



TEMAS SELECTOS DE MODELADO
Primavera 2022

PROYECTO FINAL

***Ciclo de identificación de arbitraje en el comercio de
criptomonedas***

Equipo No. 1

Integrantes

Nyrma Paulina Hernández Trejo
Aide Jazmín González Cruz
Joel Jaramillo Pacheco
Jesús Enrique Miranda Blanco

Índice

Índice	2
Objetivo	3
Introducción	3
Problema de optimización	4
Marco teórico	5
Bellman Ford	5
Algoritmo de Bellman Ford	6
Pseudocódigo	6
Complejidad del Algoritmo	7
Usos del Algoritmo Bellman Ford	8
Descripción del dataset	8
Transformación del dataset	8
Eficientar el código	10
Perfilamiento	10
Time	10
Timeit	11
cProfile	11
line_profiler	11
Memoria	12
CPU	12
Eficientar el código	13
Exchange matrix	13
Bellman Ford	14
Compilación de código con C	16
Resultados	18
Perfilamiento	19
Compilación en C	20
Aplicación del método	20
Casos de uso	21
Conclusiones	25
Referencias	26

Objetivo

Resolver el tema de “Ciclo de identificación de arbitraje en el comercio de criptomonedas”, a través de encontrar el camino más corto en un grafo dirigido ponderado, usando el método de *Bellman Ford*.

Introducción

El *trading* consiste en la compra-venta de activos cotizados con mucha liquidez en un mercado que puede ser de acciones, divisas e inversiones. Este mercado financiero es electrónico y está regulado. Su objetivo es obtener un beneficio económico cuando la operación genera una plusvalía.¹

En la última década, desde el debut del *Bitcoin* en internet, el *trading* de criptomonedas se ha vuelto cada vez más popular. Las criptomonedas son monedas digitales creadas utilizando *blockchain*, que es un conjunto de tecnologías que permiten llevar un registro seguro, descentralizado, sincronizado y distribuido de las operaciones digitales, sin necesidad de la intermediación de terceros ². Se diferencian de las monedas fiat emitidas por gobiernos de todo el mundo porque no son tangibles: en cambio, están formadas por *bits* y *bytes* de datos. Además, las criptomonedas no tienen una autoridad centralizada, como un banco central que las emite o que regule su circulación en la economía. Como las criptomonedas no son emitidas por ningún organismo gubernamental, no son consideradas como moneda legal.

Aunque las criptomonedas no son reconocidas como moneda de curso legal en la economía mundial, tienen el potencial de cambiar el panorama financiero y esto hace que sea muy difícil ignorarlas. Al mismo tiempo, la tecnología *blockchain*, que es la base de la creación de criptomonedas, ha generado nuevas oportunidades de inversión que los *traders* pueden capitalizar.³

De acuerdo a la página de CoinMarketCap a febrero de 2022 existen 17,540 criptomonedas, y existen varios tipos de ellas. ⁴ En la lista de las criptomonedas más populares se incluye el *Bitcoin*, que se considera la criptomoneda original. Cabe mencionar que existe el índice Cripto 10, que se puede comparar con un índice bursátil o de divisas, pero que está compuesto por las 10 criptomonedas más grandes y con mayor cuota de mercado.

Problema de optimización

Analizando el ciclo de identificación de arbitraje en el comercio de criptomonedas, este se puede considerar como un problema del camino o ruta más corta (*shortest path*), que es un tipo especial de problema de **programación lineal**, o bien conocido como **problema de flujo en redes**.

El problema de encontrar la ruta más corta entre dos criptomonedas, no es práctico por la cantidad enorme de alternativas que resultarían, ya que como se ha mencionado existe un gran número de ellas y por el rápido cambio que existe en los precios. Por esto se requieren algoritmos cuyo tiempo de cómputo sea pequeño o al menos razonable.

En el flujo de redes se tienen básicamente 3 elementos:

- **Nodos:** corresponde a un vértice de un grafo. En este proyecto lo conforman las criptomonedas.
- **Arcos:** corresponde a un par ordenado de vértices que representan una posible dirección de desplazamiento a través de un grafo. Para el caso de este proyecto corresponde a un par ordenado de criptomonedas.
- **Flujo:** tasa de intercambio entre criptomonedas en dólares para poder hablar de la misma unidad y poder construir más adelante la matriz de intercambio entre cada una.

Con estos elementos se pueden construir grafos, que son una herramienta matemática que se emplea para formular problemas de encaminamiento, y las cuales se forman por un conjunto de nodos (N) y arcos (E).

$$G = (N, E)$$

En la búsqueda del camino más corto, la idea principal es la de encontrar el camino con un costo mínimo entre una fuente (S) y un destino (D), donde S y D son nodos. Y el flujo se puede traducir como el costo de S a D que se puede denotar como $c(s,d)$. Si este costo se mantiene constante para todos los enlaces, la solución es la ruta de menor número de saltos.⁸

Este método como se puede observar es aplicable a grafos dirigidos, que en el caso del análisis de criptomonedas, cada criptomoneda se dirige a todas las demás monedas (cada moneda en este caso es un nodo), para realizar la búsqueda y comparación de tasas de intercambio (costo en las aristas).

Con el algoritmo de la búsqueda del camino más corto se pueden encontrar las oportunidades para el arbitraje de divisas, ya que en particular este mercado al no estar regulado existen fluctuaciones en el precio, lo que implica reaccionar rápidamente para encontrar estas rutas más cortas a todos los demás nodos en el grafo, la cual proporcionará

información sobre el intercambio que se debe realizar entre una moneda por otra y por otra, y así generar una ganancia.

