

**Análisis de**

**Algoritmos**

**Proyecto**

**Primer Parcial**

**Integrantes:**

**Allison Brito Mendoza**

**Marlon Eddie Lindao Varas**

**Vielka Villavicencio**

**Profesor:**

**Phd. Miguel Realpe**

**Paralelo:**

**2**

# Bellman Ford's Algorithm

Este algoritmo determina la ruta más corta desde un nodo origen hacia los demás nodos para ello es requerido como entrada un grafo cuyas aristas posean pesos. Siendo la diferencia de este algoritmo con los demás es que los pesos pueden tener valores negativos ya que nos permite detectar la existencia de un ciclo negativo.

## Descripción

Parte de un vértice origen que es ingresado y relaja las aristas con |V| - 1 veces, siendo |V| el número de vértices del grafo. Para detectar ciclos negativos se realiza la relajación una vez más y si se obtuvo mejores resultados e porque existe un ciclo negativo, para verificar si se tiene un ciclo negativo se puede seguir relajando las veces que queramos y seguirá obteniéndose mejores resultados

## Pseudocódigo

Bellman-Ford (G,s)

Inicializar

for cada v perteneciente a V[G]

do d[v] = infinito

p[v] = nulo

p[s] = 0

for i=1 to V[G]-1

do for cada arco (u,v) perteneciente a A[G]

Relajación

if d[v] > d[u] + w(u,v) then

d[v] = d[u] + w(u,v)

p(v) = u

for cada arco (u,v) chequea lazo de peso negativo

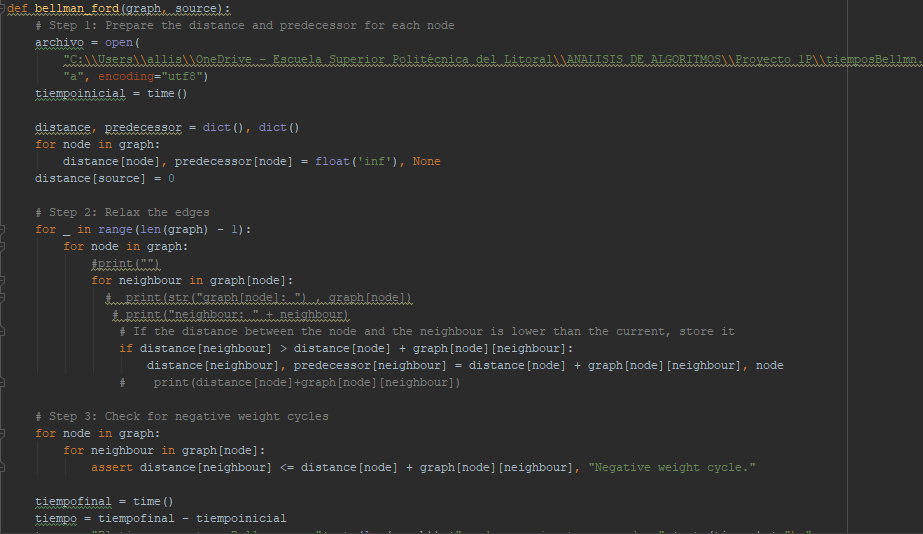
do if d[v] > d[u] + w(u,v) then

return FALSO 'el algoritmo no converge

return VERDADERO

Tiempo de ejecución O(n\*m)

## Implementación del algoritmo



Pruebas de la implementación con los siguientes tipos de grafos:

Tipo de grafo 🡪 Tiempo de ejecución

Grafo de 100 nodos 🡪 0.04488062858581543 segundos

Grafo de 200 nodos 🡪 0.07878923416137695 segundos

Grafo de 500 nodos 🡪 0.4530196189880371 segundos

Grafo de 1000 nodos 🡪 1.608994960784912 segundos

Grafo de 2000 nodos 🡪 6.389117240905762 segundos

## Tiempo de ejecución vs Asintótica estimada

|  |  |  |
| --- | --- | --- |
| Número de nodos | Tiempo de ejecución | Asintotica Estimada |
| 100 | 0.04488062858581543 | O(2\*100) |
| 200 | 0.07878923416137695 | O(2\*200) |
| 500 | 0.4530196189880371 | O(2\*500) |
| 1000 | 1.608994960784912 | O(2\*1000) |
| 2000 | 6.389117240905762 | O(2\*2000) |

# Dijkstra Algorithm

Es un algoritmo más eficiente que el algoritmo de Bellman Ford, ya que, por principio, no toma grafos con ponderaciones negativas, por lo que no realiza, el paso de verificación de las distancias. Es decir, Dijkstra encuentra caminos más cortos para grafos ponderados con peso positivo.

