# IA

# ALGORITMO

Sérgio González Martinez

Lúcia Arnaiz

## contente

# MIRAR

O objetivo deste texto é criar um algoritmo que, dado um grafo (conjunto de pontos conectados entre si por elos) e um inteiro k, encontre um subconjunto de k pontos do grafo que são conectados entre si pelo maior número possível de links. É um problema de maximização, o que significa que queremos encontrar a solução que tenha o maior valor possível.

# ALGORITMO DE PESQUISA LOCAL (GREEDY)

***Explicação de como o algoritmo resolve e funciona:***

1. Classifica a lista de tuplas em ordem decrescente pelo número de vizinhos.
2. Selecione os k vértices com mais vizinhos. Se houver empate, selecione os vértices com mais vizinhos em ordem de aparecimento na lista.
3. Conta o número de arestas entre os vértices selecionados.

***TIPO DE ALGORITMO***

Este algoritmo é uma solução heurística para o problema de encontrar um subconjunto de vértices em um grafo que maximize o número de arestas entre eles.

Este algoritmo usa uma abordagem gulosa , ou seja, toma decisões ótimas locais a cada passo na esperança de chegar a uma solução ótima global.

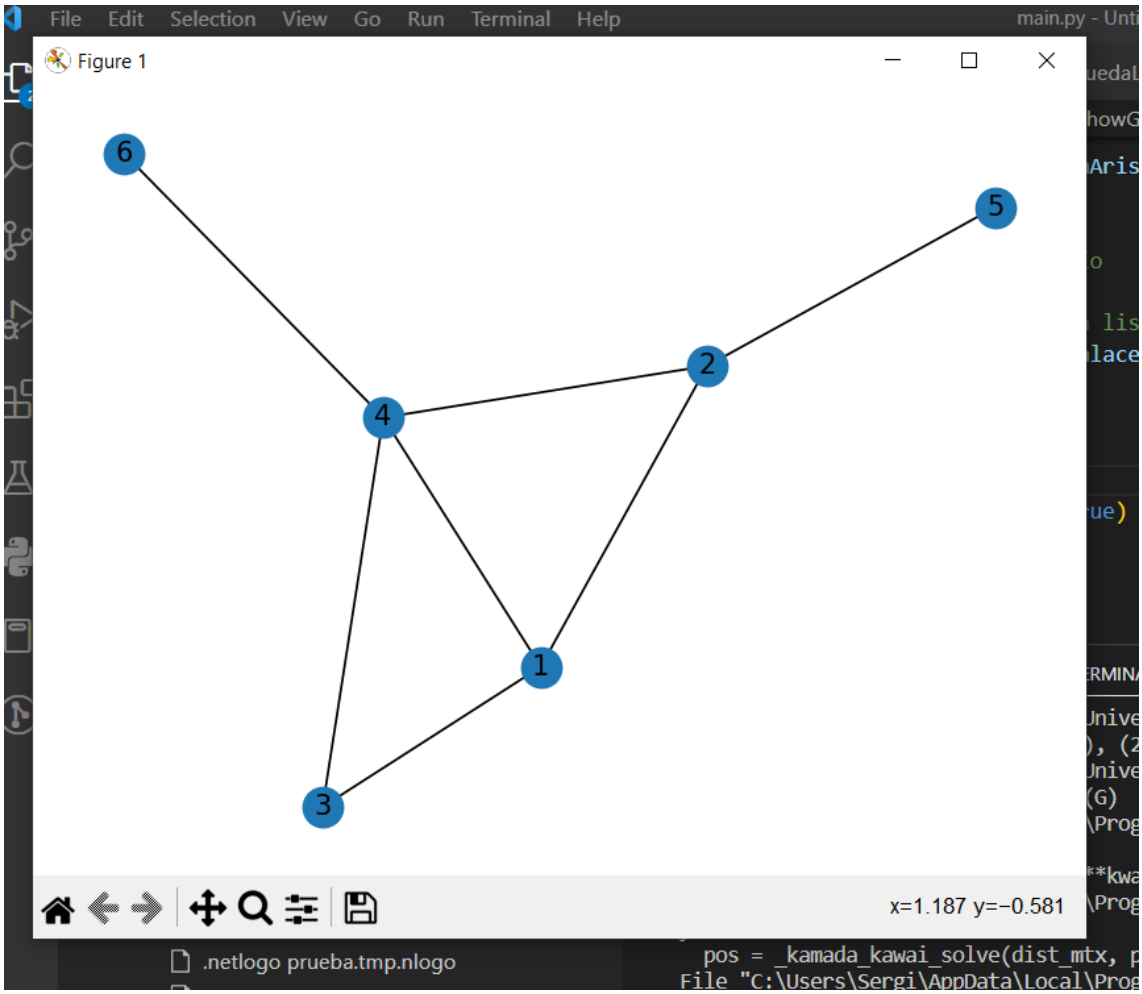
A decisão local ótima **é selecionar a cada passo o vértice com o maior número de vizinhos** , pois tem a melhor chance de ter mais arestas com outros vértices.