Los algoritmos más importantes para resolver este tipo de problemas son:

- Algoritmo de Dijkstra, resuelve el problema de los caminos más cortos desde un único vértice origen hasta todos los otros vértices del grafo.
- Algoritmo de Bellman - Ford, resuelve el problema de los caminos más cortos desde un origen si la ponderación de las aristas es negativa.
- Algoritmo de Búsqueda A*, resuelve el problema de los caminos más cortos entre un par de vértices usando la heurística para intentar agilizar la búsqueda.
- Algoritmo de Floyd - Warshall, resuelve el problema de los caminos más cortos entre todos los vértices.
- Algoritmo de Johnson, resuelve el problema de los caminos más cortos entre todos los vértices y puede ser más rápido que el de Floyd-Warshall en grafos de baja densidad.
- Algoritmo de Viterbi, resuelve el problema del camino estocástico más corto con un peso probabilístico adicional en cada vértice.⁸

En este proyecto se eligió usar el Algoritmo de Bellman - Ford, ya que ofrece una ventaja sobre los demás algoritmos, ya que soporta valores positivos y negativos en las aristas, lo que resuelve el tema de las tasas de intercambio entre monedas, donde muchas veces estas tasas serán negativas, ya que una moneda al tener un valor más bajo, no nos puede servir para comprar una criptomoneda de más valor, y por tanto generará un arco negativo.

Por lo tanto, podemos concluir que si podemos encontrar un ciclo de vértices tal que la suma de sus pesos sea negativa, entonces existe una oportunidad para el arbitraje de divisas.

Marco teórico

Bellman Ford

Alfonso Shimbel propuso el algoritmo en 1955, pero ahora lleva el nombre de Richard Bellman y Lester Ford Jr., quienes lo presentaron en 1958 y 1956. En 1959, Edward F. Moore publicó una variación del algoritmo, a veces denominado el Algoritmo de Bellman-Ford-Moore.⁶ El algoritmo de Bellman Ford calcula la ruta más corta desde un nodo origen hacia los demás nodos en un grafo dirigido con pesos en las aristas. La ventaja de este algoritmo es que los pesos pueden tomar valores negativos y, en dicho caso, se detectaría la existencia de un ciclo negativo. Un ciclo negativo se forma cuando la suma de todos sus pesos toma un valor menor a cero. Los pesos negativos se encuentran en varias aplicaciones de grafos. Por ejemplo, en lugar de pagar el costo de un camino, podemos obtener alguna ventaja si seguimos el camino.

Algoritmo de Bellman Ford

El método parte de un vértice origen y relaja las aristas para actualizar las distancias entre vértices $|V| - 1$ veces, con $|V|$ como el número de vértices en el grafo. La idea de este algoritmo es recorrer todos los bordes del grafo uno por uno en un orden aleatorio. Puede ser cualquier orden aleatorio con la única restricción que se encuentren conectados. Las reiteraciones dejan a las distancias mínimas recorrer el grafo, puesto que en la ausencia de ciclos negativos, el camino más corto solo visita cada vértice una vez.

El algoritmo de Bellman Ford funciona sobreestimando la longitud del camino desde el vértice inicial hasta todos los demás vértices. Basado en el "Principio de Relajación", los valores más precisos recuperan gradualmente una aproximación a la distancia adecuada hasta llegar finalmente a la solución óptima. Es decir, relaja iterativamente esas estimaciones al encontrar nuevos caminos que son más cortos que los caminos sobreestimados previamente. Al hacer esto repetidamente para todos los vértices, podemos garantizar que el resultado esté optimizado.

Pseudocódigo

Es un proceso iterativo en el cual comienza sobreestimando la longitud del camino desde el vértice inicial hasta todos los demás vértices. Luego, el algoritmo relaja iterativamente esas estimaciones al descubrir nuevas formas que son más cortas que las rutas sobreestimadas anteriormente.

Puede asegurarse de que el resultado esté optimizado repitiendo este proceso para todos los vértices de la siguiente forma:

1. Definir grafo dirigido con pesos
2. Elegir un nodo inicial y asignar pesos (infinitos) a las distancias de cada otro vértice
3. Visitar cada nodo y relajar el peso
4. Se actualizan $|V| - 1$ pesos de cada vértice
5. Revisar si existe un ciclo negativo

Para definir el pseudocódigo se deben en tomar en cuenta las siguientes consideraciones:

- Se debe mantener la distancia de la ruta de cada vértice. Eso se puede almacenar en una matriz V-dimensional, donde V es el número de vértices.
- No solo necesita saber la longitud del camino más corto, sino que también debe poder encontrarlo. Para lograr esto, debe asignar cada vértice al vértice que actualizó más recientemente su longitud de ruta.
- Cuando finaliza el algoritmo, puede encontrar la ruta desde el vértice de destino hasta el origen.

```
function BellmanFord(G, s = nodo inicial, distancia inicial = inf)
//G es el grafo dirigido
//s es el nodo inicial
  for each vertex V in G
    dist[V] <- infinite // dist es distancia
    prev[V] <- NULL // prev es previa
  dist[s] <- 0
  for each vertex V in G
    for each edge (u,v) in G
      temporaryDist <- dist[u] + edgeweight(u, v)
      if temporaryDist < dist[v]
        dist[v] <- temporaryDist
        prev[v] <- u
  for each edge (U,V) in G
    If dist[U] + edgeweight(U, V) < dist[V]
      Error: ¡Existe ciclo negativo!
  return dist[], prev[]
```

Complejidad del Algoritmo

A continuación se muestra la complejidad temporal del algoritmo Bellman Ford.

Para todos los casos, la complejidad de este algoritmo estará determinada por el número de comparaciones de aristas. Dependiendo de la condición:

```
if temporaryDist < dist[v]
```

Las diferencias de relajación de las distancias dependen del grafo y de la secuencia de observación de sus vértices.

Es posible que el algoritmo deba someterse a todas las repeticiones mientras se actualizan los bordes, pero en muchos casos, el resultado se obtiene en las primeras iteraciones, por lo que no se requieren actualizaciones. Convergiendo en la siguiente línea del pseudocódigo:

```
dist[v] <- temporaryDist
```

Usos del Algoritmo Bellman Ford

Hay varias aplicaciones del mundo real para el algoritmo Bellman-Ford, que incluyen:

- Identificar ciclos de peso negativos
- En una reacción química, calcular la menor ganancia/pérdida de calor posible.
- Identificar el método de conversión de moneda más eficiente.

