Implementación y Optimización del método metaheurístico Hill Climbing para resolver el Traveling Salesman Problem (TSP)

Miguel López Cruz Carlos Román López Sierra José Luis R. Zárate Cortés

Primavera 2021

Índice

Resumen

Implementar el método de Hill climbing para resolver TSP y obtener una solución suficientemente cercana a la óptima global para un conjunto aproximado de 150 ciudades en un periodo de tiempo razonable utilizando eficiencia de código y aceleración de hardware.

1. Introducción

En optimización matemática y ciencias de la computación, los métodos heurísticos y metaheurísticos se desarrollaron para resolver problemas del tipo large scale (PL´s o de optimización combinatoria) que son muy grandes y complicados como para resolverlos por medio de algoritmos exactos. Típicamente se ajustan a problemas específicos.

En nuestro caso, seleccionamos Hill Climbing, método de optimización que pertenece al grupo de búsqueda local. Encuentra la solución óptima en el caso de problemas convexos (como el algoritmo símplex que resuelve problemas de optimización convexa mediante hill climbing) y soluciones óptimas locales (no necesariamente son soluciones óptimas globales) entre todas las posibles soluciones del espacio de búsqueda. Por lo tanto, es importante remarcar que no existe garantía acerca de la calidad de la solución que se obtiene.

Existe un problema importante conocido como el Traveling Salesman Problem (TSP), el cual plantea la siguiente interrogante:

"Dado un conjunto de ciudades y la distancia que existe entre cualesquiera dos de ellas... ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y retorna a la ciudad origen?"

Responder a estar interrogante es equivalente a calcular todas las rutas posibles (permutaciones) y determinar la de menor distancia (algoritmo de fuerza bruta), lo cual implica un tiempo de ejecución que cae en un factor polinomial el orden $O(n!)$ (el factorial del número de ciudades), lo cual es impráctico tan sólo para 20 ciudades.

El método Hill Climbing se puede aplicar para resolver el TSP y obtener una solución óptima para un conjunto pequeño de ciudades de forma muy rápida. Conforme se incrementa el tamaño del conjunto, este método requiere mayor tiempo para explorar el espacio de posibles soluciones de forma suficiente para encontrar una solución óptima global. Por lo que, nuestro objetivo es explorar diversas opciones que permiten una implementación más eficiente del método utilizando alguna de las siguientes estrategias:

- Modificaciones en la técnica, i.e. random restart
- Integrar eficiencia de código como cómputo en paralelo o multiprocessing
- Utilizar estrategias de solución como el uso de Kubernetes (minikube) es una forma de tener consistencia en la generación de resultados vía contenedores de Docker.

2. Fundamento Teórico

2.1. Optimización Combinatoria

Optimización Combinatoria es un subcampo de Optimización Matemática relacionado con Investigación de Operaciones, teoría de algoritmos y complejidad computacional con importantes aplicaciones en AI, ML, ingeniería de *software*, matemáticas aplicadas, etc.

Consiste en encontrar un elemento óptimo dentro de un conjunto finito de objetos. En muchos problemas, ejecutar una búsqueda exhaustiva no es operable, por lo que opera en el dominio

de aquellos problemas de optimización en los cuales se tiene un conjunto discreto (contable) de soluciones factibles o es posible discretizarse o reducirse con el objetivo de encontrar la mejor solución. Ejemplos típicos son:

- *Traveling Salesman Problem "TSP"*, problema del vendedor viajero
- *Minimum Spanning Tree Problem*, problema del mínimo árbol de expansión

Lo anterior encuentra diversas aplicaciones en la vida diaria como en la elaboración de rutas aéreas y de transporte de mercancías, diseño de redes de distribución de agua, etc.

Los problemas en optimización combinatoria pueden interpretarse como búsqueda de el mejor elemento en un conjunto discreto de elementos; por lo que, al menos en teoría, cualquier algoritmo de búsqueda o **meta-heurístico** puede aplicarse para resolver esta clase de problemas.

Debemos precisar que algunos problemas de optimización combinatoria, tal como el TSP, son *NP-complete*

2.1.1. Definición matemática

Sea I un conjunto de instancias y dada una instancia $x \in I, f(x)$ es el conjunto finito de soluciones factibles. Dada una instancia x y una solución factible y de x, se tiene que $m(x, y)$ denota la métrica de y, usualmente en $\mathbf{R}^+$

Sea g como la función objetivo a optimizar (minimizar o maximizar), el objetivo es encontrar para alguna instancia x una *solución óptima*, esto es, una solución factible y con

$$m(x, y) = g\{m(x, y') \mid y' \in f(x)\}.$$

2.1.2. NP-complete y NP-hard

En teoría de la complejidad computacional, se tiene un problema **NP-complete** cuando:

- Se puede resolver utilizando un algoritmo de búsqueda por fuerza bruta y la solución puede validarse muy rápido.
- El problema puede utilizarse para simular otros problemas cuya solubilidad es similar.