***teste***

Vamos tentar com os dados do exemplo do arquivo . . _

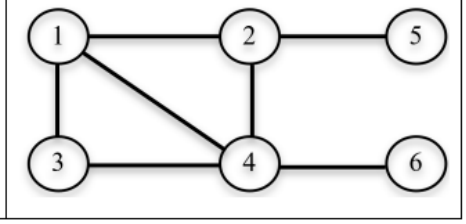
- solução mais ótima deve ser:
  - S2 = {1,2,3,4} -> Número de arestas = 5
- Podemos ver que o gráfico de exemplo é o mesmo que estamos usando usando a seguinte função:

```
def showGraph(listaEnlaces):
        # Crea un grafo vacío
    G = nx.Graph()
    # Añade los enlaces a la lista de enlaces del grafo
    G.add_edges_from(listaEnlaces)
    # Dibuja el grafo
    nx.draw(G, with_labels=True)
    # Muestra el gráfico
    plt.show()
```



**Example:**
Consider a graph with 6 vertices and 7 arcs, represented below (***test.txt*** in Moodle). The image on the right illustrates the graph. The goal is to find a solution of the problem for k=4.

```
k 4
p edge 6 7
e 1 2
e 1 3
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```




Consider the three solutions (there will be others):
S1 = {1, 2, 5, 6} - number of edges = 2
S2 = {1, 2, 3, 4} - number of edges = 5
S3 = {2, 3, 5, 6} - number of edges = 1

```
def AlgoritmoGreedy(kValue,vertices,nAristas,listaEnlaces):
```

Somamos os valores do exemplo:

```
AlgoritmoGreedy(4,6,7,[(1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (3, 4), (4, 6)])
```

E o resultado é:

```
i/Desktop/code/Universidad/Erasmus/IA/p2/Entregable/main.py
[4, 3, 2, 1] -> Number of Edges:  5
PS C:\Users\Sergi\Desktop\code\Universidad\Erasmus\IA>
```

- (Este resultado pode variar dependendo da iteração do algoritmo):

# ALGORITMO EVOLUCIONÁRIO

***Explicação de como o algoritmo resolve e funciona:***

1. Cria uma população inicial de indivíduos aleatórios (soluções candidatas). Cada indivíduo seria uma lista de vértices, de tamanho k, que representa um subconjunto de vértices do grafo.
2. Avalie cada indivíduo na população usando uma função de aptidão que conta o número de arestas entre os vértices do subconjunto.
3. Selecione dois métodos de seleção diferentes para selecionar os indivíduos mais aptos para reprodução. Por exemplo, você pode usar o método de seleção de torneio e o método de seleção de roleta.
4. Dois operadores de recombinação diferentes são aplicados a pares de indivíduos selecionados para criar novos indivíduos.

5. Ele aplica dois operadores de mutação diferentes a alguns dos novos indivíduos para introduzir variabilidade na população.
6. Se algum dos novos indivíduos for inválido (por exemplo, contém vértices repetidos ou mais de k vértices), sua aptidão é reparada ou penalizada de alguma forma.
7. Substitui alguns dos indivíduos da população original pelos novos indivíduos.
8. Repita as etapas 3 a 7 até que o critério de parada seja alcançado (por exemplo, um número máximo de gerações ou uma solução ótima).

***TIPO DE ALGORITMO***

Este código implementa um algoritmo evolutivo, que é um tipo de algoritmo de otimização inspirado no processo de evolução das espécies na natureza.

Algoritmos evolutivos consistem em uma população de indivíduos que representam possíveis soluções para um problema. Através de um processo iterativo, os indivíduos são selecionados da população para reproduzir e criar uma nova geração de indivíduos. Esses indivíduos geralmente apresentam pequenas variações em relação a seus pais, e essas variações são obtidas pela aplicação de operadores de recombinação e mutação. Os indivíduos são avaliados por uma função de aptidão, que mede sua qualidade como solução do problema , ou seja, o número de links em um subconjunto. À medida que novas gerações são geradas, espera-se que o fitness médio da população aumente e que indivíduos cada vez melhores sejam encontrados.

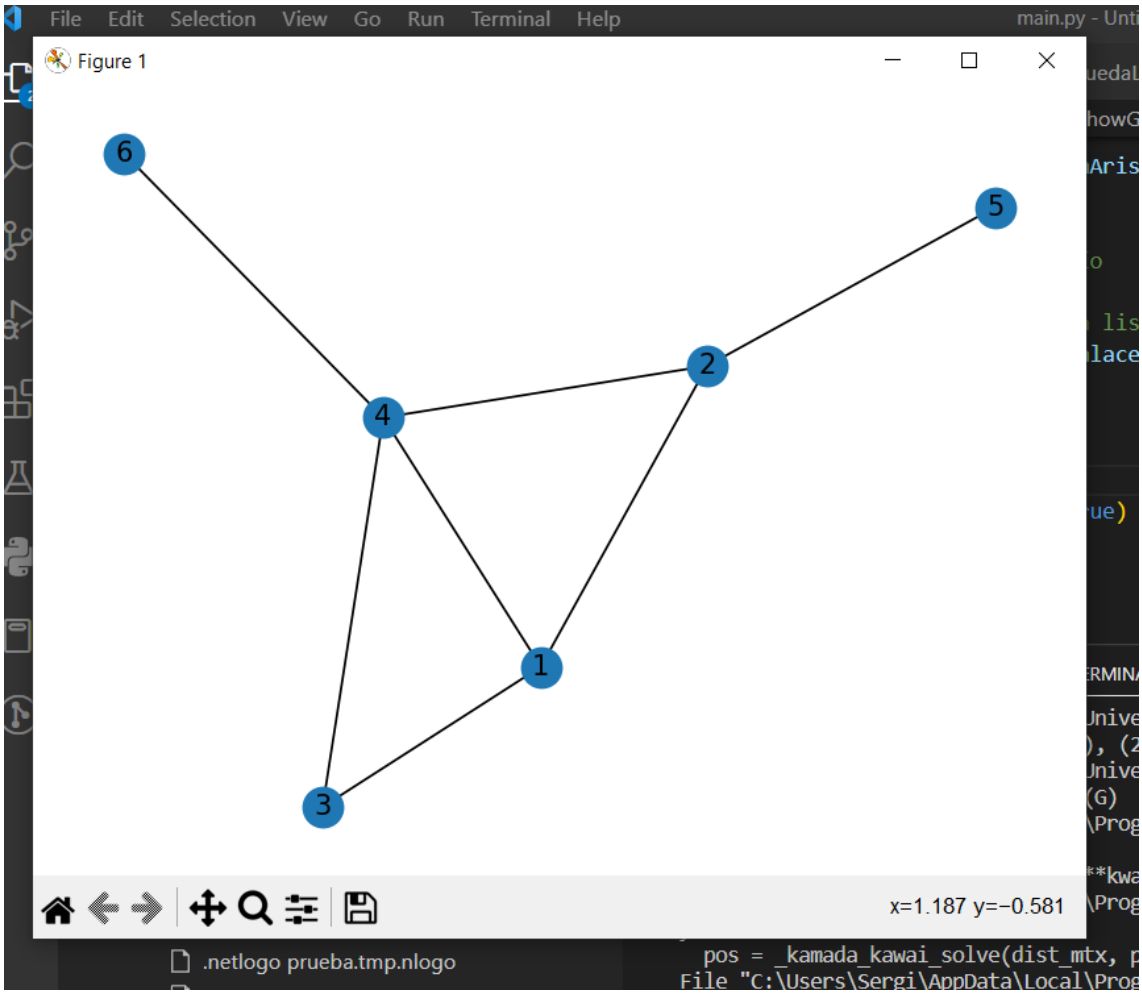
No caso desse código em particular, ele está tentando encontrar um subconjunto de vértices de um grafo que maximize o número de arestas desse subconjunto, desde que pelo menos dois vizinhos diferentes sejam explorados. Para fazer isso, cada indivíduo na população representa um subconjunto de vértices, e a função de aptidão conta o número de arestas dentro do subconjunto. Dois operadores de recombinação diferentes e dois operadores de mutação diferentes são então aplicados para criar uma nova geração a partir dos indivíduos selecionados. Por fim, obtém-se o melhor indivíduo da população.

***teste***

Vamos tentar com os dados do exemplo do arquivo . . _

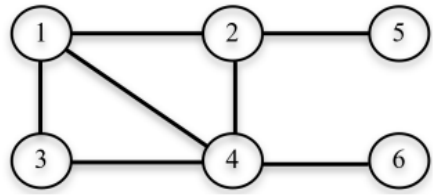
- solução mais ótima deve ser:
  - S2 = {1,2,3,4} -> Número de arestas = 5
- Podemos ver que o gráfico de exemplo é o mesmo que estamos usando usando a seguinte função:

```
def showGraph(listaEnlaces):
        # Crea un grafo vacío
    G = nx.Graph()
    # Añade los enlaces a la lista de enlaces del grafo
    G.add_edges_from(listaEnlaces)
    # Dibuja el grafo
    nx.draw(G, with_labels=True)
    # Muestra el gráfico
    plt.show()
```



**Example:**
Consider a graph with 6 vertices and 7 arcs, represented below (***test.txt*** in Moodle). The image on the right illustrates the graph. The goal is to find a solution of the problem for k=4.

```
k 4
p edge 6 7
e 1 2
e 1 3
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```




Consider the three solutions (there will be others):
S1 = {1, 2, 5, 6} - number of edges = 2
S2 = {1, 2, 3, 4} - number of edges = 5
S3 = {2, 3, 5, 6} - number of edges = 1

```python
def evolutiveAlgorithm():
    # Parámetros del algoritmo
    vertices = 6
    nAristas = 7
    kValue = 4
    listaEnlaces = [(1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (3, 4), (4, 6)]
    poblacion_inicial = 10
    max_generaciones = 100
    tasa_mutacion = 0.1
```

E o resultado é:

```
PS C:\Users\Sergi\Desktop\code\Universidad\Erasmus\IA> & C:/User
i/Desktop/code/Universidad/Erasmus/IA/p2/Entregable/main.py
[4, 2, 3, 1] Number of Edges =  5
PS C:\Users\Sergi\Desktop\code\Universidad\Erasmus\IA>
```

- (Este resultado pode variar dependendo da iteração do algoritmo):

```
i/Desktop/code/Universidad/Erasmus/IA/p2/Entregable/main.py
[5, 4, 2, 1] Number of Edges =  4
```