## Descripción

Parte de un grafo con aristas con peso positivo, y se envía un vértice inicial, después se procede a poner las distancias de los vértices a infinito, a excepción del inicial que vale 0, y se cambia la distancia de sus vecinos, según la ponderación de la arista que los une, después se escoge un vértice vecino al inicial, y se procede a cambiar las distancias de los vértices vecinos, tomando en cuenta, que el cambio se realiza, si la distancia nueva es menor a la que tienen, y se realiza este procedimiento hasta que no haya más vértices que visitar.

## Pseudocódigo

DIJKSTRA (G, w, vs)

1. Inicialización (G, vs)

2. S = ∅; Q = G.V

3. while Q 6= ∅ do

4. u = Extrae M in(Q)

5. S = S ∪ u

6. for cada vértice v ∈ Adyacentes de [u]

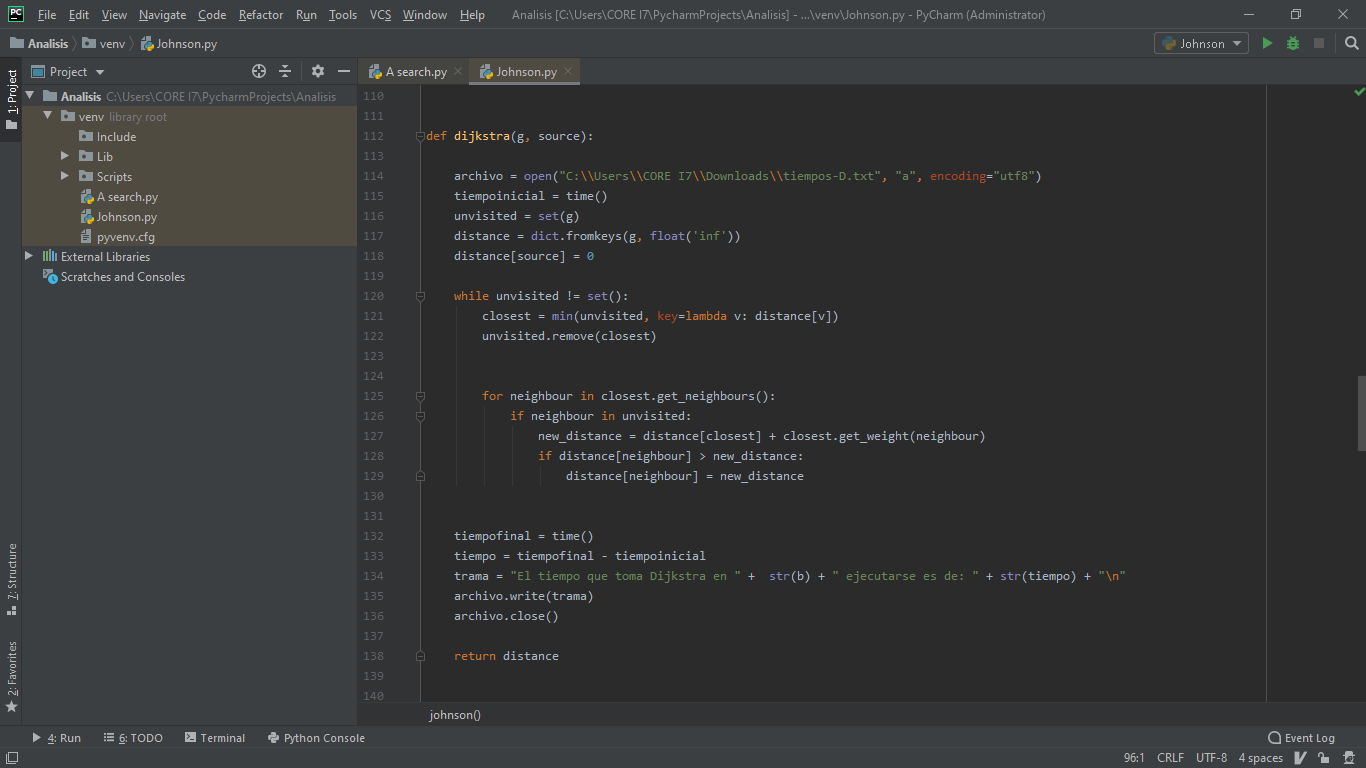
7. Comparación de etiquetas distancia (u, v, w (u, v))

