

# Projeto *Quadratic* *Unconstrained Binary* *Optimization*

UC: Projeto Integrado em Matemática e Computação

Ano letivo: 2022/2023

Curso: Mestrado em Matemática e Computação



**Alunos:** Daniela Brasileiro (pg49172) · Diogo Rosário (pg49174)  
Luís Silva (pg49177)

**Orientadores DMAT:** Cláudia Araújo · Luís Pinto

**Orientadores Fujitsu:** Bruna Calisto · José João Pereira

# Índice

Índice de Figuras .....	3
Índice de Tabelas .....	3
1. Introdução .....	4
2. Conceitos .....	5
2.1. Modelo de Ising .....	5
2.2. Modelo QUBO .....	5
2.3. Isomorfismo .....	6
2.4. Penalizações no modelo QUBO.....	7
2.5. Metodologia de busca tabu .....	8
3. Exemplos iniciais de aplicação da metodologia .....	9
4. Problema do caixeiro-viajante .....	11
4.1. Implementação para 4 cidades .....	12
4.1.1. Aplicação do modelo QUBO.....	13
4.1.2. Implementação em <i>python</i> .....	14
4.1.3. Resultados.....	15
5. Comparação do esforço computacional.....	17
6. PCV para os distritos de Portugal continental .....	19
7. Conclusão.....	24
8. Referências.....	25

## **Índice de Figuras**

Figura 1- Isomorfismo .....	7
Figura 2- Aplicação do modelo Ising .....	10
Figura 3- Aplicação do modelo QUBO .....	11
Figura 4- Matriz Q1 .....	15
Figura 5- Matriz Q2 .....	15
Figura 6- Matriz Q3 .....	15
Figura 7- Performance do algoritmo clássico.....	18
Figura 8- Performance do algoritmo QUBO .....	18
Figura 9- Cálculo das distâncias.....	19
Figura 10- Dicionário das coordenadas.....	20
Figura 11- Label encoding.....	20
Figura 12- Matriz Q4.....	21
Figura 13- Função CaminhoOptimo.....	21
Figura 14- Função custo .....	22
Figura 15- Melhor solução .....	22
Figura 16- Código para gráfico .....	23
Figura 17- Gráfico .....	23

## **Índice de Tabelas**

Tabela 1- Penalizações [1].....	8
Tabela 2-Matriz de distâncias .....	12
Tabela 3- Matriz Q1 .....	16
Tabela 4- Matriz Q2.....	16
Tabela 5- Matriz Q3.....	17

## 1. Introdução

Este trabalho foi proposto no âmbito da Unidade Curricular Projeto Integrado em Matemática e Computação na qual se desenvolveu um projeto sugerido pela Fujitsu. Este trabalho tem como objetivo resolver problemas de otimização aplicando o modelo *Quadratic Unconstrained Binary Optimization* (QUBO).

A metodologia QUBO é uma abordagem utilizada na área da otimização combinatória, que procura encontrar a melhor solução para um problema a partir de um conjunto finito de opções. Esta metodologia permite formular problemas de otimização combinatória em termos de uma função objetivo quadrática.

Uma abordagem computacional para a resolução de problemas modelados através da metodologia QUBO é inspirada na computação quântica, envolvendo a aplicação de conceitos e princípios da física quântica em algoritmos clássicos estocásticos. É importante destacar que as soluções encontradas seguindo esta abordagem computacional, geralmente, são aproximadas e não garantem a obtenção da solução ótima.

Um dos motivos pelos quais as soluções obtidas são aproximadas é o uso de algoritmos heurísticos. Muitos algoritmos utilizados na metodologia QUBO e nos processos estocásticos são baseados em métodos heurísticos, como algoritmos genéticos e algoritmos de *simulated annealing*. Estes algoritmos não garantem a descoberta da solução ótima, mas procuram encontrar uma solução que seja "suficientemente boa" dentro de um espaço de busca.

Apesar das soluções serem aproximadas, é importante ressaltar que as abordagens baseadas em QUBO e processos estocásticos podem ser eficazes na resolução de problemas de otimização complexos, especialmente quando não é possível obter soluções exatas em tempo razoável. As soluções aproximadas podem fornecer percepções significativas e resultados aceitáveis em muitos casos práticos, mesmo que não sejam ótimas.

No decorrer deste trabalho, abordou-se o problema clássico do caixeiro-viajante, aplicando o modelo QUBO mencionado anteriormente. Através da formulação do problema, procurou-se encontrar uma solução aproximada que minimize a distância total percorrida pelo caixeiro. Embora a obtenção da solução ótima não seja garantida devido à natureza aproximada dos algoritmos heurísticos utilizados, a abordagem baseada no modelo QUBO e processos estocásticos pode fornecer resultados satisfatórios para o problema do caixeiro-viajante, oferecendo uma alternativa eficaz para a resolução desse desafiante problema de otimização. O principal objetivo deste projeto foi a resolução do problema do caixeiro-viajante para os distritos de Portugal continental utilizando o modelo QUBO. Para tal, começou-se por considerar aplicações simples de ideias subjacentes à metodologia QUBO, o que, posteriormente, permitiu alcançar o objetivo principal do projeto.

## 2. Conceitos

Primeiramente, iremos explicitar alguns conceitos fundamentais para a execução deste projeto.

### 2.1. Modelo de Ising

O modelo de Ising, desenvolvido pelo físico Ernst Ising, é um modelo matemático que foi originalmente proposto para descrever o comportamento magnético de materiais, mas também tem sido amplamente utilizado em várias áreas, incluindo otimização.

No contexto de otimização, o modelo de Ising é usado para resolver problemas de minimização, nos quais o objetivo é encontrar a configuração de menor energia de um sistema. O sistema consiste num conjunto de variáveis binárias  $\sigma_1, \dots, \sigma_n$ , que representam os "spins" das partículas do sistema, sendo habitualmente designadas por "spins". Cada spin pode assumir o valor -1 (spin orientado para baixo) ou +1 (spin orientado para cima). A energia do sistema é determinada por uma função de energia que depende das interações entre os spins.

A função de energia do modelo de Ising é dada por:

$$H = - \sum_{\langle i j \rangle} J_{ij} \sigma_i \sigma_j - \sum_j h_j \sigma_j$$

onde: cada  $J_{ij}$  (com  $i$  e  $j$  a variarem entre 1 e  $n$ ) é um coeficiente que representa a interação entre os spins  $\sigma_i$  e  $\sigma_j$ ; cada  $h_j$  (com  $j$  a variar entre 1 e  $n$ ) é um coeficiente que representa o campo magnético externo aplicado ao spin  $\sigma_j$ .

O objetivo da otimização é encontrar a configuração do vetor de spins  $\sigma$  que minimize a função de energia  $H$ . Isto é, pretende-se encontrar uma atribuição de valores +1 ou -1 a cada spin que minimize a energia total do sistema.

### 2.2. Modelo QUBO

O QUBO é um modelo de otimização combinatória versátil enriquecido por uma variedade de aplicações e propriedades teóricas. As áreas de aplicação do modelo incluem finanças, análise de clusters, gestão de tráfego, programação de máquinas, conceção física VLSI (*Very Large System Integration*), física, computação quântica, engenharia e medicina. Além disso, existem reformulações QUBO de vários modelos matemáticos de otimização, abrangendo uma ampla gama de problemas. Entre eles, podemos citar o problema de atribuição de recursos limitados, o problema de partição de conjuntos, o problema de corte máximo, o problema de atribuição quadrática, o problema de otimização binária bipartida sem restrições e, neste caso específico, o problema do caixeiro-viajante. Essas reformulações QUBO possibilitam a aplicação de métodos de otimização baseados em QUBO para encontrar soluções eficientes para esses problemas complexos.

