QA - Quantum Annealing - Dwave 101

V2.1, Julio 2023

Estructura SDK Ocean y sus paquetes

https://docs.ocean.dwavesys.com/en/stable/

Documento referencial fundamental:

https://support.dwavesys.com/hc/en-us/articles/360045654674-Which-Solver-Sampler-Should-I-Use-

Documentos iniciales esencial sobre solvers, tipos etc:

https://docs.dwavesys.com/docs/latest/doc_getting_started.html

https://docs.dwavesys.com/docs/latest/c_solver_parameters.html#:~:text=1000%2C%20**reverse_anneal_params)-,annealing_time,returned%

Parte I

Arquitectura software Dwave/Ocean

Modelos: Ising,Quobo, BQM y CQM

Durante la primera parte de este tutorial se van a desplegar los recursos computacionales, tanto clásicos como cuánticos para plantear y resolver problemas de optimización, de dependencia lineal y/o cuadrática entre sus variables.

A la hora de abordar esta tarea existen 2 modelos de variable binaria bien diferenciados: el modelo Ising, con asignaciones {-1,+1}, también conocidos como problemas de variable spin, y el modelo Quobo, con asignaciones {0,1}. Ambos modelos son cuadráticos, es decir, contemplan relaciones cuadráticas entre sus variables.

El modelo Ising es inherente a una arquitectura cuántica porque la función de coste a minimizar, un hamiltoniano en realidad, se sintetiza como sumandos de productos tensoriales de puertas de pauli - {x},{I} y {Z} -, bloques fundamentales de cualquier computador cuántico.

El modelo Qubo en cambio es habitual en las ciencias de la computación como estrategia para resolver problemas de programación lineal y cuadrática de variables binarias {0,1}.

Por fortuna existe una equivalencia matemática (excepto un factor de offset) entre los modelos Ising y Quobo, por lo que, en principio, cualquier problema clásico Qubo, puede replantearse fácilmente para un computador cuántico. Se dice que ambos son modelos cuadráticos binarios o `BQM`

También, como es sabido, los problemas de programación (de variable binaria, discreta o contínua) añaden a la función objetivo una o varias ecuaciones con restricciones para sus variables, expresadas como igualdades o desigualdades.

Estas ecuaciones no siempre se pueden satisfacer en su totalidad, bien por la inherencia probabilística de la computación cuántica, la decoherencia en sus estados, o simplemente por las limitaciones en el número de iteraciones de la algoritmia. En estos casos la solución propuesta será la que arroje una función objetivo mínima, con el mayor número de restricciones satisfechas.

Ahora bien, el formalismo Qubo no contempla restricciones (de ahí su nombre). Lo mismo le sucede al modelo Ising. En ambos casos se habla de modelos BQM o cuadráticos binarios sin restricciones.

Con la finalidad de generalizar estos modelos a los problemas con restricciones, se ha recurrido tradicionalmente a la siguiente estratagia

- convertir las restricciones con desigualdades en igualdades mediante variables auxiliares (slack)
- incorporar estas restricciones a la función objetivo de los modelos BQM como cuadrados (para evitar sesgos debido al sigo) de sumandos que actuán como penalizaciones cuando se incumplen las restricciones.
- Usar un factor multiplicador, factor de Lagrange, para cada sumando añadido, y que amplifique su aporte a la función objetivo cuando no se cumple la restricción. El ajuste de este factor es heurístico y juega un papel crítico.

Bien, esta es la esencia de los modelos cuadráticos con restricciones, `CQM`, y de sus variantes de variable discreta o contínua.

Ejemplo conversión CQM-BQM

Problema:

Función objetivo: f(x,y)=-3xy + x

Restricción: x+y=1

Modelo CQM:

Función objetivo: f(x,y)=-3xy + x + 10(x+y-1)^2

La penalización P(x,y)= 10(x+y-1)^2 incluye el factor de lagrange 10. Cuando x+y=1, P(x,y)=0 y no interviene en el proceso de optimización de la función objetivo.

`solvers` y `samplers` en Dwave/Ocean

El annealing cuántico en el que se fundamentan los QPU de Dwave, está basado en la computación adiabática, con un hamiltoniano inicial de fácil preparación, y al que se hace evolucionar en el tiempo hacía el hamiltoniano problema, siendo la función de coste a minimizar, la energía de este hamiltoniano problema. En un annealer el principio de evolución adiabática solo es inspiradora, por lo que no está garantizada que el estado final corresponda al de mínima energía del problema planteado.

En el siguiente enlace se desarrolla un poco más esta base fundacional.

https://docs.dwavesys.com/docs/latest/c_qpu_annealing.html

En D-Wave, un `solver` es una abstracción de alto nivel que abarca los componentes hardware y software necesarios para resolver un problema de optimización determinado. Consiste en una colección de herramientas y algoritmos que se utilizan para mapear el problema de optimización en el hardware del annealer.

El solver encapsula todo el proceso de formulación de la función de coste del problema,y de sus restricciones, brindando un interfaz que permita a los usuarios ingresar el problema de una manera que se pueda luego transpilar en el annealer cuántico.

Los solvers también manejan varios detalles técnicos, como configurar la conexión con el annealer, envío a la nube para su ejecución y recuperación de los resultados.

Por otro lado, un `sampler` es un componente de bajo nivel que interactúa directamente con el hardware del annealer. Toma un problema como entrada y devuelve muestras de la distribución de posibles soluciones. Un sampler no realiza ninguna formulación o interpretación del problema de alto nivel, ni aplica ninguna restricción a las soluciones que devuelve. Simplemente toma un problema en un formato

particular, lo mapea en el annealer cuántico, lo ejecuta n `reads`, equivalente a los `runs`, y devuelve el conjunto de muestras o `samples`, que son las configuraciones de los qbits-variables que se estiman ser soluciones al problema.

En Dwave un ´sampler´ acepta un problema formulado según un `modelo cuadrático binario` (BQM), -mediante variables binarias-, `modelo cuadrático discreto` (DQM) -con variables discretas-, o, desde 2022, incluso como un `modelo cuadrático contínuo` (CQM), permitiendo variables reales.

En los tres casos devuelve asignaciones de variables para dicho problema. Los ´samplers´ generalmente intentan encontrar valores discretos de minimización de la función objetivo, pero también pueden muestrear distribuciones definidas por el problema.

Nota: el siguiente pdf sobre el `Hybrid Solver Service` (HSS) introduce los modelos BQM,DQM y CQM:

https://www.dwavesys.com/media/soxph512/hybrid-solvers-for-quadratic-optimization.pdf

La clase de referencia es `DWaveSampler`, que puede usar tanto solvers físicos como simuladores. En la documentación de `samplers` que se indican a continuación se pueden consultar otras clases asociadas a los recursos de `samplers`.

La adaptación del problema al hardware qpu, o transpilación, se denomina `embedding` o incrustación en este ecosistema.

Documentación específica:

Getting Started oficial:

https://docs.dwavesys.com/docs/latest/doc_getting_started.html

Parámetros Solvers:

https://docs.dwavesys.com/docs/latest/doc_solver_ref.html

Samplers

https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/samplers.html

Algoritmos de grafos proporcionados

https://docs.ocean.dwavesys.com/projects/dwave-networkx/en/latest/reference/algorithms/index.html

Instalación SDK Ocean

In [1]:
```
# pip install dwave-ocean-sdk==6.0.1
```

Configuración de la cuenta. Mejor hacerlo desde consola para mayor claridad. Activar previamente env ocean!

Solo es necesario la primera vez

Documentación de la API de acceso a Dwave:

https://docs.ocean.dwavesys.com/en/stable/overview/sapi.html

In [2]:
```
# !dwave config create -> mejor desde consola
```

Biblioteca `dimod`

En la jerarquía de la arquitecturas sofware Dwave/Ocean, que se puede consultar aquí:

https://docs.ocean.dwavesys.com/en/stable/

la biblioteca `dimod` juega un papel fundamental ya que es una especie de parser entre el problema de usuario y los recursos de computación, tanto clásicos como cuánticos.