Por otro lado, problema es **NP-hard** si un algoritmo para resolverlo puede traducirse en uno para resolver cualquier problema NP (tiempo polinomial no determinista). Por lo tanto, NP-hard significa al menos tan difícil como cualquier problema NP, aunque, de hecho, podría ser más difícil.

NP-complete o *non-deterministic polynomial-time complete*, se refiere a **non-deterministic Turing machines** (matemáticamente formaliza la idea del algoritmo de búsqueda por fuerza bruta). **Polynomial time** se refiere a la cantidad de tiempo requerida que se considera rápida"para que un algoritmo determinístico compruebe una solución específica o, para una non-deterministic Turing Machine desarrollar una búsqueda completa. **Complete** se refiere a la capacidad de simular todo lo que esté en la misma clase de complejidad.

Si bien se puede verificar rápidamente una solución a un problema del tipo *NP-complete*, no se conoce forma alguna para encontrar una solución rápidamente. Esto es, el tiempo requerido para resolver el problema utilizando cualquier algoritmo conocido, incrementa rápidamente conforme incrementa el tamaño del problema. Como consecuencia, determinar si es o no posible resolver tales problemas de forma rápida, denominado **P vs NP problem**, es uno de los problemas fundamentales no resueltos en ciencias de la computación.

2.1.3. Time complexity

En ciencias de la computación, *time complexity* es la complejidad computacional que describe la cantidad de tiempo máquina que le toma ejecutar un algoritmo. Comunmente se estima contando el número de operaciones elementales realizadas por el algoritmo, asumiendo que cada operación le toma un tiempo fijo, por lo que puede considerarse como un factor constante.

Existen varios tipos comunes de *time complexity* tales como timpo constante, tiempo logarítmico, lineal, polinomial, exponencial, factorial, etc. En nuestro caso de interés, *superpolynomial time*, se dice que un algoritmo tiene un tiempo superpolinomial si $T(n)$ es no acotado superiormente por algún polinomio. Por ejemplo, un algoritmo que ejecuta 2^n pasos para un *input* de tamaño n, requiere de un tiempo super-polinómico, específicamente tiempo exponencial.

2.1.4. Resolver problemas NP-complete

Los algoritmos conocidos actualmente para problemas *NP-complete*, requieren de un tiempo *superpolynomial* en relación al tamaño del *input*. Algunas técnicas utilizadas para resolver problemas computacionales y que usualmente incrementan sustancialmente la velocidad de los algoritmos son:

- Aproximación. Búsqueda de una solución que representa un factor de una solución óptima en lugar de buscar esta última
- Aleatorización. Uso de aleatorización para lograr tiempos de ejecución promedio más rápidos.
- Restricción. Restringir la estructura del *input* hace que se logren algoritmos más rápidos y eficientes.
- Parametrización. Usualmente existen algoritmos que son más rápidos si se fijan algunos parámetros.
- Heurística. Los algoritmos heurísticos en la mayoría de los casos "trabajan bien", pero no existe prueba de que siempre sean rápidos o que generen un buen resultado.

2.2. Algoritmos Meta-heurísticos

En ciencias de la computación y optimización matemática, una **meta-heurística** es un procedimiento (heurística) de alto-nivel diseñado para encontrar, generar o seleccionar una heurística (algoritmo de búsqueda parcial). Ésto provee una solución "suficientemente buena"para un problema de optimización, especialmente en los casos de información incompleta/imperfecta o capacidad computacional limitada. Realizan un muestreo de un subconjunto de soluciones de un conjunto que es muy grande para poder enumerarse o explorarse completamente, utilizando pocas suposiciones acerca del problema de optimización que se resuelve, por lo que son utilizados en una variedad de problemas.

En comparación con algoritmos de optimización y métodos iterativos, los algoritmos meta-heurísticos no garantizan que se pueda encontrar una solución óptima global en cierta clase de problemas. En el caso de optimización combinatoria, al buscar en un conjunto grande de soluciones factibles, estos algoritmos usualmente encuentran buenas soluciones con una menor demanda computacional que los algoritmos de optimización, métodos iterativos o simples heurísticas.

