# Actividad grupal: Minimizar costos publicitarios

May 12, 2024

Miembros del grupo:

- Miguel Aliende García
- Elisabeth Ortega Carrasco

## 1 Enunciado

En esta actividad resolveremos el el siguiente problema utilizado el algorimo QAOA (*Quantum Aproximation Optimization Algorithm*) utilizando 3 capas: Somos una empresa interesada en que nuestro producto se conozca entre todos los miembros de una red social. Tenemos la siguiente información de dicha red:

- Mario es amigo de Sarah, Raúl y Ana.
- Enrique es amigo de Sarah y Raúl.
- Saúl es amigo de Ana.

Todas las relaciones de amistad son recíprocas, es decir, si Mario es amigo de Raúl, entonces Raúl lo es de Mario. Nuestra idea es pagar a ciertas personas para que hablen de nuestro producto a sus contactos, pero queremos pagar al menor número de personas posibles, ¿a qué personas deberíamos contratar con este fin?

### 1.1 Introducción al algortimo QAOA

En este algortimo tratremos de encotrar el maximo autovalor de un operador $C$, que está asociado a una función objetivo $f$, es decir:

$$C|z\rangle = \sum_{i=1}^{m} C_i|z\rangle = f(z)|z\rangle$$

Podemos ver una similitud muy clara entre este operador C y un Hamiltoniano tipo Ising, por lo tanto, nuestro objetivo será encontrar el autoestado que corresponda al mayor valor de energía de $H$. Dado que $H$ no es unitario, lo que hacemos es calcular $e^{iH}$, ya que es fácilemente demostrable que podemos contruir esta exponencial con puertas $\sigma_z$ (cuando $H$ sea un Hamiltoniano tipo Ising). Una vez hecho esto, nuestro objetivo pasa de ser calcular el estado con el mayor módulo a calcular el de mayor fase.

1. Preparamos todos los estados a un estado de superposición equiprobable.

2. Aplicamos el operador $U(C, \gamma) = e^{-i\gamma C}$. Este operador lo que hace es desplazar del origen en el que se econtraban todos los estados, a otros puntos del eje $z$ en función de su fase, es decir, aquellos estados con mas fase se desplazaran más.

3. Posteriormente, aplicamos el operador $U(B, \beta) = e^{-i\beta B}$, siendo $B = \sum_{i=1}^{m} C_i^x$. Intuitivamente es otra rotación, y el resultado de aplicarla después de haber deplazado los estados en función de su fase es aquellos estados con mayor fase tendrán mayor probabilidad. De esta manera, hemos conseguido que aquellos con mayor energía de nuestro Hamiltoniano inicial, sean los más probables.

4. Si continuamos aplicando estos operadores consecutivamente conseguiremos mejorar aún más las probabilidades. También podemos utilizar algoritmos de optimización clásicos para calcular los valores más óptimos de $\gamma$ y $\beta$.

## 2 Desarrollo de la solución

El problema del enunciado se puede solucionar usando un algoritmo de cobertura de vértices y tiene una complejidad de NP-Completo, por lo que es un ejemplo ideal para ejecutarse en un computador cuántico. En pocas palabras, este algoritmo encuentra el número mínimo de vértices que se deben recorrer para visitar todos los nodos. Aplicado a nuestro problema, se traduce en el número mínimo de influencers que hay que pagar para que hablen de nosotros a sus contactos.

En las siguientes líneas explicamos cómo hemos resuelto este ejercicio.

[8]:
```
import networkx as nx
import pennylane as qml
from matplotlib import pyplot as plt
from pennylane import qaoa
from pennylane import numpy as np
```

Leemos el input del usuario con el nombre de las amistades para parametrizar el problema. El código está preparado para adaptarse a esta lista de *input* generada por el usuario.

[9]:
```
amistades = [["Mario", "Sarah"],
             ["Mario", "Raul"],
             ["Mario", "Ana"],
             ["Enrique", "Sarah"],
             ["Enrique", "Raul"],
             ["Saul", "Ana"]]
```

Creamos una función para leer el *input* y convertirlo a lista de conectores (edges) y etiquetas (labels):

[10]:
```
def input_to_data(inp):
    labels = {}
    edges = []
    element = -1
    for i in range(len(inp)):
        if inp[i][0] not in labels.values():
            element += 1
            labels[element] = str(inp[i][0])
```

