IA

ALGORITHM

Sergio González Martínez

Lucia Arnaiz

Content

OBJETIVO

El objetivo de este texto es crear un algoritmo que, dado un grafo (un conjunto de puntos conectados entre sí por enlaces) y un número entero k, encuentre un subconjunto de k puntos del grafo que estén conectados entre sí por el mayor número posible de enlaces. Es un problema de maximización, lo que significa que queremos encontrar la solución que tenga el mayor valor posible.

ALGORITMO BUSQUEDA LOCAL(GREEDY)

Explicación de como resuelve y funciona el algoritmo:

1. Ordena la lista de tuplas en orden descendente por el número de vecinos.
2. Selecciona los k vértices con más vecinos. Si hay empate, selecciona los vértices con más vecinos en orden de aparición en la lista.
3. Cuenta el número de aristas entre los vértices seleccionados.

TIPO DE ALGORITMO

Este algoritmo es una solución heurística para el problema de encontrar un subconjunto de vértices en un grafo que maximice el número de aristas entre ellos.

Este algoritmo utiliza un enfoque greedy, es decir, va tomando decisiones locales óptimas en cada paso con la esperanza de llegar a una solución global óptima.

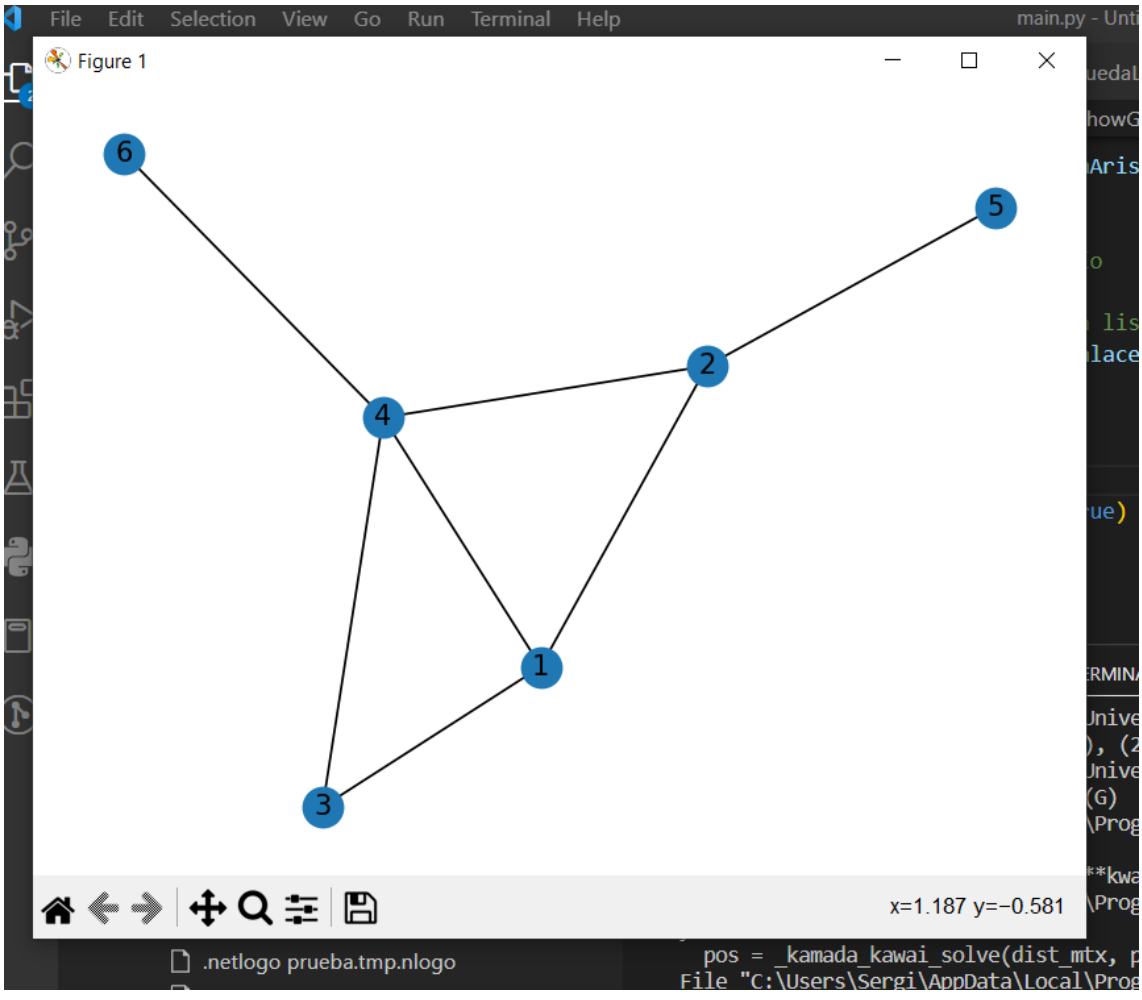
La decisión local óptima **es seleccionar en cada paso el vértice con más vecinos**, ya que tiene más posibilidades de tener más aristas con otros vértices.