Descripción del dataset

Para la extracción de datos se hizo un web scraping para obtener los símbolos de las criptomonedas, se usó el módulo de “*pandas datareader*” tomando el API de yahoo finanzas. Se descargaron 10,105 monedas del 2022-05-01 al 2022-05-13, con los precios en dólares.

Con el siguiente layout:

	High	Low	Open	Close	Volume	Adj Close	ticker
Date							
2022-05-01	38627.859375	37585.789062	37713.265625	38469.093750	2.700276e+10	38469.093750	BTC
2022-05-02	39074.972656	38156.562500	38472.187500	38529.328125	3.292264e+10	38529.328125	BTC
2022-05-03	38629.996094	37585.621094	38528.109375	37750.453125	2.732694e+10	37750.453125	BTC
2022-05-04	39902.949219	37732.058594	37748.011719	39698.371094	3.675440e+10	39698.371094	BTC
2022-05-05	39789.281250	35856.515625	39695.746094	36575.140625	4.310626e+10	36575.140625	BTC

Donde:

- Date: Fecha de los precios
- High: Precio más alto del día
- Low: Precio más bajo del día
- Open: Precio de apertura
- Close: Precio de cierre
- Volume: Volumen de transacciones
- Adj Close: Precio del cierre ajustado (en este caso es el mismo que el cierre)
- Ticker: Símbolo de la criptomoneda

Transformación del dataset

La primera parte de la transformación implica aplicar un *spread* aleatorio a los precios que va de $[0, .05]$, para simular la fluctuación de precios en el mercado. Después estos datos se convierten en una matriz, donde las criptomonedas se encuentran tanto en las columnas y renglones, y cada celda nos indica la tasa de intercambio entre cada una de ellas

	0	1	2	3	4	5	6
0	1.00	2.10	156.56	38.22	6.32	10.73	4.33
1	0.50	1.00	76.17	18.60	3.08	5.22	2.11
2	0.01	0.01	1.00	0.25	0.04	0.07	0.03
3	0.03	0.06	4.14	1.00	0.17	0.28	0.11
4	0.17	0.35	25.77	6.29	1.00	1.77	0.71
5	0.10	0.20	15.18	3.71	0.61	1.00	0.42
6	0.25	0.50	37.30	9.11	1.51	2.56	1.00

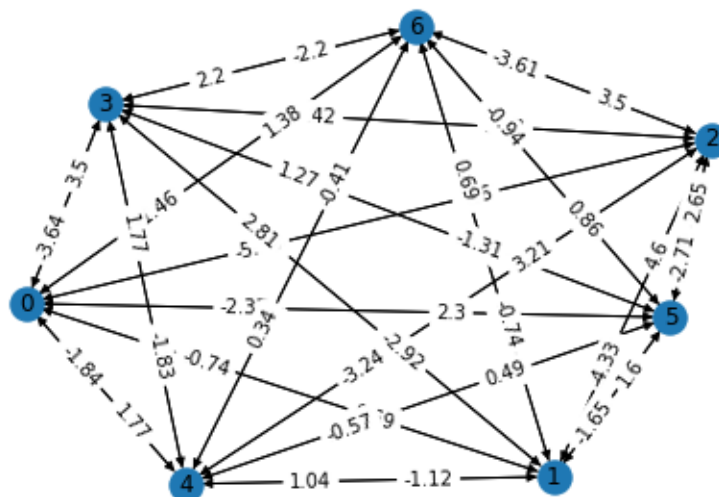
El recuadro azul nos dice que de la *moneda 0* nos alcanza comprar 0.5 de la *moneda 1*, lo cuál no es viable, pero en el caso contrario (cuadro verde), nos indica que de la *moneda 1* podemos comprar 2.1 *monedas 0*.

Esta representación nos da una primera idea de cómo se está comportando el mercado, sin embargo para manejar de una manera más sencilla estas representaciones, se aplicó logaritmo a los precios, para obtener las tasas de intercambio con signo:

	0	1	2	3	4	5	6
0	-0.000000	-0.741937	-5.053439	-3.643359	-1.843719	-2.373044	-1.465568
1	0.693147	-0.000000	-4.332968	-2.923162	-1.124930	-1.652497	-0.746688
2	4.605170	4.605170	-0.000000	1.386294	3.218876	2.659260	3.506558
3	3.506558	2.813411	-1.420696	-0.000000	1.771957	1.272966	2.207275
4	1.771957	1.049822	-3.249211	-1.838961	-0.000000	-0.570980	0.342490
5	2.302585	1.609438	-2.719979	-1.311032	0.494296	-0.000000	0.867501
6	1.386294	0.693147	-3.618993	-2.209373	-0.412110	-0.940007	-0.000000

De esta manera con el signo, nos podemos dar idea que nos conviene encontrar una combinación negativa, lo que indica una oportunidad para el arbitraje de divisas.

Finalmente se transforma esta matriz a un grafo dirigido, donde cada nodo es una criptomoneda y las aristas son las tasas de intercambio entre una moneda y otra, que servirá de entrada al método de *bellman ford* para encontrar el ciclo negativo que da mayor ganancia.



Eficientar el código

Parte del objetivo de este proyecto es hacer más eficiente el código que desarrollamos, para lograr esto, nos apoyamos de herramientas que permiten hacer un análisis de perfilamiento, es decir revisar el comportamiento del código con respecto al uso de los recursos (CPU y memoria) para posteriormente reescribirlo usando técnicas que ayudan a hacer más eficiente el código e incluso hacer uso de lenguajes más eficientes como por ejemplo C.

Perfilamiento

Para realizar el proceso de eficientar el código se realizaron los siguientes perfilamientos.