A função custo do modelo QUBO é dada por:

$$f(x) = \sum_i a_i x_i + \sum_{\langle i, j \rangle} b_{ij} x_i x_j = \sum_i \sum_j Q_{ij} x_i x_j$$

Nesta fórmula, cada termo  $Q_{ij}$  representa o coeficiente associado à interação entre os nós  $i$  e  $j$ , e  $x_i$  e  $x_j$  são as variáveis binárias correspondentes a esses nós.

Assim, o modelo QUBO é expresso pelo seguinte problema de otimização:

$$\text{minimizar } y = x^t Q x,$$

onde  $x$  é um vetor binário com as variáveis de decisão e  $Q$  é uma matriz quadrada que contém as constantes do problema. Cada elemento da matriz  $Q$  representa os custos ou as interações entre as variáveis do problema. É comum assumir que a matriz  $Q$  seja simétrica ou triangular superior, o que simplifica a formulação do problema.

Simétrica:

Para todo  $i$  e  $j$ , exceto quando  $i = j$ , substituir  $q_{ij}$  por  $(q_{ij} + q_{ji})/2$

Triangular Superior:

Para todo  $i$  e  $j$  com  $j > i$ , substituir  $q_{ij}$  por  $q_{ij} + q_{ji}$ . Em seguida, substituir todo o  $q_{ij}$  quando  $j < i$  por 0.

Os problemas do modelo QUBO enquadram-se numa classe de problemas conhecida como NP-difíceis, o que implica desafios significativos na obtenção de soluções ótimas. Os otimizadores exatos, concebidos para encontrar essas soluções, geralmente enfrentam dificuldades ao lidar com problemas de tamanho realista, muitas vezes exigindo dias ou até semanas de tempo de computação sem garantia de resultados de alta qualidade. Felizmente, estão a ser alcançados progressos impressionantes com métodos meta-heurísticos modernos, desenhados para encontrar soluções de alta qualidade, embora não necessariamente ótimas, num tempo de computação razoável. Estas abordagens estão a abrir possibilidades valiosas para a combinação da computação clássica e quântica, tirando partido dos benefícios de ambas as áreas. Ao unir essas disciplinas, é possível procurar soluções eficientes para problemas QUBO, representando um avanço promissor na resolução de problemas complexos.

### 2.3. Isomorfismo

O modelo de Ising e o modelo QUBO são duas formulações matemáticas amplamente utilizadas na área de otimização combinatória. Embora estes modelos sejam formulados de forma diferente são matematicamente equivalentes, o que significa que é possível transformar um problema de otimização QUBO num problema de otimização de Ising, e vice-versa.

A prova do isomorfismo entre o modelo de Ising e o modelo QUBO é fundamentada por uma série de igualdades entre as variáveis e os coeficientes das funções objetivo dos dois modelos. Essas igualdades estabelecem uma correspondência precisa entre as variáveis binárias do QUBO e as variáveis do modelo de Ising, bem como entre os coeficientes das funções objetivo. Esta prova é apresentada em [2].

Esta equivalência matemática permite que os problemas de otimização sejam formulados e resolvidos em ambos os modelos, dependendo da preferência ou da natureza específica do problema em questão.

Implementou-se em *python* duas funções para transformar o modelo de Ising no modelo QUBO e vice-versa. Visualizam-se estas funções na Figura 1.

```
def ising_to_qubo(h, J):
    n = len(h)
    Q = np.zeros((n, n))
    for i in range(n):
        Q[i][i] = h[i] / 2
        for j in range(i+1, n):
            Q[i][j] = J[i][j] / 4
            Q[j][i] = Q[i][j]
    return Q

def qubo_to_ising(Q):
    n = Q.shape[0]
    J = np.zeros((n, n))
    h = np.zeros(n)
    for i in range(n):
        h[i] = 2 * Q[i][i]
        for j in range(i+1, n):
            J[i][j] = 4 * Q[i][j]
            J[j][i] = J[i][j]
    return h, J
```

Figura 1- Isomorfismo

A função **ising\_to\_qubo** recebe um vetor **h** que representa os termos do campo magnético e uma matriz **J** que representa os coeficientes de interação entre as variáveis no modelo de Ising. Esta função retorna uma matriz **Q** que representa a função objetivo quadrática no modelo QUBO.

A função **qubo\_to\_ising** recebe uma matriz **Q** que representa a função objetivo quadrática no modelo QUBO. Esta função retorna um vetor **h** que representa os termos do campo magnético e uma matriz **J** que representa os coeficientes de interação entre as variáveis no modelo de Ising.

## 2.4. Penalizações no modelo QUBO

A forma do modelo QUBO referida anteriormente não contém restrições para além das que exigem que as variáveis sejam binárias. No entanto, a maioria dos problemas de otimização envolve restrições adicionais que devem ser atendidas durante a busca por soluções. Essas restrições podem ser incorporadas ao modelo QUBO através da introdução de termos de penalização quadráticos na função objetivo, em vez de serem explicitamente impostas na formulação clássica. As penalizações são formuladas de forma a serem iguais a zero para soluções

admissíveis e iguais ao valor da penalização para soluções que não são admissíveis ao problema. Assim, a situação em que a função objetivo é mínima no problema de minimização ocorre quando as penalizações são eliminadas, ou seja, quando as restrições são satisfeitas.

Na tabela 1 são apresentados exemplos de penalizações frequentemente utilizadas em problemas de otimização. O parâmetro  $P$  é um número positivo usado como penalização. Este número  $P$  deve ser selecionado de forma adequada para assegurar que o termo de penalização seja efetivamente equivalente à restrição clássica. Ou seja,  $P$  deve ter um valor suficientemente grande para o problema em questão.

Tabela 1- Penalizações [1]

Restrição Clássica	Penalização equivalente
$x + y \leq 1$	$P(xy)$
$x + y \geq 1$	$P(1 - x - y + xy)$
$x + y = 1$	$P(1 - x - y + 2xy)$
$x \leq y$	$P(x - xy)$
$x_1 + x_2 + x_3 \leq 1$	$P(x_1x_2 + x_1x_3 + x_2x_3)$
$x = y$	$P(x + y - 2xy)$

A escolha do valor da penalização requer um cuidadoso ajuste, levando em consideração a natureza do problema, a escala dos termos envolvidos e os resultados obtidos através de experiência e análise. Em geral, encontrar os valores ideais das penalizações é um desafio complexo, e diferentes abordagens podem ser combinadas ou adaptadas de acordo com a natureza específica do problema. É importante equilibrar a imposição das restrições desejadas com a necessidade de evitar soluções sub-ótimas devido a penalizações excessivamente altas. A experimentação, o ajuste iterativo e o uso de conhecimento especializado são componentes-chave para determinar valores de penalização eficazes.

## 2.5. Metodologia de busca tabu

Nesta subseção, será explicada a metodologia de busca tabu (*Tabu Search*), uma vez que na parte prática deste projeto se utilizou a classe **TabuSampler()**. Esta é uma classe da D-Wave e pertence à biblioteca de software D-Wave Ocean.

A metodologia de busca tabu é um método heurístico utilizado para encontrar soluções aproximadas em problemas de otimização, especialmente em espaços de busca complexos. Esta abordagem baseia-se no princípio de manter uma lista de movimentos "tabu" ou proibidos, de forma a evitar a repetição de soluções já exploradas anteriormente.