Pruebas

Probaremos con los datos del ejemplo del .pdf.

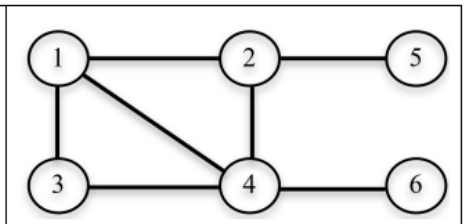
- La solución mas optima debería ser:
  - S2 = {1,2,3,4} -> Number of Edges = 5
- Podemos ver que el grafo del ejemplo es el mismo que el que estamos utilizando mediante la siguiente función:

```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

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

Le añadimos los valores del ejemplo:

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

Y el resultado es:

```
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 puede variar dependiendo la iteración del algoritmo):

ALGORITMO EVOLUTIVO

Explicación de como resuelve y Funciona el algoritmo:

1. Crea una población inicial de individuos (soluciones candidatas) aleatorios. Cada individuo sería una lista de vértices, de tamaño k, que representa un subconjunto de vértices del grafo.
2. Evalúa cada individuo de la población utilizando una función de aptitud que cuente el número de aristas entre los vértices del subconjunto.
3. Selecciona dos métodos de selección diferentes para seleccionar a los individuos más aptos para reproducirse. Por ejemplo, podrías utilizar el método de selección por torneo y el método de selección por ruleta.
4. Se aplica dos operadores de recombinación diferentes a pares de individuos seleccionados para crear nuevos individuos.

5. Aplica dos operadores de mutación diferentes a algunos de los nuevos individuos para introducir variabilidad en la población.
6. Si alguno de los nuevos individuos es inválido (por ejemplo, contiene vértices repetidos o más de k vértices), Se repara o penaliza su aptitud de alguna manera.
7. Reemplaza a algunos de los individuos de la población original con los nuevos individuos.
8. Repite los pasos 3 a 7 hasta que se alcance el criterio de parada (por ejemplo, un número máximo de generaciones o una solución óptima).

TIPO DE ALGORITMO

Este código implementa un algoritmo evolutivo, que es un tipo de algoritmo de optimización inspirado en el proceso de evolución de las especies en la naturaleza.

Los algoritmos evolutivos consisten en una población de individuos que representan posibles soluciones a un problema. A través de un proceso iterativo, se seleccionan individuos de la población para reproducirse y crear una nueva generación de individuos. Estos individuos suelen tener pequeñas variaciones respecto a sus progenitores, y estas variaciones se consiguen aplicando operadores de recombinación y mutación. Los individuos se evalúan mediante una función de aptitud, que mide su calidad como solución al problema es decir, el número de enlaces de un subconjunto. A medida que se van generando nuevas generaciones, se espera que la aptitud media de la población aumente y que se encuentren individuos cada vez mejores.