Efectivamente, consultando el siguiente documento:

https://docs.ocean.dwavesys.com/en/stable/docs_dimod/index.html

Se dice que dentro del SDK Ocean, la biblioteca `dimod` es una API para samplers, proporcionado por ejemplo la clase `BQM` (binary quadratic model) que permite plantear problemas de optimización usando:

- Modelo Ising para variables {-1,1}
- Modelos Qubo, tanto cuadráticos como de orden superior!- para variables {0,1}

`dimod` también permite usar varios samplers para probar y depurar los algoritmos con cpu local, y que no consumen por tanto tiempo-máquina:

Una lista de los mismos:

https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/sampler_composites/samplers.html

Pero especialmente, estamos interesados en samplers cuánticos reales (QPUs), para los que también actúa como interfase:

https://docs.ocean.dwavesys.com/en/stable/docs_system/reference/samplers.html

Dwave y los modelos Ising y Qubo

Se empezará esta inmersión al ecosistema Ocean/Dwave con un problema Ising.

Un problema Ising es un problema BQM que asigna valores {-1,+1} a cada variable del problema. En su adaptación a la QC estos valores los vincularemos a la base computacional. Así el valor +1 lo asociaremos a un cúbit en estado |0> y un valor -1 al estado |1>.

De este modo, si un problema Ising de dos variables, v1 y v2, minimiza su energía para los valores {v1,v2}={-1,-1}, esto equivaldrá en formalismo cuántico al estado |q1q2>=|11>= [0001]

El modelo Ising se inspira en el hamiltoniano ising de la física cuántica, exitoso a la hora de explicar el ferromagnetismo. Un recordatorio del mismo.

https://en.wikipedia.org/wiki/Ising_model

Por otro lado, los problemas QUBO tienen su origen en las ciencias de la computación, para plantear y resolver problemas de optimización de variables binarias {0} y {1}.

Ambos enfoques son modelos cuadráticos binarios (BQM), **equivalentes**, y la conversión entre ellos es trivial.

En el siguiente documento se explica su impronta en el ecosistema Dwave.

https://docs.dwavesys.com/docs/latest/c_gs_3.html#getting-started-concepts

Observación importante:

Tal como recoge el documento anterior, es habitual en QC que el signo (-) que precede a los sumatorios de las componentes longitudinales y transversales del formalismo Ising queda codificado en los propios parámetros Jij y hi. Bajo esta consideración:

Parámetro de acoplamiento cuadrático J:

. $\mathbf{J} < 0$: Acoplamiento ferromagnético; los qubits acoplados tienden a estar en el mismo estado, (1,1) o (–1,–1)

. $\mathbf{J} > 0$: Acoplamiento antiferromagnético; los qubits acoplados tienden a estar en estados opuestos, (–1,1) o (1,–1)

. $\mathbf{J} = 0$: Sin acoplamiento; Los estados de qubit no se afectan entre sí.

Parámetro de acoplamiento lineal h:

Debido a que el annealing cuántico minimiza la función de energía del hamiltoniano y hi es el sesgo de la variable i, los estados vi que contribuyen a la minimización energética son aquellos de signo contrario al del sesgo. Así, para hi =-1, el estado vi que favorece la minimización es vi=+1

La clase `sample_ising`

Heredado de la clase `sample()`, convierte un problema Ising en un modelo BQM y luego invoca a `sample()`.

- Parámetros
  - h (dict/list) – Sesgos lineales del problema de Ising. Si es un dict, debe tener la forma {v: sesgo, ...} donde `v` es una variable de espín (+1,-1) y el sesgo es su sesgo asociado. Si es una lista, se trata como una lista de sesgos donde los índices son las etiquetas de las variables.
  - J (dict[(variable i, variable j), bias]) – Sesgos cuadráticos del problema Ising.

La clase `sample_quobo`

De modo equivalente, este método convierte un modelo QUBO (0,1) a un modelo BQM, y luego invoca `sample()` -

- Parámetros
  - Parámetros - Q (dict): coeficientes de sesgos de la matriz Q de un problema QUBO. Es un dict de la forma {(i, j): aij, ...} donde i, j, son variables con valores binarios y aij es su coeficiente asociado.

Nota importante sobre `sample()`:

`sample()` se aplica estrictamente a un modelo BQM, que ha de ser compatible con la lista de nodos y de aristas de la geometría QPU que se esté usando.

El siguiente ejemplo de código permite obtener tales recursos para un sistema Dwave concreto, y posteriormente asignar con determinada estrategia nodos a qb_a, qb_b y qb_c.

In [3]:
```python
# Interactuando con la arquitectura física

from dwave.system import DWaveSampler
sampler = DWaveSampler()

# sampler.nodelist
# sampler.edgelist

qb_a = sampler.nodelist[0] # asigna primer nodo de la QPU
nodos_ady= iter(sampler.adjacency[qb_a]) #obtiene todos los adyacentes al nodo asignado a qb_a
qb_b = next(nodos_ady) #asigna primer adyacente a qb_b
qb_c = next(nodos_ady) #asigna segundo adyacente a qb_c

print(qb_a)
print(sampler.adjacency[qb_a])
print(qb_b,qb_c)
```

```
30
{2985, 2955, 45, 2970, 2940, 31}
2985 2955
```

Documentación oficial:

https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/generated/dwave.system.samplers.DWaveSampler.sample_ising.html#d

https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/generated/dwave.system.samplers.DWaveSampler.sample_qubo.html#d

https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/generated/dwave.system.samplers.DWaveSampler.sample.html#dwave.

Casos de estudio

Introducidos los conceptos fundamentales y las nociones básicas del ecosistema Dwave, a continuación se van a desplegar recursos clave de este SDK para resolver problemas germinales sencillos, usando diferentes modelos y técnicas.

Caso 1: Ising antiferromagnético sin campo longitudinal

Enlazando con la sección anterior, se trata de encontrar la energía mínima de un sistema binario usando el modelo Ising, y empezando por este caso, con sesgos h nulos.

In [4]:
```
# caso 1: Ising antiferromagnético sin campo longitudinal

from dwave.system import DWaveSampler, EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())
h = {'a': 0, 'b': 0}   # sin sesgo longitudinal favorecedor
J = {('a', 'b'): 1.5}  # J>0, favorece estados opuestos
result = sampler.sample_ising(h, J, num_reads=10)
print(result)
result.first.energy
```

```
   a  b energy num_oc. chain_.
0 -1 +1   -1.5       7     0.0
1 +1 -1   -1.5       3     0.0
['SPIN', 2 rows, 10 samples, 2 variables]
```

Out[4]:
```
-1.5
```

Caso 2: Ising antiferromagnético con campo longitudinal

In [5]:
```
# caso 2:  Ising antiferromagnético con campo longitudinal

from dwave.system import DWaveSampler, EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())
h = {'a': -1, 'b': 2}  # favorece el estado b=-1 (|1>)
J = {('a', 'b'): 1.5}  # J>0, favorece estados opuestos
result = sampler.sample_ising(h, J, num_reads=10)
```

```
print(result)
result.first.energy
```

```
   a  b energy num_oc. chain_.
0 +1 -1   -4.5       10     0.0
['SPIN', 1 rows, 10 samples, 2 variables]
-4.5
```

Out[5]:

Caso 3: Ising ferromagnético con campo longitudinal

In [6]:

```
# caso 3: Ising ferromagnético con campo longitudinal

from dwave.system import DWaveSampler, EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())
h = {'a': -1, 'b': 2}  # favorece el estado b=-1 (|1>)
J = {('a', 'b'): -2}  # J>0, favorece estados iguales
result = sampler.sample_ising(h, J, num_reads=10)
print(result)
result.first.energy
```

```
   a  b energy num_oc. chain_.
0 -1 -1   -3.0        9     0.0
1 +1 +1   -1.0        1     0.0
['SPIN', 2 rows, 10 samples, 2 variables]
-3.0
```

Out[6]:

Observamos que los resultados son consistentes con el argumentario inicial.