Time

El método de tiempo de Python `time()` devuelve el tiempo como un número de punto flotante expresado en segundos desde la época, en UTC. (La época es el punto donde comienza el tiempo y depende de la plataforma. Para Unix, la época es el 1 de enero de 1970, 00:00:00 (UTC)).

```
start_time_c = time.time()
```

```
df = get_data('data/historical_data.csv', '2022-05-12')
G = create_grap(df)
start_time_m = time.time()
bf_negative_cycle(G)
end_time = time.time()
```

Timeit

%timeit ejecuta el código dado muchas veces, luego devuelve la velocidad de ejecución del resultado más rápido.

```
%timeit bf_negative_cycle(G)
```

cProfile

cProfile proporciona un perfil deterministas de los programas de Python. Un perfil es un conjunto de estadísticas que describe con qué frecuencia y durante cuánto tiempo se ejecutan varias partes del programa.

cProfile se recomienda para la mayoría de los usuarios; es una extensión de C con una sobrecarga razonable que la hace adecuada para crear perfiles de programas de ejecución prolongada. Basado en Isprof, aportado por Brett Rosen y Ted Czotter.

```
cprof = cProfile.Profile()
cprof.enable()
res = bf_negative_cycle(G)
cprof.disable()
```

line_profiler

line_profiler es un módulo para hacer perfiles de funciones línea por línea.

```
line_prof = line_profiler.LineProfiler()
```

```
print(line_prof(bf_negative_cycle)(G))
print(line_prof.print_stats())
```

Memoria

`memory_usage` es un módulo de python para monitorear el consumo de memoria de un proceso, así como el análisis línea por línea del consumo de memoria para programas de python. Es un módulo de Python puro que depende del módulo `psutil`.

```
t = (bf_negative_cycle, (G,1,100000))

print(memory_usage(t, max_usage=True))

start_mem = memory_usage(max_usage=True)

res = memory_usage(t, max_usage=True, retval=True)
```

CPU

`perf` es una herramienta de análisis de rendimiento en Linux, disponible desde la versión 2.6.31 del kernel de Linux en 2009.[3] Se accede a la utilidad de control del espacio de usuario, denominada `perf`, desde la línea de comandos y proporciona una serie de subcomandos; es capaz de generar perfiles estadísticos de todo el sistema (tanto el kernel como el código del espacio de usuario). en nuestro caso se utilizó para medir ciclos, instrucciones, cache-references, cache-misses

```
%%bash

perf stat -S -e cycles,instructions,cache-references,cache-misses -r 20 python3
src/opt2/pipeline.py
```

Estadísticas por core

```
%%bash

perf stat -S --all-cpus -A -e cycles,instructions,cache-references,cache-misses -r 20
python3 src/opt2/pipeline.py
```

Estadísticas

```
%%bash
```

```
perf stat -S -r 20 python3 src/opt2/pipeline.py
```

Eficientar el código

Posteriormente se realizaron las siguientes acciones para efficientar el código, donde se muestran sólo parte del código que es relevante a la mejora marginal obtenida. para ver el código completo referirse al repositorio del proyecto. También nos referimos a Código Original al código que se generó en la práctica 1 donde no había efficientado el código.

Se realizaron este proceso en dos funciones de nuestro código, una referente a la función de *exchange matrix*, usada para variar el precio de las criptomonedas un 5% para simular la fluctuación de sus valores, y otro propiamente del método de *Bellman Ford*.

Exchange matrix

Referente a esta función se modificó el código derivado de las siguientes situaciones:

1.- Generaba un warning al correr la función

```
/tmp/ipykernel_78/1278838402.py:12: SettingWithCopyWarning:
A value is trying to be set on a copy of a slice from a DataFrame.
Try using .loc[row_indexer,col_indexer] = value instead

See the caveats in the documentation: https://pandas.pydata.org/pandas-docs/stable/user_guide/indexing.html#returning-a-view-versus-a-copy
c1[i] = aux/c1[['Precio']].values[i]*(1+random.uniform(0,max_spread_pct))
```

2.- Al usar esta función para variar el precio usando todo el dataset de criptomonedas consumía mucho tiempo.

En el **método** `exchange_rate_matrix(data)` se realizó el siguiente cambio de código;

Código Original

```
for i in range(n):

    c1[i]=aux/c1[['Precio']].values[i]*(1+random.uniform(0,max_
spread_pct))

    c1.drop(columns=['Precio'],inplace=True)
```

```

for i in range(len(c1.index)):
    for j in range(len(c1.columns)):
        if i==j:
            c1.loc[i,j] = 1

return c1

```

Código mejorado marginalmente en términos de eficiencia.

```

for i in range(n):
    c1[i] = aux/c1.loc[i]['Precio']*(1+random.uniform(0,max_spread_pct))

c1.drop(columns=['Precio'],inplace=True)

for i in range(len(c1.index)):
    for j in range(len(c1.columns)):
        if i==j:
            c1.loc[i,j] = 1

return c1

```

Bellman Ford

Derivado de los resultados obtenidos con las herramientas de perfilamiento anteriormente descritos, y basándose un poco más en la herramienta de **line_prof** donde se observaron las partes de código que tenía más hits y consumo de tiempo, fue un indicador para probar reescribir el código para que estos parámetros se disminuyera.

En el método

`bf_negative_cycle(graph, node_ini=None, distance_ini=np.inf)` se realizó el siguiente cambio de código;

Código Original

```
def bf_negative_cycle(graph, node_ini=None, distance_ini=np.inf):
```

```
    if node_ini is None:
```

```
        n_nodes = len(graph.nodes())
```

```
    else:
```

```
        assert node_ini <= len(graph.nodes), f"El nodo definido es mayor a los del grafo. Deberia de ser menor a {len(graph.nodes)}."
```

```
        n_nodes = node_ini
```

```
    n = len(graph.nodes()) + 1
```

```
    # Remove nan borders inside graph
```

```
    edges = []
```

```
    for edge in graph.edges().data():
```

```
        if ~np.isnan(edge[2]['weight']):
```

```
            edges.append(edge)
```

```
    # Initialize distances of nodes and predecessors
```

```
    distance= np.ones(n) * distance_ini # Starting distances with infinite values
```

```
    predecessors = np.ones(n) * -1 # Starting predecessors with -1 values
```

Código mejorado marginalmente en términos de eficiencia.

```
def bf_negative_cycle2(graph, node_ini=None, distance_ini=np.inf):
```

```
    n_nodes = len(graph.nodes)
```

```
    if node_ini is not None:
```

```
        assert node_ini <= n_nodes, f"El nodo definido es mayor a los del grafo. Deberia de ser menor a {n_nodes}."
```

```
        n_nodes = node_ini
```

```

n = n_nodes + 1

# Remove nan borders inside graph

edges = [edge for edge in graph.edges().data() if ~np.isnan(edge[2]['weight'])]

# Initialize distances of nodes and predecessors

# https://codingdeekshi.com/initialize-an-array-in-python/

distance= [distance_ini]*n

distance[n_nodes] = 0

predecessors = [-1]*n

```

Compilación de código con C

Posteriormente se hizo una transformación a código C con la librería de Cython.

