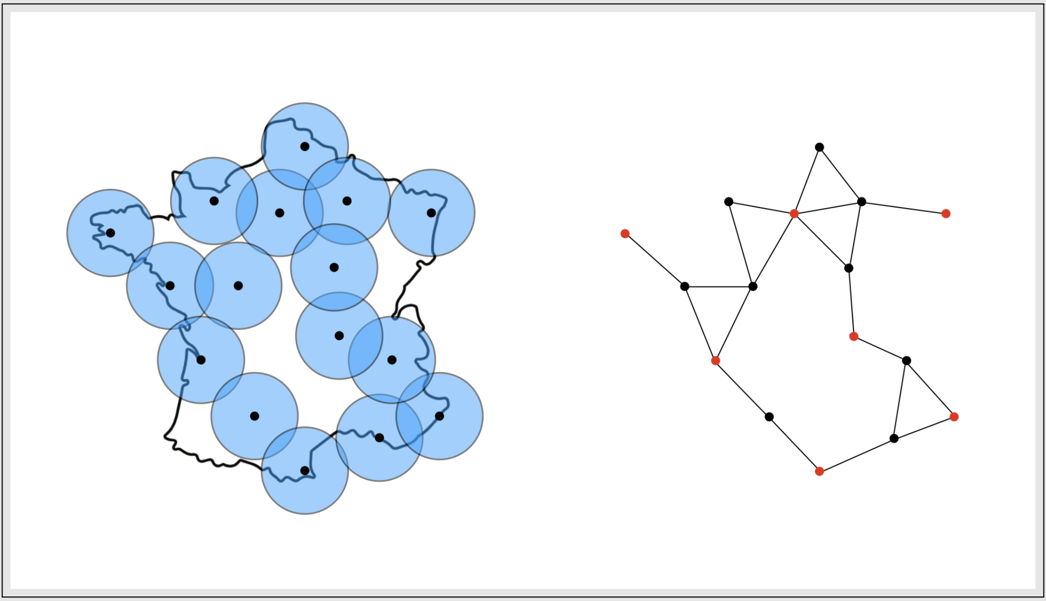
Problem - Solving the MIS problem with a hardware-efficient implementation of QAOA
Consider an undirected graph $G(V, E)$ composed of a set of vertices V connected by unweighted edges E. An independent set of this graph is a subset of vertices where any two elements of this subset are not connected by an edge. The Maximum Independent Set (MIS) corresponds to the largest of such subsets, and it is in general an NP-complete problem to determine the MIS of a graph. The MIS problem has several interesting applications, such as portfolio diversification in finance, or broadcast systems (wifi or cellular network) optimization.

For example, assume an ensemble of identical radio transmitters over French cities that each have the same radius of transmission. It was quickly realized that two transmitters with close or equal frequencies could interfere with one another, hence the necessity to assign non-interfering frequencies to overlapping transmitting towers. Because of the limited amount of bandwidth space, some towers have to be assigned the same or close frequencies. The MIS of a graph of towers indicate the maximum number of towers that can have close or equal given frequency (red points).



When looking for the MIS of a graph, we separate the vertices into two distinct classes: an independence one and the others. We can attribute a status z to each vertex, where $z_i = 1$ if vertex i is attributed to the independent set, and $z_i = 0$ otherwise. The Maximum Independent Set corresponds to the minima of the following cost function:

$$C(z_1, \dots, z_N) = -\sum_{i=1}^{N} z_i + U \sum_{(i,j)\in E} z_i z_j,$$

In this cost function, we want to promote a maximal number of vertices to the 1 state, but the fact that $U \gg 1$ strongly penalizes two adjacent vertices in state. The minimum of $C(z_1, \dots, z_N)$ therefore corresponds to the maximum independent set of the graph.