Problema BLP

A continuación se plantea un problema clásico de programación lineal binaria (BLP) con restricciones, y para cuya resolución se usa un enfoque BQM, pero con coeficientes cuadráticos nulos. Esta variante recibe el nombre de `CQM` (modelo cuadrático con restricciones).

Ocean dispone de una clase CQM, `ConstrainedQuadraticModel()`, con métodos tanto para resolver modelos en formalismo ising, `BinaryQuadraticModel.from_ising`, como qubo, `ConstrainedQuadraticModel().from_bqm`.

Si se prefiere, también se le puede declarar la función objetivo y las restricciones directamente con los métodos `set_objective` y `add_constrains`, alternativa que se sigue en el siguiente ejemplo.

Documentación:

https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/generated/dimod.ConstrainedQuadraticModel.from_quadratic_model.html

In [142…
```
# Modelo binario cuadrático con restricciones: CQM

# Definición de las variables binarias {0,1}

x0 = dimod.Binary("x0")
x1 = dimod.Binary("x1")
x2 = dimod.Binary("x2")
x3 = dimod.Binary("x3")
```

In [143…
```
#Definición de un problema de programación lineal binario (blp) con modelo CQM

blp = dimod.ConstrainedQuadraticModel() #Modelo CQM

blp.set_objective(-2*x0-3*x1+4*x2-x3)  # Función objetivo

blp.add_constraint(x0 + 2*x1 <= 2, "Primera restricción")
blp.add_constraint(x2 + x3 <= 1, "Segunda restricción")
```

Out[143]:
```
'Segunda restricción'
```

In [144…
```
print("Variables:")
print(blp.variables)
print("Objetivo:")
print(blp.objective)
print("Restricciones:")
print(blp.constraints)
```

```
Variables:
Variables(['x0', 'x1', 'x2', 'x3'])
Objetivo:
ObjectiveView({'x0': -2.0, 'x1': -3.0, 'x2': 4.0, 'x3': -1.0}, {}, 0.0, {'x0': 'BINARY', 'x1': 'BINARY', 'x2': 'BINAR
Y', 'x3': 'BINARY'})
Restricciones:
{'Primera restricción': Le(ConstraintView({'x0': 1.0, 'x1': 2.0}, {}, 0.0, {'x0': 'BINARY', 'x1': 'BINARY'}), 2.0), 'S
egunda restricción': Le(ConstraintView({'x2': 1.0, 'x3': 1.0}, {}, 0.0, {'x2': 'BINARY', 'x3': 'BINARY'}), 1.0)}
```

Veamos ahora

- Ensayos de soluciones adhoc
- Métricas de restricciones:
  - +m: la solución excede en +m unidades a la restricción
  - 0: la restricción se cumple en su igualdad
  - -m: la restricción se cumple con margen de m unidades
- Sampler de fuerza bruta

In [145…
```
sample1 = {"x0":1, "x1":1, "x2":1, "x3":1}
print("Muestra", sample1)
print("Su coste", blp.objective.energy(sample1))
print("Es factible?",blp.check_feasible(sample1))
print("Métrica restricciones:")
print(blp.violations(sample1))
```

```
Muestra {'x0': 1, 'x1': 1, 'x2': 1, 'x3': 1}
Su coste -2.0
Es factible? False
Métrica restricciones:
{'Primera restricción': 1.0, 'Segunda restricción': 1.0}
```

In [146…
```
sample2 = {"x0":0, "x1":0, "x2":1, "x3":1}
print("Muestra", sample2)
print("Su coste", blp.objective.energy(sample2))
print("Es factible?",blp.check_feasible(sample2))
print("Métrica restricciones:")
print(blp.violations(sample2))
```

```
Muestra {'x0': 0, 'x1': 0, 'x2': 1, 'x3': 1}
Su coste 3.0
Es factible? False
Métrica restricciones:
{'Primera restricción': -2.0, 'Segunda restricción': 1.0}
```

Samplers locales

Como es obvio, la verificación adhoc de problemas de optimización solo tiene valor demostrativo, no resolutivo. Es conveniente automatizar este proceso, una forma de barrer el espacio de entrada de manera automática. El paquete `dimod` proporciona para ello varias clases de samplers:

- exactos: barren todo el espacio de entrada. Son samplers de fuerza bruta. Reciben el nombre de `solvers exactos`. Este espacio de entrada admite cualquier modelo, incluso ising (clase `SimulatedAnnealingSampler`)
- discretos: barren un subconjunto del espacio de entrada, por ejemplo siguiendo una estrategia aleatoria.

Por definición estos samplers usan cpu local, no qpu, y por tanto están muy indicados para el prototipado de problemas.

Por supuesto que también se pueden usar samplers cuánticos Dwave con diferentes estrategias.

Documentación

samplers dimod locales:

https://docs.ocean.dwavesys.com/en/stable/docs_dimod/reference/sampler_composites/samplers.html

samplers Q, y cuántico-clásicos Dwave:

https://docs.ocean.dwavesys.com/en/stable/docs_samplers/index.html#

ExactCQMSolver

En el problema blp planteado se optará por un sampler de fuerza bruta que soporte el modelo CQM.

La clase `ExactCQMSolver()` proporciona las soluciones para los 2^n estados del espacio de entradas, ordenando las mismas en función de energía creciente de la función de coste, y comprobando las restricciones del problema

In [147…

```
# Solver fuerza bruta ExactCQMSolver

solver = dimod.ExactCQMSolver()
solution = solver.sample_cqm(blp)

print("Lista de asignaciones")
print(solution)
```

```
Lista de asignaciones
   x0 x1 x2 x3 energy num_oc. is_sat. is_fea.
11  1  1  0  1   -6.0       1 arra...   False
3   1  1  0  0   -5.0       1 arra...   False
9   0  1  0  1   -4.0       1 arra...    True
1   0  1  0  0   -3.0       1 arra...    True
10  1  0  0  1   -3.0       1 arra...    True
2   1  0  0  0   -2.0       1 arra...    True
15  1  1  1  1   -2.0       1 arra...   False
7   1  1  1  0   -1.0       1 arra...   False
8   0  0  0  1   -1.0       1 arra...    True
0   0  0  0  0    0.0       1 arra...    True
13  0  1  1  1    0.0       1 arra...   False
5   0  1  1  0    1.0       1 arra...    True
14  1  0  1  1    1.0       1 arra...   False
6   1  0  1  0    2.0       1 arra...    True
12  0  0  1  1    3.0       1 arra...   False
4   0  0  1  0    4.0       1 arra...    True
['INTEGER', 16 rows, 16 samples, 4 variables]
```

De la tabla anterior se observa que la solución óptima, función de coste mínima, corresponde a:

[x0x1x2x3x4]=[0101]

con valor f=-4.

Extrayendo la solución óptima

Se pueden extraer la solución factible de `menor energía` con el siguiente método:

In [148…
```
feasible_sols = solution.filter(lambda s: s.is_feasible)
feasible_sols.first
```

Out[148]:
```
Sample(sample={'x0': 0, 'x1': 1, 'x2': 0, 'x3': 1}, energy=-4.0, num_occurrences=1, is_satisfied=array([ True,  Tru
e]), is_feasible=True)
```

Nota importante

Al final de esta Parte I se volverá a este tipo de problemas, pero reformulándolo como un problema `bqm`, sin restricciones, y haciendo uso ya de samplers cuánticos.