Se compilaron tanto el código inicial como el código perfilado.

```

%%file bf_cython.pyx

import numpy as np

def bf_negative_cycle_p(graph, node_ini=None, distance_ini=np.inf):

    if node_ini is None:
        n_nodes = len(graph.nodes())
    else:
        assert node_ini <= len(graph.nodes), f"El nodo definido es mayor a los del grafo. Deberia de ser menor a {len(graph.nodes)}."
        n_nodes = node_ini

    n = len(graph.nodes()) + 1

    # Remove nan borders inside graph

```



```

edges = []

for edge in graph.edges().data():
    if ~np.isnan(edge[2]['weight']):
        edges.append(edge)

# Initialize distances of nodes and predecessors
distance= np.ones(n) * distance_ini # Starting distances with infinite values
predecessors = np.ones(n) * -1 # Starting predecessors with -1 values

%%file bf_cython2.pyx
import numpy as np
def bf_negative_cycle_cc(graph, node_ini=None, unsigned int distance_ini=np.inf):

    cdef unsigned int i,n,n_nodes

    n_nodes = len(graph.nodes)

    if node_ini is not None:
        assert node_ini <= n_nodes, f"El nodo definido es mayor a los del grafo. Deberia de ser menor a {n_nodes}."
        n_nodes = node_ini

    n = n_nodes + 1

    # Remove nan borders inside graph
    edges = [edge for edge in graph.edges().data() if ~np.isnan(edge[2]['weight'])]

    # Initialize distances of nodes and predecessors
    # https://codingdeekshi.com/initialize-an-array-in-python/
    distance= [distance_ini ]*n

```

```
distance[n_nodes] = 0
```

```
predecessors = [-1]*n
```

Aunque la herramienta de `line_profiler` sirvió para indicar la sección del código factible de eficientar y se realizaron las adecuaciones pertinentes, también la salida html de **Cython**, sirvió para hacer una segunda revisión y mejorar las líneas que la herramienta colocaba en color amarillo, las cuales se mostraron en el código anterior.

En la siguiente sección se muestran los resultados obtenidos puntualmente por estas herramientas usadas en el proyecto.

Resultados

Se logró implementar el método de Bellman Ford, el cuál fue mejorado desde la primera programación del método, gracias a los conceptos vistos en la materia de optimización, ya que de manera práctica quedaron más claros al aplicarse en un tema real.

La parte que se trabajó referente a hacer más eficiente el código fue fundamental para mejorar la programación realizada, derivado de los resultados del perfilamiento, que ayudó a enfocar las partes de código a mejorar, y donde la compilación con C redujo marginalmente el tiempo de ejecución, como se observará a continuación. Para ello se usó una máquina de AWS con las siguientes características:

```
%%bash
lscpu
```

```
Architecture:           x86_64
CPU op-mode(s):         32-bit, 64-bit
Byte Order:             Little Endian
Address sizes:          46 bits physical, 48 bits virtual
CPU(s):                 2
On-line CPU(s) list:    0,1
Thread(s) per core:     2
Core(s) per socket:     1
Socket(s):              1
NUMA node(s):           1
Vendor ID:              GenuineIntel
CPU family:             6
Model:                 85
Model name:             Intel(R) Xeon(R) Platinum 8259CL CPU @ 2.50GHz
Stepping:               7
CPU MHz:                3100.063
BogoMIPS:               4999.99
Hypervisor vendor:      KVM
Virtualization type:    full
L1d cache:              32 KiB
L1i cache:              32 KiB
L2 cache:               1 MiB
L3 cache:               35.8 MiB
NUMA node0 CPU(s):      0,1
```

```
%%bash
sudo lshw -C memory
```

```
*-memory
  description: System memory
  physical id: 0
  size: 15GiB
```

Perfilamiento

A continuación se muestra una tabla comparativa de las herramientas aplicadas en el método de *Bellman Ford*:

Herramienta usada en la comparación	Código antes de Eficientar	Código Eficientado
Time	Tomo 2.65 segundos en correr	Tomo 2.09 segundos en correr
Timeit	2.67 s \pm 13.6 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)	2.04 s \pm 182 ms per loop (mean \pm std. dev. of 7 runs, 1 loop each)
cProfile	29979 function calls (29977 primitive calls) in 1.371 seconds	20063 function calls (20062 primitive calls) in 1.089 seconds
line_profiler	Timer unit: 1e-06 s Total time: 4.06207 s	Timer unit: 1e-06 s Total time: 4.12395 s
memory_profiler	memory_usage 120.125	memory_usage 120.1640625
perf	<p>2.52719 +- 0.00924 seconds time elapsed (+- 0.37%)</p> <p>Estadísticas por core: 2.52596 +- 0.00867 seconds time elapsed (+- 0.34%)</p> <p>Estadísticas generales:</p> <ul style="list-style-type: none"> - 2575.60 msec task-clock # 1.019 CPUs utilized (+- 0.33%) - 27 context-switches # 0.010 K/sec (+- 2.43%) - 0 cpu-migrations # 0.000 K/sec (+- 58.49%) - 26093 page-faults # 0.010 M/sec (+- 0.04%) 	<p>2.22617 +- 0.00621 seconds time elapsed (+- 0.28%)</p> <p>Estadísticas por core: 2.21754 +- 0.00313 seconds time elapsed (+- 0.14%)</p> <p>Estadísticas generales:</p> <ul style="list-style-type: none"> - 2268.51 msec task-clock # 1.022 CPUs utilized (+- 0.25%) - 27 context-switches # 0.012 K/sec (+- 2.33%) - 0 cpu-migrations # 0.000 K/sec (+- 35.04%) - 26132 page-faults # 0.012 M/sec (+- 0.15%)

Se puede observar que el tiempo de proceso disminuyó marginalmente con las acciones realizadas, sin embargo aumentó el uso de memoria, que era de esperarse ya que se sacrifica un recurso por otro.

