Universidad Nebrija

Escuela Politécnica Superior

Máster Universitario en Computación Cuántica

Modelos de programación

Coloreado de mapas en D-Wave

Guillermo Fuentes Morales

gfuentesm@alumnos.nebrija.es

Fecha: 18 de enero de 2022

Resumen

En esta práctica vamos a resolver el problema del coloreado de mapas en los ordenadores cuánticos de D-Wave. Este se basa en ser capaz de colorear un mapa sin que 2 provincias limítrofes tengan el mismo color. Plantearemos el ejercicio como un ejercicio de grafos con sus correspondientes restricciones y lo resolveremos tanto con la topología Quimera como con Pegasus. Después de unas cuantas ejecuciones guardaremos los resultados y los analizaremos para ver cual de las topologías se adapta mejor a nuestro problema.

En este caso es un problema no muy grande pero lo suficiente para ver como se plantearían otros más grandes y que "solver"sería mejor escoger para otros problemas parecidos al nuestro.

Palabras clave: D-Wave, coloreado de mapas, computación cuántica, grafos.

Abstract

In this practice we are going to solve the map coloring problem in D-Wave quantum computers. This is based on being able to color a map without 2 bordering provinces having the same color. We will pose the exercise as a graph exercise with its corresponding restrictions and we will solve it both with the Chimera topology and with Pegasus. After a few runs we will save the results and analyze them to see which of the topologies best suits our problem.

In this case, it is not a very big problem, but enough to see how other bigger ones would arise and which "solver"would be better to choose for other problems similar to ours.

Keywords: D-Wave, map coloring, quantum computing, graphs.

Índice

Índice de figuras

Índice de código

1. Introducción

Hemos enfocado esta práctica a resolver el problema del coloreado de mapas, en este caso vamos a usar como ejemplo el mapa de comunidades autónomas de España.

Para ello hemos realizado una lista con todas las comunidades autónomas y la relación que existen entre ellas. Enfocaremos el problema como si de un ejercicio de nodos se tratase, donde cada comunidad es un nodo y las aristas las comunidades con las que están relacionadas y añadiremos 2 restricciones, una de que cada comunidad solo puede tener un color y otra para indicar que 2 comunidades colindantes no pueden tener igual color.

Finalmente, le pasaremos el problema a los 2 solver de D-Wave tanto el de arquitectura Pegasus como el de Quimera. El "embedding" debido al número de variables existentes y la cantidad de relaciones resulta muy dificil hacerlo manual asique lo realizará automáticamente el propio sistema de D-Wave.

2. Procedimiento

2.1. Lista de comunidades autónomas:

Lo primero a realizar para este problema es obtener las 17 comunidades autónomas y las 2 ciudades autónomas de Ceuta y Melilla y sus correspondientes relaciones. A la hora de pasar esto a Python creamos 2 listas una con todas las comunidades y otra con las tuplas de las relaciones de vecindad que existan entre comunidades, por ejemplo Galicia tendría una tupla con Asturias y otra con Castilla y León. Además, para que a la hora de representar el gráfico quede más o menos organizado hemos asignado unas posiciones por defecto a cada nodo. Como se observa en el código 12.

2.2. Restricciones

"Teorema de los 4 colores: Dado cualquier mapa geográfico con regiones continuas, este puede ser coloreado con cuatro colores diferentes, de forma que no queden regiones adyacentes con el mismo color."Wikipedia (2021)

Luego basandonos en este Teorema aplicaremos una restricción que indique que cada comunidad puede ser pintada de 4 colores pero solo se seleccionará uno. Cabe destacar que tanto para esta como la siguiente restricción indicaremos los colores como (0,1,2,3) *Map Coloring* (s.f.). Añadiremos una restricción para cada comunidad.

```
one_color_configurations = {(0, 0, 0, 1), (0, 0, 1, 0), (0, 1, 0, 0),
   (1, 0, 0, 0)}
for ca in ccaa:
    variables = [ca+"_"+str(i) for i in range(len(
   one_color_configurations))]
    print(variables)
```

```
    #Y add a las restricciones de que solamente puede tener un color
    csp.add_constraint(one_color_configurations, variables)
```