Pasemos ahora a otro tipo interesante de problema clásico-cuántico: el maxcut.

Un problema Maxcut

El problema maxcut es un problema clásico de clasificación del ML. En un grafo dado, con una topología de interconexiones arbitraria, se trata de maximizar el número de aristas que interconectan nodos de diferente clase.

Se va a ejemplarizar con un grafo sencillo:

- 5 nodos
- tolología planar de base cuadrada: 4 nodos en los vértices de las aristas de un cuadrado, y un quinto nodo en el centro del mismo.

En esta sección se van a indicar varias estrategías para afrontar este tipo de problemas, tanto con computación cuántica en nube, como con cpu local.

Maxcut y modelo Ising

El formalismo Ising se adapta especialmente bien a este tipo de problemas, favoreciendo con un peso -1 a las configuraciones nodales de diferente clase, o penalizando con un peso de +1 cuando los nodos pertenecen a la misma clase.

Por cada configuración nodal (espacio de entrada) se obtendrá una función de coste asociada, con el balance final entre premios y penalizanciones según que las diferentes aristas interconecten nodos de igual o diferente clase.

Cada arista maxcut contribuye con -1 a la energía, cada arista no maxcut contribuye con +1. Así, la energía maxcut óptima es:

- Emin= Num_aristas * -1

En esta sección se vas a utilizar tanto samplers CPU como samplers QPU (DwaveSampler)

Los parámetros del problema: J, h, BQM etc se pueden consultar en el siguiente link:

https://docs.dwavesys.com/docs/latest/c_solver_problems.html

Y de nuevo el enlace de referencia al modelo Ising.

https://docs.dwavesys.com/docs/latest/c_gs_3.html#getting-started-concepts

Especificación del problema maxcut con formalismo Ising

A continuación se va a codificar el problema maxcut propuesto bajo el modelo Ising, con sesgos hi=0

In [30]:
```
# Modelo Maxcut Ising

# prisma de base cuadrada,  4 nodos, `0-3`, en los vértices, con un quinto, `4`, como nodo central.

J = {(0,1):1, (0,3):1, (0,4):1,(1,2):1, (1,4):1, (2,3):1, (2,4):1, (3,4):1}
h = {} # sin campo externo
```

Solución con sampler de fuerza bruta

Con el sampler `ExactSolver()` y el método `sample_ising()`, se obtiene la función de coste para todo el espacio de entradas, es decir, para todas las configuraciones nodales, 2^5=32, en este caso.

In [31]:
```
import dimod

maxcut_cpu = dimod.ExactSolver().sample_ising(h, J)
print("Soluciones Ising")
print(maxcut_cpu)
print("\nAristas totales:",len(J))
print("Aristas óptimas:",int(maxcut_cpu.first.energy+len(J)))
```

```
Soluciones Ising
    0  1  2  3  4 energy num_oc.
4  -1 +1 -1 +1 -1   -4.0       1
11 -1 +1 -1 +1 +1   -4.0       1
17 +1 -1 +1 -1 +1   -4.0       1
30 +1 -1 +1 -1 -1   -4.0       1
5  +1 +1 -1 +1 -1   -2.0       1
8  -1 -1 -1 +1 +1   -2.0       1
12 -1 +1 -1 -1 +1   -2.0       1
14 +1 -1 -1 -1 +1   -2.0       1
16 -1 -1 +1 -1 +1   -2.0       1
25 +1 -1 +1 +1 -1   -2.0       1
27 -1 +1 +1 +1 -1   -2.0       1
29 +1 +1 +1 -1 -1   -2.0       1
2  +1 +1 -1 -1 -1    0.0       1
6  +1 -1 -1 +1 -1    0.0       1
9  +1 -1 -1 +1 +1    0.0       1
13 +1 +1 -1 -1 +1    0.0       1
15 -1 -1 -1 -1 +1    0.0       1
19 -1 +1 +1 -1 +1    0.0       1
23 -1 -1 +1 +1 +1    0.0       1
24 -1 -1 +1 +1 -1    0.0       1
26 +1 +1 +1 +1 -1    0.0       1
28 -1 +1 +1 -1 -1    0.0       1
1  +1 -1 -1 -1 -1    2.0       1
3  -1 +1 -1 -1 -1    2.0       1
7  -1 -1 -1 +1 -1    2.0       1
10 +1 +1 -1 +1 +1    2.0       1
18 +1 +1 +1 -1 +1    2.0       1
20 -1 +1 +1 +1 +1    2.0       1
22 +1 -1 +1 +1 +1    2.0       1
31 -1 -1 +1 -1 -1    2.0       1
0  -1 -1 -1 -1 -1    8.0       1
21 +1 +1 +1 +1 +1    8.0       1
['SPIN', 32 rows, 32 samples, 5 variables]

Aristas totales: 8
Aristas óptimas: 4
```

Solución con sampler cuántico sobre modelo Ising

Lanzar el problema maxcut sobre procesador cuántico solo requiere migrar el sampler cuántico e invocar el mismo método `sample_ising()`

In [32]:
```
from dwave.system import DWaveSampler, EmbeddingComposite

sampler = EmbeddingComposite(DWaveSampler())
maxcut_qpu = sampler.sample_ising(h, J, num_reads=10)
```

Resultados:

In [33]:
```
print("Soluciones Ising")
print(maxcut_qpu)
print("\nAristas totales:",len(J))
print("Aristas óptimas:",int(maxcut_qpu.first.energy+len(J)))
```

```
Soluciones Ising
   0  1  2  3  4 energy num_oc. chain_.
0 -1 +1 -1 +1 -1   -4.0       4     0.0
1 +1 -1 +1 -1 -1   -4.0       1     0.0
2 -1 +1 -1 +1 +1   -4.0       4     0.0
3 +1 -1 +1 -1 +1   -4.0       1     0.0
['SPIN', 4 rows, 10 samples, 5 variables]

Aristas totales: 8
Aristas óptimas: 4
```

Se han proporcionado 4 soluciones con el mismo coste (-4) al problema maxcut planteado. Incluso no se puede descartar que en algunos runs se proporcione soluciones con coste -2

Solución con sampler cuántico sobre modelo BQM

En lugar de usar el método `sample_ising`, se puede pasar directamente el problema al sampler usando un modelo BQM, sin restricciones. (recordemos que el CQM usado en el problema blp era con restricciones.

In [34]:
```
# Sampler CPU exacto modelo BQM con variables SPIN (-1,1)

import dimod

maxcut_bqm = dimod.BinaryQuadraticModel(h, J, 0.0, dimod.SPIN)  #BQM
print("El problema a resolver es:")
print(maxcut_bqm)
```

```
sampler = EmbeddingComposite(DWaveSampler())
maxcut_qpu2 = sampler.sample(maxcut_bqm, num_reads=10)
```

```
El problema a resolver es:
BinaryQuadraticModel({0: 0.0, 1: 0.0, 3: 0.0, 4: 0.0, 2: 0.0}, {(1, 0): 1.0, (3, 0): 1.0, (4, 0): 1.0, (4, 1): 1.0,
(4, 3): 1.0, (2, 1): 1.0, (2, 3): 1.0, (2, 4): 1.0}, 0.0, 'SPIN')
```

Resultados:

In [35]:
```
print("Soluciones Ising")
print(maxcut_qpu2)
print("\nAristas totales:",len(J))
print("Aristas óptimas:",int(maxcut_qpu2.first.energy+len(J)))
```

```
Soluciones Ising
   0  1  2  3  4 energy num_oc. chain_.
0 -1 +1 -1 +1 -1   -4.0       6     0.0
1 +1 -1 +1 -1 +1   -4.0       1     0.0
2 -1 +1 -1 +1 +1   -4.0       1     0.0
3 +1 -1 +1 -1 -1   -4.0       2     0.0
['SPIN', 4 rows, 10 samples, 5 variables]

Aristas totales: 8
Aristas óptimas: 4
```

Veamos ahora un modelo BQM con restricciones:

- Modelo CQM, cuadrático con restricciones

Reformulación de problemas CQM a BQM

Plantear un problema en formalismo qubo obliga a incorporar las restricciones a la función objetivo como penalizaciones, usando para ello relaciones lineales o cuadráticas, generalmente con la ayuda de `variables slack`, y `multiplicadores o factores de Lagrange` que modulen el nivel de penalización por incumplimiento de la restricciones.