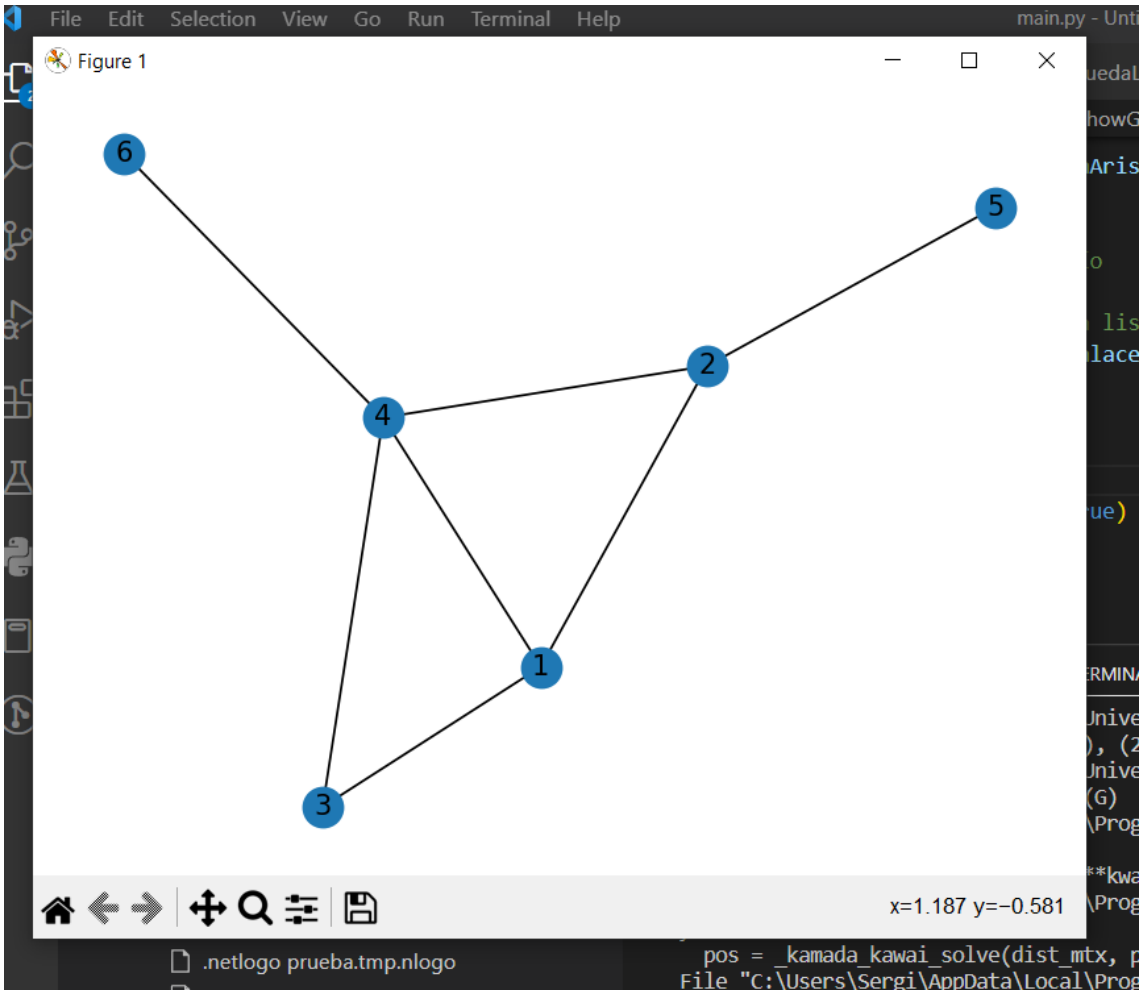
En el caso de este código en particular, se está tratando de encontrar un subconjunto de vértices de un grafo que maximice el número de aristas dentro de ese subconjunto, siempre y cuando al menos dos vecinos diferentes sean explorados. Para ello, cada individuo de la población representa un subconjunto de vértices, y la función de aptitud cuenta el número de aristas que hay dentro del subconjunto. Luego, se aplican dos operadores de recombinación y dos operadores de mutación diferentes para crear una nueva generación a partir de los individuos seleccionados. Finalmente, se obtiene el mejor individuo de la población.

Pruebas

Probaremos con los datos del ejemplo del .pdf.

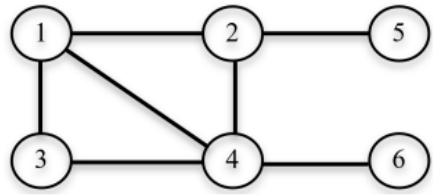
- La solución mas optima debería ser:
  - S2 = {1,2,3,4} -> Number of Edges = 5
- Podemos ver que el grafo del ejemplo es el mismo que el que estamos utilizando mediante la siguiente función:

```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

```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
```

Y el resultado es:

```
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 puede variar dependiendo la iteración del algoritmo):