Tiempo de ejecución O (m + n\*log(n))

## Implementación de Dijkstra

El código del algoritmo de Dijkstra se encuentra en los anexos, en el archivo Johnson.py

Aquí se muestra una foto del algoritmo implementado



Pruebas de la implementación con los siguientes tipos de grafos:

Tipo de grafo 🡪 Tiempo de ejecución

Grafo de 100 nodos 🡪 0.015624284744262695 segundos

Grafo de 200 nodos 🡪 0.01564192771911621 segundos

Grafo de 500 nodos 🡪 0.06247663497924805 segundos

Grafo de 1000 nodos 🡪 0.37120604515075684 segundos

Grafo de 2000 nodos 🡪 1.2342875003814697 segundos

## Tiempo de ejecución vs Asintótica estimada

|  |  |  |
| --- | --- | --- |
| Número de nodos | Tiempo de ejecución | Asintótica Estimada |
| 100 | 0.015624284744262695 | O(5+100lg(100)) |
| 200 | 0.01564192771911621 | O(5+200lg(200)) |
| 500 | 0.06247663497924805 | O(5+500lg(500)) |
| 1000 | 0.37120604515075684 | O(5+1000lg(1000)) |
| 2000 | 1.2342875003814697 | O(5+2000lg(2000)) |