A continuación se presenta a modo de ejemplo un sencillo problema `blp` con este nuevo enfoque

In [130…
```
# Problema de programación lineal cqm con reformulación bqm (qubo)

x0 = dimod.Binary("x0")
x1 = dimod.Binary("x1")
```

```
cqm = dimod.ConstrainedQuadraticModel() #CQM

cqm.set_objective(-2*x0-3*x1)  # Función objetivo

cqm.add_constraint(x0 + 2*x1 <= 2, 'restricción')

qubo, invert = dimod.cqm_to_bqm(cqm, lagrange_multiplier = 5)
print(qubo)
```

```
BinaryQuadraticModel({'x0': -17.0, 'x1': -23.0, 'slack_v1a9892a316b3477d9e7307c243a4a9fd_0': -15.0, 'slack_v1a9892a316
b3477d9e7307c243a4a9fd_1': -15.0}, {('x1', 'x0'): 20.0, ('slack_v1a9892a316b3477d9e7307c243a4a9fd_0', 'x0'): 10.0, ('s
lack_v1a9892a316b3477d9e7307c243a4a9fd_0', 'x1'): 20.0, ('slack_v1a9892a316b3477d9e7307c243a4a9fd_1', 'x0'): 10.0, ('s
lack_v1a9892a316b3477d9e7307c243a4a9fd_1', 'x1'): 20.0, ('slack_v1a9892a316b3477d9e7307c243a4a9fd_1', 'slack_v1a9892a3
16b3477d9e7307c243a4a9fd_0'): 10.0}, 20.0, 'BINARY')
```

In [132…

```
# Transpilado  (embedding) a Qpu

sampler = EmbeddingComposite(DWaveSampler())
result = sampler.sample(qubo, num_reads=10)

print("Soluciones encontradas:")
print(result.data)
```

```
Soluciones encontradas:
<bound method SampleSet.data of SampleSet(rec.array([([0, 0, 0, 1], -3., 5, 0.), ([1, 0, 1, 0], -2., 3, 0.),
           ([0, 1, 1, 0], -2., 1, 0.), ([0, 0, 1, 1],  0., 1, 0.)],
          dtype=[('sample', 'i1', (4,)), ('energy', '<f8'), ('num_occurrences', '<i8'), ('chain_break_fraction', '<f
8')]), Variables(['slack_v1a9892a316b3477d9e7307c243a4a9fd_0', 'slack_v1a9892a316b3477d9e7307c243a4a9fd_1', 'x0', 'x
1']), {'timing': {'qpu_sampling_time': 836.0, 'qpu_anneal_time_per_sample': 20.0, 'qpu_readout_time_per_sample': 43.0
6, 'qpu_access_time': 16594.77, 'qpu_access_overhead_time': 2279.23, 'qpu_programming_time': 15758.77, 'qpu_delay_time
_per_sample': 20.54, 'total_post_processing_time': 162.0, 'post_processing_overhead_time': 162.0}, 'problem_id': 'e444
ecac-d779-4654-95c1-907ce2aa4ea3'}, 'BINARY')>
```

Pero no todas estas soluciones son factibles, pueden incumplir una o varias restricciones, aunque su función de coste pueda ser la mínima.

A continuación se hace un filtrado de estos resultados:

In [128…

```
samples = []
occurrences = []
for s in result.data():
    samples.append(invert(s.sample))
    occurrences.append(s.num_occurrences)
sampleset = dimod.SampleSet.from_samples_cqm(samples,cqm,
```

```
    num_occurrences=occurrences)
print("Soluciones (función de coste vs factibilidadd):")
print(sampleset)
```

```
Soluciones (función de coste vs factibilidadd):
  x0 x1 energy num_oc. is_sat. is_fea.
0  0  1   -3.0       6 arra...    True
1  1  0   -2.0       4 arra...    True
['INTEGER', 2 rows, 10 samples, 2 variables]
```

In [129…

```
final_sols = sampleset.filter(lambda s: s.is_feasible)
final_sols = final_sols.aggregate()
print("\nSoluciones finales:")
print(final_sols)
```

```
Soluciones finales:
  x0 x1 energy num_oc. is_sat. is_fea.
0  0  1   -3.0       6 arra...    True
1  1  0   -2.0       4 arra...    True
['INTEGER', 2 rows, 10 samples, 2 variables]
```

Este enfoque general será el empleado ahora en este otro problema con dos restricciones.

BLP como problema BQM

Volviendo al problema blp planteado al principio de este cuaderno, y que se había resuelto mediante un sampler de fuerza bruta, vamos ahoa a replantearlo como problema BQM, incorporando las dos restricciones como penalizaciones de la función objetivo.

In [134…

```
sampler = EmbeddingComposite(DWaveSampler("Advantage_system4.1"))

# Definición del problema

x0 = dimod.Binary("x0")
x1 = dimod.Binary("x1")
x2 = dimod.Binary("x2")
x3 = dimod.Binary("x3")

# Modelo CQM con restricciones
blp = dimod.ConstrainedQuadraticModel()

# Función objetivo
blp.set_objective(-2*x0-3*x1+4*x2-x3)
```

```
# Restricciones
blp.add_constraint(x0 + 2*x1 <= 2, "Primera restricción")
blp.add_constraint(x2 + x3 <= 1, "Segunda restricción")

# Conversión a qubo (cqm-bqm) con multiplicador de Lagrange

fl=10


qubo, invert = dimod.cqm_to_bqm(blp, lagrange_multiplier = fl)
result = sampler.sample(qubo, num_reads=100)

# Agregación de los n reads
samples = []
occurrences = []

for s in result.data():
    samples.append(invert(s.sample))
    occurrences.append(s.num_occurrences)

    sampleset = dimod.SampleSet.from_samples_cqm(samples,blp,num_occurrences=occurrences)
```

In [135…

```
# Resultados
print("\nFactor de Lagrange:", fl)
print("\nLas soluciones factibles al problema son:\n")
print(sampleset.filter(lambda s: s.is_feasible).aggregate())
```

```
Factor de Lagrange: 10

Las soluciones factibles al problema son:

  x0 x1 x2 x3 energy num_oc. is_sat. is_fea.
0  0  1  0  1   -4.0      19 arra...    True
1  1  0  0  1   -3.0      18 arra...    True
2  0  1  0  0   -3.0      24 arra...    True
3  1  0  0  0   -2.0      21 arra...    True
4  0  0  0  1   -1.0       5 arra...    True
5  0  0  0  0    0.0       4 arra...    True
6  0  1  1  0    1.0       3 arra...    True
7  1  0  1  0    2.0       3 arra...    True
8  0  0  1  0    4.0       2 arra...    True
['INTEGER', 9 rows, 99 samples, 4 variables]
```

Influencia del factor de Lagrange:

Se va a repetir el problema usando factores de lagrange 3 y 1, para "calibrar" el efecto amplificación de las penalizaciones por no cumplimiento de las restricciones.