Compilación en C

A continuación se muestra una tabla de comparaciones del método *Bellman Ford*, antes y después de la compilación con C:

Herramienta usada en la comparación	Código antes de Eficientar	Código Eficientado
Cython	Bellman Ford tomó 0.8361616134643555 segundos	Bellman Ford tomó 0.527334451675415 segundos

Se puede observar que disminuyó el tiempo de procesamiento.

Aplicación del método

El método implementado consta de 5 parámetros para modificar la entrada de datos a evaluar:

```
def bf_negative_cycle_cc(graph, node_ini=None, distance_ini=np.inf):
```

Donde:

- **graph**: Corresponde al grafo de entrada a evaluar
- **node_ini**: Por default se toma el último nodo para ejecutar los ciclos de comparación, pero puede cambiarse pasando el índice del nodo.
- **distance_ini**: Por default es infinito, pero se usó para evaluar los parámetros a optimizar.

Adicionalmente se tienen funciones para manipular los datos de entrada y su transformación:

get_data: Usado para carga de datos, recibe como parámetro la ruta del archivo con la información de cryptomonedas (que para la implementación es un histórico de algunos días del mes de mayo 2022), la fecha a evaluar, el tamaño de datos a trabajar, y el precio que se va a tomar para la evaluación.

exchange_rate_matrix: se usa para variar la fluctuación en un 5% el valor de las cryptomonedas.

log_transformed_rep: el método necesita trabajar con cifras negativas para su convergencia, por ello se realiza esta transformación de datos.

create_grap: crea el grafo, y se encarga de crear el grafo, unificando las funciones anteriormente mencionadas.

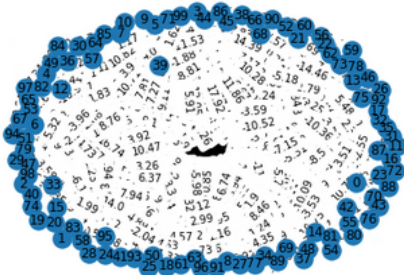
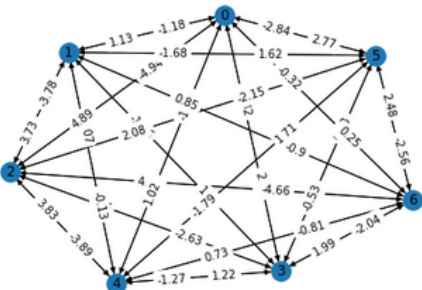
show_graph: pinta el grafo.

Casos de uso

Para probar la herramienta se trabajaron 2 casos de uso:

1. Toma al azar de criptomonedas.
2. Analizar un grupo de criptomonedas de interés.

En el primer caso se tomó un subconjunto de 100 monedas, y en el segundo caso se eligieron las siguiente lista de criptomonedas: ['LTC','NEO','XMR','EOS','BCH','DASH','ETC']

	Caso 1	Caso 2																																										
Datos	<p>(100, 2)</p> <table> <thead> <tr> <th></th><th>Símbolo</th><th>Precio</th></tr> </thead> <tbody> <tr> <td>0</td><td>REAPER</td><td>0.000903</td></tr> <tr> <td>1</td><td>SQUA</td><td>5.031241</td></tr> <tr> <td>2</td><td>DHR</td><td>0.003381</td></tr> <tr> <td>3</td><td>GSAIL</td><td>0.119018</td></tr> <tr> <td>4</td><td>vBUSD</td><td>0.021519</td></tr> </tbody> </table>		Símbolo	Precio	0	REAPER	0.000903	1	SQUA	5.031241	2	DHR	0.003381	3	GSAIL	0.119018	4	vBUSD	0.021519	<table> <thead> <tr> <th></th><th>Símbolo</th><th>Precio</th></tr> </thead> <tbody> <tr> <td>0</td><td>BCH</td><td>263.117533</td></tr> <tr> <td>1</td><td>DASH</td><td>82.388514</td></tr> <tr> <td>2</td><td>EOS</td><td>1.930541</td></tr> <tr> <td>3</td><td>ETC</td><td>26.157788</td></tr> <tr> <td>4</td><td>LTC</td><td>92.195750</td></tr> <tr> <td>5</td><td>NEO</td><td>16.032580</td></tr> <tr> <td>6</td><td>XMR</td><td>198.262798</td></tr> </tbody> </table>		Símbolo	Precio	0	BCH	263.117533	1	DASH	82.388514	2	EOS	1.930541	3	ETC	26.157788	4	LTC	92.195750	5	NEO	16.032580	6	XMR	198.262798
	Símbolo	Precio																																										
0	REAPER	0.000903																																										
1	SQUA	5.031241																																										
2	DHR	0.003381																																										
3	GSAIL	0.119018																																										
4	vBUSD	0.021519																																										
	Símbolo	Precio																																										
0	BCH	263.117533																																										
1	DASH	82.388514																																										
2	EOS	1.930541																																										
3	ETC	26.157788																																										
4	LTC	92.195750																																										
5	NEO	16.032580																																										
6	XMR	198.262798																																										
Grafo																																												
Lista solución	[99, 98, 99]	[5, 6, 5]																																										

Resultado	Símbolo Precio			Símbolo Precio		
	99	DTH	0.005611	5	NEO	16.032580
	98	DEL	0.065131	6	XMR	198.262798
	99	DTH	0.005611	5	NEO	16.032580

Estos casos se ejecutaron al mismo tiempo usando un pipeline en infraestructura de AWS, y usando las herramientas de **Kale** y **Kubeflow**.