Código 1: Creación de la restricción para que solo pueda tener un color

La siguiente restricción es la de los vecinos, que indicamos que 2 comunidades colindantes no pueden tener el mismo color. Es decir, crearemos una función que indique que no puede ser esa tupla (Ambas comunidades del mismo color). Añadiremos una restricción 4 veces (Por los 4 colores) para cada vecino.

```
def different_colours(ca1, ca2):
    return not (ca1 and ca2)
for neighbour in neighbours:
    ca1, ca2 = neighbour
    for i in range(len(one_color_configurations)):
        variables = [ca1+"_"+str(i), ca2+"_"+str(i)]
        csp.add_constraint(different_colours, variables)
```

Código 2: Creación de la restricción para que 2 vecinos no tenga el mismo color

2.3. Resolución de BQM

Una vez ya tenemos todas nuestras restricciones, solo tenemos que seleccionar que transformarlo en BQM y enviarselo al sampler que resolverá nuestro problema.

```
bqm = dwavebinarycsp.stitch(csp)
if(QUIMERA_PEGASUS):
    sampler = EmbeddingComposite(DWaveSampler(solver='DW_2000Q_6'))
else:
    sampler = EmbeddingComposite(DWaveSampler(solver={'topology__type':
    'pegasus'}))
#Realizamos 100 lecturas
response = sampler.sample(bqm, num_reads=100)
```

Código 3: Selección del sampler y resolución

Una vez tenemos la respuesta a nuestro problema, seleccionamos la de más baja energía y comprobamos si la respuesta cumple las restricciones impuestas, de ser así, calculamos el número de respuestas con esa mínima energía para ver cuantas soluciones correctas ha encontrado el "solver"(Porque lógicamente el mapa tiene más de una solución correcta).

```
sample = response.first.sample
#Comprobamos si la de menor energia cumple con nuestras restricciones y
    si las cumple pintamos
if not csp.check(sample):
    print("Failed to color map")
```

```
else:
    print(response.first.energy)
    total_ok=0
    for energy, occurences in response.data(['energy', 'num_occurrences
    ']):
        if(energy>response.first.energy):
            break
        total_ok += occurences
    print("Ha habido un total de " +str(total_ok)+" soluciones
    correctas")
    print("Problema resuelto, guardando el mapa..")
    dwave.inspector.show(response)
    plot_map(sample)
```

Código 4: Comprobación de las soluciones del grafo

2.4. Creación del grafo:

Para la creación y manipulación del grafo vamos a usar la librería NetworkX. Lo primero que haremos será añadir todos los nodos que tienen conexión con otros, de esta manera no tenemos que introducir manualmente la relación existente entre ellos y finalmente añadir aquellos que no tienen ninguna relación. En este caso son las Islas Baleares, Islas Canarias y las ciudades autónomas mencionadas anteriormente.

```
G = nx.Graph(neighbours)
#Obtenemos las comunidades sin vecinos
lone_nodes = set(ccaa) - set(G.nodes)
for lone_node in lone_nodes:
    G.add_node(lone_node)
```

Código 5: Creación del grafo

Ahora deberemos separar el color(0,1,2,3) que habiamos asignado a la comunidad de su nombre y se lo añadiremos a los datos del grafo y lo dibujamos.

```
color_labels = [k for k, v in sample.items() if v == 1]
for label in color_labels:
    name, color = label.split("_")
    color = int(color)
    G.nodes[name]["color"] = color
color_map = [color for name, color in G.nodes(data="color")]
nx.draw_networkx(G, with_labels=True, pos = node_positions,
    node_color=color_map, font_color="w", node_size=500)
```

Código 6: Dibujo del grafo

2.5. Analisis de los resultados

Para analizar los resultados hemos realizado varias ejecuciones y guardado los siguientes datos en formato JSON de cada una de ellas, el "solverüsado, el tiempo de acceso a la QPU, el número de qubits fisicos que se ha usado durante el embedding y el número de soluciones fallidas.