In [140…
```
# FL=4
fl=3
qubo, invert = dimod.cqm_to_bqm(blp, lagrange_multiplier = fl)
result = sampler.sample(qubo, num_reads=100)

# Agregación de los n reads
samples = []
occurrences = []

for s in result.data():
    samples.append(invert(s.sample))
    occurrences.append(s.num_occurrences)
sampleset = dimod.SampleSet.from_samples_cqm(samples,blp,
    num_occurrences=occurrences)
```

In [141…
```
# Resultados
print("\nFactor de Lagrange:", fl)
print("\nLas soluciones factibles al problema son:\n")
print(sampleset.filter(lambda s: s.is_feasible).aggregate())
```

```
Factor de Lagrange: 3

Las soluciones factibles al problema son:

  x0 x1 x2 x3 energy num_oc. is_sat. is_fea.
0  0  1  0  1   -4.0      57 arra...    True
1  0  1  0  0   -3.0      10 arra...    True
2  1  0  0  1   -3.0      21 arra...    True
3  1  0  0  0   -2.0       4 arra...    True
['INTEGER', 4 rows, 92 samples, 4 variables]
```

In [138…
```
# FL1
fl=1
qubo, invert = dimod.cqm_to_bqm(blp, lagrange_multiplier = fl)
result = sampler.sample(qubo, num_reads=100)

# Agregación de los n reads
```

```
samples = []
occurrences = []

for s in result.data():
    samples.append(invert(s.sample))
    occurrences.append(s.num_occurrences)
sampleset = dimod.SampleSet.from_samples_cqm(samples,blp,
    num_occurrences=occurrences)
```

In [139…

```
# Resultados
print("\nFactor de Lagrange:", fl)
print("\nLas soluciones factibles al problema son:\n")
print(sampleset.filter(lambda s: s.is_feasible).aggregate())
```

```
Factor de Lagrange: 1

Las soluciones factibles al problema son:

Empty SampleSet
Record Fields: ['sample', 'energy', 'num_occurrences', 'is_satisfied', ...]
Variables: ['x0', 'x1', 'x2', 'x3']
['INTEGER', 0 rows, 0 samples, 4 variables]
```

Con fl=3 las soluciones factibles se han reducido considerablemente, pero la de menor energía coincide.

Con fl=1, el sistema simplemente no ha encontrado ninguna solución factible.

Parte II

Administración QPUs Dwave

Topologías Dwave

Dwave construye QPUs de propósito específico con arreglos de qbits especialmente pensados para los problemas de programación lineal y cuadrática.

Las diferentes generaciónes han ido proponiendo diferentes topologías. En el siguiente documento se pueden consultar.

https://docs.dwavesys.com/docs/latest/c_gs_4.html

Es importante percibir la importancia central que juegan estas arquitecturas en el éxito comercial de Dwave, que sin duda descansa en su efectividad a la hora de resolver estos problemas de optimización, de extremada importancia en algunos sectores como las finanzas y la industria, bajo el paradigma cuántico.

En las próximas secciones se van a poder consultar y manipular algunos de estos atributos.

Caracterización física de QPUs asignables a Cliente

La instrucción `!dwave config create` , que ha de ejecutarse una primera y única vez en `local` , o en cada sesión si se hace desde `colab` , instancia el objeto Client que configura el entorno operativo y la disponibilidad de recursos de cómputo asociados.

Se puede obtener info relevante usando diferentes métodos del mismo.

In [3]:
```
# Interactuando con la arquitectura física

from dwave.system import DWaveSampler
sampler = DWaveSampler()

# sampler.nodelist
# sampler.edgelist

qb_a = sampler.nodelist[0] # asigna primer nodo de la QPU
nodos_ady= iter(sampler.adjacency[qb_a]) #obtiene todos los adyacentes al nodo asignado a qb_a
qb_b = next(nodos_ady) #asigna primer adyacente a qb_b
qb_c = next(nodos_ady) #asigna segundo adyacente a qb_c

print(qb_a)
print(sampler.adjacency[qb_a])
print(qb_b,qb_c)
```

```
30
{2985, 2955, 45, 2970, 2940, 31}
2985 2955
```

In [49]:
```
# Funciones auxiliares informativas

import random
from dwave.system import DWaveSampler
```

```
from dwave.cloud import Client

def client_info():
    print("Solvers:")
    for solver in Client.from_config().get_solvers():
        print(solver)

def dwave_info(sampler,modo=0):
    print("Nombre:",sampler.properties["chip_id"])
    print("No. qubits:",sampler.properties["num_qubits"])
    print("Categoría:",sampler.properties["category"])
    print("Problemas soportados:",sampler.properties["supported_problem_types"])
    print("Topología:",sampler.properties["topology"])
    print("Fuerza de acoplamiento", sampler.properties["h_range"])
    print("Rango de 'reads':",sampler.properties["num_reads_range"])
    print("Annealing time (defecto)",sampler.properties["default_annealing_time"],"microsecs")
    print("Rango annealing time (us)",sampler.properties["annealing_time_range"])
    if modo:
        print("Acoplamientos:",sampler.properties["couplers"]) #muestra geometría
        print(sampler.adjacency) # muestra adyacencias
```

with Client.from_config() as client: solver = client.get_solver()

```
# Build a random Ising model to exactly fit the graph the solver supports
linear = {index: random.choice([-1, 1]) for index in solver.nodes}
quad = {key: random.choice([-1, 1]) for key in solver.undirected_edges}
```

In [18]:
```
#solver.undirected_edges
```

In [51]:
```
# solvers disponibles para cuenta de usuario (puede variar)
client_info()
```

```
Solvers:
BQMSolver(id='hybrid_binary_quadratic_model_version2')
DQMSolver(id='hybrid_discrete_quadratic_model_version1')
StructuredSolver(id='Advantage_system4.1')
CQMSolver(id='hybrid_constrained_quadratic_model_version1')
StructuredSolver(id='Advantage2_prototype1.1')
StructuredSolver(id='Advantage_system6.2')
```

Los solvers anteriores tienen asociado samplers cuyas características se pueden consultar:

In [54]:
```
# sampler=DWaveSampler(solver='DW_2000Q_6')
sampler=DWaveSampler(solver='Advantage_system6.2')
dwave_info(sampler)
```

```
Nombre: Advantage_system6.2
No. qubits: 5760
Categoría: qpu
Problemas soportados: ['ising', 'qubo']
Topología: {'type': 'pegasus', 'shape': [16]}
Fuerza de acoplamiento [-4.0, 4.0]
Rango de 'reads': [1, 10000]
Annealing time (defecto) 20.0 microsecs
Rango annealing time (us) [0.5, 2000.0]
```

In [52]:
```
sampler=DWaveSampler(solver='Advantage_system4.1')
dwave_info(sampler)
```

```
Nombre: Advantage_system4.1
No. qubits: 5760
Categoría: qpu
Problemas soportados: ['ising', 'qubo']
Topología: {'type': 'pegasus', 'shape': [16]}
Fuerza de acoplamiento [-4.0, 4.0]
Rango de 'reads': [1, 10000]
Annealing time (defecto) 20.0 microsecs
Rango annealing time (us) [0.5, 2000.0]
```

In [43]:
```
sampler=DWaveSampler(solver='Advantage2_prototype1.1')
dwave_info(sampler)
```

```
Nombre: Advantage2_prototype1.1
No. qubits: 576
Categoría: qpu
Problemas soportados: ['ising', 'qubo']
Topología: {'type': 'zephyr', 'shape': [4, 4]}
Fuerza de acoplamiento [-4.0, 4.0]
Rango de 'reads': [1, 10000]
Annealing time (defecto) 20.0 microsecs
Rango annealing time (us) [1.0, 2000.0]
```

In [45]:
```
sampler=DWaveSampler(solver='Advantage_system6.2')
dwave_info(sampler)
```

```
Nombre: Advantage_system6.2
No. qubits: 5760
Categoría: qpu
Problemas soportados: ['ising', 'qubo']
Topología: {'type': 'pegasus', 'shape': [16]}
Fuerza de acoplamiento [-4.0, 4.0]
Rango de 'reads': [1, 10000]
Annealing time (defecto) 20.0 microsecs
Rango annealing time (us) [0.5, 2000.0]
```

Maxcut con qpu específica

Retomando el problema maxcut anterior, se va a resolver ahora con qpu específica y reetiquetando los nodos 0-4 con el objeto de evidenciar algunos aspectos importantes de la arquitectura hardwarer subyacente.