Getting Started <https://ec2-35-175-127-139.compute-1.amazonaws.com:30001/myurl/lab?>

File Edit View Run Kernel Tabs Settings Help

Pipeline Name
proyecto-final

Pipeline Description
Proyecto final opt2

Run

HP Tuning with Katib ☐

SET UP KATIB JOB

Advanced Settings

Docker image
joelitam2021/pkg_proy_final:0.1

Validating notebook... ☒

Compiling notebook... [Done](#) ☒

Uploading pipeline... [Done](#) ☒

Running pipeline... [View](#) ☒

COMPILE AND RUN

root@jupyterlab-opt-0-1-host x pipeline_bellman_ford.ipynb x

grafo dirigido ponderado, usando el método de Bellman Ford.

Importando librerías

```
[11]: from opt2.preprocessing import get_data, data_to_graph, create_grap, show_graph
      from opt2.bf_cython2 import bf_negative_cycle_cc
      import numpy as np
      import networkx as nx
      import matplotlib.pyplot as plt
      import time
      import pandas as pd
      from datetime import datetime
```

Carga de datos

Caso 1: Datos al azar

step: caso1_carga

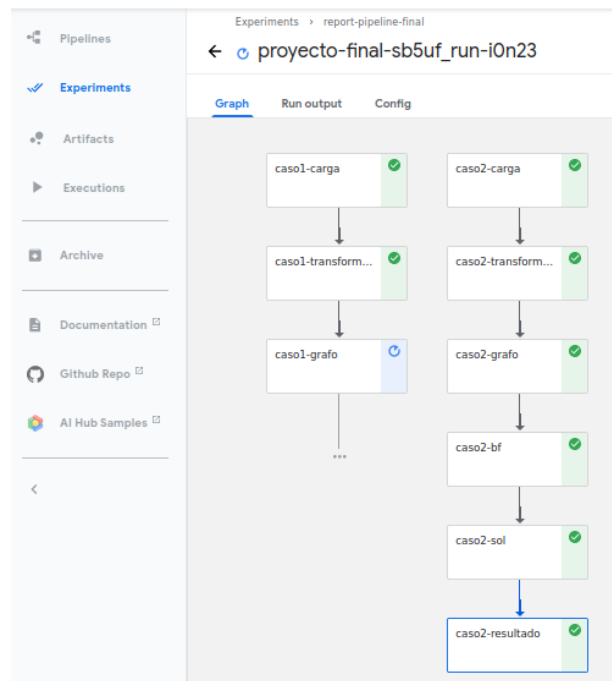
```
[22]: df1 = get_data('../data/historical_data.csv', '2022-05-01', 100, 'Open')
      print(df1.shape)
      df1.head()
```

```
(100, 2)
[22]:
```