# ALGORITMO ESTOCÁSTICO DE ESCALADA DE MONTANHAS

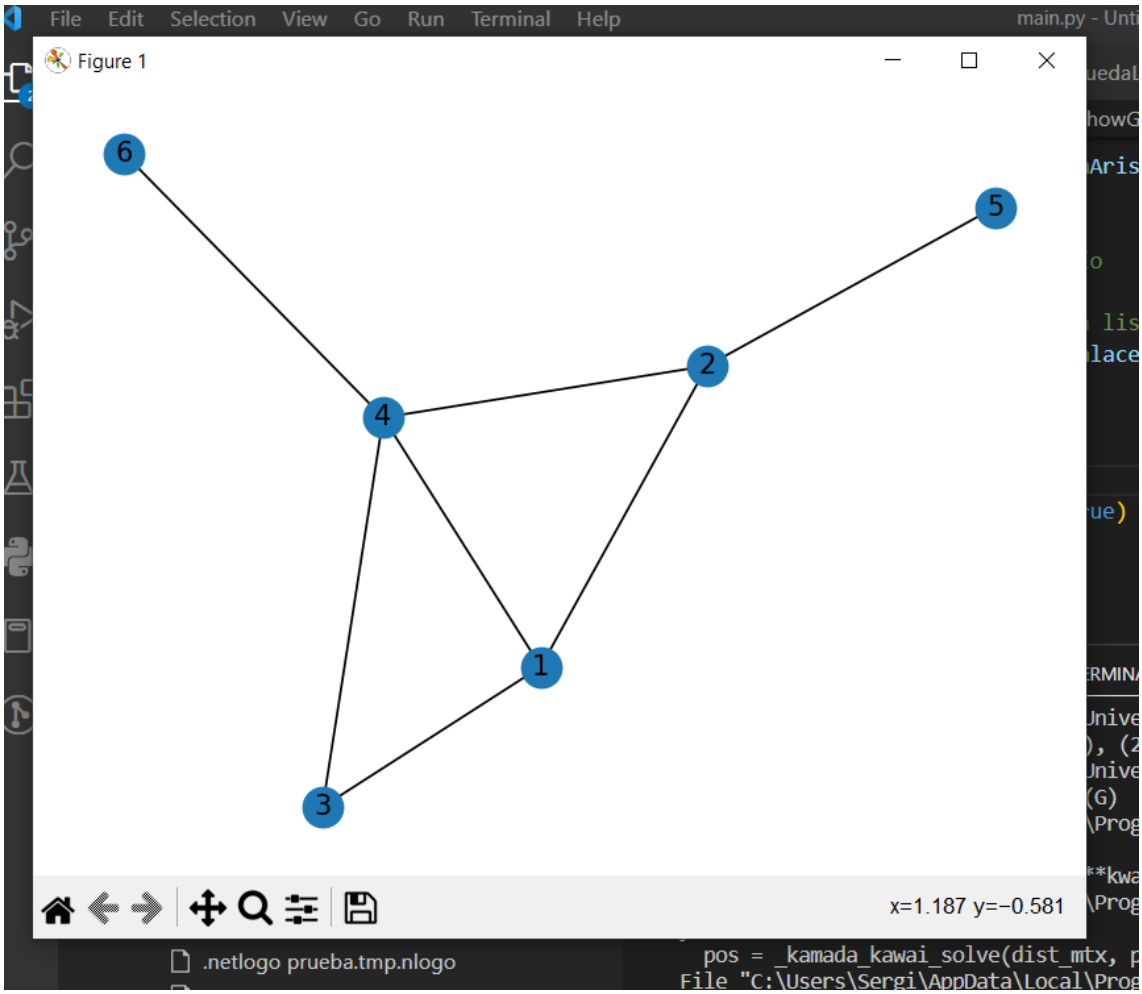
Este algoritmo é uma implementação do algoritmo estocástico de subida de encosta. É um tipo de algoritmo de busca local usado para encontrar uma solução ótima (ou muito próxima da ótima) para um determinado problema. O algoritmo começa com uma solução inicial e, em seguida, iterativamente faz pequenas alterações na solução atual na tentativa de melhorá-la. Se uma alteração resultar em uma solução melhor, ela se tornará a nova solução e o processo continuará. Se não houver como melhorar a solução atual, o algoritmo termina e retorna a solução final.

***teste***

Vamos tentar com os dados do exemplo do arquivo . . _

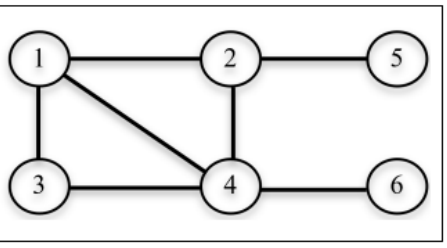
- solução mais ótima deve ser:
  - S2 = {1,2,3,4} -> Número de arestas = 5
- Podemos ver que o gráfico de exemplo é o mesmo que estamos usando usando a seguinte função:

```
def showGraph(listaEnlaces):
        # Crea un grafo vacío
    G = nx.Graph()
    # Añade los enlaces a la lista de enlaces del grafo
    G.add_edges_from(listaEnlaces)
    # Dibuja el grafo
    nx.draw(G, with_labels=True)
    # Muestra el gráfico
    plt.show()
```




**Example:**
Consider a graph with 6 vertices and 7 arcs, represented below (***test.txt*** in Moodle). The image on the right illustrates the graph. The goal is to find a solution of the problem for k=4.

```
k 4
p edge 6 7
e 1 2
e 1 3
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```




Consider the three solutions (there will be others):
S1 = {1, 2, 5, 6} - number of edges = 2
S2 = {1, 2, 3, 4} - number of edges = 5
S3 = {2, 3, 5, 6} - number of edges = 1

E o resultado é:

```
**********          ALGORITMO Hibrido ESTOCASTICO - EVOLUTIVO      **********
[4, 3, 2, 1] -> Number of Edges =  5
```

# ALGORITMO EVOLUTIVO HÍBRIDO DE GREEDY

**Explicação de como o algoritmo resolve e funciona:**

Este algoritmo híbrido combina elementos de um algoritmo evolutivo e um algoritmo de busca local guloso .

O algoritmo evolutivo é usado para encontrar uma boa solução para um problema usando técnicas de seleção, recombinação e mutação. Neste caso particular, são utilizados dois métodos de selecção (selecção por torneio e selecção por roleta) para escolher os indivíduos a reproduzir, e são aplicados dois operadores de recombinação (recombinação de um ponto e recombinação uniforme) para produzir dois grupos de indivíduos. Dois operadores de mutação (mutação de troca e mutação de inserção) são então aplicados a cada um dos dois conjuntos de filhos.

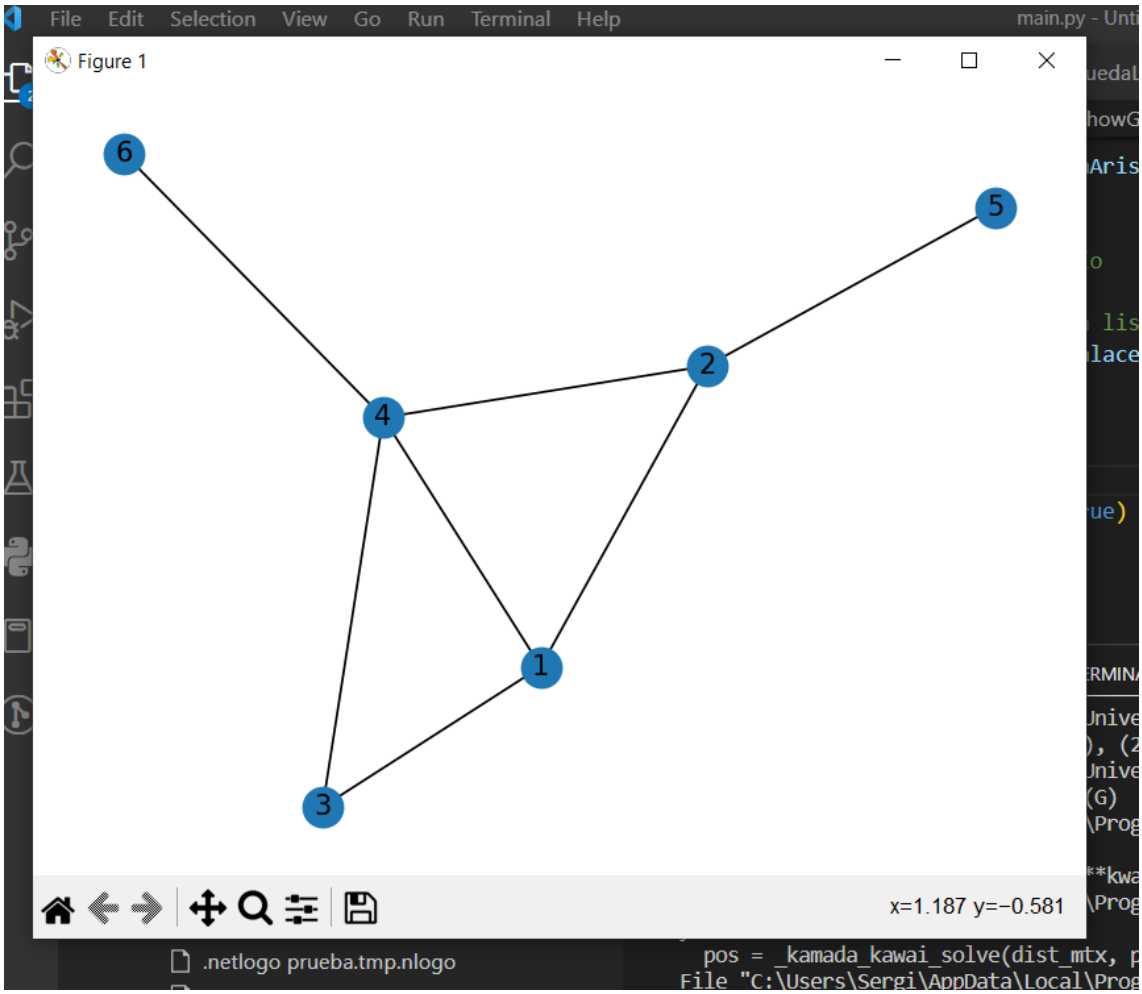
de busca local guloso usa uma heurística para encontrar uma solução ótima em uma vizinhança de soluções.

***teste***

Vamos tentar com os dados do exemplo do arquivo . . _

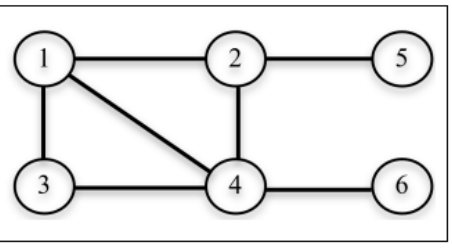
- solução mais ótima deve ser:
  - S2 = {1,2,3,4} -> Número de arestas = 5
- Podemos ver que o gráfico de exemplo é o mesmo que estamos usando usando a seguinte função:

```
def showGraph(listaEnlaces):
        # Crea un grafo vacío
    G = nx.Graph()
    # Añade los enlaces a la lista de enlaces del grafo
    G.add_edges_from(listaEnlaces)
    # Dibuja el grafo
    nx.draw(G, with_labels=True)
    # Muestra el gráfico
    plt.show()
```




**Example:**
Consider a graph with 6 vertices and 7 arcs, represented below (***test.txt*** in Moodle). The image on the right illustrates the graph. The goal is to find a solution of the problem for k=4.

```
k 4
p edge 6 7
e 1 2
e 1 3
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```




Consider the three solutions (there will be others):
S1 = {1, 2, 5, 6} - number of edges = 2
S2 = {1, 2, 3, 4} - number of edges = 5
S3 = {2, 3, 5, 6} - number of edges = 1

```python
def evolutiveAlgorithm():
    # Parámetros del algoritmo
    vertices = 6
    nAristas = 7
    kValue = 4
    listaEnlaces = [(1, 2), (1, 3), (1, 4), (2, 4), (2, 5), (3, 4), (4, 6)]
    poblacion_inicial = 10
    max_generaciones = 100
    tasa_mutacion = 0.1
```

E o resultado é:

```
[4, 2, 3, 1] Number of Edges =  5
PS C:\Users\Sergi\Desktop\code\Universidad\Erasmus\IA>
```

- (Este resultado pode variar dependendo da iteração do algoritmo):

```
[5, 4, 2, 1] Number of Edges =  4
```

# ALGORITMO ESTOCÁSTICO EVOLUTIVO HÍBRIDO

***Explicação de como resolve O algoritmo funciona:***

Este algoritmo híbrido combina o algoritmo estocástico de escalada e o algoritmo evolutivo para encontrar uma solução ótima para obter o subconjunto com o maior número de arestas. O algoritmo estocástico de escalada é usado para gerar soluções iniciais, enquanto o algoritmo evolutivo é usado para melhorar essas soluções iniciais.