A busca tabu começa da mesma forma que uma busca local ou de vizinhança. A partir de um ponto inicial (solução), o algoritmo avança iterativamente para soluções vizinhas, utilizando operações chamadas de movimentos. Ao contrário de uma busca local simples, em que apenas são permitidos movimentos para soluções



vizinhas que melhoram o valor da função objetivo atual, a busca tabu permite movimentos que podem deteriorar o valor da função objetivo atual. A lista tabu determina os movimentos que são proibidos e evita que soluções já visitadas sejam exploradas novamente [3].

A busca tabu também utiliza memórias de curto e longo prazo para guiar o processo de busca. A memória de curto prazo mantém informações sobre as soluções recentemente visitadas e as características que mudaram durante a busca. Certas características das soluções recentes são rotuladas como ativas na lista tabu, e soluções que contenham essas características (ou combinações específicas delas) são consideradas tabu. Isso impede que soluções recentes façam parte da vizinhança atual e sejam revisitadas. A memória de longo prazo permite considerar soluções que não estão normalmente presentes na vizinhança, como soluções encontradas e avaliadas anteriormente ou soluções de alta qualidade relacionadas a essas soluções passadas [3].

A metodologia do **TabuSampler()** também envolve uma busca equilibrada entre intensificação e diversificação. As estratégias de intensificação procuram explorar combinações de movimentos e características de soluções historicamente boas. Por outro lado, as estratégias de diversificação procuram incorporar novas características e combinações de características não incluídas nas soluções geradas anteriormente. A diversificação é importante para explorar regiões diferentes do espaço de busca e evitar ficar preso em soluções sub-ótimas locais [3].

Assim, pode-se concluir que a busca tabu é uma técnica heurística de otimização que procura soluções melhores a partir da exploração do espaço de busca. Esta permite movimentos que pioram temporariamente o valor da função objetivo, evitando que a procura fique presa em ótimos locais. Isso ajuda a explorar novas regiões do espaço de busca na procura de soluções melhores. No entanto, a busca tabu não garante encontrar a solução ótima global. É uma abordagem de otimização aproximada que visa encontrar soluções aceitáveis num tempo computacionalmente viável. A qualidade da solução encontrada pode variar dependendo da complexidade do problema e das configurações específicas da busca tabu.

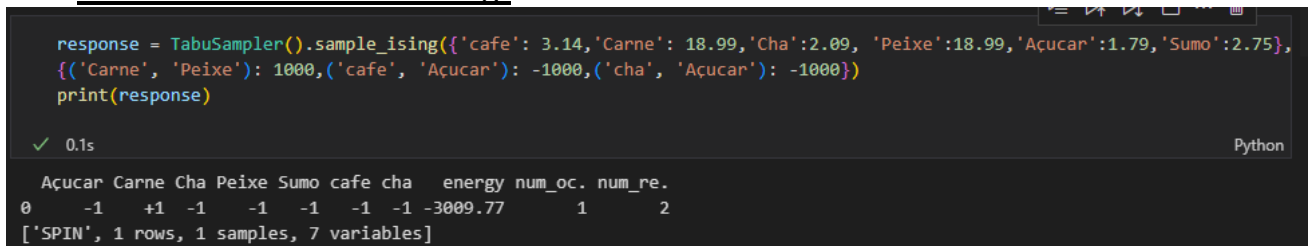
### 3. Exemplos iniciais de aplicação da metodologia

Na fase inicial do projeto, foram abordados pequenos exemplos de aplicação do método *Tabu Search* na resolução de problemas, relativos a representação e explicação de tomadas de decisão. Para tal, implementou-se um exemplo no qual se pretende proceder à compra de produtos alimentares com valorações/preferências independentes e a valoração da relação entre as decisões. Para tal, decidiu-se aplicar decisões independentes, decisões em conjunto (comprar café implica comprar açúcar e comprar chá implica comprar açúcar) e decisões

que não podem ser tomadas em conjunto (não comprar carne e peixe em simultâneo).

Numa primeira aplicação utilizou-se o método **sample\_ising** da classe **TabuSampler** que aplica o método *Tabu Search* ao modelo de Ising. Na segunda aplicação utilizou-se o método **sample\_qubo**, da mesma classe, dado que aplica o método *Tabu Search* ao modelo QUBO.

### Utilizando o modelo de Ising:



```
response = TabuSampler().sample_ising({'cafe': 3.14, 'Carne': 18.99, 'Cha': 2.09, 'Peixe': 18.99, 'Açucar': 1.79, 'Sumo': 2.75},
{{'Carne', 'Peixe': 1000, ('cafe', 'Açucar'): -1000, ('cha', 'Açucar'): -1000}}
print(response)
```

✓ 0.1s Python

	Açucar	Carne	Cha	Peixe	Sumo	cafe	cha	energy	num_oc.	num_re.
0	-1	+1	-1	-1	-1	-1	-1	-3009.77	1	2

['SPIN', 1 rows, 1 samples, 7 variables]

Figura 2- Aplicação do modelo Ising

Através da Figura 2, visualiza-se que se começou por atribuir o custo de cada produto e de seguida a relação entre os produtos, em caso de omissão considera-se a relação nula (significando decisões independentes).

Como referido anteriormente o modelo de Ising pretende minimizar a energia, e desta forma, interações negativas como café e açúcar igual a -1000 são as interações que o modelo seleciona como “boas” (sendo estas as decisões de compra em simultâneo). Por fim, as interações positivas no exemplo entre a carne e o peixe são interações que o modelo elege como “más”, desta forma correspondem à decisão de não comprar em simultâneo.

Desta forma, para este exemplo a notação da energia para o modelo de Ising é dada por:

$$H = -(1000\sigma_1\sigma_3 - 1000\sigma_0\sigma_4 - 1000\sigma_2\sigma_4) - (3.14\sigma_0 + 18.99\sigma_1 + 2.09\sigma_2 + 18.99\sigma_3 + 1.79\sigma_4 + 2.75\sigma_5),$$

onde os a correspondência entre os índices e os produtos é a seguinte: 0: café, 1: carne, 2: chá, 3: peixe, 4: açúcar, 5: sumo.

A solução é obtida para a energia igual a -3009.77. Neste caso, o modelo selecionou todos os produtos, exceto a carne. Pode-se concluir que a escolha em simultâneo da carne e do peixe é desfavorável, resultando numa penalização. O modelo também poderia ter excluído o peixe e mantido a carne, maximizando assim a escolha de produtos para um consumo de energia menor.

### Utilizando o modelo QUBO:

Utilizou-se o mesmo exemplo para aplicar o modelo QUBO. Neste caso é necessário criar a matriz QUBO: na diagonal tem o custo dos produtos e nas restantes entradas a interação entre os produtos.

```
sampler = TabuSampler()
Q = {
    (0, 0): 3.14, # x_0 (café)
    (1,1): 18.99, #x_1 (carne)
    (2,2): 2.09, #x_2 (chá)
    (3,3):18.99, # x_3 (peixe)
    (4,4):1.79, # x_4 (açúcar)
    (5,5):2.75, #x_5 (sumo)
    (1, 3): 1000, # x_1 com x_3 (carne e peixe)
    (0,4): -1000, # x_0 com x_4 (café e açúcar)
    (2,4):-1000 #x_2 com x_4 (chá e açúcar)
}
response = sampler.sample_qubo(Q)
print(response)
```

✓ 0.0s

	0	1	2	3	4	5	energy	num_oc.	num_re.
0	1	0	1	0	1	0	-1992.98	1	2

['BINARY', 1 rows, 1 samples, 6 variables]

Figura 3- Aplicação do modelo QUBO

A matriz QUBO associada a este exemplo é:

$$\begin{bmatrix} 3.14 & 0 & 0 & 0 & -1000 & 0 \\ 0 & 18.99 & 0 & 1000 & 0 & 0 \\ 0 & 0 & 2.09 & 0 & -1000 & 0 \\ 0 & 1000 & 0 & 18.99 & 0 & 0 \\ -1000 & 0 & -1000 & 0 & 1.79 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2.75 \end{bmatrix},$$

onde os a correspondência entre os índices e os produtos é a seguinte: 0: café, 1: carne, 2: chá, 3: peixe, 4: açúcar, 5: sumo.