Una vez almacenados los datos, los hemos analizado en un cuaderno de Jupyter aparte. Lo primero es realizar la lectura de estos.

```
#Obtenemos los datos del archivo Python
with open(FILE) as f:
    data_ccaa = json.load(f)
for i in data_ccaa:
    architecture =i['architecture']
    time=i['qpu_time']
    num_sol=i['num_sols']
    qubits=i['qubits']
    if(architecture=='Quimera'):
        total_quimera +=1
        total_quimera_time+=time
        total_quimera_sols+=num_sol
        qubits_sol_quimera.append((qubits,num_sol))
    else:
        total_pegasus +=1
        total_pegasus_time+=time
        total_pegasus_sols+=num_sol
        qubits_sol_pegasus.append((qubits,num_sol))
```

Código 7: Lectura de los datos del JSON

Lo primero que vamos a comparar es si existe una relación lineal entre el número de Qubits sobre los que se ha embebido y el número de soluciones obtenidas para ambas arquitecturas.

```
#Representamos los datos de Pegasus y calculamos su regresion lineal
if(total_pegasus>0):
    y= [i[0] for i in qubits_sol_pegasus]
    x= [i[1] for i in qubits_sol_pegasus]
    stats_linregress = stats.linregress(x, y)
    plt.plot(x,y,'o',label='Qubits/n soluciones')
    plt.plot(x,stats_linregress.intercept + stats_linregress.slope*np.
   array(x), 'r', label='fitted line')
    plt.ylabel("Qubits")
    plt.xlabel("Soluciones")
    plt.legend(loc='best')
    plt.show()
```

```
    print("Valor de r cuadrado en pegasus : "+str(stats_linregress.
    rvalue))
```

Código 8: Regresión lineal del número de Qubits físicos y soluciones aportadas de Pegasus

Lo siguiente que vamos a comparar es el tiempo de acceso a la QPU de cada "solver".

```
if(total_pegasus>0):
#Comparamos el tiempo de acceso a la QPU
if(total_quimera == 0 or total_pegasus == 0):
    print("No hay datos de alguno de los solvers luego no se puede
    comparar")
else:
    avg_time_quimera=total_quimera_time/total_quimera
    avg_time_pegasus=total_pegasus_time/total_pegasus
    print("El tiempo de uso de la QPU de Quimera es "+str(round(
    avg_time_quimera/avg_time_pegasus,2))+" veces en comparacion con
    pegasus")
```

Código 9: Analisis del tiempo de acceso a la QPU

También compararemos el número de soluciones de media que encuentran al programa cada "solver".

```
#Comparamos el numero de soluciones apoortadas
if(total_quimera == 0 or total_pegasus == 0):
    print("No hay datos de alguno de los solvers luego no se puede
    comparar")
else:
    avg_sols_quimera=total_quimera_sols/total_quimera
    avg_sols_pegasus=total_pegasus_sols/total_pegasus
    print("Quimera da un total de  "+str(round(avg_sols_quimera/
    avg_sols_pegasus,2))+" soluciones en comparacion con pegasus")
if(total_quimera==NUM_EXE):
    print("Quimera encontro soluciones correctas en todas las
    ejecuciones")
else:
    print("Quimera no encontro soluciones al problema en "+ str(NUM_EXE
    -total_quimera)+" veces")
if(total_pegasus==NUM_EXE):
    print("Quimera encontro soluciones correctas en todas las
    ejecuciones")
else:
    print("Quimera no encontro soluciones al problema en "+ str(NUM_EXE
    -total_pegasus)+" veces")
```

Código 10: Analisis del número de soluciones aportadas

Finalmente, compararemos el número de qubits fisicos de media que se han usado para embeber el problema.

```
#Comparamos la cantidad de Qubits fisicos usados para el embedding
if(total_quimera == 0 or total_pegasus == 0):
    print("No hay datos de alguno de los solvers luego no se puede
   comparar")
else:
    qubits_quimera = sum([i[0] for i in qubits_sol_quimera])
    qubits_pegasus = sum([i[0] for i in qubits_sol_pegasus])
    avg_qubits_quimera=qubits_quimera/total_quimera
    avg_qubits_pegasus=qubits_pegasus/total_pegasus
    print("Quimera usa "+str(round(avg_qubits_quimera/
   avg_qubits_pegasus,2))+" qubits en comparacion con pegasus")
```