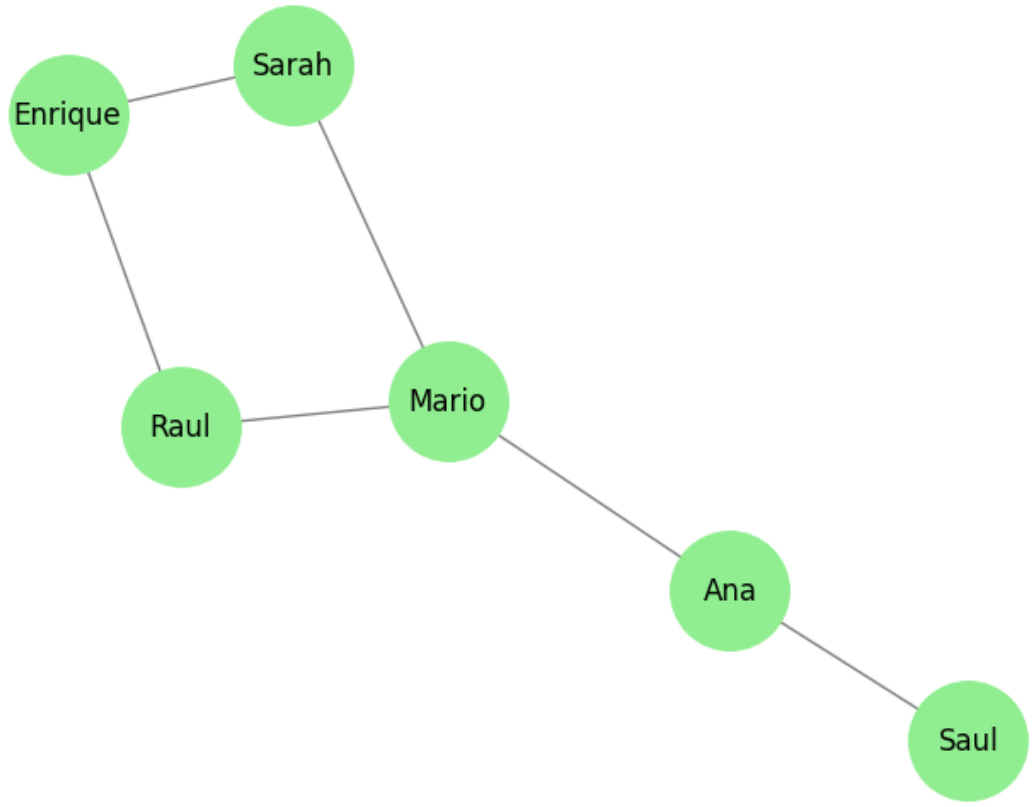
```
        if inp[i][1] not in labels.values():
            element += 1
            labels[element] = str(inp[i][1])

    labels_list = list(labels.values())
    for a in amistades:
        edges.append((labels_list.index(a[0]), labels_list.index(a[1])))

    return edges, labels
```

Ahora dibujamos el grafo que representa la red de contactos:

[11]:
```
edges, int2label = input_to_data(amistades)
graph = nx.Graph(edges)
nx.draw(graph, labels=int2label, with_labels=True, node_size=2500,␣
 ↪node_color='lightgreen', edge_color='grey')
plt.show()
```




Usamos la libería de *min vertex cover* para generar el Hamiltoniano de coste U(C) y el mixer U(B):

[12]:
```
cost_h, mixer_h = qaoa.min_vertex_cover(graph, constrained=False)

print("Cost Hamiltonian", cost_h)
print("Mixer Hamiltonian", mixer_h)
```

```
Cost Hamiltonian 0.75 * (Z(0) @ Z(1)) + 0.75 * Z(0) + 0.75 * Z(1) + 0.75 * (Z(0)
@ Z(2)) + 0.75 * Z(0) + 0.75 * Z(2) + 0.75 * (Z(0) @ Z(3)) + 0.75 * Z(0) + 0.75
* Z(3) + 0.75 * (Z(1) @ Z(4)) + 0.75 * Z(1) + 0.75 * Z(4) + 0.75 * (Z(2) @ Z(4))
+ 0.75 * Z(2) + 0.75 * Z(4) + 0.75 * (Z(3) @ Z(5)) + 0.75 * Z(3) + 0.75 * Z(5) +
-1.0 * Z(0) + -1.0 * Z(1) + -1.0 * Z(2) + -1.0 * Z(3) + -1.0 * Z(4) + -1.0 *
Z(5)
Mixer Hamiltonian 1 * X(0) + 1 * X(1) + 1 * X(2) + 1 * X(3) + 1 * X(4) + 1 *
X(5)
```

Vamos definir el bloque que consiste en aplicar la puerta $U(C, \gamma)$ y $U(B, \beta)$:

[13]:
```
def qaoa_layer(gamma, beta):
    qaoa.cost_layer(gamma, cost_h)
    qaoa.mixer_layer(beta, mixer_h)
```

Circuito que inicializa todas los estados a un estado de superposición equiprobable con puertas Hadamard, tambien aplicaremos la "layer" definida arriba un numero de veces concreto determinado por el parametro "depth".