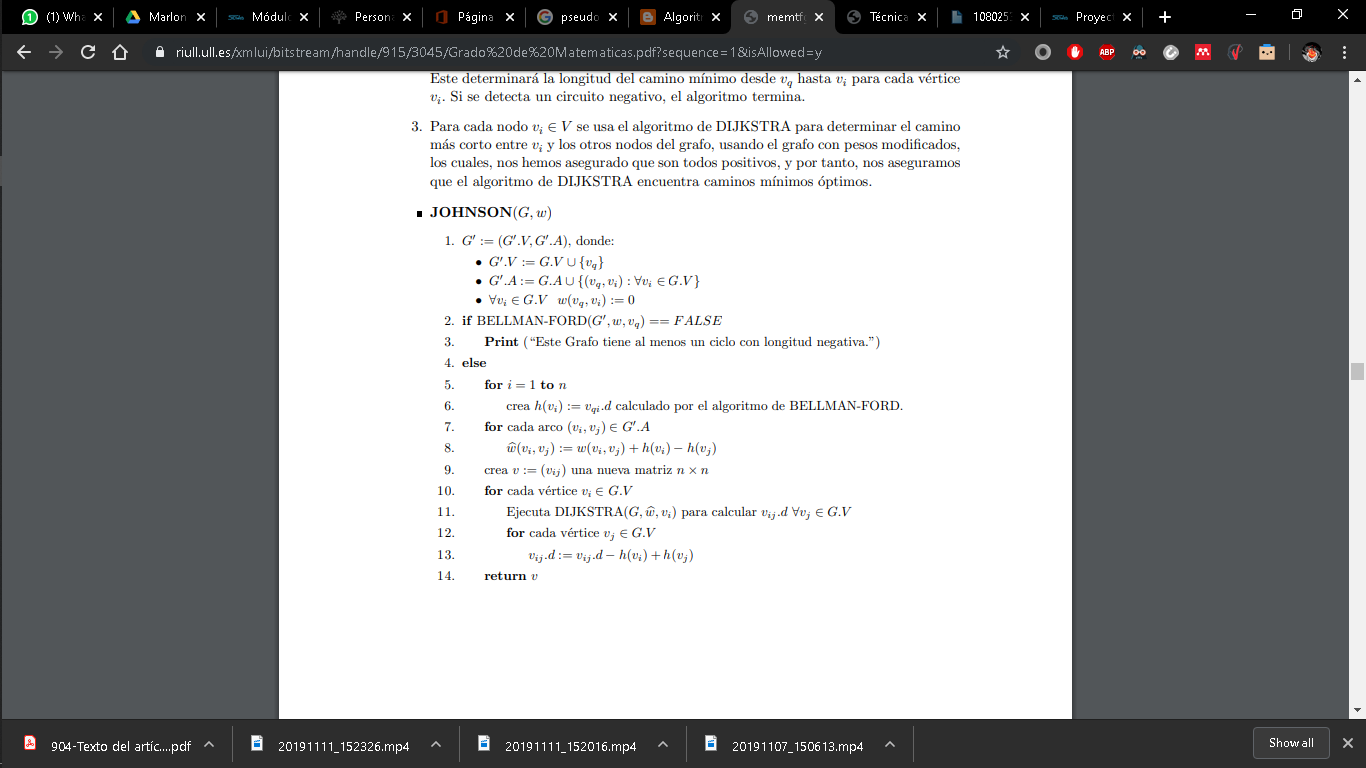
# Johnson Algorithm

Este algoritmo une las ventajas del algoritmo de Dijkstra y Bellman Ford, para hacerlo más rápido que el algoritmo de Floyd Warshall, para grafos de poca densidad.

## Descripción

1. Se añade un nuevo nodo al grafo, y se lo conecta a cada nodo del grafo, con una arista de peso cero.
2. Se usa el algoritmo de Bellman Ford, para calcular los caminos más cortos del nuevo nodo hasta cada vértice del grafo original. Si se encuentra un ciclo negativo, el algoritmo termina.
3. Si no hay ciclos negativos, se procede a realizar Dijkstra para conocer los caminos más cortos entre cada par de vértice del grafo, usando los pesos modificados del grafo obtenidos en Bellman Ford, y cuyos valores se conoce que son positivos por haber pasado el paso 2.