Código 11: Analisis del número de Qubits usados

3. Resultados

Para hacer una comparación de los resultados obtenidos vamos a usar el Notebook explicado anteriormente. Hemos ejecutado el programa con 200 lecturas y 50 ejecuciones con cada "solver". Probamos con menos lecturas pero de esas 50 ejecuciones ocurrían más fallos al pintar el mapa y con 200 el número de fallos se redujo significativamente, solo hubo 1 al usar la topología Quimera.

Lo primero que observamos es que la correlación entre la forma de embeber el problema y el número de soluciones es bastante baja. Es algo mayor en el caso de la topología de Quimera pero sigue sin ser relevante. (2) y (3)

Luego, podemos observar que el tiempo de uso de la QPU es 1,52 veces mayor en el caso de la topología Quimera, teniendo una media de $59904\mu s$ y $39378\mu s$ en el caso de Pegasus. (4)

Respecto a las soluciones del problema, el "solver"Quimera a pesar de fallar una vez aportó de media 9 soluciones mientras que Pegasus solo 4. (5)

Finalmente, como habiamos estudiado teoricamente, el número de Qubits físicos usados por Quimera es muy superior respecto a Pegasus siendo este 321 en contraposición a 151, lo que significa que son 2,1 veces más los qubits que necesita Quimera para embeber el problema. Esta parte además se puede observar muy facilmente a través de la herramienta de OASIS "Problem Inspector". (6), (7) y (8)

4. Conclusiones

Con los resultados analizados, parece más lógico hacer uso de la topología de Pegasus en vez de la Quimera, principalmente porque no ha fallado ninguna vez y como para este problema solo necesitamos una solución nos da un resultado igualmente valido en un tiempo de acceso a la QPU menor.

La realización de este ejercicio nos ha servido para aprender a como modelar un problema algo más complicado en formato CSP, además de poder comparar los 2 procesadores cuánticos principales de D-Wave, aunque la diferencia no haya sido muy notable para este problema, podemos observar que Pegasus a pesar de encontrar menos soluciones nos ofrece un rednmiento más rápido que es lo que necesitariamos para este problema.

Hubiera sido bueno, probar con otros nodos algo más grandes y ver si el rendimiento de este último sigue siendo superior. Por otro lado, podemos concluir que para otro tipo de problemas en el que nos interese que se encuentren todas las soluciones validas en vez de 1 sola, Quimera sería bastante superior.

En conclusión, cada "solver"tienes sus cualidades como queda claramente reflejado en el analisis de los resultados, además nos ha permitido profundizar un poco más en la creación de problemas de tipo CSP. El mapa de España obtenido es el siguiente:

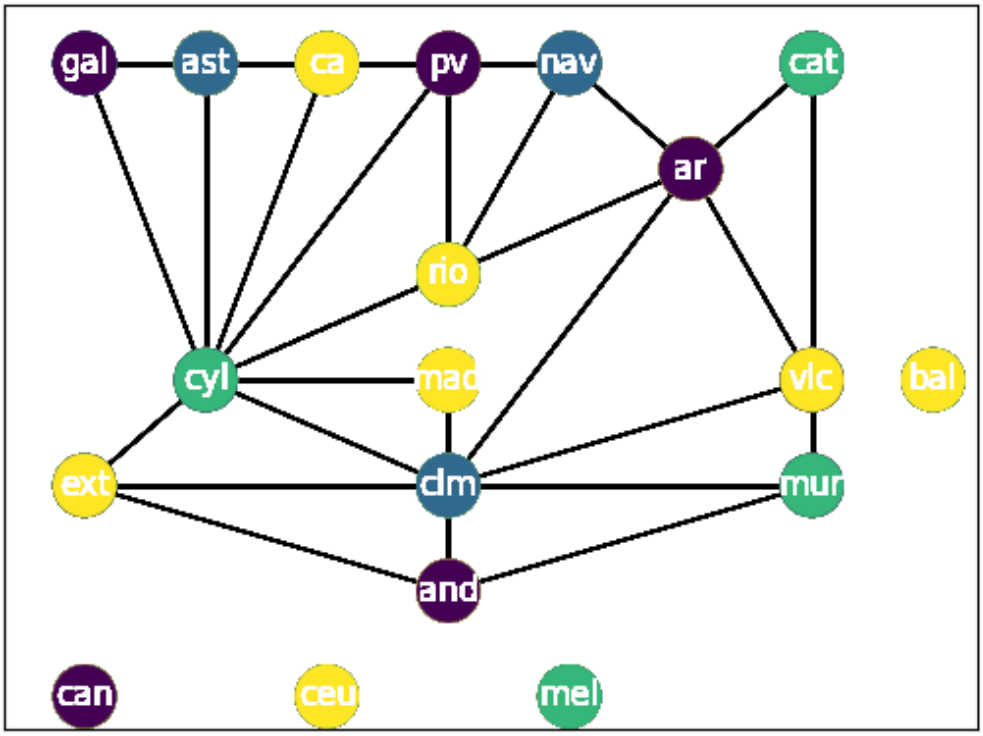


Figura 1: Mapa de españa

Referencias

Map coloring. (s.f.). Descargado de `https://docs.ocean.dwavesys.com/en/latest/examples/map_coloring.html` ([Internet; descargado 16-enero-2022])

Wikipedia. (2021). *Teorema de los cuatro colores — wikipedia, la enciclopedia libre.* Descargado de `https://es.wikipedia.org/w/index.php?title=Teorema_de_los_cuatro_colores&oldid=139558867`

A. Código

```
#Ponemos las posiciones del nodo para representar el mapa de espana de
    una manera mas organizada
node_positions = {"can": (0, 0),
                  "ceu": (2, 0),
                  "mel": (4, 0),
                  "and": (3, 1),
                  "ext": (0, 2),
                  "clm": (3, 2),
                  "mur": (6, 2),
                  "cyl": (1, 3),
                  "mad": (3, 3),
                  "rio": (3, 4),
                  "ar": (5, 5),
                  "vlc": (6, 3),
                  "cat": (6, 6),
                  "bal": (7, 3),
                  "nav": (4, 6),
                  "pv": (3, 6),
                  "ca": (2, 6),
                  "ast": (1, 6),
                  "gal": (0,6)}
#Lista de CCAA
ccaa = ['can','and','ext','clm','cyl','mur','vlc','mad','ar',
'cat','rio','pv','nav','ca','ast','gal',
'bal','ceu','mel']
#Lista de vecinos
neighbours = [('and','ext'),('and','clm'),('and','mur'),
('ext','cyl'),('clm','ext'),('clm','mad'),('clm','mur'),('clm','vlc'),
('clm','ar'),('clm','cyl'),('vlc','mur'),('cyl','mad'),('cyl','ar'),
('cyl','rio'),('cyl','pv'),('cyl','ca'),('cyl','ast'),('cyl','gal'),
('cat','ar'),('nav','ar'),('rio','ar'), ('rio','nav'),
('rio','pv'),('pv','nav'),('pv','ca'),('ca','ast'),
('ast','gal'),('ar','vlc'),('cat','vlc')
]
```

