the constraints).

## Tune parameters

Great! You've learned how to model a cost function, run a solver, and verify the solution of an optimization problem using Azure Quantum. Using your knowledge, you successfully repaired your ship! However, you may have been wondering how exactly the weights that appear in the cost function were chosen. Let's take a look at a general method that can help you balance the different components that make up a cost function.

If you recall, the cost function is made up of four components, one for each constraint and one to minimize the makespan:

$$H(x) = \alpha \cdot f(x) + \beta \cdot g(x) + \gamma \cdot h(x) + \delta \cdot k(x)$$

The importance attributed to each term can be adjusted using the weights (coefficients) $\alpha, \beta, \gamma, \text{ and } \delta$. The process of adjusting these weights is referred to as *parameter tuning*. In general, there's no absolute rule to determine the optimal value for each weight, and you might have to use some trial and error to figure out what works best for your problem. However, the guidelines below can help you get good starting point.

### Adjusting the optimization term weight

Intuitively, it should be clear that satisfying the constraints is more important than minimizing the makespan. An invalid solution, even with a very small makespan, would be useless to you. The weights of the cost function can be used to reflect this fact. As a rule of thumb, breaking a single constraint should be around 5-10x more expensive than any valid solution.

Let's start with an upper bound on the value of the cost function for any valid solution. At worst, a valid solution (meaning that $f(x) = g(x) = h(x) = 0$) contributes at most $m \cdot w_{T-1+max(p_i)}$ to the cost function. This is the case when $m$ operations, all taking $max(p_i)$ to complete, are scheduled at the last time step $T-1$. For convenience, let's say that this should result in a cost function value of $1$. You can compute what the value of $\delta$ should be to achieve this value. The code example you've been working with uses the following parameters:

$$m = 3,\ T = 21,\ max(p_i) = 3,\ M_{lb} = 8,\ w_t = \frac{m^{t-M_{lb}}}{m-1}$$

First, calculate the latest time an operation could finish. This is given by the max time $T$ (minus one because you are using 0-based indexing), plus the longest processing time for any operation ($max(p_i)$):

$$t_{max} = T - 1 + max(p_i) = 21 - 1 + 3 = 23$$

Then, calculate $w_{t_{max}}$:

$$w_{t_{max}} = \frac{m^{t_{max} - M_{lb}}}{m - 1} = \frac{3^{23-8}}{3-1} = \frac{3^{15}}{2} = 7,174,453.5$$

The upper bound is then:

$$m \cdot w_{t_{max}} = 3 \times 7,174,453.5 = 21,523,360.5$$

To obtain the desired value of $1$, you can approximately set the weight to:

$$\delta = \frac{1}{m \cdot w_{t_{max}}} = \frac{1}{21,523,360.5} = 0.00000005$$

### Adjusting the constraint weights

As mentioned in the previous section, breaking a single constraint should incur a penalty roughly 5-10x higher than that of the worst valid solution. Assuming that breaking one constraint adds a value of $1$ to the cost function, you can set the remaining weights to:

$$\alpha = \beta = \gamma = 5$$

Now, you can run a problem instance and use the verifier to check if any constraints are being broken. If all constraints are satisfied, congratulations! You should have obtained a good solution from the optimizer.

If instead one constraint is consistently broken, you probably need to increase its weight compared to the others.

### Further adjustments

You may also come across situations in which constraints are being broken without a particular preference for which. In this case, make sure the time $T$ given a large enough value. If $T$ is too small, there may not even exist a valid solution, or the solver could be too constrained to feasibly find one.

Optionally, if you're looking for better solutions than the ones obtained so far, you may always try to lower the value of $T$, or increase the importance of the makespan component $\delta$. A tighter bound on the makespan can help the solver find a more optimal solution, as can increasing the weight $\delta$. You may also find that doing so increases the speed at which a solution is found. If any problems pop up with broken constraints, you went too far and need to change the parameters in the other direction again.

## Next steps

Now that you understand the problem scenario and how to define the cost function, there are a number of experiments you can perform to deepen your understanding and improve the solution defined above:

- Modify the problem definition:
  - Change the number of jobs, operations, and/or machines
    - Vary the number of operations in each job
    - Change operation runtimes
    - Change machine assignments
    - Add/remove machines
- Rewrite the penalty functions to improve their efficiency
- Tune the parameters
- Try using a different solver (such as `ParallelTempering`)