Na figura 3 observa-se que a solução é dada para a energia de -1992.98 na qual se escolhe os seguintes produtos: café, chá e açúcar. No modelo QUBO, ao contrário do modelo de Ising, não foram selecionados todos os produtos, mesmo que estes não tenham uma penalização associada. O sumo, a carne e o peixe foram excluídos provavelmente de forma a obter o menor número de energia possível, pois excluí qualquer produto que não tenha uma combinação que reduza a energia.

## 4. Problema do caixeiro-viajante

O Problema do Caixeiro Viajante (PCV) /*The Traveling Salesman Problem (TSP)* é um problema clássico de otimização combinatória. Neste problema, pretende-se visitar um conjunto de cidades, passando por cada cidade uma única vez, e regressar ao ponto de partida, percorrendo o menor caminho possível. O objetivo é encontrar a rota que minimiza a distância total percorrida.

De uma forma mais concreta, no PCV é dada uma matriz D (de dimensão nxn) das distâncias entre n cidades, na qual o elemento  $d_{ij}$  representa a distância do percurso entre as cidades i e j e pretende-se determinar a rota de distância mínima que passa por cada cidade uma e uma só vez.

Considerando que este é um problema de otimização, vamos assumir que o custo da viagem é representado pela distância total percorrida.

Existem diferentes maneiras de expressar este problema. Consideremos a seguinte formalização:

Variáveis de decisão:

$$x_{t,i} = \begin{cases} 1, & \text{se a cidade } i \text{ é visitada no instante/posição } t \\ 0, & \text{caso contrário} \end{cases}$$

Função objetivo (minimizar o custo (distância)):

$$\text{Min} \sum_{i=0}^{n-1} \sum_{j=0}^{n-1} \sum_{t=0}^{n-1} d_{ij} x_{ti} x_{t+1,j}$$

Sendo que, para  $t = n - 1$ ,  $t + 1 = n$ , e esta posição terá o significado de voltar à primeira posição da rota, consideramos, para  $t = n - 1$ ,  $t + 1 = 0$

Sujeito a:

Restrição 1: (Garantir que cada cidade é visitada exatamente uma vez)

$$\forall i \in \{0, 1, \dots, n\}: \sum_{t=0}^n x_{t,i} = 1$$

Restrição 2: (Garantir que apenas é visitada uma cidade por cada instante)

$$\forall t \in \{0, 1, \dots, n\}: \sum_{i=0}^n x_{t,i} = 1$$

#### 4.1. Implementação para 4 cidades

Nesta secção iremos resolver o problema do caixeiro-viajante para 4 cidades: vamos supor que se pretende encontrar o caminho que minimiza as distâncias entre 4 cidades (Braga, Barcelos, Guimarães e Famalicão) passando uma única vez por cada cidade com exceção da cidade na posição final que coincide com a cidade correspondente ao ponto de partida.

A matriz D das distâncias, onde representa a distância entre a cidade i e a cidade j, é dada na tabela seguinte:

Tabela 2-Matriz de distâncias

	Braga	Barcelos	Guimarães	Famalicão
Braga	0	23	23	24
Barcelos	23	0	40	20
Guimarães	23	40	0	23

Famalicão	24	20	23	0
-----------	----	----	----	---

Variáveis de decisão:

$$x_{t,i} = \begin{cases} 1, & \text{se a cidade } i \text{ é visitada no instante/posição } t \\ 0, & \text{caso contrário} \end{cases},$$

$$t \in \{0,1,2,3\}, i \in \{0,1,2,3\}$$

Função objetivo (minimizar o custo (distância)):

$$\text{Min} \sum_{i=0}^3 \sum_{j=0}^3 \sum_{t=0}^3 d_{ij} x_{ti} x_{t+1,j}$$

sendo que, para  $t = 3$ , consideramos  $t + 1 = 0$ .

Sujeito a:

Restrição 1: (Garantir que cada cidade é visitada exatamente uma vez)

$$\forall i \in \{0,1,2,3\}: \sum_{t=0}^3 x_{t,i} = 1$$

Restrição 2: (Garantir que apenas é visitada uma cidade por cada instante)

$$\forall t \in \{0,1,2,3\}: \sum_{i=0}^3 x_{t,i} = 1$$

#### 4.1.1. Aplicação do modelo QUBO

Criaram-se três matrizes auxiliares denominadas de Q1, Q2 e Q3 sendo que a matriz QUBO corresponderá a soma destas três matrizes. As matrizes Q1, Q2 e Q3 representarão a função objetivo, a restrição 1 e restrição 2, respetivamente.

$$\begin{aligned} \triangleright Q_1 &= \sum_{i=0}^3 \sum_{j=0}^3 \sum_{t=0}^3 d_{ij} x_{ti} x_{t+1,j} \\ \triangleright Q_2 &= \forall i \in \{0,1,2,3\}: P_2 ((\sum_{t=0}^3 x_{t,i}) - 1)^2 \\ \triangleright Q_3 &= \forall t \in \{0,1,2,3\}: P_3 ((\sum_{i=0}^3 x_{t,i}) - 1)^2 \end{aligned}$$

$$Q = Q_1 + Q_2 + Q_3,$$

onde  $P_2$  e  $P_3$  representam as penalizações associadas às restrições 1 e 2, respetivamente.

Aa matrizes Q1, Q2 e Q3 terão a dimensão 16x16 devido à existência de 16 combinações possíveis para as variáveis de decisão  $x_{t,i}$ .

**Desenvolvimento de Q2:**

Quando  $i=0$ :

$$\begin{aligned}
 & (x_{0,0} + x_{1,0} + x_{2,0} + x_{3,0} - 1)^2 \\
 &= x_{0,0}^2 + x_{1,0}^2 + x_{2,0}^2 + x_{3,0}^2 + 2(x_{0,0}x_{1,0} + x_{0,0}x_{2,0} + x_{0,0}x_{3,0} + x_{1,0}x_{2,0} \\
 &\quad + x_{1,0}x_{3,0} + x_{2,0}x_{3,0}) - 2(x_{0,0} + x_{1,0} + x_{2,0} + x_{3,0}) + 1 \\
 &= -x_{0,0}^2 - x_{1,0}^2 - x_{2,0}^2 - x_{3,0}^2 + 2(x_{0,0}x_{1,0} + x_{0,0}x_{2,0} + x_{0,0}x_{3,0} + x_{1,0}x_{2,0} \\
 &\quad + x_{1,0}x_{3,0} + x_{2,0}x_{3,0}) + 1
 \end{aligned}$$

Importante referir que  $x_{t,i}^2 = x_{t,i}$ . Obtém-se o resultado de modo análogo para  $i=1,2$  e  $3$ .