2.2.1. Propiedades

Las siguientes son propiedades que caracterizan a la mayoría de los algoritmos meta-heurísticos:

- Son estrategias que guían el proceso de búsqueda

- El objetivo es explorar de forma eficiente el espacio de búsqueda para encontrar soluciones cercanas a la óptima.
- Las técnicas que constituyen los algoritmos meta-heurísticos van desde un procedimiento simple de búsqueda local hasta procesos complejos de aprendizaje.
- Son aproximaciones usualmente no determinísticas.

2.2.2. Clasificación

Los métodos heurísticos y meta-heurísticas (estrategias para mejorar o diseñar métodos heurísticos) encuentran una buena solución factible que al menos está razonablemente cerca de ser óptima, también pueden reportar que no se encontró tales soluciones. Sin embargo son métodos que no dan una garantía acerca de la calidad de la solución que se obtiene.

Los métodos heurísticos se desarrollaron principalmente para el manejo de problemas *large scale* sean PL's o de optimización combinatoria y típicamente se han ajustado a problemas específicos en lugar de una variedad de aplicaciones.

Existe una amplia variedad de meta-heurísticas y de sus propiedades, en función de lo que son clasificadas. Para nuestro caso de estudio particular, nos enfocaremos en **búsqueda local vs búsqueda global**.

Los algoritmos de búsqueda local simple pueden ser una mejora en la estrategia de búsqueda. Un algoritmo muy bien conocido de búsqueda local es *hill climbing method*, utilizado para encontrar óptimos locales pero el cual no garantiza encontrar soluciones óptimas globales.

Existe otra clasificación de nuestro interés que es *parallel metaheurístics*, técnica que utiliza *parallel programming* para correr múltiples búsquedas en paralelo; pueden variar de simples esquemas distribuidos hasta ejecuciones con búsqueda concurrente que interactúan para mejorar la solución.

Otras clasificaciones son *single-solution vs populated-based; hybridization and memetic algortithms; nature inspired and metaphor-based metaheuristics; ancient inspired metaheurístics.*

3. Búsqueda local en problemas de optimización

Los algoritmos de búsqueda local operan buscando desde un estado inicial a estados vecinos, sin realizar un seguimiento de las rutas ni del conjunto de estados que se han alcanzado. Eso significa que no son sistemáticos; es posible que nunca exploren una parte del espacio de búsqueda donde un la solución reside realmente. Sin embargo, tienen dos ventajas clave que son:

1. Utilizan muy poca memoria
2. A menudo pueden encontrar soluciones razonables en espacios de estados grandes o infinitos para los cuales los algoritmos sistemáticos no son adecuados.

Para comprender la búsqueda local, considere los estados de un problema presentado en un espacio de estados, como se muestra en la Figura 1. Cada punto (estado) tiene una elevación, definida por el valor de la función objetivo. Entonces el objetivo es encontrar el pico más alto, un máximo global.

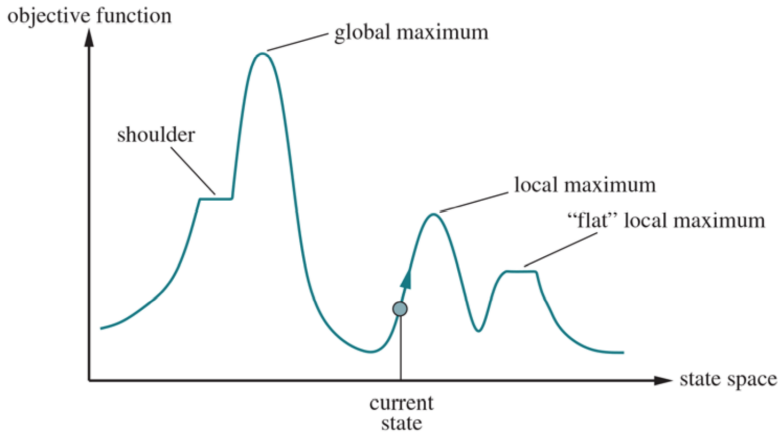

Figura 1: Espacio de estados para la función objetivo

4. Hill Climbing

4.1. Definición del algoritmo

El algoritmo de Hill Climbing realiza un seguimiento de un estado actual y en cada iteración se mueve al estado vecino con el valor más alto, es decir, se dirige en la dirección que proporciona el ascenso más empinado. Termina cuando alcanza un "Pico" donde ningún vecino tiene un valor más alto. Hill Climbing no mira hacia el futuro más allá de los vecinos inmediatos del estado actual. Esto se asemeja a tratar de encontrar la cima del Monte Everest en una espesa niebla mientras sufre de amnesia.