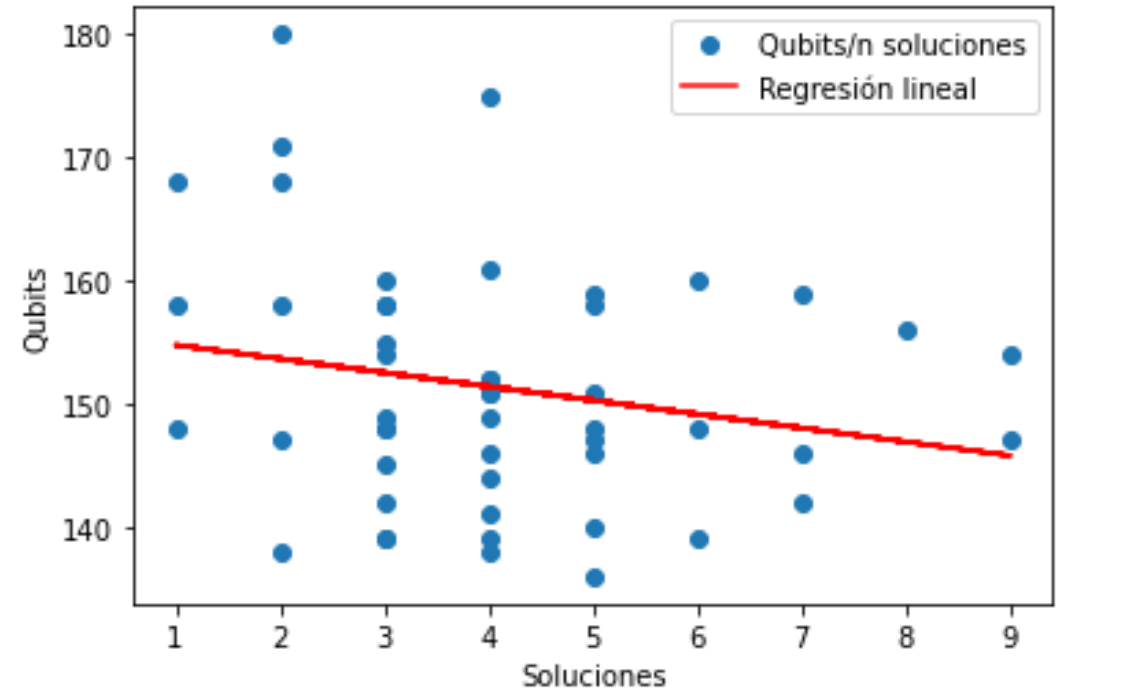
Código 12: Lista de comunidades autónomas

```
############ CONSTANTES ############
#Ruta al archivo de los datos
FILE="restricciones/data_ccaa.json"
NUM_EXE=50
############ VARIABLES ############
```

```
total_quimera=0
total_pegasus=0
total_quimera_time=0
total_pegasus_time=0
qubits_sol_quimera=[]
qubits_sol_pegasus=[]
total_quimera_sols=0
total_pegasus_sols=0
```

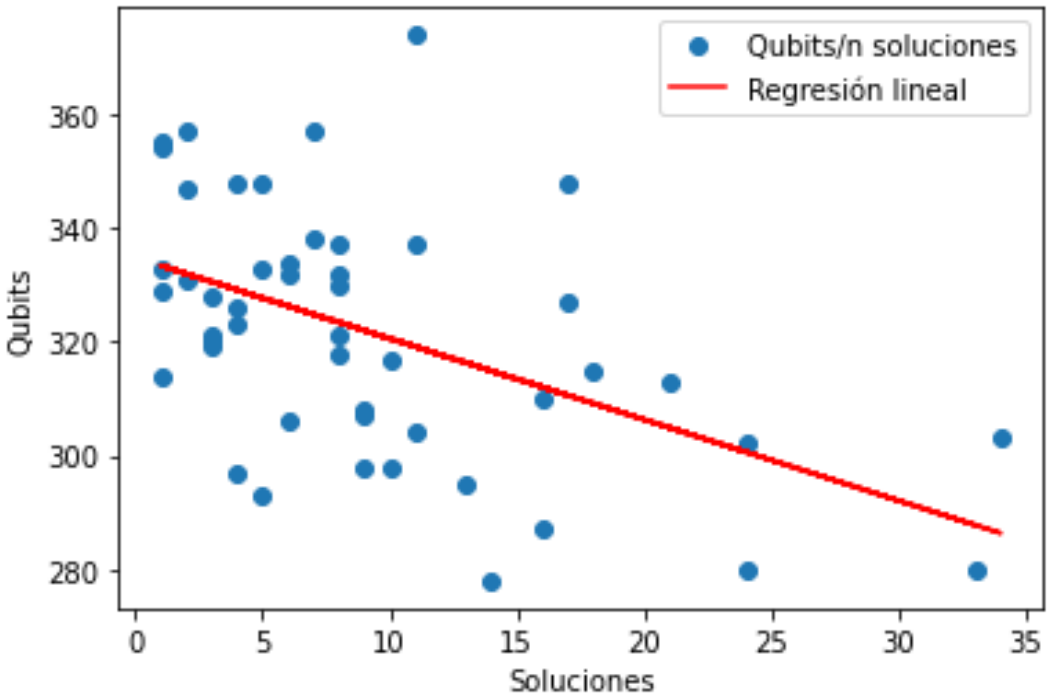
Código 13: Constantes y variables del Notebook