**Desenvolvimento de Q3:**

Quando  $t=0$ :

$$\begin{aligned}
 & (x_{0,0} + x_{0,1} + x_{0,2} + x_{0,3} - 1)^2 = x_{0,0}^2 + 2x_{0,0}x_{0,1} + 2x_{0,0}x_{0,2} + 2x_{0,0}x_{0,3} \\
 &\quad - 2x_{0,0} + x_{0,1}^2 + 2x_{0,1}x_{0,2} + 2x_{0,1}x_{0,3} - 2x_{0,1} + x_{0,2}^2 + 2x_{0,2}x_{0,3} \\
 &\quad - 2x_{0,2} + x_{0,3}^2 - 2x_{0,3} + 1 \\
 &= x_{0,0}^2 + x_{0,1}^2 + x_{0,2}^2 + x_{0,3}^2 + 2(x_{0,0}x_{0,1} + x_{0,0}x_{0,2} + x_{0,0}x_{0,3} + x_{0,1}x_{0,2} \\
 &\quad + x_{0,1}x_{0,3} + x_{0,2}x_{0,3}) - 2(x_{0,0} + x_{0,1} + x_{0,2} + x_{0,3}) + 1 \\
 &= -x_{0,0}^2 - x_{0,1}^2 - x_{0,2}^2 - x_{0,3}^2 + 2(x_{0,0}x_{0,1} + x_{0,0}x_{0,2} + x_{0,0}x_{0,3} + x_{0,1}x_{0,2} \\
 &\quad + x_{0,1}x_{0,3} + x_{0,2}x_{0,3}) + 1
 \end{aligned}$$

Notar que  $x_{t,i}^2 = x_{t,i}$

Quando  $t=1$ :

$$\begin{aligned}
 & (x_{1,0} + x_{1,1} + x_{1,2} + x_{1,3} - 1)^2 \\
 &= x_{1,0}^2 + x_{1,1}^2 + x_{1,2}^2 + x_{1,3}^2 + 2(x_{1,0}x_{1,1} + x_{1,0}x_{1,2} + x_{1,0}x_{1,3} + x_{1,1}x_{1,2} \\
 &\quad + x_{1,1}x_{1,3} + x_{1,2}x_{1,3}) - 2(x_{1,0} + x_{1,1} + x_{1,2} + x_{1,3}) + 1 \\
 &= -x_{1,0}^2 - x_{1,1}^2 - x_{1,2}^2 - x_{1,3}^2 + 2(x_{1,0}x_{1,1} + x_{1,0}x_{1,2} + x_{1,0}x_{1,3} + x_{1,1}x_{1,2} \\
 &\quad + x_{1,1}x_{1,3} + x_{1,2}x_{1,3}) + 1
 \end{aligned}$$

E assim sucessivamente para  $t=2$  e  $3$ .

**4.1.2. Implementação em *python***

Iniciou-se a implementação deste problema em *python* com a criação da matriz de distâncias (Tabela 2). Em seguida, foi desenvolvida uma função auxiliar denominada de **tradutor** para calcular o índice de cada cidade nas matrizes

QUBO. Uma vez que as matrizes QUBO para as 4 cidades tem dimensão 16x16, foi necessário realizar essa tradução.

Posteriormente, elaboraram-se as matrizes Q1, Q2 e Q3 mencionadas anteriormente, conforme demonstrado no código apresentado nas Figuras 4, 5 e 6, respetivamente.

```
Q1 = np.zeros((16, 16))
for i in range(n):
    for j in range(n):
        for t in range(n):
            linha = tradutor(t, i,n)
            if t == 3:
                coluna = tradutor(0, j,n)
            else:
                coluna = tradutor(t+1, j,n)
            Q1[linha][coluna] = d[i][j]/2
            Q1[coluna][linha] = d[i][j]/2
```

Figura 4- Matriz Q1

```
Q2 = np.zeros((16, 16))
P2= 50
for i in range(n):
    for t in range(n):
        linha = tradutor(t, i,n)
        for tt in range(n):
            coluna = tradutor(tt, i,n)
            if t == tt:
                Q2[linha][coluna] -= 1
            else:
                Q2[linha][coluna] += 2/2
```

Figura 5- Matriz Q2

```
Q3 = np.zeros((16, 16))
P3=50
for t in range(n):
    for i in range(n):
        linha = tradutor(t, i,n)
        for j in range(n):
            coluna = tradutor(t, j,n)
            if i == j:
                Q3[linha][coluna] -= 1
            else:
                Q3[linha][coluna] += 2/2
```

Figura 6- Matriz Q3

Por fim, a matriz Q foi calculada como a soma das matrizes Q1, Q2 e Q3 ( $Q = Q1 + Q2 + Q3$ ). Em seguida, o modelo QUBO foi aplicado para obter a solução. Para isso, foram utilizadas as seguintes funções:

**sampler = TabuSampler()** e **response = sampler.sample\_qubo(Q)**.

### 4.1.3. Resultados

A solução é dada pela sequência resultante do **response**. Numa das simulações, a solução obtida foi [0, 1, 0, 0, 1, 0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 1] que representa o percurso [1, 0, 2, 3, 1]. Ou seja, a solução corresponde ao caminho Barcelos-Braga-Guimarães-Famalicão-Barcelos.

A matriz Q que levou a este resultado foi obtida a partir das matrizes Q1, Q2 e Q3 representadas na tabela 3, 4 e 5, respetivamente.

Tabela 3- Matriz Q1

	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$
$x_{0,0}$	0	0	0	0	0	11,5	11,5	12	0	0	0	0	0	11,5	11,5	12
$x_{0,1}$	0	0	0	0	11,5	0	20	10	0	0	0	0	11,5	0	20	10
$x_{0,2}$	0	0	0	0	11,5	20	0	11,5	0	0	0	0	11,5	20	0	11,5
$x_{0,3}$	0	0	0	0	12	10	11,5	0	0	0	0	0	12	10	11,5	0
$x_{1,0}$	0	11,5	11,5	12	0	0	0	0	0	11,5	11,5	12	0	0	0	0
$x_{1,1}$	11,5	0	20	10	0	0	0	0	11,5	0	20	10	0	0	0	0
$x_{1,2}$	11,5	20	0	11,5	0	0	0	0	11,5	20	0	11,5	0	0	0	0
$x_{1,3}$	12	10	11,5	0	0	0	0	0	12	10	11,5	0	0	0	0	0
$x_{2,0}$	0	0	0	0	0	11,5	11,5	12	0	0	0	0	0	11,5	11,5	12
$x_{2,1}$	0	0	0	0	11,5	0	20	10	0	0	0	0	11,5	0	20	10
$x_{2,2}$	0	0	0	0	11,5	20	0	11,5	0	0	0	0	11,5	20	0	11,5
$x_{2,3}$	0	0	0	0	12	10	11,5	0	0	0	0	0	12	10	11,5	0
$x_{3,0}$	0	11,5	11,5	12	0	0	0	0	0	11,5	11,5	12	0	0	0	0
$x_{3,1}$	11,5	0	20	10	0	0	0	0	11,5	0	20	10	0	0	0	0
$x_{3,2}$	11,5	20	0	11,5	0	0	0	0	11,5	20	0	11,5	0	0	0	0
$x_{3,3}$	12	10	11,5	0	0	0	0	0	12	10	11,5	0	0	0	0	0