La QPU seleccionada es la `Advantage_system6.2`, de 5760 cúbits con arquitectura `Pegasus` y cuyo resto de características figuran en el listado inmediato anterior.

In [78]:
```
# Problema maxcut de 5 nodos

J = {('a','b'):1, ('a','d'):1, ('a','e'):1,('b','c'):1, ('b','e'):1, ('c','d'):1, ('c','e'):1, ('d','e'):1}
h = {}

prisma = dimod.BinaryQuadraticModel(h, J, 0.0, dimod.SPIN)

# embedding y run en annealer 'Advantage_system6.2'

sampler = EmbeddingComposite(DWaveSampler(solver = 'Advantage_system6.2'))
result = sampler.sample(prisma, num_reads=10,
    return_embedding = True)
```

In [79]:
```
print("'Samples' obtenidos:\n")
print(result)
print("\nLa incrustación asignada fue:\n")
print(result.info["embedding_context"])
```

```
'Samples' obtenidos:

   a  b  c  d  e energy num_oc. chain_.
0 +1 -1 +1 -1 +1   -4.0       2     0.0
1 -1 +1 -1 +1 -1   -4.0       1     0.0
2 +1 -1 +1 -1 -1   -4.0       5     0.0
3 -1 +1 -1 +1 +1   -4.0       1     0.0
4 +1 +1 -1 +1 -1   -2.0       1     0.0
['SPIN', 5 rows, 10 samples, 5 variables]

La incrustación asignada fue:

{'embedding': {'b': (2046,), 'a': (2061, 4195), 'd': (2002,), 'e': (4285,), 'c': (4270,)}, 'chain_break_method': 'majo
rity_vote', 'embedding_parameters': {}, 'chain_strength': 2.529440096147762}
```

Examinando la asignación de nodos de la QPU al grafo, esta parece caprichosa. Es más, la lista de los 5 primeros nodos de la máquina seleccionada sería:

In [73]:
```
qpu=DWaveSampler(solver = 'Advantage_system6.2')

for i in range(5):
 print(qpu.nodelist[i])
```

```
30
31
32
33
34
```

Entonces?, pues es fácil de entender si asumimos que las qpu operan en la nube en un contexto multiusuario. El administrador de jobs de la qpu va distribuyendo los nodos entre los trabajos concurrentes.

Ahora bien, si examinamos los nodos adyacentes a, por ejemplo el nodo 'a', asignado al nodo qpu 4190, podremos comprobar que los nodos 'b','d' y 'e' a los que se conecta el nodo 'a', pertenecen a esa lista.

In [71]:
```
qpu=DWaveSampler(solver = 'Advantage_system6.2')
print(qpu.adjacency[4190])
```

```
{1056, 1026, 996, 1161, 1132, 1102, 1071, 4175, 1041, 1011, 4189, 1146, 1117, 4191, 1087}
```

Otra singularidad, el nodo central 'e', está interconectado con los otros 4. Si el incrustador no logra asignarle nodo con adyacencias a ellos, puede optar por realizar una doble asignación, y por eso en la lista anterior aparece con asignación doble: 4175 y 1041.

In [75]:
```
qpu=DWaveSampler(solver = 'Advantage_system6.2')
print(qpu.adjacency[1041])
print(qpu.adjacency[4175])
```

```
{4160, 4129, 1026, 4099, 4070, 4205, 4175, 1040, 4145, 1042, 4114, 4084, 4055, 4220, 4190}
{1056, 1026, 996, 1161, 1132, 1102, 1071, 4174, 1041, 4176, 1011, 1146, 1117, 4190, 1087}
```

Examinando el nodo físico 1041, vemos que es adyacente al de asignación para los nodos 'a', 'c' y 'e', pero no del 'b', del que sí lo es el 4176.

Concluímos que la incrustación automática ha optimizado la asignación nodal en la qpu para que los vértices interconectados del problema maxcut tengan nodos adyacentes.

Annealing time

Como se mencionó al comienzo del cuaderno, el annealer hace evolucionar un hamiltoniano inicial, de setup sencillo, a uno final, que representa el problema a minimizar.

Esta evolución temporal queda caracterizada por el `annealing time`, uno de los parámetros fundamentales de un determinado `read`.

El rango de valores admitidos depende del qpu seleccionado. En el Advantage_system6.2 por ejemplo varía entre 0.5 y 2000 μs, con una resolución de 0,02 μs, siendo 20 μs el valor por defecto.

Cuanto mayor sea este tiempo, **más se acercará la evolución al principio adiabático** , pero más tiempo máquina se consumirá.

Consultar la descripción proporcionada por `dwave_info(sampler)` para otros samplers, y el siguiente doc:

https://docs.dwavesys.com/docs/latest/c_solver_parameters.html#annealing-time

Se va a repetir el ejemplo anterior usando un At= 100μs

In [82]:
```
# 'Advantage_system6.2 con annealing_time =100 us

J = {(0,1):1, (0,3):1, (0,4):1,(1,2):1, (1,4):1, (2,3):1, (2,4):1, (3,4):1}
h = {}

prisma = dimod.BinaryQuadraticModel(h, J, 0.0, dimod.SPIN)
```

```
sampler = EmbeddingComposite(DWaveSampler(solver = 'Advantage_system6.2'))
result = sampler.sample(prisma, num_reads=10,
    return_embedding = True)
result = sampler.sample(prisma, num_reads=10, annealing_time = 100)

print("'Samples' obtenidos:\n")
print(result)
```

```
'Samples' obtenidos:

   0  1  2  3  4 energy num_oc. chain_.
0 +1 -1 +1 -1 +1   -4.0       2     0.0
1 -1 +1 -1 +1 -1   -4.0       3     0.0
2 -1 +1 -1 +1 +1   -4.0       2     0.0
3 +1 -1 +1 -1 -1   -4.0       2     0.0
4 -1 -1 +1 -1 +1   -2.0       1     0.0
['SPIN', 5 rows, 10 samples, 5 variables]
```

Programación hacía adelante (`forward sheduling`)

La evolución temporal del hamiltoniano durante el `annealing time` es lineal, con una pendiente constante entre t=0 y el tiempo de annealing. Durante este intervalo el parámetro del hamiltoniano, s(t), varía linealmente entre s=0 y s=1, de modo que:

$$H(s) = sH0 + (1-s)Hf,\ s(t) = (1/ta)* t,\ ta = \text{annealing time}$$

Pero este mapeo lineal se puede cambiar fijando los valores de s para determinado tiempo de annealing.

Se usa para ello listas con pares de números en coma flotante. El primer elemento del par es el tiempo t en microsegundos con una granularidad de 0.01 u 0.0.2 μs según sampler, y el segundo elemento, su valor `s` en ese instante.