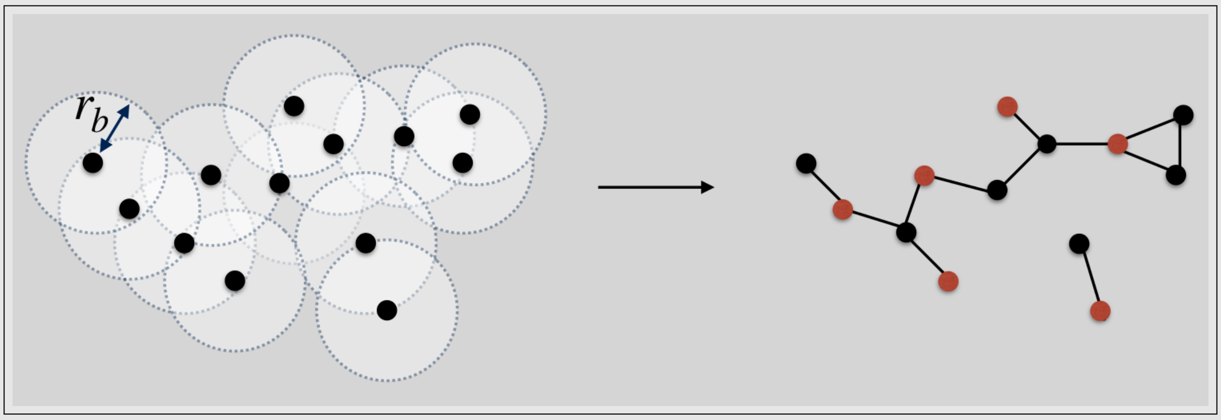
The MIS problem on unit disks graphs can be tackled by using an ensemble of interacting cold neutral atoms as a quantum resource, where each atom represents a vertex of the graph under study.

As with any quantum system, the dynamics of the atoms are governed by the Schrödinger equation, involving a Hamiltonian depending on the atomic positions, the electronic energy levels and their interactions. As seen in the course, placing N atoms at positions $\mathbf{r}_j$ in a plane, and coupling the ground state $|0\rangle$ to the Rydberg state $|1\rangle$ with a laser enables the realization of the Ising Hamiltonian:

$$H = \sum_{i=1}^{N} \frac{\hbar\Omega}{2}\sigma_i^x - \sum_{i=1}^{N} \frac{\hbar\delta}{2}\sigma_i^z + \sum_{j<i} \frac{C_6}{|\mathbf{r}_i - \mathbf{r}_j|^6} n_i n_j.$$

where $n_i = \frac{1}{2}(\sigma_i^z + 1)$ counts the number of Rydberg excitations at position i.

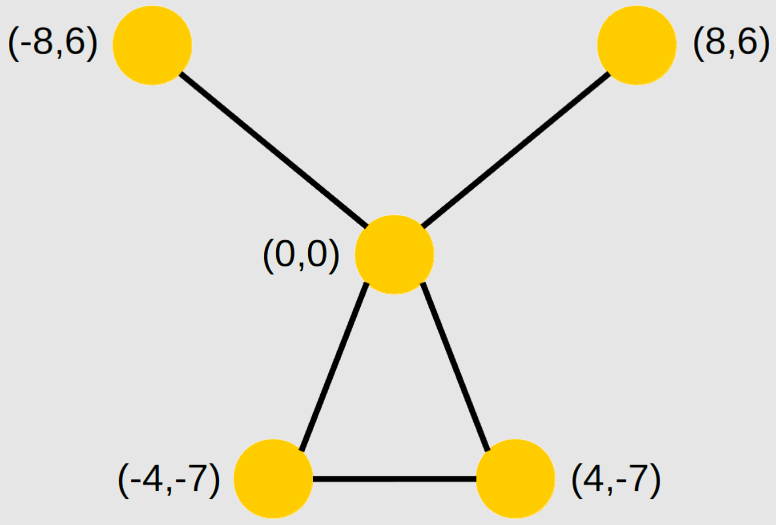
Interestingly, the physical interactions encoded in this Hamiltonian constrain the dynamics to only explore independent sets of the graph under study, then leading to an efficient search in the set of possible solutions. More precisely, the levels of two Rydberg atoms strongly interact if the distance between the atoms is smaller than the Rydberg blockade radius r_b, see left part of the figure below, resulting in the impossibility for the two atoms to be both in the same state at the same time. This naturally corresponds to the independent set constraint in the graph defined by the atoms, with edges linking atoms that sit at a distance closer than r_b, see right part of the figure where red vertices are forming one independent set.