Tabela 4- Matriz Q2

	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$
$x_{0,0}$	-1	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0
$x_{0,1}$	0	-1	0	0	0	1	0	0	0	1	0	0	0	1	0	0
$x_{0,2}$	0	0	-1	0	0	0	1	0	0	0	1	0	0	0	1	0
$x_{0,3}$	0	0	0	-1	0	0	0	1	0	0	0	1	0	0	0	1
$x_{1,0}$	1	0	0	0	-1	0	0	0	1	0	0	0	1	0	0	0
$x_{1,1}$	0	1	0	0	0	-1	0	0	0	1	0	0	0	1	0	0
$x_{1,2}$	0	0	1	0	0	0	-1	0	0	0	1	0	0	0	1	0



$x_{1,3}$	0	0	0	1	0	0	0	-1	0	0	0	1	0	0	0	1
$x_{2,0}$	1	0	0	0	1	0	0	0	-1	0	0	0	1	0	0	0
$x_{2,1}$	0	1	0	0	0	1	0	0	0	-1	0	0	0	1	0	0
$x_{2,2}$	0	0	1	0	0	0	1	0	0	0	-1	0	0	0	1	0
$x_{2,3}$	0	0	0	1	0	0	0	1	0	0	0	-1	0	0	0	1
$x_{3,0}$	1	0	0	0	1	0	0	0	1	0	0	0	-1	0	0	0
$x_{3,1}$	0	1	0	0	0	1	0	0	0	1	0	0	0	-1	0	0
$x_{3,2}$	0	0	1	0	0	0	1	0	0	0	1	0	0	0	-1	0
$x_{3,3}$	0	0	0	1	0	0	0	1	0	0	0	1	0	0	0	-1

Tabela 5- Matriz  $Q3$ 

	$x_{0,0}$	$x_{0,1}$	$x_{0,2}$	$x_{0,3}$	$x_{1,0}$	$x_{1,1}$	$x_{1,2}$	$x_{1,3}$	$x_{2,0}$	$x_{2,1}$	$x_{2,2}$	$x_{2,3}$	$x_{3,0}$	$x_{3,1}$	$x_{3,2}$	$x_{3,3}$
$x_{0,0}$	-1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
$x_{0,1}$	1	-1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
$x_{0,2}$	1	1	-1	1	0	0	0	0	0	0	0	0	0	0	0	0
$x_{0,3}$	1	1	1	-1	0	0	0	0	0	0	0	0	0	0	0	0
$x_{1,0}$	0	0	0	0	-1	1	1	1	0	0	0	0	0	0	0	0
$x_{1,1}$	0	0	0	0	1	-1	1	1	0	0	0	0	0	0	0	0
$x_{1,2}$	0	0	0	0	1	1	-1	1	0	0	0	0	0	0	0	0
$x_{1,3}$	0	0	0	0	1	1	1	-1	0	0	0	0	0	0	0	0
$x_{2,0}$	0	0	0	0	0	0	0	0	-1	1	1	1	0	0	0	0
$x_{2,1}$	0	0	0	0	0	0	0	0	1	-1	1	1	0	0	0	0
$x_{2,2}$	0	0	0	0	0	0	0	0	1	1	-1	1	0	0	0	0
$x_{2,3}$	0	0	0	0	0	0	0	0	1	1	1	-1	0	0	0	0
$x_{3,0}$	0	0	0	0	0	0	0	0	0	0	0	0	-1	1	1	1
$x_{3,1}$	0	0	0	0	0	0	0	0	0	0	0	0	1	-1	1	1
$x_{3,2}$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	-1	1
$x_{3,3}$	0	0	0	0	0	0	0	0	0	0	0	0	1	1	1	-1

## 5. Comparação do esforço computacional

Com o objetivo de comparar o esforço computacional necessário para resolver o problema do caixeiro-viajante utilizando o modelo clássico e o modelo QUBO, decidiu-se verificar o número máximo de cidades que cada modelo conseguiria processar dentro do período de 1 minuto em cada algoritmo.

- **Modelo Clássico**

Para avaliar o esforço computacional utilizando o modelo clássico implementou-se o ciclo ilustrado na Figura 7. O algoritmo é inicializado com 3 cidades no qual o ciclo é interrompido caso o algoritmo exceda 1 minuto de execução.

```

while True:
    start_time = time.time()
    # Add a new city with random distances
    city_count += 1
    new_city = f"City{city_count}"
    cidades.append(new_city)
    for city in cidades[:-1]:
        dist = random.randint(20, 400)
        distancias[(city, new_city)] = dist
        distancias[(new_city, city)] = dist
    # Gera todas as permutações das cidades
    permutacoes = itertools.permutations(cidades)
    # Inicializa a menor distância como infinito
    menor_distancia = float('inf')
    # Itera sobre todas as permutações e calcula a distância total
    for permutacao in permutacoes:
        distancia_total = 0
        for i in range(len(permutacao) - 1):
            distancia = distancias.get((permutacao[i], permutacao[i+1]), 0)
            if distancia == 0:
                distancia = distancias.get((permutacao[i+1], permutacao[i]), 0)
            distancia_total += distancia
        distancia_total += distancias.get((permutacao[-1], permutacao[0]), 0)
        # Atualiza a menor distância
        if distancia_total < menor_distancia:
            menor_distancia = distancia_total
            melhor_rota = permutacao
    # Calculate elapsed time
    elapsed_time = time.time() - start_time
    print(elapsed_time)
    if elapsed_time > 60:
        break

```

Figura 7- Performance do algoritmo clássico

Na execução deste algoritmo, o ciclo foi interrompido após processar 11 cidades, resultando no tempo de execução de 106 segundos. Isto indica que o número máximo de cidades para obter uma solução em 1 minuto é 10, demorando 9 segundos. Esta discrepância acentuada de tempo revela o verdadeiro custo computacional do modelo clássico, devido ao seu crescimento fatorial, podendo assim em casos práticos revelar-se demasiado dispendioso.

- **Modelo QUBO**

Para avaliar o esforço computacional utilizando o modelo QUBO criou-se a função **CidadesMax()** representada na Figura 8 na qual se irá criar a matriz QUBO e aplicá-la ao sampler parando quando exceder 1 minuto.

```

def CidadesMax():
    sampler = TabuSampler()
    TEMPO=0
    n=131
    while TEMPO < 60:
        n += 1
        start = time.time()
        matriz=gerar_matriz(n)
        Q1,Q2,Q3,Q4 = QubosCreator(n,matriz,"Braga")
        Q = Q1+Q2+Q3+Q4
        response = sampler.sample_qubo(Q)
        end = time.time()
        TEMPO = end - start
        print(TEMPO)
    print(n)
    print(TEMPO)

```

Figura 8- Performance do algoritmo QUBO

A execução da função foi interrompida ao processar 132 cidades, com um tempo de execução de 67,5 segundos. Portanto, o número máximo de cidades para o algoritmo encontrar uma solução em 1 minuto é de 131 cidades, com um tempo de execução de 53,18 segundos.

É importante ressaltar que esses valores podem variar dependendo da capacidade computacional da máquina em que o código é executado.

No entanto, embora podendo não encontrar a solução ótima, pode-se concluir que o modelo QUBO apresenta um desempenho superior ao modelo clássico na resolução do problema do caixeiro-viajante, especialmente quando se aborda problemas de grande escala.

## 6. PCV para os distritos de Portugal continental

Nesta secção pretende-se solucionar o problema do caixeiro-viajante para os 18 distritos de Portugal continental, considerando o custo como sendo a distância entre os distritos. A formulação deste problema é análoga à explicitada no exemplo para 4 cidades. No entanto, neste caso, foi determinado que a cidade inicial será fixa, ou seja, a primeira cidade do percurso será definida. Tal restrição será especificada mais adiante.