[14]:
```
n_wires = range(len(graph))
depth = 3  # Definimos 3 capas tal y como manda el enunciado

def circuit(params, **kwargs):
    for w in n_wires:
        qml.Hadamard(wires=w)
    qml.layer(qaoa_layer, depth, params[0], params[1])
```

La función de coste que queremos minimizar es el valor esperado de "cost_h". utiliamos la función expval() que devuelve el valor esperado del Hamiltoniano respecto al estado del circuito.

[15]:
```
dev = qml.device("default.qubit", wires = n_wires)

@qml.qnode(dev)
def cost_function(params):
    circuit(params)
    return qml.expval(cost_h)
```

Para mejorar nuestro algortimo vamos a utilizar "GradientDescentOptimizer()" para optimizar los valores de $\gamma$ y $\beta$. Durante el transcurso del desarrollo de la solución hemos ido modificando los parámetros del número de pasos (steps) y los valores de partida de $\gamma$ (guess_g) y $\beta$ (guess_b) que nos dieran mejor resultado.

[35]:
```
optimizador = qml.GradientDescentOptimizer()
steps = 50
guess_g = 0.3
```

```
guess_b = 0.3
params = np.array([[guess_g, guess_g, guess_g], [guess_b, guess_b, guess_b]],␣
↪requires_grad=True)

for i in range(steps):
    params = optimizador.step(cost_function, params)

print("Parametros optimos: ", params)
```

```
Parametros optimos:  [[ 0.21788827  0.29934398 -0.25646866]
 [-0.25812299  0.65982112  0.68241898]]
```

Creamos el algortimo con estos parametros optimos, pero primero añadimos una función que nos ayudará a interpretar el resultado.

[36]:
```
# Obtenemos el valor de la máxima probabilidad y su estado correspondiente

def get_max_prob(probs):
    max_prob = probs[0]
    max_index = 0
    for i, prob in enumerate(probs):
        if prob > max_prob:
            max_prob = prob
            max_index = i

    state =  bin(max_index)[2:].zfill(len(n_wires))
    return max_prob, state
```

Ejecutamos el circuito que nos dará las probabilidades de cada estado y obtenemos el estado más probable

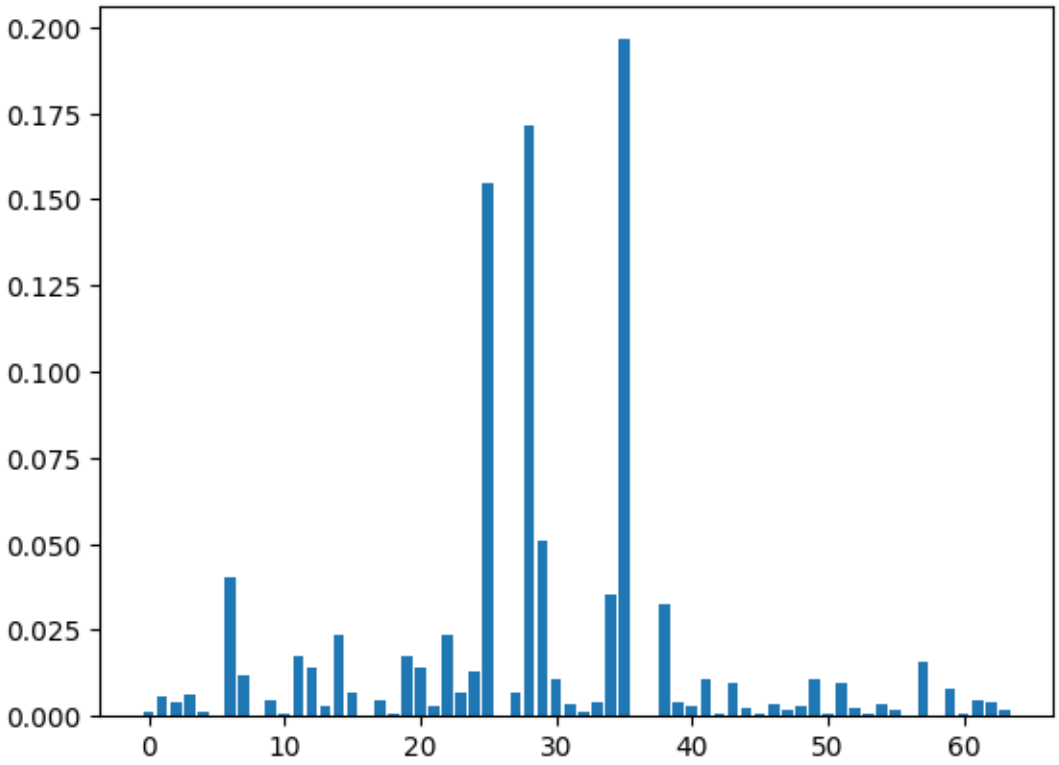
[37]:
```
@qml.qnode(dev)
def probaility_circut (gamma, beta):
    circuit([gamma, beta])
    return qml.probs(wires=n_wires)


probs= probaility_circut(params[0], params[1])

prob, state = get_max_prob(probs)
print ("Estado más probable es", state,", con una probabilidad de:", prob)
```

```
Estado más probable es 100011 , con una probabilidad de: 0.19644374726886943
```