From a spatial configuration of Rydberg atoms to a unit-disk MIS graph problem.

In the context of QAOA, keeping the last two terms of H, i.e. by taking $\Omega = 0$, thus corresponds to the quantum cost Hamiltonian H_C . A mixer Hamiltonian H_M can then be realized by keeping the first term and canceling the second term of H, i.e. by taking $\delta = 0$.

We will now illustrate how one can use Pulser to tackle the MIS problem. The graph $G(V, E)$ we propose you to study is the following:



First of all, copy/past the following imports in a new notebook:

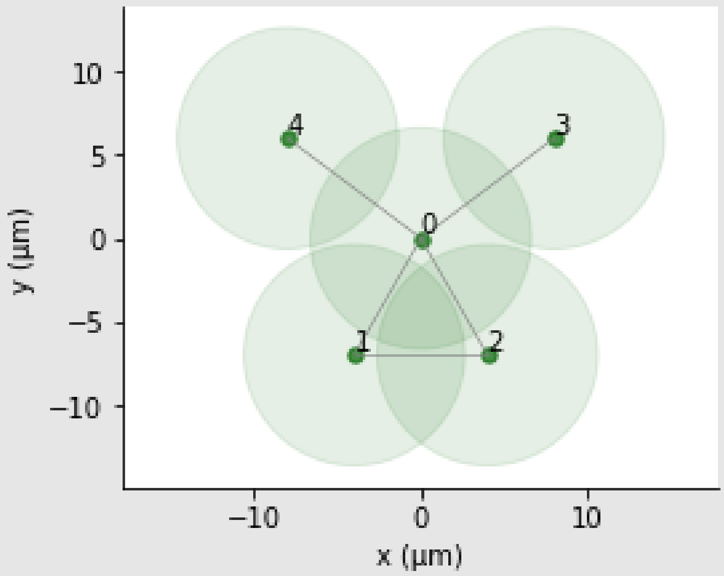
```
!pip install pulser
!pip install igraph

from pulser import Pulse, Sequence, Register
from pulser_simulation import Simulation
from pulser.devices import Chadoq2
from itertools import combinations
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import numpy as np
import igraph
```

All along this problem, you will take $\Omega = 1$ rad/µs.

a. (Graph architecture) Using the `Chadoq2` device available in Pulser, insert the missing information in the following lines of code in order to build a register that will reproduce the graph to study:

```
qubits = ...
reg = Register(...)
reg.draw(
    blockade_radius=...,
    draw_graph=True,
    draw_half_radius=True,
)
```

When executing, your code should produces the following figure:



b. (Maximum Independent Set) How many maximal independent sets do you identify on this graph? Give the corresponding configuration(s) in binary.

c. (Building the parameterized sequence) Now, you must build the quantum part of the QAOA. Initially, all atoms must be set in the ground state of the `ground-rydberg` basis. Once that done, next step consists in applying layers of alternating non-commutative Hamiltonians H_C and H_M. For the k-th optimization cycle, it consists in first applying the driver Hamiltonian evolution $\mathrm{e}^{-i\beta[k]H_M}$ by setting $\Omega = 1$ rad/µs and $\delta = 0$ rad/µs, then the cost Hamiltonian evolution $\mathrm{e}^{-i\gamma[k]H_C}$ by setting $\Omega = 0$ rad/µs and $\delta = 1$ rad/µs.