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

ALGORITMO TREPA COLINAS ESTOCASTICO

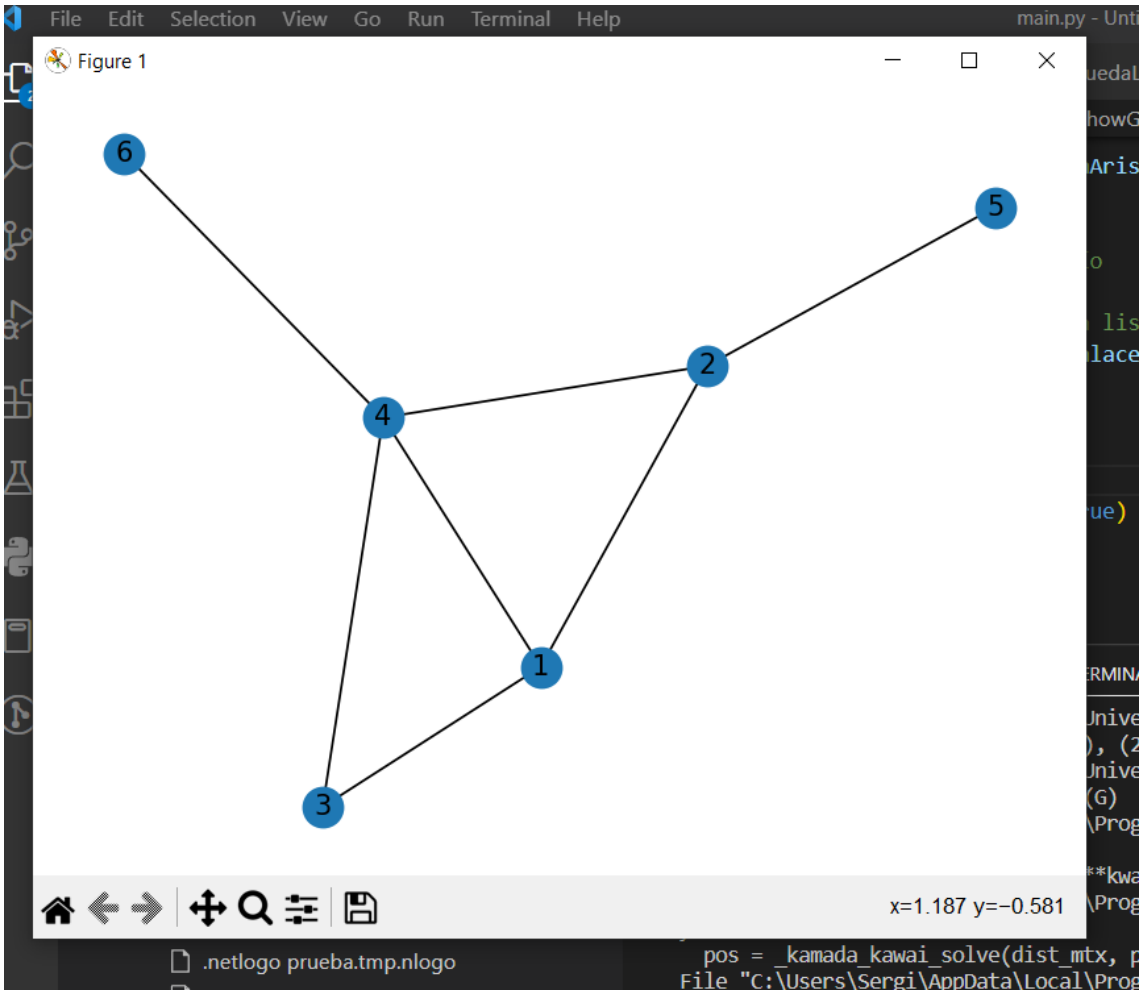
Este algoritmo es una implementación del algoritmo de escalada de colina estocástico. Es un tipo de algoritmo de búsqueda local que se utiliza para encontrar una solución óptima (o muy cercana a la óptima) para un problema dado. El algoritmo comienza con una solución inicial, y luego realiza iterativamente cambios pequeños a la solución actual en un intento de mejorarla. Si un cambio resulta en una solución mejor, se convierte en la nueva solución y el proceso continúa. Si no hay forma de mejorar la solución actual, el algoritmo finaliza y devuelve la solución final.

Pruebas

Probaremos con los datos del ejemplo del .pdf.