Em cada iteração do algoritmo evolutivo, os indivíduos a serem criados são primeiro selecionados usando um método de seleção, como seleção por torneio ou seleção por roleta. Os operadores de reprodução e mutação são então aplicados aos indivíduos selecionados para criar novos indivíduos (chamados descendentes). Em seguida, avalia-se a aptidão da prole e seleciona-se o indivíduo mais apto como a nova solução atual.

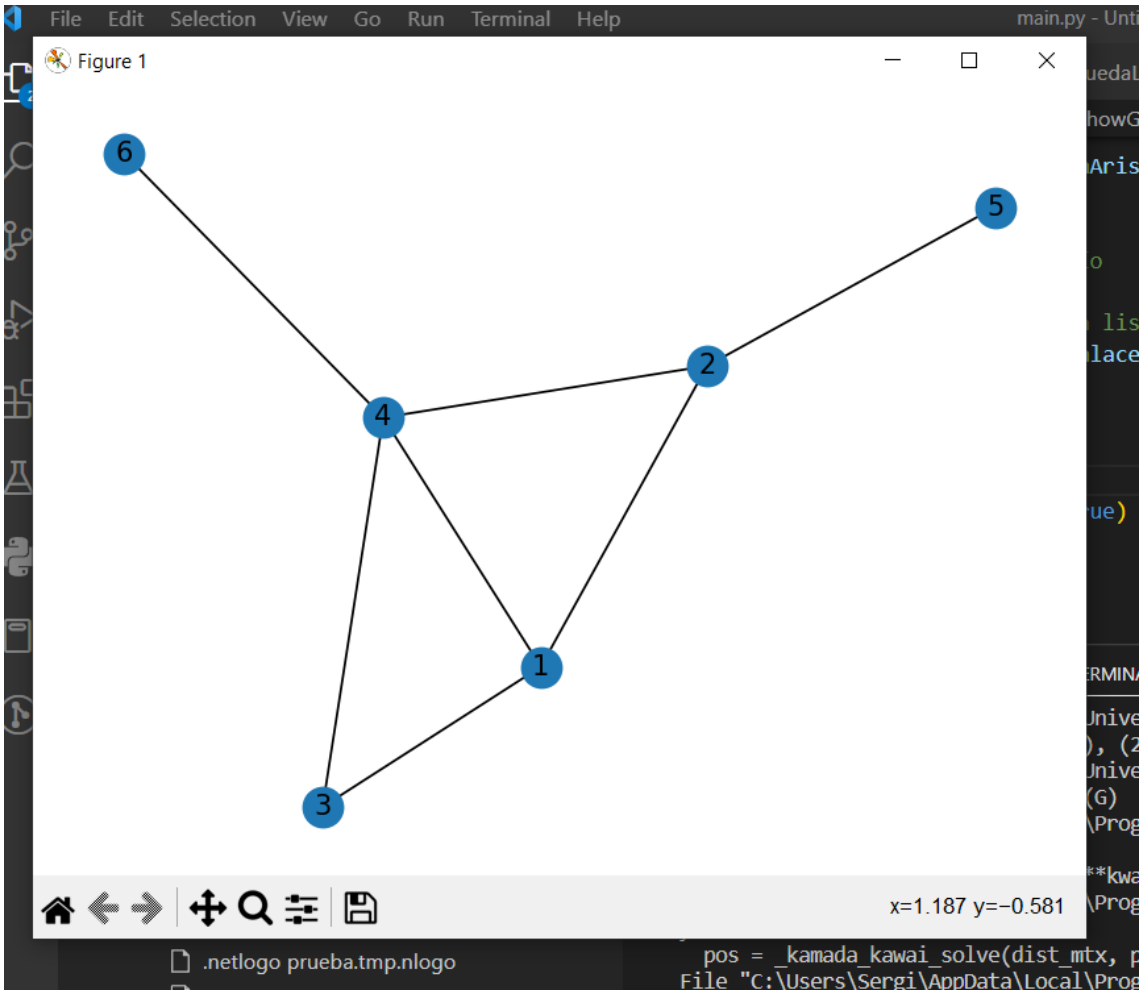
Uma das vantagens de usar um algoritmo híbrido é que você pode aproveitar os pontos fortes de cada algoritmo individual e minimizar seus pontos fracos. Nesse caso, o algoritmo estocástico de subida de encosta é rápido e pode encontrar soluções quase ótimas, enquanto o algoritmo evolutivo é capaz de explorar um espaço de solução maior e encontrar soluções ótimas. Combinando os dois algoritmos, podemos obter uma solução final que tenha uma boa combinação de velocidade e otimização.