Instead of creating a new `Sequence` object everytime the quantum loop is called, you are going to create a parameterized sequence and later give that to the quantum loop. So for that, insert the missing information in the following lines of code:

```python
# Number of optimization cycles for QAOA
LAYERS = 2 # This value can be later changed

# Parametrized sequence
seq = Sequence(..., ...)
seq.declare_channel(..., ...)

beta_list = seq.declare_variable("beta_list", size=...)
gamma_list = seq.declare_variable("gamma_list", size=...)

if LAYERS == 1:
  beta_list = [beta_list]
  gamma_list = [gamma_list]

for beta, gamma in zip(beta_list, gamma_list):
  beta_pulse = Pulse.ConstantPulse(1000 * beta, ..., ..., 0)
  gamma_pulse = Pulse.ConstantPulse(1000 * gamma, ..., ..., 0)

  seq.add(..., ...)
  seq.add(..., ...)

seq.measure(...)
```

d. (QAOA with random parameters) Once you have the parameters that we want to apply - you don't have them yet, then use the `.build()` method to assign these values into a `assigned_seq` sequence. It is this sequence which is simulated every time the quantum loop is called. All these operations are synthesized in the function `quantum_loop` below where you have to insert the missing information:

```python
# Building the quantum loop
def quantum_loop(parameters):

  params = np.array(parameters)
  beta_params, gamma_params = np.reshape(params.astype(int), (2, LAYERS))
  assigned_seq = seq.build(beta_list=..., gamma_list=...)

  sim = Simulation(..., sampling_rate=0.01)
  res = ...
  counts = ...  # sample from the state vector

  return counts
```

Let's see the way it works by initializing with random parameters for two cycles of optimization.

For that, define the $\beta[k]$ and the $\gamma[k]$ by sampling uniformly at random between 1 and 10:

```
np.random.seed(123)  # ensures reproducibility of the tutorial

random_beta = np.random.uniform(..., ..., ...)
random_gamma = ...
```

and use these parameters as inputs for your function `quantum_loop`:

```
random_counts = quantum_loop(np.r_[random_beta, random_gamma])
```

The returned counts can finally be displayed in the form of an histogram with the following plot function that displays the MIS contributions in red and the other contributions in green:

```
def plot_distribution(counts):
    counts = dict(sorted(counts.items(), key=lambda item: item[1], reverse=True))
    indexes = ["01011", "00111"]  # MIS indexes
    color_dict = {key: "r" if key in indexes else "g" for key in counts}
    plt.figure(figsize=(12, 6))
    plt.xlabel("bitstrings")
    plt.ylabel("counts")
    plt.bar(counts.keys(), counts.values(), width=0.5, color=color_dict.values())
    plt.xticks(rotation="vertical")
    plt.show()
```

As the $\beta[k]$ and $\gamma[k]$ were chosen randomly, we don't expect here the MIS contributions to be the dominant contributions.

e. (Defining a cost function for the classical optimizer) For your quantum loop to return the MIS contributions as dominant contributions, you need first to define a cost function for your problem then minimize this function thanks to a classical optimizer.

To ease your work here, first convert your `Register` object into a `Graph` object of the `igraph` library. For that, copy/past the following lines of code in your notebook:

```
def pos_to_graph(pos):
    rb = Chadoq2.rydberg_blockade_radius(1.0)
    g = igraph.Graph()
    N = len(pos)
    edges = [
        [m, n]
        for m, n in combinations(range(N), r=2)
        if np.linalg.norm(pos[m] - pos[n]) < rb
    ]
    g.add_vertices(N)
    g.add_edges(edges)
    return g

pos = np.array([[0.0, 0.0], [-4, -7], [4, -7], [8, 6], [-8, 6]])
G = pos_to_graph(pos)
```