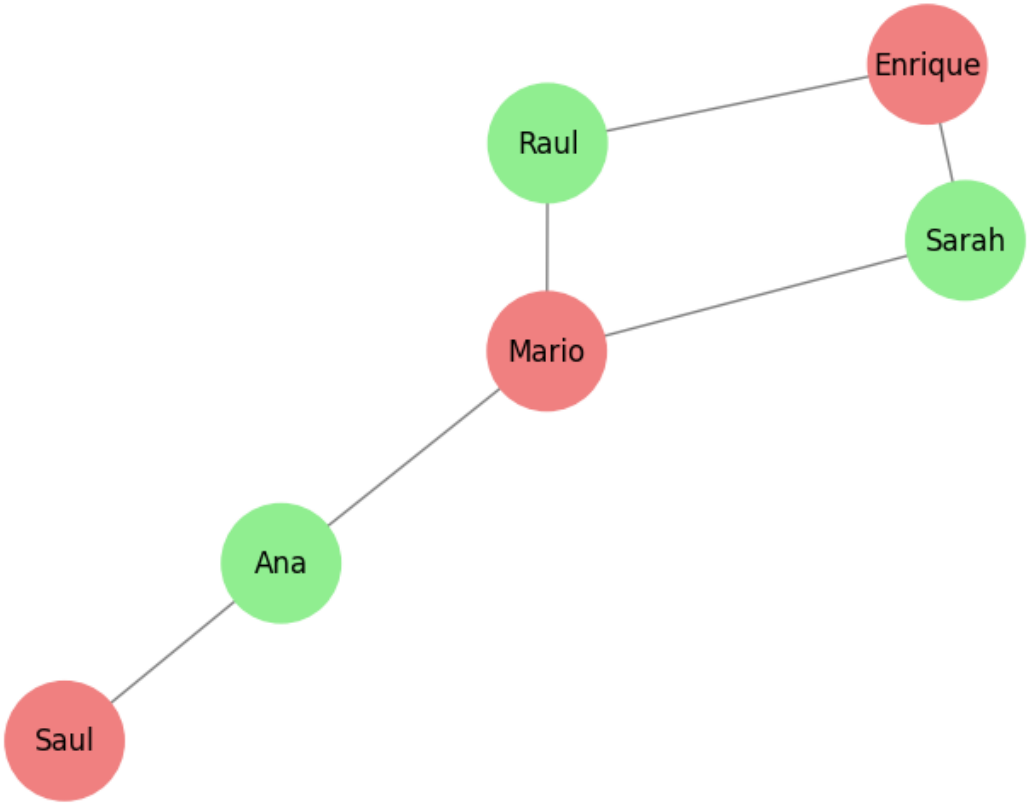
[38]:
```
plt.bar(range(2 ** len(n_wires)), probs)
plt.show()
```




El estado de máxima probabilidad se asocia al siguiente grafo. Los influencers a los que hay que pagar son los que están marcados en rojo.

[42]:
```
colors = []
for b in list(state):
    colors.append("lightgreen" if b == '0' else "lightcoral")

nx.draw(graph, labels=int2label, node_color=colors, with_labels=True,␣
 ↪node_size=2500, edge_color='grey')
```




El resultado obtenido es que tenemos que pagar a Saúl, a Marío y a Enrique.