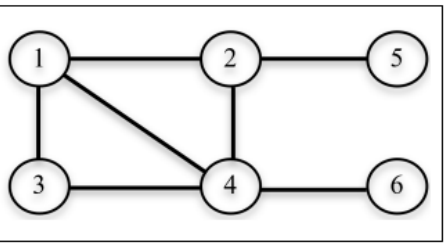
- La solución mas optima debería ser:
  - S2 = {1,2,3,4} -> Number of Edges = 5
- Podemos ver que el grafo del ejemplo es el mismo que el que estamos utilizando mediante la siguiente función:

```
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

Y el resultado es:

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

ALGORITMO HIBRIDO GREEDY-EVOLUTIVO

Explicación de como resuelve y Funciona el algoritmo:

Este algoritmo híbrido combina elementos de un algoritmo evolutivo y un algoritmo de búsqueda local greedy.

El algoritmo evolutivo se utiliza para encontrar una buena solución a un problema mediante el uso de técnicas de selección, recombinación y mutación. En este caso en particular, se utilizan dos métodos de selección (selección por torneo y selección por ruleta) para elegir a los individuos que se reproducirán y se aplican dos operadores de recombinación (recombinación en un punto y recombinación uniforme) para producir dos grupos de hijos. Luego, se aplican dos operadores de mutación (mutación por intercambio y mutación por inserción) a cada uno de los dos grupos de hijos.

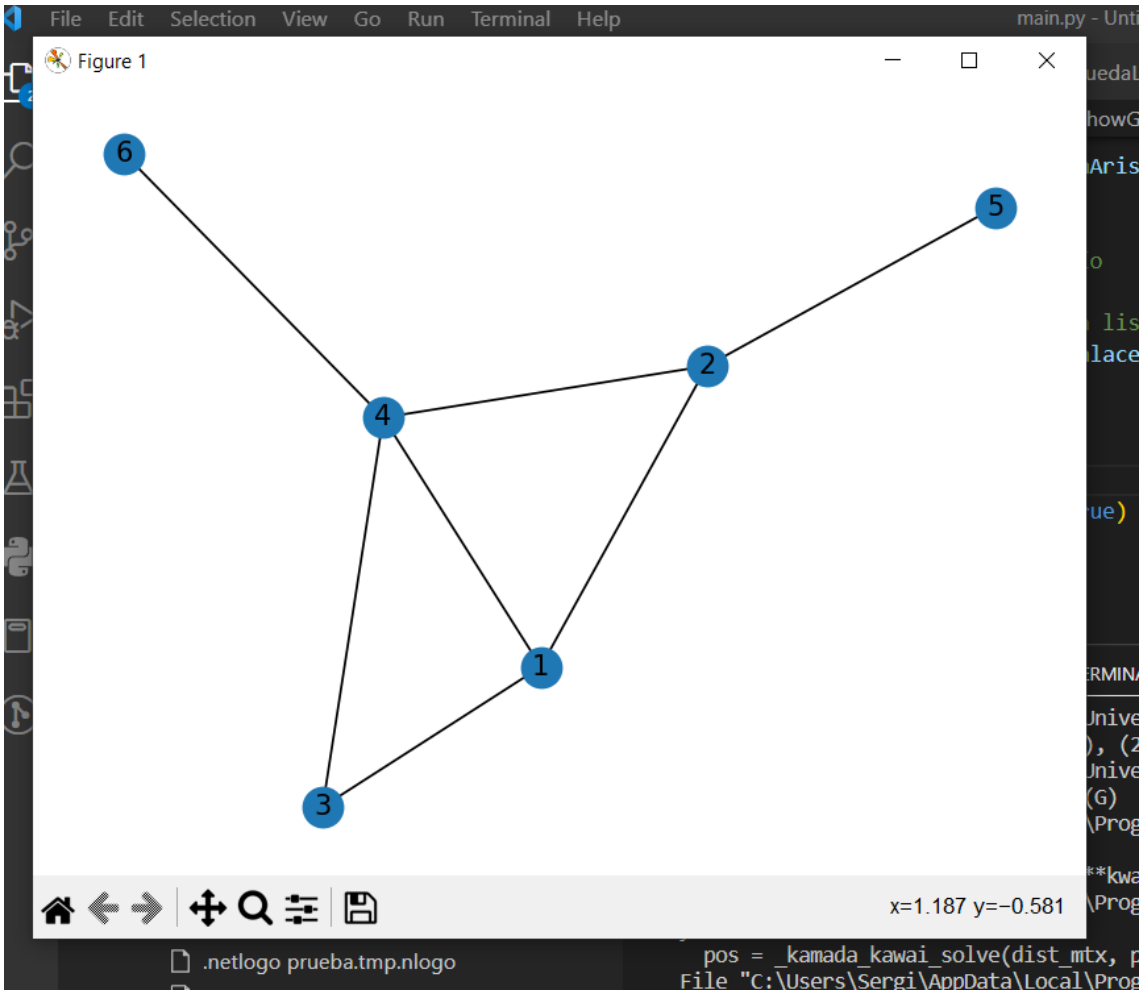
Por otro lado, el algoritmo de búsqueda local greedy utiliza una heurística para encontrar una solución óptima en un vecindario de soluciones.

Pruebas

Probaremos con los datos del ejemplo del .pdf.

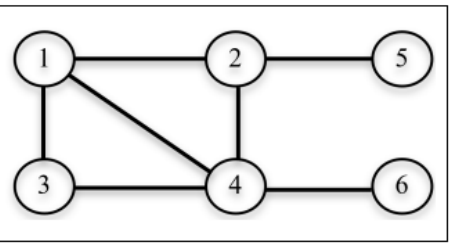
- La solución mas optima debería ser:
  - S2 = {1,2,3,4} -> Number of Edges = 5
- Podemos ver que el grafo del ejemplo es el mismo que el que estamos utilizando mediante la siguiente función:

```
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
```

Y el resultado es:

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

- (Este resultado puede variar dependiendo la iteración del algoritmo):

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

ALGORITMO HIBRIDO ESTOCASTICO EVOLUTIVO

Explicación de como resuelve Funciona el algoritmo:

Este algoritmo híbrido combina el algoritmo de trepa colinas estocástico y el algoritmo evolutivo para encontrar una solución óptima para obtener el subconjunto con mayor número de aristas. El algoritmo de trepa colinas estocástico se utiliza para generar soluciones iniciales, mientras que el algoritmo evolutivo se utiliza para mejorar esas soluciones iniciales.

En cada iteración del algoritmo evolutivo, primero se seleccionan los individuos que se reproducirán utilizando un método de selección, como la selección por torneo o la selección por ruleta. Luego, se aplican los operadores de reproducción y mutación a los individuos seleccionados para crear nuevos individuos (llamados descendientes). A continuación, se evalúa la aptitud de los descendientes y se selecciona el individuo más apto como la nueva solución actual.

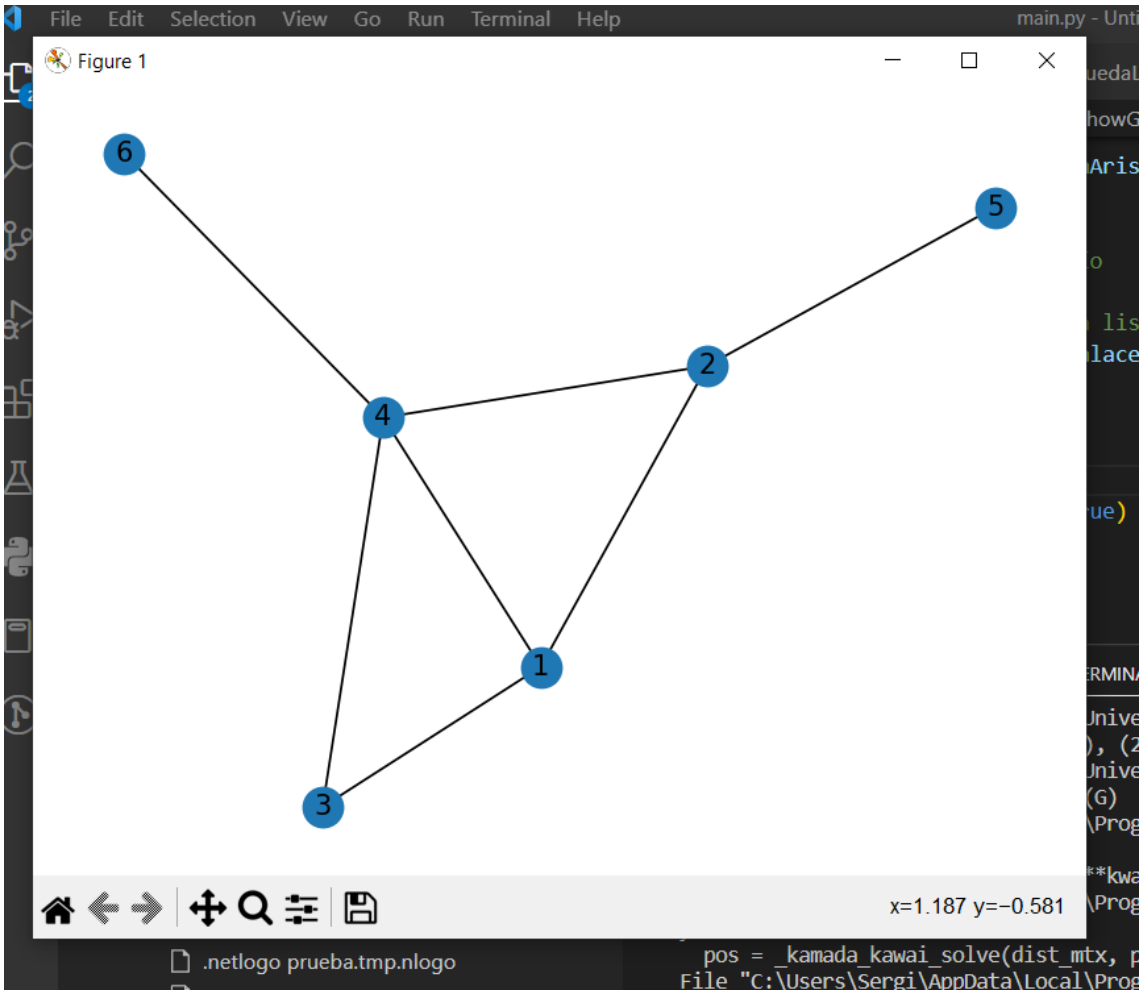
Una de las ventajas de utilizar un algoritmo híbrido es que puede aprovechar las fortalezas de cada algoritmo individual y minimizar sus debilidades. En este caso, el algoritmo de trepa colinas estocástico es rápido y puede encontrar soluciones cercanas al óptimo, mientras que el algoritmo evolutivo es capaz de explorar un espacio de soluciones más amplio y encontrar soluciones óptimas. Al combinar ambos algoritmos, podemos obtener una solución final que tenga una buena combinación de velocidad y optimización.

Pruebas

Probaremos con los datos del ejemplo del .pdf.

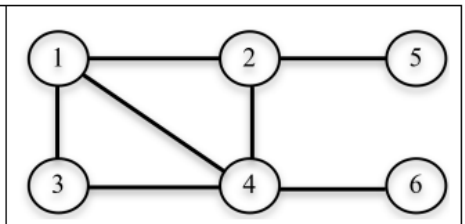
- La solución mas optima debería ser:
  - S2 = {1,2,3,4} -> Number of Edges = 5
- Podemos ver que el grafo del ejemplo es el mismo que el que estamos utilizando mediante la siguiente función:

```
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

Y el resultado es:

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

- (Este resultado puede variar dependiendo la iteración del algoritmo):

ALGORITMO HIBRIDO ESTOCASTICO EVOLUTIVO 2

El algoritmo híbrido estocástico evolutivo2 comienza generando una solución inicial utilizando el algoritmo de trepa colinas estocástico y luego utiliza un proceso evolutivo para mejorar esa solución iterativamente durante varias generaciones.