We estimate the cost of a sampled state vector by making an average over the samples. This is done by taking the corresponding bitstring $\mathbf{z} = (z_1, \dots, z_N)$ and calculating:

$$C(\mathbf{z}) = -\sum_i z_i + \sum_{i \geq j} p A_{ij} z_i z_j = p\left(\mathbf{z}^\top \cdot A^{\mathsf{U}} \cdot \mathbf{z}\right) - |\mathbf{z}|_0,$$

where A^{U} is the upper triangular part of the adjacency matrix of the graph, $|\cdot|_0$ gives the sum of non-zero terms of the bitstring, and p is the "penalty" introduced by the magnitude of the quadratic term.

Determining the cost of a given bitstring takes polynomial time. The average estimate is then used in the classical loop to optimize the variational parameters $\beta[k]$ and $\gamma[k]$:

```
# Calculating the cost of a single configuration of the graph
def get_cost(bitstring, G, penalty=10):
    z = np.array(list(bitstring), dtype=int)
    A = np.array(G.get_adjacency().data)
    # Add penalty and bias:
    cost = penalty * (z.T @ np.triu(A) @ z) - np.sum(z)
    return cost

# Weighted average over all the configurations of the graph
def get_avg_cost(counts, G):
    avg_cost = sum(counts[key] * get_cost(key, G) for key in counts)
    avg_cost = avg_cost / sum(counts.values())  # Divide by total samples
    return avg_cost
```

The function to minimize is then:

```
# Cost function to minimize
def func(param, *args):
    G = args[0]
    C = quantum_loop(param)
    avg_cost = get_avg_cost(C, G)
    return avg_cost
```

f. (QAOA with optimized parameters)

Let's now use a classical optimizer `minimize` in order to find the best variational parameters. This function takes as arguments the cost function `func` previously defined, the graph `G` and an initial `x0` point (i.e. random $\beta[k]$ and $\gamma[k]$) for the simplex in Nelder-Mead minimization. As the optimizer might get trapped in local minima, we repeat the optimization 20 times and select the parameters that yield the best approximation ratio.

```
# Trying 20 different initial guess
scores = []
params = []

for repetition in range(20):

  random_beta = np.random.uniform(1, 10, LAYERS)
  random_gamma = np.random.uniform(1, 10, LAYERS)
```

```
        try:
          res = minimize(
            func,
            args=G,
            x0=np.r_[random_beta, random_gamma],
            method="Nelder-Mead",
            tol=1e-5,
            options={"maxiter": 10},
          )
          scores.append(res.fun)
          params.append(res.x)
        except Exception as e:
          pass
```

Finally, use the parameters found by the optimizer to access the MIS configurations of your graph:

```
# QAOA with optimized parameters
optimal_count_dict = quantum_loop(...)
plot_distribution(...)
```