## Pseudocódigo

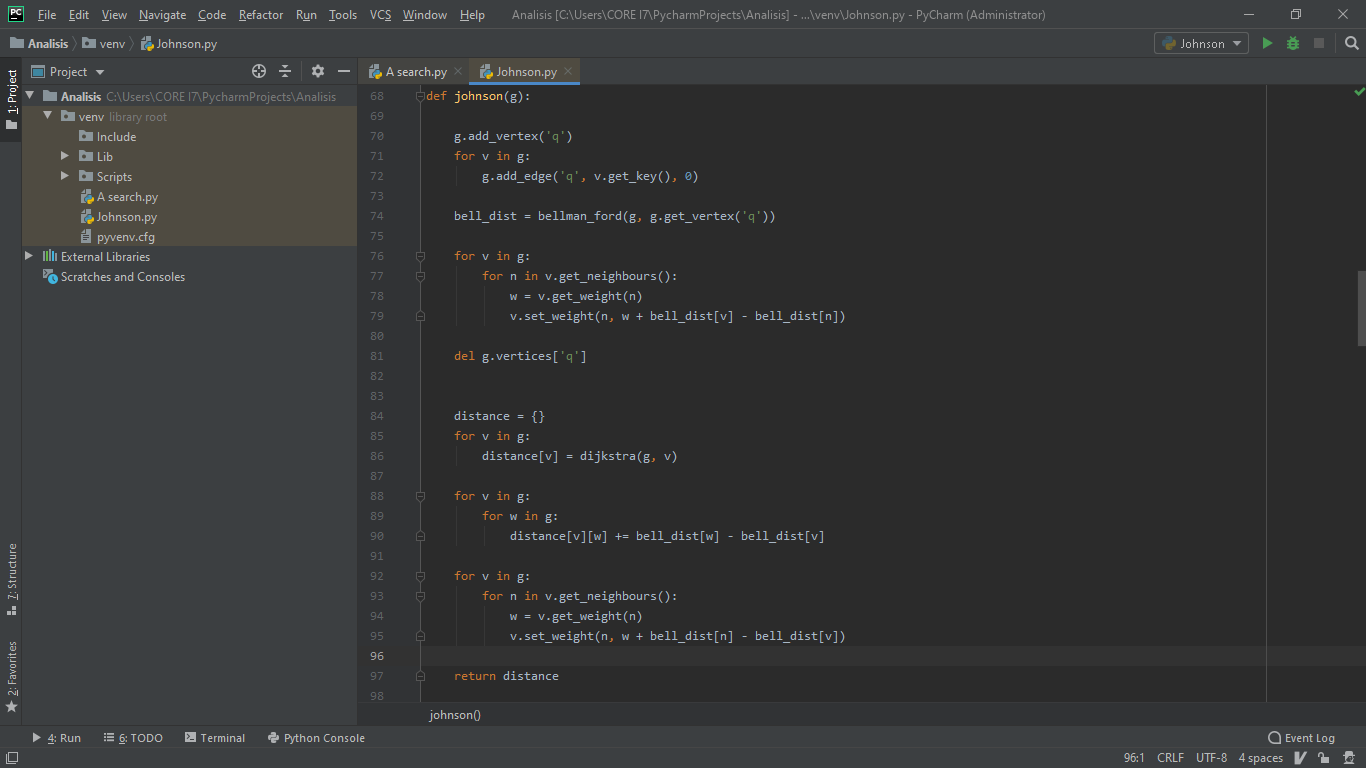


Tiempo de ejecución O (n\*(m + n\*log(n)))

## Implementación del Johnson

El código del algoritmo de Johnson se encuentra en los anexos, en el archivo Johnson.py

Aquí se muestra una foto del algoritmo implementado:



Pruebas de la implementación con los siguientes tipos de grafos:

Tipo de grafo 🡪 Tiempo de ejecución

Grafo de 100 nodos 🡪 0.15623784065246582 segundos

Grafo de 200 nodos 🡪 2.7185559272766113 segundos

Grafo de 500 nodos 🡪 45.293697118759155 segundos

Grafo de 1000 nodos 🡪 501.4607729911804 segundos

Grafo de 2000 nodos 🡪 4758.636205196381 segundos

## Tiempo de ejecución vs Asintótica estimada

|  |  |  |
| --- | --- | --- |
| Número de nodos | Tiempo de ejecución | Asintótica Estimada |
| 100 | 0.15623784065246582 | O(100\*((100\*5+5) +100lg(100))) |
| 200 | 2.7185559272766113 | O(200\*((200\*5+5) +200lg(200))) |
| 500 | 45.293697118759155 | O(500\*((500\*5+5) +500lg(500))) |
| 1000 | 501.4607729911804 | O(1000\*((1000\*5+5) +1000lg(1000))) |
| 2000 | 4758.636205196381 | O(2000\*((2000\*5+5) +2000lg(2000))) |

# Floyd–Warshall's Algorithm

Es un algoritmo de análisis de grafos que encuentra los caminos más cortos de un grafo en el cual las aristas tengan un costo (distancia entre nodo y nodo, duración del viaje entre nodo, etc.). El algoritmo al ser ejecutado encontrara el camino menor o más corto entre todos los pares de vértices, pero no devuelve los detalles de los caminos en sí. Cabe mencionar que el algoritmo tiene problemas con ciclos negativos.

## Descripción

Para la ejecución de este algoritmo no es necesario elegir un nodo origen o destino. El algoritmo nos proporciona todos los posibles caminos mínimos entra cada par de nodos origen y destino.

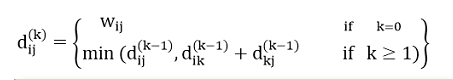
Suponiendo que se debe llegar del vértice i al j de forma óptima contando con k-1 puntos intermedios. Luego se quiere encontrar distancias más cortas agregando un k-ésimo punto intermedio. Se puede ir de i a k y después a j o mantener la distancia que ya se tenía.

En ambos casos se considera los k-1 puntos como intermedios. Al hacer que k tome el valor de todos los vértices, obtenemos la distancia más corta entre todas las parejas.

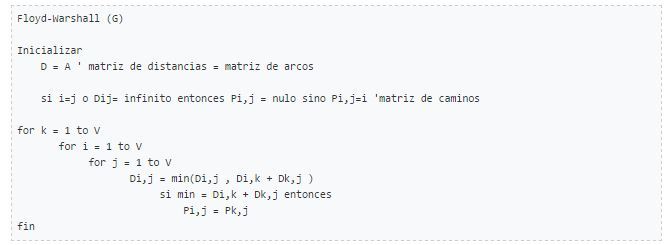
Cuando se analiza un grafo G con peso y n vértices, se puede obtener la matriz de peso.

Si 1,2,3, 4 …, n son vértices, entonces el elemento de la fila i y la columna j, es 0 si i=j. Si no existiese arista directa de i a j, toma un valor de -1. Si de i a j hay arista directa, entonces se toma el valor del peso de la arista. Así se forma la matriz de adyacencias.