***teste***

Vamos tentar com os dados do exemplo do arquivo . . _

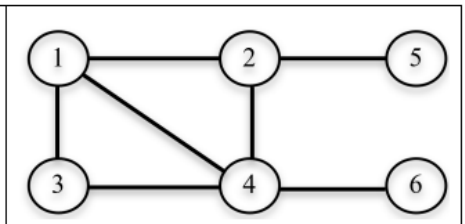
- solução mais ótima deve ser:
    - S2 = {1,2,3,4} -> Número de arestas = 5
- Podemos ver que o gráfico de exemplo é o mesmo que estamos usando usando a seguinte função:

```python
def showGraph(listaEnlaces):
        # Crea un grafo vacío
    G = nx.Graph()
    # Añade los enlaces a la lista de enlaces del grafo
    G.add_edges_from(listaEnlaces)
    # Dibuja el grafo
    nx.draw(G, with_labels=True)
    # Muestra el gráfico
    plt.show()
```



**Example:**
Consider a graph with 6 vertices and 7 arcs, represented below (***test.txt*** in Moodle). The image on the right illustrates the graph. The goal is to find a solution of the problem for k=4.

```
k 4
p edge 6 7
e 1 2
e 1 3
e 1 4
e 2 4
e 2 5
e 3 4
e 4 6
```




Consider the three solutions (there will be others):
S1 = {1, 2, 5, 6} - number of edges = 2
S2 = {1, 2, 3, 4} - number of edges = 5
S3 = {2, 3, 5, 6} - number of edges = 1

E o resultado é:

```
**********          ALGORITMO Hibrido ESTOCASTICO - EVOLUTIVO      **********
[4, 3, 2, 1] -> Number of Edges =  5
```

- (Este resultado pode variar dependendo da iteração do algoritmo):

# ALGORITMO HÍBRIDO ESTOCÁSTICO EVOLUTIVO 2

O algoritmo evolutivo estocástico híbrido2 começa gerando uma solução inicial usando o algoritmo estocástico de escalada e, em seguida, usa um processo evolutivo para melhorar essa solução iterativamente ao longo de várias gerações.

# ESTUDAR

Melhores soluções de estudo:

| File | Vertices | Edges | K | Best solution |
|---|---|---|---|---|
| teste.txt | 6 | 7 | 4 | 5 |
| file1.txt | 28 | 210 | 8 | 21 |
| file2.txt | 64 | 704 | 7 | 16 |
| file3.txt | 70 | 1855 | 16 | 112 |
| file4.txt | 200 | 1534 | 14 | 79 |
| file5.txt | 500 | 4459 | 15 | 98 |

## Algoritmo guloso :

Podemos realizar um estudo com o número de iterações máximas e o número de interações máximas SEM melhoria:

- Podemos ver que: Este tipo de algoritmo não é muito eficaz, ao contrário, é o mais rápido de todos devido ao seu curto tempo de execução. Também podemos observar que o primeiro resultado com o número máximo de interações sem melhoria e com melhoria é sempre o mais eficiente, pois não há variação entre os resultados. Podemos concluir que este algoritmo guloso não é EFICIENTE em grafos grandes, ao invés disso é em grafos pequenos como o arquivo teste.txt se for.
- O melhor resultado é com:
  - Arquivo1.txt
    - Iterações : 100
    - MaxIteration Sem melhoria: 100
    - Resultado: [8, 7, 6, 5, 4, 3, 2, 1] -> n: 9
  - arquivo2.txt
    - Iterações : 100
    - MaxIteration Sem melhoria: 100
    - Resultado: [] -> n: 0
  - Arquivo3.txt

        - Iterações : 100
        - MaxIteration Sem melhoria: 100
        - Resultado: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1] -> n: 57
    - Arquivo4.txt
        - Iterações : 100
        - MaxIteration Sem melhoria: 100
        - Resultado: [39, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] -> n: 14
    - Arquivo5.txt
        - Iterações : 100
        - MaxIteration Sem melhoria: 100
        - Resultado: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] -> n: 14

### Algoritmo Stochastic Hill Climb:

Podemos realizar um estudo com o número máximo de iterações, pois o número máximo de iterações será o número de bairros que o algoritmo visita:

- Nós podemos ver isso:
    - Os resultados são muito bons, a diferença com a melhor solução é mínima, portanto podemos concluir que o algoritmo é muito eficiente e cumpre o que promete.
    - Ele tende a encontrar a solução mais eficiente nas primeiras iterações.
- O melhor resultado é com:
    - Arquivo1.txt
        - Iterações : 3200
        - Resultado: [26, 24, 21, 20, 12, 6, 4, 1] -> n = 20
    - arquivo2.txt
        - Iterações : 100
        - Resultado: [63, 50, 42, 39, 29, 20, 9] -> n = 15

- Arquivo3.txt
  - Iterações : 20000
  - Resultado: (64, 60, 53, 51, 46, 43, 38, 35, 29, 28, 25, 20, 16, 11, 9, 3) -> n = 112
- Arquivo4.txt
  - Iterações : 200
  - Resultado: (195, 159, 158, 122, 121, 86, 85, 84, 49, 48, 47, 12, 11, 10) -> n = 79
- Arquivo5.txt
  - Iterações : 100
  - Resultado: (331, 330, 252, 251, 250, 172, 171, 170, 91, 90, 89, 13, 12, 11, 10) -> n = 93

### Algoritmo Evolutivo :

Podemos realizar um estudo com o número máximo de gerações, a população inicial e a taxa de mutação.

- Nós podemos ver isso:
  - Nos três primeiros arquivos o Algoritmo é EFICIENTE, pois a distância da solução obtida com a distância ideal é mínima.
  - Nos arquivos 4 e 5 o algoritmo perde eficiência e os resultados não são próximos aos desejados.
  - Ele tende a encontrar a melhor solução com um limite de geração maior .
  - Nenhuma mudança significativa na população e na taxa de mutação pode ser observada.