Solution: This graph has two maximal independent sets: $(1, 3, 4)$ and $(2, 3, 4)$, respectively 01011 and 00111 in binary.

```
!pip install pulser
!pip install igraph

from pulser import Pulse, Sequence, Register
from pulser_simulation import Simulation
from pulser.devices import Chadoq2
from itertools import combinations
from scipy.optimize import minimize
import matplotlib.pyplot as plt
import numpy as np
import igraph

# Reproducing the graph architecture
qubits = {'0':(0, 0), '1':(-4, -7), '2':(4, -7), '3':(8, 6), '4':(-8, 6)}
reg = Register(qubits)
reg.draw(
    blockade_radius=Chadoq2.rydberg_blockade_radius(1.0),
    draw_graph=True,
    draw_half_radius=True,
)

# Number of optimization cycles for QAOA
LAYERS = 2

# Parametrized sequence
seq = Sequence(reg, Chadoq2)
seq.declare_channel("ch0", "rydberg_global")

beta_list = seq.declare_variable("beta_list", size=LAYERS)
gamma_list = seq.declare_variable("gamma_list", size=LAYERS)
```

```
if LAYERS == 1:
  beta_list = [beta_list]
  gamma_list = [gamma_list]

for beta, gamma in zip(beta_list, gamma_list):
  beta_pulse = Pulse.ConstantPulse(1000 * beta, 1.0, 0.0, 0.0)
  gamma_pulse = Pulse.ConstantPulse(1000 * gamma, 0.0, 1.0, 0.0)

  seq.add(beta_pulse, "ch0")
  seq.add(gamma_pulse, "ch0")

seq.measure("ground-rydberg")

# Building the quantum loop
def quantum_loop(parameters):

  params = np.array(parameters)
  beta_params, gamma_params = np.reshape(params.astype(int), (2, LAYERS))

  assigned_seq = seq.build(beta_list=beta_params, gamma_list=gamma_params)

  sim = Simulation(assigned_seq, sampling_rate=0.01)
  res = sim.run()
  counts = res.sample_final_state(N_samples=1000)  # sample from the state vector

  return counts

# Displaying results
def plot_distribution(C):
    C = dict(sorted(C.items(), key=lambda item: item[1], reverse=True))
    indexes = ["01011", "00111"]  # MIS indexes
    color_dict = {key: "r" if key in indexes else "g" for key in C}
    plt.figure(figsize=(12, 6))
    plt.xlabel("bitstrings")
    plt.ylabel("counts")
    plt.bar(C.keys(), C.values(), width=0.5, color=color_dict.values())
    plt.xticks(rotation="vertical")
    plt.show()


# QAOA with random parameters
np.random.seed(123)  # ensures reproducibility of the tutorial

random_beta = np.random.uniform(1, 10, LAYERS)
random_gamma = np.random.uniform(1, 10, LAYERS)

random_counts = quantum_loop(np.r_[random_beta, random_gamma])
plot_distribution(random_counts)


# Defining the cost function for the classical optimizer
## Building a graph object with igraph
```

```python
def pos_to_graph(pos):
    rb = Chadoq2.rydberg_blockade_radius(1.0)
    g = igraph.Graph()
    N = len(pos)
    edges = [
        [m, n]
        for m, n in combinations(range(N), r=2)
        if np.linalg.norm(pos[m] - pos[n]) < rb
    ]
    g.add_vertices(N)
    g.add_edges(edges)
    return g

pos = np.array([[0.0, 0.0], [-4, -7], [4, -7], [8, 6], [-8, 6]])
G = pos_to_graph(pos)

## # Calculating the cost of a single configuration of the graph
def get_cost(bitstring, G, penalty=10):
    z = np.array(list(bitstring), dtype=int)
    A = np.array(G.get_adjacency().data)
    # Add penalty and bias:
    cost = penalty * (z.T @ np.triu(A) @ z) - np.sum(z)
    return cost

## # Weighted average over all the configurations of the graph
def get_avg_cost(counts, G):
    avg_cost = sum(counts[key] * get_cost(key, G) for key in counts)
    avg_cost = avg_cost / sum(counts.values())  # Divide by total samples
    return avg_cost

## Cost function to minimize
def func(param, *args):
    G = args[0]
    C = quantum_loop(param)
    avg_cost = get_avg_cost(C, G)
    return avg_cost

# Trying 20 different initial guess
scores = []
params = []

for repetition in range(20):
  random_beta = np.random.uniform(1, 10, LAYERS)
  random_gamma = np.random.uniform(1, 10, LAYERS)

  try:
    res = minimize(
      func,
      args=G,
      x0=np.r_[random_beta, random_gamma],
      method="Nelder-Mead",
      tol=1e-5,
      options={"maxiter": 10},
    )
    scores.append(res.fun)
    params.append(res.x)
  except Exception as e:
    pass

# QAOA with optimized parameters
optimal_count_dict = quantum_loop(params[np.argmin(scores)])
plot_distribution(optimal_count_dict)
```