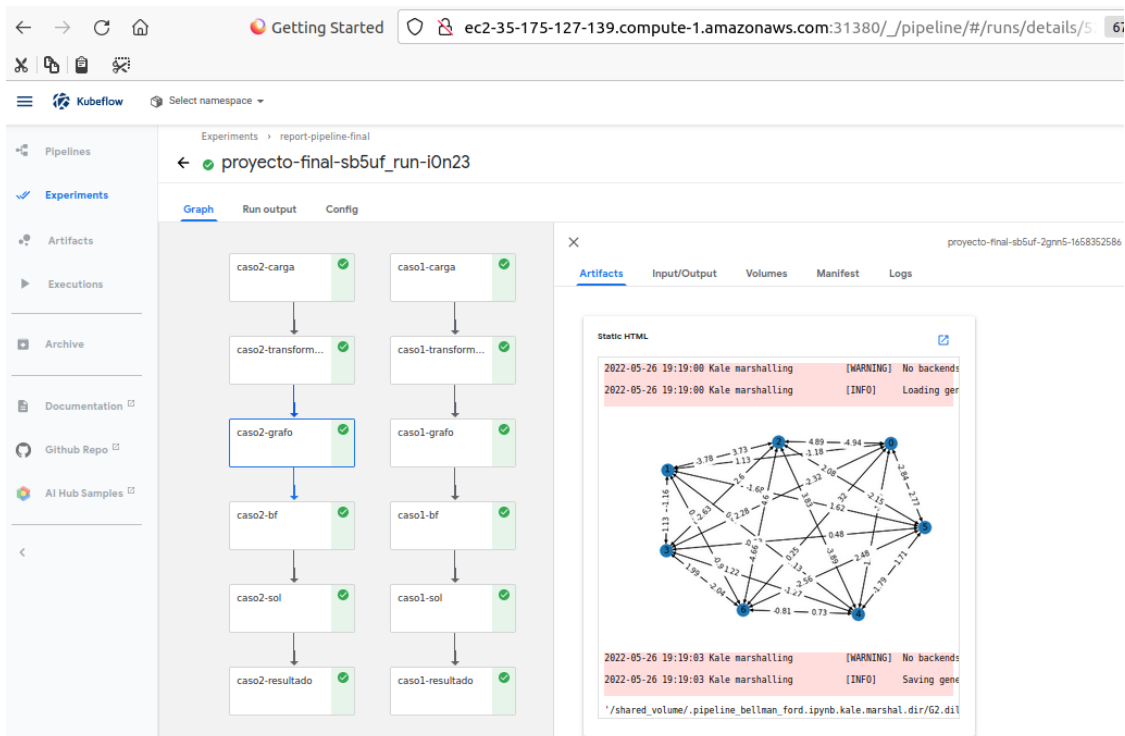
	Símbolo	Precio
0	REAPER	0.000903

1 0 Python 3 (ipykernel) | Idle Saving completed Mode: Command

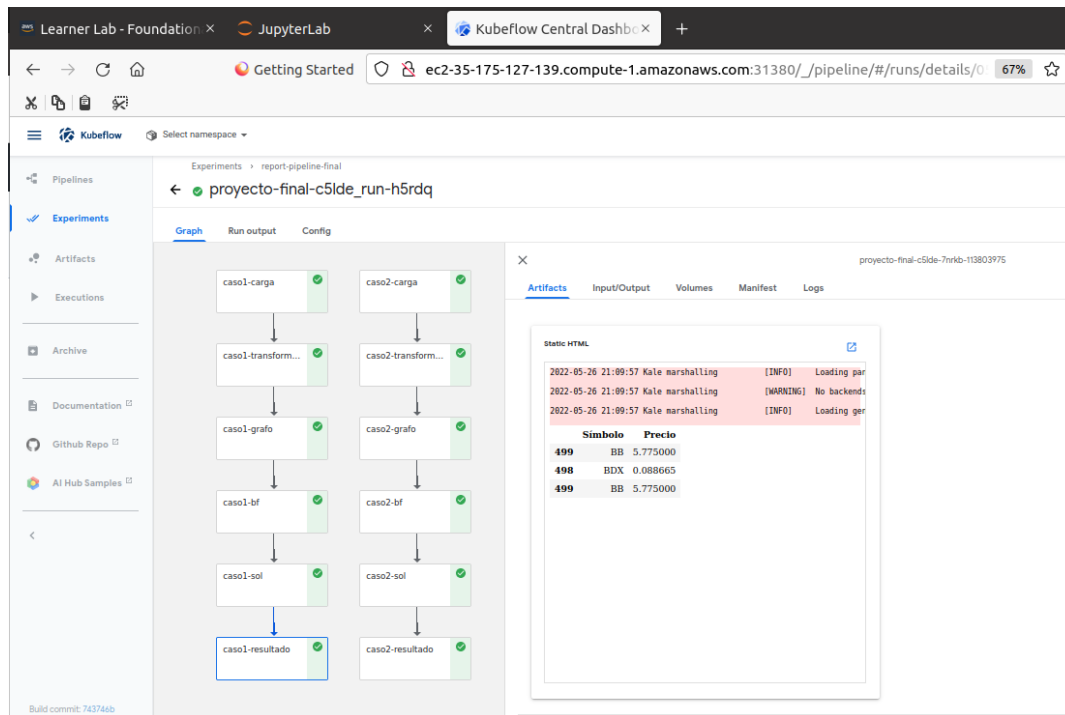
como era de esperarse el grafo con menor número de nodos término en menor tiempo.



Finalmente se muestra la corrida del pipeline completo, usando 100 nodos, en el caso que toma criptomonedas al azar.



Y el siguiente es una corrida con 500 criptomonedas tomadas al azar, donde se omitió el pintado del grafo, porque su consumo de tiempo, y ya no es tan viable observado, porque la cantidad de enlaces entre nodos no da mucha información.



The screenshot shows the Kubeflow Central Dashboard interface. The main view displays a pipeline execution for 'proyecto-final-c5Ide_run-h5rdq'. The pipeline graph consists of two parallel paths of tasks: 'caso1-carga' to 'caso1-resultado' and 'caso2-carga' to 'caso2-resultado'. The 'Artifacts' tab is open, showing a table of cryptocurrency data.

Simbolo	Precio
499	BB 5.775000
498	BDX 0.088665
499	BB 5.775000

Conclusiones

Durante el desarrollo del proyecto, se aplicaron diversos conceptos y temas vistos en clase, los cuales se reforzaron de forma práctica en cada etapa del proyecto, desde la definición de un problema de optimización, los métodos disponibles para la solución de los mismos, perfilamiento y eficientar el código.

De la mano también se puso en práctica el uso nuevas herramientas que se podrán aplicar en nuestra carrera como data scientist para solucionar diversos problemas de optimización (*AWS, Kale, docker, python*, etc.). En particular nos gustaron las herramientas de *Kale* y *Kubeflow* en la implementación de pipelines, que fue una aplicación que se realizó de una manera muy sencilla comparado con otras herramientas como son *luigi* o *airflow*, por mencionar algunos.

Todo este bagaje de herramientas permitió encontrar una solución en el campo de trading sobre criptomonedas, donde una vez identificado la clasificación de problema que implicaba, el encontrar los diferentes métodos que están disponibles para la solución del costo mínimo fue más sencillo, y donde se logró seleccionar un método adecuado, como fue el algoritmo *Bellman Ford*.

La evaluación del método sobre datos reales nos emocionó, ya que no se sabía si el algoritmo encontraría la solución al problema, ya que era la prueba final de todo el trabajo desarrollado.

Algo que no nos agrado mucho, es que no se pudo eficientar mucho la implementación realizada desde la primera versión, hasta el resultado obtenido después de aplicar el perfilamiento y la compilación con C. Como resultado de la eficiencia del código se pasó de 0.836 a 0.527 segundos, lo que muestra una pequeña disminución en el tiempo de procesamiento, sin embargo analizando los resultados no fue tan grande la diferencia debido a que ya era rápida la primera implementación, ya que cuando se evaluó el método ingresando 500 criptomonedas tardó alrededor de 4 minutos, lo cuál se nos hizo aceptable, por el procesamiento que implicaba.

Sin embargo, nos pudimos percatar que con la disminución de tiempo que se logró hacer, es decir la disminución de procesamiento, se tuvo un aumento en el uso de memoria, que era de esperarse ya que se sacrifica un recurso por otro.

Para finalizar, aunque no se logró disminuir un poco más la ejecución del código como se hubiera deseado, creemos que se aprendió bastante, ya que la división del proyecto en entregas parciales permitió estudiar por separado los conceptos y permitió concentrarnos en cada etapa, que de alguna manera se iba acumulando, pero se reforzaban los conceptos que se repasaban nuevamente.

Referencias

- ¹ [¿Qué es el 'trading'? . BBVA. INVERSIONES Act. 17 feb 2021.](#)
- ² [¿Qué es y para qué sirve la tecnología blockchain?](#)
- ³ [¿Qué es el Trading de Criptomonedas? . Plus500.](#)
- ⁴ [CoinMarketCap](#)
- ⁵ [Algoritmo Bellman Ford](#)
- ⁶ [Bellman Ford](#)
- ⁷ [Libro de optimización, subtemas 5.2 y 5.3.](#)
- ⁸ [Tema 2. Algoritmos de encaminamiento](#)
- ⁹ [Problema del camino más corto](#)
- ¹⁰ [Optimización en Redes](#)