Tende a encontrar a melhor solução com maior geração máxima

- O melhor resultado é com:
  - Arquivo1.txt
    - Gerações máximas: 100
    - População inicial: 10
    - Taxa de mutação: 0,3
    - Resultado: [27, 23, 20, 19, 13, 8, 4, 2] -> Número de arestas = 20
  - arquivo2.txt
    - Gerações máximas: 500
    - População inicial: 16
    - Taxa de mutação: 0,1
    - Resultado: [56, 42, 39, 29, 20, 14, 3] -> Número de arestas = 15
  - Arquivo3.txt
    - Gerações máximas: 500
    - População inicial: 16
    - Taxa de mutação: 0,1
    - Resultado: [67, 65, 61, 56, 49, 44, 42, 35, 34, 21, 20, 17, 10, 9, 4, 1] -> Número de arestas = 104
  - Arquivo4.txt

        - Gerações máximas: 500
        - População inicial: 16
        - Taxa de mutação: 0,3
        - Resultado: (197, 196, 195, 195, 194, 123, 121, 48, 13, 12, 11, 10, 9, 8) -> Número de arestas = 39
    - Arquivo5.txt
        - Gerações máximas: 500
        - População inicial: 20
        - Taxa de mutação: 0,1
        - Resultado: (483, 482, 411, 402, 401, 325, 322, 241, 82, 10, 9, 4, 3, 2, 1) -> Número de arestas = 44

### Algoritmo Híbrido 1:

Podemos realizar um estudo com o número máximo de iterações, pois o número máximo de iterações será o número de vizinhanças que o algoritmo visitar, a população inicial e o número máximo de gerações.

- Nós podemos ver isso:
  - Os resultados do algoritmo híbrido são quase perfeitos, obtendo o resultado ideal em 3/5 arquivos.
  - Podemos observar que na maioria das vezes não é necessário aumentar a população inicial e as gerações para obter uma solução melhor.
  - Podemos observar que devido à eficiência deste algoritmo não é necessário aumentar as interações para o estudo destes arquivos.
- O melhor resultado é com:
  - Arquivo1.txt
    - Gerações máximas: 100
    - População inicial: 10
    - Iterações: 100
    - Resultado: [27, 22, 19, 17, 14, 9, 7, 3] -> Número de arestas = 20
  - arquivo2.txt
    - Gerações máximas: 100
    - População inicial: 10
    - Iterações: 100
    - Resultado: [52, 37, 35, 31, 25, 16, 6] -> Número de arestas = 15
  - Arquivo3.txt
    - Gerações máximas: 500
    - População inicial: 10
    - Iterações: 100
    - Resultado: [70, 59, 58, 50, 47, 46, 45, 40, 31, 26, 24, 21, 17, 13, 12, 1] -> Número de arestas = 112
  - Arquivo4.txt

        - Gerações máximas: 1000
        - População inicial: 10
        - Iterações: 100
        - Resultado : (196, 195, 160, 159, 158, 122, 121, 85, 84, 49, 48, 47, 11, 10) -> Número de arestas = 79
    - Arquivo5.txt
        - Gerações máximas: 100
        - População inicial: 20
        - Iterações: 500
        - Resultado: (496, 495, 417, 416, 415, 336, 335, 256, 255, 176, 175, 96, 95, 16, 15) -> Número de arestas = 98

### Algoritmo Híbrido 1:

Este algoritmo usa os mesmos parâmetros do algoritmo anterior.

- Nós podemos ver isso:
  - O resultado ideal é alcançado em 2/5 limas e os demais resultados aproximam-se da solução ideal, portanto podemos dizer que o resultado é muito bom.
  - Observa-se que não é necessário aumentar na maioria dos casos a população inicial na maioria dos casos para se ter uma boa solução.
  - Nenhum padrão é visto entre as outras variáveis para desenhar uma observação.
- O melhor resultado é com:
  - Arquivo1.txt
    - Gerações máximas: 100
    - População inicial: 10
    - Iterações: 100
    - Resultado: [25, 24, 17, 16, 13, 12, 10, 7] -> Número de arestas = 20
  - arquivo2.txt
    - Gerações máximas: 500
    - População inicial: 10
    - Iterações: 100
    - Resultado: [60, 45, 33, 27, 22, 8, 3] -> Número de arestas = 15
  - Arquivo3.txt
    - Gerações máximas: 1000
    - População inicial: 10
    - Iterações: 1000
    - Resultado: [60, 57, 52, 51, 46, 45, 43, 42, 34, 29, 28, 25, 20, 14, 11, 1] -> Número de arestas = 110
  - Arquivo4.txt
    - Gerações máximas: 1000
    - População inicial: 10
    - Iterações: 100

        - Resultado: (195, 194, 158, 157, 121, 120, 84, 83, 82, 47, 46, 45, 10, 9) -> Número de arestas = 79
    - Arquivo5.txt
        - Gerações máximas: 1000
        - População inicial: 20
        - Iterações: 500
        - Resultado: (494, 493, 414, 413, 334, 333, 254, 253, 175, 174, 173, 94, 93, 14, 13) -> Número de arestas = 98

**Diferença entre os dois algoritmos híbridos:**

- Não há diferenças notáveis entre os dois algoritmos, pois o resultado de ambos é bom.
- No primeiro algoritmo o resultado ideal foi alcançado em 3/5 arquivos e no segundo algoritmo foi alcançado em 2/5
- Pode-se dizer que pelos resultados obtidos o algoritmo 1 é mais eficiente que o algoritmo 2.

*Nota* : Não foram realizados estudos sobre o algoritmo híbrido guloso e evolutivo uma vez que os resultados do algoritmo local não foram próximos dos desejados.