Por otro lado, otra forma de utilizar Hill Climbing consiste en utilizar el negativo de una función de coste heurística como función objetivo; que ascenderá localmente al estado con menor distancia heurística a la meta.

A este tipo de algoritmos a veces se les llama búsqueda local greedy porque toma un buen estado vecino sin pensar en los estados futuros a dónde moverse. Hill Climbing puede progresar rápidamente hacia una solución porque, por lo general, es bastante fácil mejorar un mal estado. Desafortunadamente, Hill Climbing tiene problemas en áreas donde espacio de posibles soluciones puede atascarse, estos son:

- **Máximo Local:** Es un pico que es más alto que los estados vecinos pero menor que el máximo global. Los algoritmos Hill Climbing que alcanzan las proximidades de un máximo local tomaran estados vecinos que lo lleven en esta dirección, pero luego de llegar a este punto les será imposible salir de él.
- **Cordilleras y corredores:** Este tipo de puntos en el espacio de soluciones resulta en una secuencia de máximos locales que es muy difícil de navegar para los algoritmos greedy, esto por el gran número de soluciones vecinas que te llevan a puntos no óptimos de la función objetivo.
- **Mesetas:** Esta se encuentra cuando el espacio de búsqueda es plano, o suficientemente plano de manera tal que el valor retornado por la función objetivo es indistinguible de los valores retornados por regiones cercanas debido a la precisión utilizada por la máquina para representar su valor. En estos casos el algoritmo puede que no sea capaz de determinar en qué dirección debe continuar y puede tomar un camino que nunca conlleve a una mejoría de la solución.

4.2. Problema de Optimización, Definición Matemática Formal

El método de optimización *Hill Climbing* intenta maximizar o miminizar una función objetivo $f(\mathbf{x})$, donde $\mathbf{x}$ es un vector de valores continuos o discretos. En cada iteración, este método modifica algún componente en $\mathbf{x}$ y determina si tal modificación mejora el valor de $f(\mathbf{x})$, aceptando cualquier mejora y continua el proceso hasta no encontrar un cambio que que mejore el valor de $f(\mathbf{x})$, por lo que $\mathbf{x}$ se conoce como óptimo local.

4.3. Aplicación del algoritmo *Hill Climbing* para resolver el *Traveling Salesman Problem*

Este problema se enuncia de la siguiente forma:

"Dado un conjunto de ciudades y la distancia que existe entre cada par de ellas... ¿cuál es la ruta más corta posible que visita cada ciudad exactamente una vez y retornar a la ciudad orígen?"

El TSP es un problema de optimización **NP-hard** en optimización combinatoria, importante para la teoría de ciencias de la computación e investigación de operaciones. Esta clase de problemas son al menos tan complejos como los problemas más complejos de la clase **NP**, los cuales (tal como mencionamos anteriormente) se refieren a problemas que son solubles en tiempo polinómico por una máquina de Turing no-determinista. De esta forma, es posible que el peor caso en tiempo de ejecución para cualquier algoritmo que resuelva el TSP incremente de forma superpolinomial (no más allá de exponencial) conforme incrementa el número de ciudades.

Tradicionalmente, las líneas de ataque a los problemas **NP-hard** son:

- Formular algoritmos exactos, los cuales son razonablemente rápidos para problemas de *small scale* (pocas ciudades).
- Formular algoritmos heurísticos, los cuales generan una solución aproximada en un tiempo razonable.
- Encontrar casos especiales ("subproblemas") para los cuales es posible encontrar heurísticas mejores o exactas.

En nuestro caso, realizaremos la implementación del método metaheurístico *Hill Climbing*.

4.4. Variantes de Hill climbing

Se han inventado muchas variantes de este algoritmo. Por ejemplo *Hill climbing* estocástica elige al azar entre los movimientos cuesta arriba; la probabilidad de selección puede variar con la inclinación del movimiento cuesta arriba. Esto suele converger más lentamente que el *steepest ascent*, pero en algunos puntos de espacio de soluciones, encuentra mejores soluciones. *First-choice hill climbing* implementa el método estocástico generando sucesores aleatoriamente hasta que se genera uno que es mejor que el estado actual. Ésta es una buena estrategia cuando un estado tiene muchos posibles estados siguientes a donde moverse.

También existe el algoritmo *random-restart Hill climbing* está construido sobre la base de *hill climbing*. Esto se realiza, iterativamente, cada vez con una condición inicial aleatoria . Una primera mejor solución es guardada; si una nueva corrida del produce una mejor que el estado guardado, lo reemplaza. *Random-restart* es un método sorprendentemente efectivo en muchos casos. De esto se deriva que con frecuencia es mejor gastar tiempo de CPU explorando el espacio, que cuidadosamente optimizar desde una condición inicial.