La pendiente máxima de cualquier segmento de curva no debe ser mayor que el inverso del tiempo mínimo soportado por el annealer. Así, para una QPU con rango de annealing de 0.5-2000 μs, la pendiente máxima que nunca se debe superar es:

- m= (sf-si)/(tf-ti)=(1-0)/(0.5-0)=2μs–1

Así, entre los dos intervalos siguientes: [0.0, 0.0], [5.0, 0.25], el annealer empieza en t=0 con s=0, variando este parámetro linealmente hasta el valor s=0.25 para t=5 us. La pendiente de intervalo es por tanto:

- m = (0.25-0/5-0)= 0.05 μs–1

El mapeo s(t) resultante es la curva lineal por partes que interconecta los puntos proporcionados, y que determinan la rapidez con la que evoluciona el annealer en cada intervalo

En el siguiente doc se puede ampliar la info sobre este proceso:

https://docs.dwavesys.com/docs/latest/c_solver_parameters.html#param-anneal-sched

Importante:

Recordemos que por el principio adiabático, una evolución demasiado rápida puede provocar que el estado final no corresponda al de mínima energía

A continuación se repetirá el problema maxcut anterior, usando una programación de 4 etapas, con una pronunciada pendiente inicial, muy suave entre 10-40μs, y de nuevo pronunciada en los últimos 10μs, alcanzándose el hamiltoniano final a los 50 μs (s=1)

In [83]:
```
forward_schedule=[[0.0, 0.0], [10.0, 0.25], [40, 0.75], [50, 1.0]]

sampler = EmbeddingComposite(DWaveSampler())

result = sampler.sample(prisma, num_reads=10,
    anneal_schedule = forward_schedule)

print("'Samples' obtenidos:\n")
print(result)
```

```
'Samples' obtenidos:

   0  1  2  3  4 energy num_oc. chain_.
0 -1 +1 -1 +1 -1   -4.0       2     0.0
1 +1 -1 +1 -1 +1   -4.0       5     0.0
2 -1 +1 -1 +1 +1   -4.0       1     0.0
3 +1 -1 +1 -1 -1   -4.0       2     0.0
['SPIN', 4 rows, 10 samples, 5 variables]
```

Programación inversa (`reverse sheduling`)

El annealer también soporta programación inversa, es decir, tramos en los cuales la variación s(t) es de la forma:

$$s(t) = 1 - 1/ta * t$$

No obstante, una programación de este tipo **debe de empezar y terminar con s=1** , por lo que es imperativo que el último tramo sea de programación directa.

Este modo también obliga a indicar un estado inicial para s=1. Se hará mediante parejas clave:valor (índice_qb, estado)

- -1 / 1 : Ising, activos
- 0 / 1 : QUBO, activos
- 3 : sin usar o inactivos

https://docs.dwavesys.com/docs/latest/c_solver_parameters.html#param-initial-state

Si se han programado múltiples `reads` mediante una llamada única a la API del solver, existen dos enfoques para el estado inicial del siguiente `read` :

- `reinitialize_state=true` : reinicializa al estado inicial especificado en cada `read` .
- `reinitialize_state=false` : solo se fija el estado inicial en el primer read. Los siguientes parten `del estado final` del `read` anterior.

La programación siguiente parte en t=0 con s=1, haciendo una programación inversa de 10μs hasta s=0.5, y a continuación una directa para completar el annealing a los 20μs, regresando a s=1.

In [84]:
```
reverse_schedule=[[0.0, 1.0], [10.0, 0.5], [20, 1.0]]
estado_inicial = {0:-1, 1:-1, 2:-1, 3:1, 4:1}  # q4q3q2q1q0=-1-1-111

sampler = EmbeddingComposite(DWaveSampler())
result = sampler.sample(prisma, num_reads=10,
    anneal_schedule = reverse_schedule,
    reinitialize_state=False, initial_state = estado_inicial)

print("'Samples' obtenidos:\n")
print(result)
```

```
'Samples' obtenidos:

   0  1  2  3  4 energy num_oc. chain_.
0 -1 +1 -1 +1 +1   -4.0       1     0.0
1 -1 +1 -1 +1 -1   -4.0       1     0.0
2 -1 +1 -1 +1 -1   -4.0       1     0.0
3 +1 -1 +1 -1 -1   -4.0       1     0.0
4 +1 -1 +1 -1 -1   -4.0       1     0.0
5 +1 -1 +1 -1 +1   -4.0       1     0.0
7 +1 -1 +1 -1 +1   -4.0       1     0.0
8 +1 -1 +1 -1 +1   -4.0       1     0.0
9 +1 -1 +1 -1 -1   -4.0       1     0.0
6 +1 -1 -1 -1 +1   -2.0       1     0.0
['SPIN', 10 rows, 10 samples, 5 variables]
```

Anexo

Samplers y solvers alternativos

Un `sampler` acepta un problema en formato de modelo cuadrático binario (BQM) o modelo cuadrático discreto (DQM) y devuelve asignaciones de variables. Los samplers generalmente intentan minimizar una función objetivo, pero también pueden muestrear distribuciones definidas por el problema.

https://docs.ocean.dwavesys.com/projects/system/en/stable/reference/samplers.html

https://docs.ocean.dwavesys.com/en/stable/docs_system/reference/samplers.html

SteepestDescentSolver()

https://docs.ocean.dwavesys.com/projects/greedy/en/latest/reference/generated/greedy.sampler.SteepestDescentSolver.sample.html

In [15]:
```python
import greedy
import dimod

J = {(0,1):1, (0,3):1, (0,4):1,(1,2):1, (1,4):1, (2,3):1, (2,4):1, (3,4):1}
h = {}

prisma = dimod.BinaryQuadraticModel(h, J, 0.0, dimod.SPIN)
```

```
# Sampler con  SteepestDescentSolver

solver = greedy.SteepestDescentSolver()
solution = solver.sample(prisma, num_reads = 10)

print(solution.aggregate())
```

```
   0  1  2  3  4 energy num_oc. num_st.
0 +1 -1 +1 -1 +1   -4.0       3       1
1 -1 +1 -1 +1 +1   -4.0       1       2
2 -1 +1 -1 +1 -1   -4.0       3       1
3 +1 -1 +1 -1 -1   -4.0       3       1
['SPIN', 4 rows, 10 samples, 5 variables]
```

In [30]:
```
import tabu

solver = tabu.TabuSampler()
solution = solver.sample(prisma, num_reads = 15)

print(solution.aggregate())
```

```
   0  1  2  3  4 energy num_oc. num_re.
0 +1 -1 +1 -1 -1   -4.0       5       1
1 -1 +1 -1 +1 +1   -4.0       4       1
2 +1 -1 +1 -1 +1   -4.0       4       1
3 -1 +1 -1 +1 -1   -4.0       2       1
['SPIN', 4 rows, 15 samples, 5 variables]
```

SimulatedAnnealingSampler()

Sampler dimod que utiliza un algoritmo de annealing simulado, un método heurístico de optimización para ordenadores clásicos.

https://docs.ocean.dwavesys.com/projects/neal/en/latest/reference/sampler.html

In [85]:
```
import neal

sampler = neal.SimulatedAnnealingSampler()
solution = sampler.sample(prisma, num_reads = 10)

print(solution.aggregate())
```

```
   0  1  2  3  4 energy num_oc.
0 +1 -1 +1 -1 -1   -4.0       3
1 -1 +1 -1 +1 -1   -4.0       3
2 -1 +1 -1 +1 +1   -4.0       2
3 +1 -1 +1 -1 +1   -4.0       2
['SPIN', 4 rows, 10 samples, 5 variables]
```

Samplers Dwave

https://docs.ocean.dwavesys.com/en/stable/docs_system/reference/samplers.html

In [90]:
```
import dwave.system

sampler = dwave.system.LeapHybridSampler()
```

In [91]:
```
import dwave.system

sampler = EmbeddingComposite(DWaveSampler())
solution = sampler.sample(prisma, num_reads = 10)

print(solution.aggregate())
```

```
   0  1  2  3  4 energy num_oc. chain_.
0 +1 -1 +1 -1 -1   -4.0       1     0.0
1 -1 +1 -1 +1 -1   -4.0       6     0.0
2 -1 +1 -1 +1 +1   -4.0       2     0.0
3 +1 +1 +1 -1 -1   -2.0       1     0.0
['SPIN', 4 rows, 10 samples, 5 variables]
```

DWaveCliqueSampler

Sampler para resolver Clique BQM en los sistemas D-Wave

In [ ]:
```
```