ESTUDIO

Mejores soluciones para el estudio:

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 greedy:

Podemos realizar un estudio con el numero de iteraciones máximas y el número de interacciones máximas SIN mejora:

- Podemos ver que: Este tipo de algoritmo no es muy efectivo, en cambio si que es el mas rápido de todos debido a su corto tiempo de ejecución. Podemos observar también que siempre el primer resultado con el numero de interaciones máximas sin mejora y con mejora siempre es el mas eficiente ya que no hay variación entre los resultados. Podemos concluir que este algoritmo greedy no es EFICIENTE en grafos grandes, en cambio es grafos pequeños como el archivo teste.txt si lo es.
- El mejor resultado es con:
  - File1.txt
    - Iterations: 100
    - MaxIteration Sin mejora: 100
    - Resultado: [8, 7, 6, 5, 4, 3, 2, 1] -> n: 9
  - File2.txt
    - Iterations: 100
    - MaxIteration Sin mejora: 100
    - Resultado: [] -> n: 0
  - File3.txt

        - Iterations: 100
        - MaxIteration Sin mejora: 100
        - Resultado: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2, 1] -> n: 57
    - File4.txt
        - Iterations: 100
        - MaxIteration Sin mejora: 100
        - Resultado: [39, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] -> n: 14
    - File5.txt
        - Iterations: 100
        - MaxIteration Sin mejora: 100
        - Resultado: [16, 15, 14, 13, 12, 11, 10, 9, 8, 7, 6, 5, 4, 3, 2] -> n: 14

Algoritmo Trepa Colinas estocástico:

Podemos realizar un estudio con el número de iteraciones máximas, ya que el número de iteraciones máximas será la cantidad de vecindarios que el algoritmo visite:

- Podemos ver que:
    - Los resultados son muy buenos, la diferencia con la mejor solución es mínima, por tanto podemos concluir que el algoritmo es muy eficiente y da lo que promete.
    - Tiende a encontrar la solución mas eficiente en las primeras iteraciones.
- El mejor resultado es con:
    - File1.txt
        - Iterations: 3200
        - Resultado: [26, 24, 21, 20, 12, 6, 4, 1] -> n = 20
    - File2.txt
        - Iterations: 100
        - Resultado: [63, 50, 42, 39, 29, 20, 9] -> n = 15

- File3.txt
  - Iterations: 20000
  - Resultado: (64, 60, 53, 51, 46, 43, 38, 35, 29, 28, 25, 20, 16, 11, 9, 3) -> n = 112
- File4.txt
  - Iterations: 200
  - Resultado: (195, 159, 158, 122, 121, 86, 85, 84, 49, 48, 47, 12, 11, 10) -> n = 79
- File5.txt
  - Iterations: 100
  - Resultado: (331, 330, 252, 251, 250, 172, 171, 170, 91, 90, 89, 13, 12, 11, 10) -> n = 93

Algoritmo Evolutivo:

Podemos realizar un estudio con el número de generaciones máximas, la población inicial y la tasa de mutación.

- Podemos ver que:
  - En los primeros tres ficheros el Algoritmo es EFICIENTE, ya que la distancia de la solución obtenida con la distancia ideal es mínima.
  - En los ficheros 4 y 5 el algoritmo pierde eficiencia y los resultados no se acercan a los deseados.
  - Tiende a encontrar la mejor solución con un limite de generaciones mayor.
  - No se pueden observar cambios significativos en la población y en la tasa de mutación.

Tiende a encontrar la mejor solución un un máximo de generación mayor

- El mejor resultado es con:
  - File1.txt
    - Generaciones máximas: 100
    - Población inicial: 10
    - Tasa mutación: 0.3
    - Resultado: [27, 23, 20, 19, 13, 8, 4, 2] -> Number of Edges = 20
  - File2.txt
    - Generaciones máximas: 500
    - Población inicial: 16
    - Tasa mutación: 0.1
    - Resultado: [56, 42, 39, 29, 20, 14, 3] -> Number of Edges = 15
  - File3.txt
    - Generaciones máximas: 500
    - Población inicial: 16
    - Tasa mutación: 0.1
    - Resultado: [67, 65, 61, 56, 49, 44, 42, 35, 34, 21, 20, 17, 10, 9, 4, 1] -> Number of Edges = 104