Dado que será necessário obter a matriz das distâncias, decidiu-se criar essa matriz através das coordenadas de cada distrito. Para isso, foi necessária a elaboração de algumas funções auxiliares.

Na Figura 9, apresentam-se as funções **degrees\_to\_radians(degrees)** e **calcular\_distancia(coord1, coord2)**. A primeira converte o valor de graus em radianos e a segunda calcula a distância em quilómetros entre duas coordenadas geográficas.

```
def degrees_to_radians(degrees):
    return degrees * np.pi / 180.0

# Função para calcular a distância em km entre duas coordenadas geográficas
def calcular_distancia(coord1, coord2):
    lat1, lon1 = coord1
    lat2, lon2 = coord2
    radius = 6371 # Raio médio da Terra em km
    dlat = degrees_to_radians(lat2 - lat1)
    dlon = degrees_to_radians(lon2 - lon1)
    a = np.sin(dlat/2) * np.sin(dlat/2) + np.cos(degrees_to_radians(lat1)) * np.cos(degrees_to_radians(lat2)) * np.sin(dlon/2) * np.sin(dlon/2)
    c = 2 * np.arctan2(np.sqrt(a), np.sqrt(1-a))
    distance = radius * c
    return distance
```

Figura 9- Cálculo das distâncias

Desta forma, criou-se um dicionário com as coordenadas correspondentes a cada distrito (Figura 10) e um dicionário contendo o *label encoding* para cada cidade (Figura 11).

```

coordenadas = {
    'Aveiro': (40.6405, -8.6538),
    'Beja': (38.0151, -7.8631),
    'Braga': (41.5454, -8.4265),
    'Bragança': (41.8071, -6.7583),
    'Castelo Branco': (39.8238, -7.4937),
    'Coimbra': (40.2110, -8.4293),
    'Évora': (38.5737, -7.9077),
    'Faro': (37.0179, -7.9307),
    'Guarda': (40.5370, -7.2673),
    'Leiria': (39.7442, -8.8070),
    'Lisboa': (38.7223, -9.1393),
    'Portalegre': (39.2938, -7.4313),
    'Porto': (41.1496, -8.6109),
    'Santarém': (39.2362, -8.6850),
    'Setúbal': (38.5244, -8.8945),
    'Viana do Castelo': (41.6918, -8.8349),
    'Vila Real': (41.3005, -7.7437),
    'Viseu': (40.6610, -7.9097)
}

```

Figura 10- Dicionário das coordenadas

```

legenda = {
    'Aveiro': 0,
    'Beja': 1,
    'Braga': 2,
    'Bragança': 3,
    'Castelo Branco': 4,
    'Coimbra': 5,
    'Évora': 6,
    'Faro': 7,
    'Guarda': 8,
    'Leiria': 9,
    'Lisboa': 10,
    'Portalegre': 11,
    'Porto': 12,
    'Santarém': 13,
    'Setúbal': 14,
    'Viana do Castelo': 15,
    'Vila Real': 16,
    'Viseu': 17
}

```

Figura 11- Label encoding

Depois disto, aplicaram-se as coordenadas à função **calcular\_distancia(coord1, coord2)** e armazenaram-se as distâncias obtidas numa matriz (**matriz\_distancias**).

Com o objetivo de otimizar o código, elaborou-se uma função denotada de **QubosCreator** que recebe como parâmetros o número de cidades a matriz de distâncias e qual cidade correspondente ao ponto de partida. Esta função cria as matrizes Q1, Q2, Q3 e Q4. Estas matrizes são desenvolvidas de modo análogo ao exemplo para as 4 cidades. A matriz Q4 representa a restrição na qual é fixada a cidade inicial.

### Restrição 3:

$x_{0,i} = 1$  , onde i corresponde à cidade visitada na posição inicial

$$Q_4 = \{i : \text{cidade inicial}\} : P_3 (x_{0,i} - 1)^2$$

Desta forma, implementou-se o código presente na Figura 12, na qual a variável **posicao** corresponde ao *label encoding* da cidade definida como cidade inicial.

Assim, observa-se que se atribui um valor negativo (-1) para as entradas que representam a inicialização na cidade definida e um valor positivo (1) para as restantes entradas de inicialização (instante 0). Isto garante que a solução cumpre a restrição 3 devido ao facto de se pretender minimizar a função objetivo.

```

Q4 = np.zeros((dimensao**2, dimensao**2))
P4=400
posicao=legenda[inicio]
for i in range(n):
    linha = tradutor(0, i,n)
    for j in range(n):
        coluna = tradutor(0, j,n)
        if i == posicao or j == posicao:
            Q4[linha][coluna] -= 1
        else:
            Q4[linha][coluna] += 2/2
Q4 = P4 * Q4

```

Figura 12- Matriz Q4

Além disso, também se desenvolveu a função **CaminhoOptimo**, representada na Figura 13, que recebe como parâmetros a matriz de distâncias e a cidade inicial. Esta função irá invocar a função **QubosCreator** para obter as matrizes Q1, Q2, Q3 e Q4 e para calcular a matriz QUBO final ( $Q=Q1+Q2+Q3+Q4$ ), obtendo a solução através da função **sampler.sample\_qubo()**. De seguida, a função **CaminhoOptimo** descodifica a solução obtida, isto é, encontra qual o percurso percorrido.

```

def CaminhoOptimo(d,inicio):
    sampler = TabuSampler()
    n = len(d)
    Q1,Q2,Q3,Q4 = QubosCreator(n,d,inicio)
    Q = Q1+Q2+Q3+Q4
    response = sampler.sample_qubo(Q)
    list_response = list(response.samples())
    #Converter sampler em caminho
    cidades_visitadas = []
    # transformar dicionario em matrix
    caminhoBinario = np.array(list(list_response[0].values()))
    shape = int(math.sqrt(len(caminhoBinario)))
    matriz = np.reshape(caminhoBinario, (shape, shape))
    n = len(matriz)
    for t in range(n):
        cidade = [i for i in range(n) if caminhoBinario[tradutor(t, i,n)] == 1]
        if cidade:
            cidades_visitadas.append(cidade[0])
    cidades_visitadas.append(cidades_visitadas[0])
    print(f"Solução:{cidades_visitadas}")
    return cidades_visitadas

```

Figura 13- Função CaminhoOptimo

Seguidamente, elaborou-se a função **custo**, como se observa na Figura 14, que através do resultado da função **CaminhoOptimo** calcula a distância total percorrida.

```
def custo(d,legenda,inicio):
    caminho= CaminhoOptimo(d,inicio)
    caminho_legendado=[]
    for i in caminho:
        for cidade, indx in legenda.items():
            if i ==indx:
                caminho_legendado.append(cidade)
    print(f"Caminho ótimo: {caminho_legendado}")
    custo=0
    counter = len(caminho)
    for i, j in enumerate(caminho):
        if i == counter-1:
            break
        cidade1= j
        cidade2= caminho[i+1]
        custo+= d[cidade1,cidade2]
    return caminho_legendado,custo
```