Como formula se puede ver que:

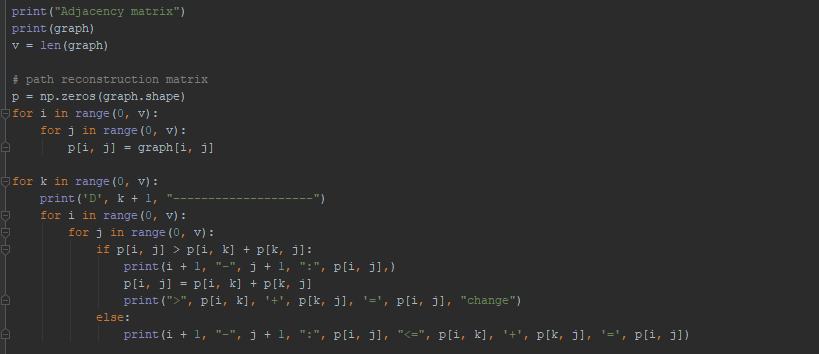


## Pseudocódigo



Complejidad: O(n3)

## Algoritmo implementado



Pruebas de la implementación con los siguientes tipos de grafos:

Tipo de grafo 🡪 Tiempo de ejecución

Grafo de 100 nodos 🡪 168.27494549751282 segundos

Grafo de 200 nodos 🡪 1517.6769881248474 segundos

Grafo de 500 nodos 🡪 21303.930930137634 segundos

## Tiempo de ejecución vs Asintótica estimada

|  |  |  |
| --- | --- | --- |
| Número de nodos | Tiempo de ejecución | Asintótica Estimada |
| 100 | 168.27494549751282 | O(1000000) |
| 200 | 1517.6769881248474 | O(8000000) |
| 500 | 45.293697118759155 | O(75000000) |
|  |  |  |
|  |  |  |

# A\* search Algorithm

Es un algoritmo de búsqueda que puede ser empleado para el cálculo de caminos mínimos en una red. Una de las principales características es que hará uso de una función de evaluación heurística, mediante la cual etiquetar los diferentes nodos de la red y que servirá para determinar l probabilidad de los nodos de pertenecer al camino óptimo.

Esta función de evaluación está compuesta a su vez de otras dos funciones y es aquella que etiquetará los nodos de la red compuesta. Una función indica la distancia del camino del nodo origen hasta el nodo a etiquetar y la otra función expresa la distancia estimada desde este nodo a etiquetar hasta el nodo destino hasta el que se pretende encontrar un camino mínimo. Esto quiere decir que, si se desea encontrar el camino más corto desde el nodo origen a hasta el nodo destino b, un nodo intermedio de la red n tendría la siguiente función de evaluación f(n) como etiqueta:

f(n)=g(n)+h(n)

h(n) es una función heurística, expresa la idea de cuán lejos aún se está de alcanzar el nodo destino, y de su correcta elección dependerá el funcionamiento correcto del algoritmo A\* al aplicarlo en una red. En el caso de que la función heurística nunca sobreestime el valor de la distancia entre el nodo y el destino, se dice que es admisible y está garantizada la solución óptima; mientras que si se da el caso de que la función no sea admisible no se puede garantizar el hallazgo de la solución para el problema del camino más corto.

## Pseudocódigo

1.- Establecer el nodo s como origen. Hacer f(s)=0, y f(i)=∞ para todos los nodos i diferentes del nodo s. Iniciar el conjunto Q vacío.

2.- Calcular el valor de f(s) y mover el nodo s al conjunto Q.

3.- Seleccionar el nodo i del conjunto Q que presente menor valor de la función f(i) y eliminarlo del conjunto Q.

4.- Analizar los nodos vecinos j de i. Para cada enlace (i, j) con coste cij hacer:

4.1.-Calcular: f(j)’=g(i)+cij +h(j)

4.2.-Si f(j)’<f(j),

4.2.1.-Actualizar la etiqueta de j y su nuevo valor será: f(j)= g(i)+cij +h(i)

4.2.2.-Insertar el nodo j en Q

4.3.-Si f(j)’≥f(j)

4.3.1.-Dejar la etiqueta de j como está, con su valor f(j)

5.- Si Q está vacío el algoritmo se termina. Si no está vacío, volver al paso 3.