B. Figuras



Valor de r cuadrado en pegasus : -0.21330810740545303

Figura 2: Regresión lineal Pegasus




```
Valor de r cuadrado en quimera : -0.5039151381428222
```

Figura 3: Regresión lineal Quimera

```
Tiempo medio de Quimera (micro s) 59904.30612244898
Tiempo medio de Pegasus (micro s) 39378.38
El tiempo de uso de la QPU de Quimera es 1.52 veces en comparacion con pegasus
```

Figura 4: Tiempo medio de acceso a la QPU

```
Soluciones medias de Quimera 9.244897959183673
Soluciones medias de Pegasus 4.08
Quimera da un total de  2.27 soluciones en comparacion con pegasus
Quimera no encontro soluciones al problema en 1 veces
Pegasus encontro soluciones correctas en todas las ejecuciones
```

Figura 5: Número de soluciones medias

```
Qubits fisicos medias de Quimera 321.6734693877551
Qubits fisicos de Pegasus 151.32
Quimera usa 2.13 qubits en comparacion con pegasus
```

Figura 6: Número de qubits físicos medios

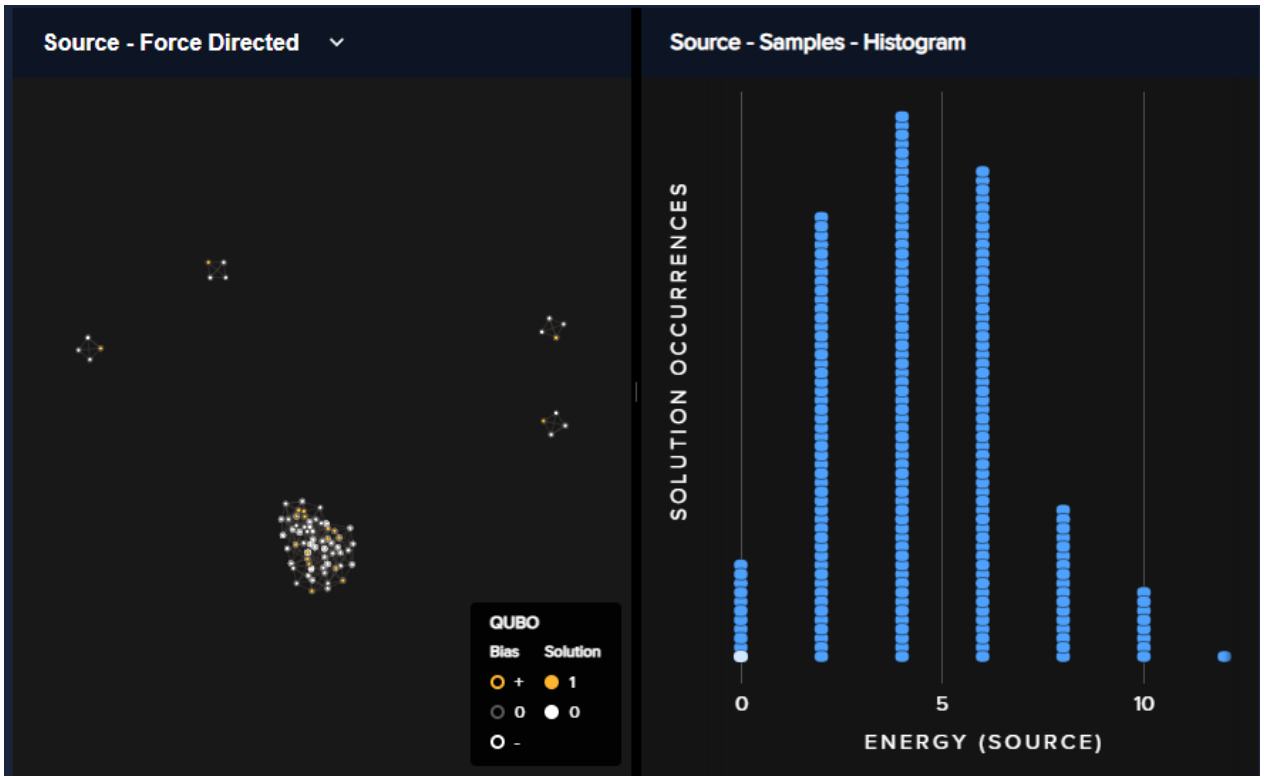


Figura 7: Problem inspector de Quimera

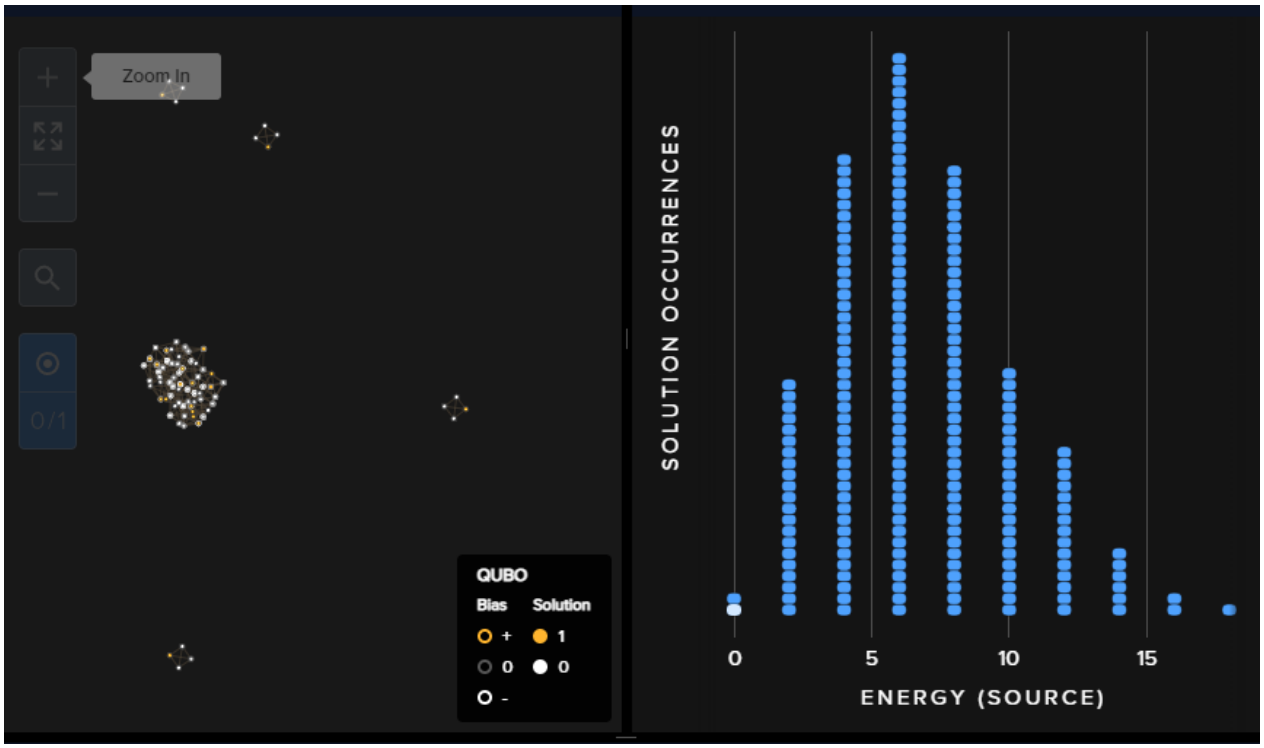


Figura 8: Problem inspector de Pegasus