- File4.txt
  - Generaciones máximas: 500
  - Población inicial: 16
  - Tasa mutación: 0.3
  - Resultado: (197, 196, 195, 195, 194, 123, 121, 48, 13, 12, 11, 10, 9, 8) -> Number of Edges = 39
- File5.txt
  - Generaciones máximas: 500
  - Población inicial: 20
  - Tasa mutación: 0.1
  - Resultado: (483, 482, 411, 402, 401, 325, 322, 241, 82, 10, 9, 4, 3, 2, 1) -> Number of Edges = 44

Algoritmo Hibrido 1:

Podemos realizar un estudio con el número de iteraciones máximas, ya que el número de iteraciones máximas será la cantidad de vecindarios que el algoritmo visite, la población inicial y la el número de generaciones máximas.

- Podemos ver que:
  - Los resultados de el algoritmo hibrido son casi perfectos, obteniendo el resultado ideal en 3/5 ficheros.
  - Podemos observar que la mayoría de las veces no hace falta aumentar las población inicial y las generaciones para obtener una mejor solución.
  - Podemos observar que debido a la eficiencia de este algoritmo no hace falta aumentar las interaciones para el estudio de estos ficheros.
- El mejor resultado es con:
  - File1.txt
    - Generaciones máximas: 100
    - Población inicial: 10
    - Iteraciones: 100
    - Resultado: [27, 22, 19, 17, 14, 9, 7, 3] -> Number of Edges = 20
  - File2.txt
    - Generaciones máximas: 100
    - Población inicial: 10
    - Iteraciones: 100
    - Resultado: [52, 37, 35, 31, 25, 16, 6] -> Number of Edges = 15
  - File3.txt
    - Generaciones máximas: 500
    - Población inicial:10
    - Iteraciones:100
    - Resultado: [70, 59, 58, 50, 47, 46, 45, 40, 31, 26, 24, 21, 17, 13, 12, 1] -> Number of Edges = 112

- File4.txt
  - Generaciones máximas:1000
  - Población inicial:10
  - Iteraciones:100
  - Resultado: (196, 195, 160, 159, 158, 122, 121, 85, 84, 49, 48, 47, 11, 10) -> Number of Edges = 79
- File5.txt
  - Generaciones máximas:100
  - Población inicial:20
  - Iteraciones:500
  - Resultado: (496, 495, 417, 416, 415, 336, 335, 256, 255, 176, 175, 96, 95, 16, 15) -> Number of Edges = 98

Algoritmo Hibrido 1:

Este algoritmo usa los mismos parámetros que el anterior algoritmo.

- Podemos ver que:
  - Se logra el resultado ideal en 2/5 ficheros y los demás resultados se acercan a la solución ideal, por tanto podemos decir que el resultado es muy bueno.
  - Se observa que no se necesita aumentar en la mayoría de casos la población inicial en la mayorías de los casos para tener una solución buena
  - No se ven patrones entre las demás variables como para sacar una observación.
- El mejor resultado es con:
  - File1.txt
    - Generaciones máximas:100
    - Población inicial:10
    - Iteraciones:100
    - Resultado: [25, 24, 17, 16, 13, 12, 10, 7] -> Number of Edges = 20
  - File2.txt
    - Generaciones máximas:500
    - Población inicial:10
    - Iteraciones:100
    - Resultado: [60, 45, 33, 27, 22, 8, 3] -> Number of Edges = 15
  - File3.txt
    - Generaciones máximas: 1000
    - Población inicial: 10
    - Iteraciones: 1000
    - Resultado: [60, 57, 52, 51, 46, 45, 43, 42, 34, 29, 28, 25, 20, 14, 11, 1] -> Number of Edges = 110
  - File4.txt
    - Generaciones máximas: 1000
    - Población inicial: 10
    - Iteraciones: 100

        - Resultado: (195, 194, 158, 157, 121, 120, 84, 83, 82, 47, 46, 45, 10, 9) -> Number of Edges = 79
    - File5.txt
        - Generaciones máximas: 1000
        - Población inicial: 20
        - Iteraciones: 500
        - Resultado: (494, 493, 414, 413, 334, 333, 254, 253, 175, 174, 173, 94, 93, 14, 13) -> Number of Edges = 98

Diferencia entre los dos algoritmos híbridos:

- No hay diferencias destacables entre los dos algoritmos ya que el resultado de ambas es bueno.
- En el primer algoritmo el resultado ideal fue conseguido en 3/5 ficheros y en el segundo algoritmo fue conseguido en 2/5
- Se podría decir que por los resultados obtenidos el algoritmo 1 es mas eficiente que el 2.

Nota: No se realizaron estudios sobre el algoritmo hibrido greedy y evolutivo ya que los resultados del algoritmo local no se acercaban a los deseados.