## Complejidad:

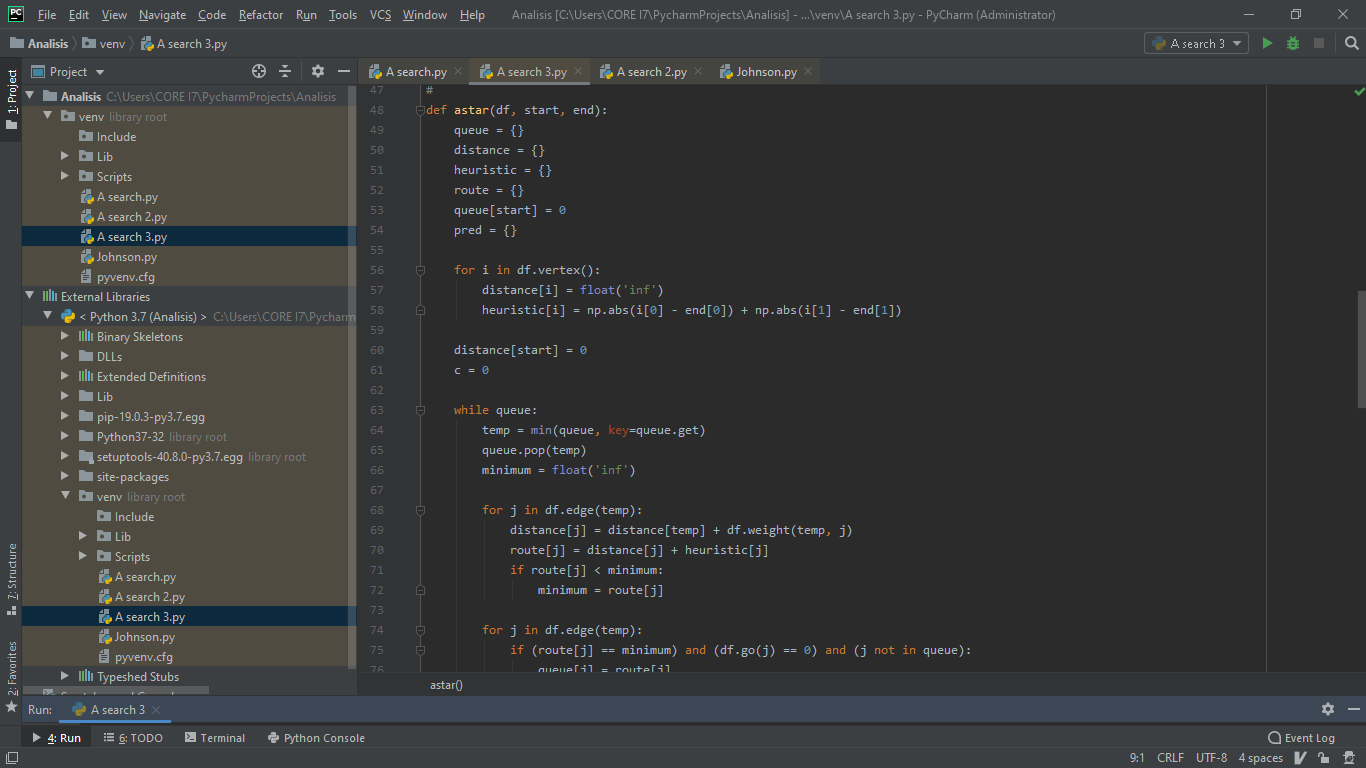
Su tiempo depende mucho de la función heurística. Si la función es O (1), el tiempo del algoritmo se vuelve el de Dijkstra siendo:

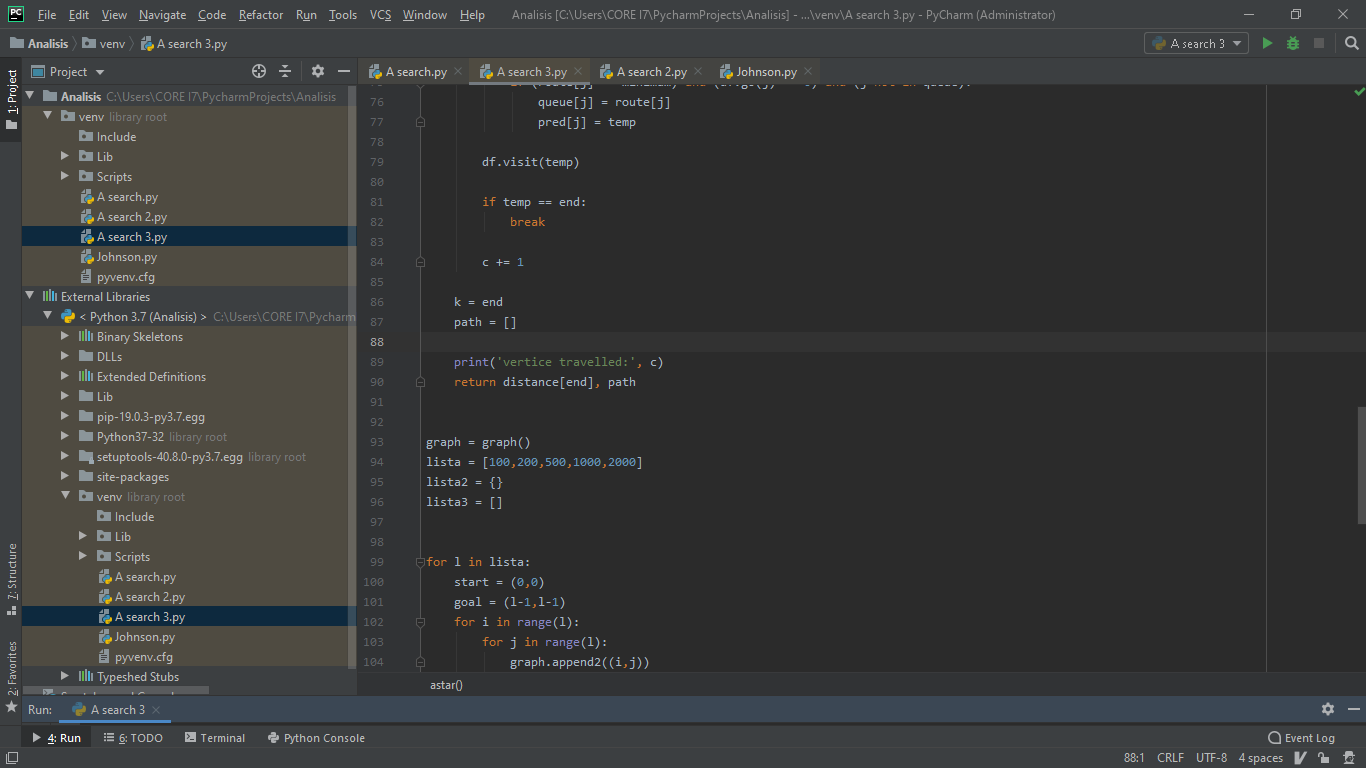
O (m + n\*log(n))

Si es usado para una búsqueda espacial sin barreras, el tiempo se vuelve a:

O(bd) donde b es el factor de ramificación y d es la profundidad de la solución.

## Algoritmo Implementado





Pruebas de la implementación con los siguientes tipos de grafos:

Tipo de grafo 🡪 Tiempo de ejecución

Grafo de 100 nodos 🡪 0.031002283096313477 segundos

Grafo de 200 nodos 🡪 0.125929594039917 segundos

Grafo de 500 nodos 🡪 0.8514919281005859 segundos

Grafo de 1000 nodos 🡪 3.2671217918395996 segundos

Grafo de 2000 nodos 🡪 13.47526216506958 segundos

## Tiempo de ejecución vs Asintótica estimada

|  |  |  |
| --- | --- | --- |
| Número de nodos | Tiempo de ejecución | Asintótica Estimada |
| 100 | 0.031002283096313477 | O((100\*100\*2) +100log(100)) |
| 200 | 0.125929594039917 | O((200\*200\*2) +200log(200)) |
| 500 | 0.8514919281005859 | O((500\*500\*2) +500log(500)) |
| 1000 | 3.2671217918395996 | O((1000\*1000\*2) +1000log(1000)) |
| 2000 | 13.47526216506958 | O((2000\*2000\*2) +2000log(2000)) |

<http://idelab.uva.es/algoritmo>

https://escarbandocodigo.wordpress.com/2011/07/11/1051/