Figura 14- Função custo

Como mencionado anteriormente, são empregues métodos de busca local para otimização matemática. Neste sentido, foi decidido realizar um ciclo de 100 iterações, com o objetivo de encontrar uma solução 100 vezes e armazenar aquela que apresentar a menor distância entre todas as soluções obtidas.

```
guardar_resultados={}
for i in range(100):
    ordem_distritos,custo_total=custo(matriz_distancias,legenda,"Braga")
    guardar_resultados[i]= ordem_distritos, custo_total

melhor_custo = float('inf') # inicializar com um valor infinito para encontrar o mínimo
melhor_ordem_distritos = None

for i in guardar_resultados:
    ordem_distritos, custo_total = guardar_resultados[i]
    if custo_total < melhor_custo:
        melhor_custo = custo_total
        melhor_ordem_distritos = ordem_distritos

print("Menor custo_total:", melhor_custo,"km")
print("Ordem distritos correspondente:", melhor_ordem_distritos)
```

Figura 15- Melhor solução

Finalmente, o código previamente mencionado foi aplicado, tendo como ponto de partida inicial a cidade de Braga. A melhor solução foi encontrada para o seguinte percurso ['Braga', 'Vila Real', 'Bragança', 'Guarda', 'Viseu', 'Castelo Branco', 'Portalegre', 'Évora', 'Beja', 'Faro', 'Setúbal', 'Lisboa', 'Santarém', 'Coimbra', 'Leiria', 'Aveiro', 'Porto', 'Viana do Castelo', 'Braga'] correspondente à distância de 1505.4370875674633 km.

Com o propósito de obter uma visualização do percurso obtido recorreu-se à biblioteca *folium* e aplicou-se a função **grafico\_caminho** presente na Figura 16.

```
def grafico_caminho(distritos):
    # Extrai as coordenadas dos distritos
    coordenadas = grafico_coordenadas(distritos)
    # Cria um mapa centrado em Portugal continental
    mapa = folium.Map(location=[39.5, -8], zoom_start=7, tiles='CartoDB Positron')
    # Adiciona marcadores para cada distrito
    for i, coord in enumerate(coordenadas):
        distrito = distritos[i]
        folium.Marker(coord, popup=distrito, icon=folium.Icon(color="red")).add_to(mapa)
    # Adiciona uma linha para ligar os marcadores
    folium.PolyLine(coordenadas, color='black', weight=2.5, opacity=1).add_to(mapa)
    # Guarda o mapa como um ficheiro HTML
    mapa.save("mapa.html")
```

Figura 16- Código para gráfico

Na Figura 17 observa-se o resultado obtido da aplicação da função **grafico\_caminho** ao percurso com menor distância. Visualiza-se assinalado a verde o ponto de partida. Neste caso corresponde à cidade de Braga.

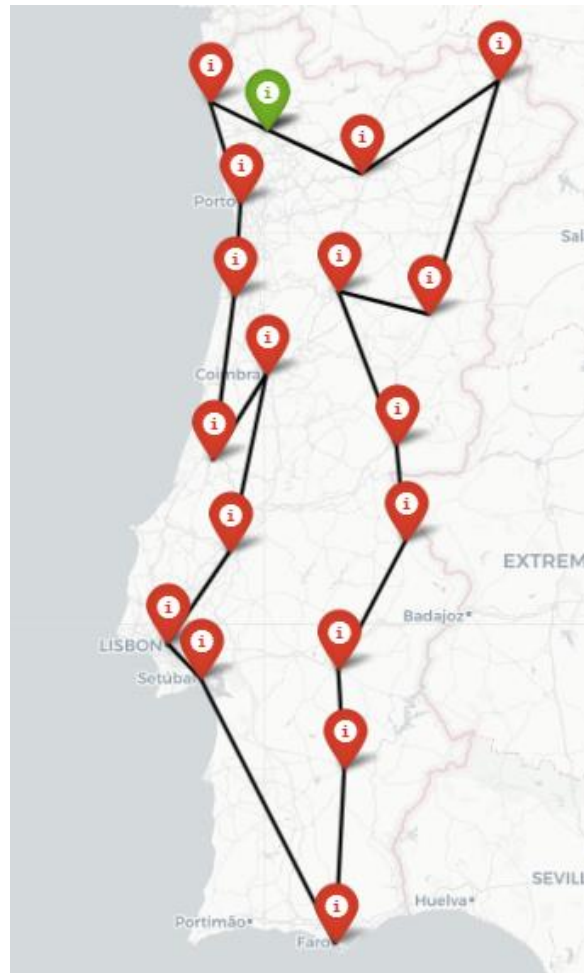


Figura 17- Gráfico

A aparência deste gráfico sugere que representa um percurso eficiente.

## 7. Conclusão

Durante este projeto, aplicamos as competências adquiridas ao longo do nosso percurso acadêmico, como conhecimentos em Álgebra Linear, Algoritmia, Otimização e programação em *Python*. Além disso, aprimoramos habilidades essenciais como trabalho em equipa e gestão de recursos em grupo, especialmente durante o período prolongado de execução do projeto. Além disso, desenvolvemos novas competências como é o caso do conhecimento e aplicação dos modelos de Ising e QUBO ou do método *Tabu Search*.

O principal objetivo do projeto foi resolver problemas de otimização utilizando o modelo QUBO, através de algoritmos inspirados em computação quântica.

Embora as abordagens baseadas em QUBO e processos estocásticos não garantam a solução ótima, estas têm mostrado ser eficazes na resolução de problemas de otimização complexos. Estas metodologias oferecem resultados aproximados eficazes e aceitáveis em muitos cenários práticos, com um custo computacional bastante mais reduzido, destacando a importância de ajustar adequadamente os parâmetros, como as penalizações, para obter soluções de alta qualidade.

A abordagem do modelo QUBO para resolver o problema do caixeiro-viajante para os 18 distritos de Portugal continental mostrou-se promissora e eficiente. A utilização de técnicas de otimização combinatória, juntamente com a aplicação de métodos de busca local, permitiram obter uma solução provavelmente próxima da ótima, com distância percorrida relativamente pequena. Esta abordagem pode ser aplicada em diversos contextos e mostrou ter potencial para otimizar problemas de rotas em diferentes escalas.

Com esforços contínuos de pesquisa e desenvolvimento, espera-se que essas abordagens continuem a avançar e contribuir para a resolução de problemas de otimização cada vez mais desafiadores. Um exemplo promissor de combinação de problemas é a integração do Problema do Caixeiro Viajante (PCV) com aspetos do Problema da Mochila (*Knapsack*), conforme proposto no desafio inicial apresentado pela Fujitsu. Essa combinação apresenta um novo desafio ao considerar a otimização da rota do caixeiro-viajante, levando em conta restrições adicionais relacionadas à capacidade da “mochila”. Ao aplicar a metodologia QUBO, espera-se que seja possível encontrar soluções aproximadas que atendam tanto aos requisitos do PCV quanto do problema *Knapsack*, contribuindo para a resolução eficiente desse problema combinado. Esse exemplo ilustra a versatilidade e o potencial das abordagens baseadas no modelo QUBO para enfrentar desafios de otimização complexos e diversificados.



## **8. Referências**

- [1] Fred Glover, Gary Kochenberger, Yu Du (2018). Quantum Bridge Analytics I: A Tutorial on Formulating and Using QUBO Models. Disponível em <https://arxiv.org/ftp/arxiv/papers/1811/1811.11538.pdf>
- [2] Melody W (outubro 05, 2018). Difference between BQM, Ising, and QUBO problems? Disponível em <https://support.dwavesys.com/hc/en-us/community/posts/360017439853-Difference-between-BQM-Ising-and-QUBO-problems->
- [3] Fred Glover, Rafael Martí (julho, 2008) TABU SEARCH. Disponível em <https://www.uv.es/~rmarti/paper/docs/ts2.pdf>