4.5. Programación

Lo siguiente es revisar el pseudocódigo de programación que encierra las ideas elementales de la implementación de nuestro algoritmo:

```
Pseudocode
algorithm Discrete Space Hill Climbing is
    currentNode := startNode  # Asignamos a currenNode una
    loop do
        L := NEIGHBORS(currentNode)
        nextEval := INF
        nextNode := NULL
        for all x in L do
            if EVAL(x) > nextEval then
                nextNode := x
                nextEval := EVAL(x)
        if nextEval  EVAL(currentNode) then
            // Return current node since no better neighbors exist
            return currentNode
        currentNode := nextNode
```

5. Implementación

5.1. Código

Antes de implementar random-restart nuestro algoritmo encontraba una solución rápida terminando rutas de 100 ciudades en menos de 30 segundos, pero esto lo efectuaba con poca exactitud. De la teoría conocemos que es un método con el problema de quedar atascado en zonas puntuales de espacio de soluciones, cómo mínimos locales y mesetas. Hecho que en la depuración encontramos, que después de cierto número de iteraciones repetía la misma ruta. Otra situación encontrada en la revisión fue que arrojaba soluciones hasta un 40 % mayor en la distancia recorrida por la ruta establecida cómo óptima por otros paquetes cómo OR-Tools, esto también derivado de los problemas inherentes al algoritmo. Habiéndonos dado cuenta de estas situaciones agregamos al método un random-restart, que le dio la capacidad de comenzar en un nuevo punto aleatorio para sacarlo del ciclo donde caía después de un cierto número de iteraciones y con este misma modificación acercarlo más a un valor menor con un mayor número de iteraciones en diferentes puntos del espacio solución.

Nuestro método se compone de cinco funciones incluyendo `best_solution` que efectúa el algoritmo *Hill climbing*. A continuación describimos las funciones que componen nuestro código:

- `distance_matrix`: Esta función calcula la matriz de distancias de los puntos que componen el *dataset* de las ciudades a visitar. Esta genera una matriz simétrica de tamaño nxn.
- `random_solution`: Genera soluciones aleatorias para el *dataset* propuesto.Esta función retorna una lista con el orden en el que se visitaran las ciudades.
- `calculate_distance`: Calcula la distancia euclidiana que se recorre dada una solución propuesta. La salida de esta función es un número real correspondiente a la distancia recorrida en dicha solución propuesta.

- `neighbors`: *Hill climbing* funciona en parte generando todas las soluciones vecinas a la solución actual. Una solución vecina es una solución que es solo ligeramente diferente de la solución actual, esto cuidando que cada ciudad se visite una sola vez. Crear soluciones ligeramente diferentes lo logramos intercambiando ciudades de lugar a través de ciclos `for`. Después con un conjunto de este tipo de soluciones, el algoritmo busca la que recorre la menor distancia, encontrado así el vecino más óptimo.

- `best_solution`: Primero, crea una solución aleatoria y calcula la longitud de su ruta. Después a partir de esta soluciones crea el conjunto de soluciones vecinas y con ello encuentra la mejor. A partir de ahí, siempre que el mejor vecino sea mejor que la solución actual se repite el mismo patrón con la solución actual cada vez que se actualiza con el mejor vecino. Para no quedar estancado en mínimos locales implementamos un random-restart, generando un nueva ruta aleatoria. Si esta nueva solución es mejor que la anterior, se remplaza. Esto se repite un número determinado de veces establecido en los parámetros de la función.

5.2. *Dataset*

El dataset elegido para resolver TSP se extrajo de la pagina National Traveling Salesman Problem de concord, el cual contiene un total de 4,663 ciudades canadienses cada una con su latitud y longitud.

5.3. Instancia

La instancia de AWS utilizada para el desarrollo de nuestras pruebas y experimentos tiene las siguientes características:

5.4. Ejecución

En los tiempos de ejecución (wall clock o elapsed time) desde el inicio de los statements hasta su finalización es exactamente los 2.58 segundos que coinciden con user en este caso.

Después ejecutando el comando `%timeit` evaluamos nuestro método, considerando promediar los tiempos de n=7 ejecuciones calculando su desviación estándar. Este proceso se ejecutó 11 veces, dando el siguiente resultado:

También observamos que el requerimiento de memoria de nuestro código fue de 11.8 MB, cómo se muestra a continuación

5.5. Paralelización

Numba nos permite, gracias a JIT, hacer compilación durante la ejecución. Entonces menos tiempo dedicado a la compilación inicial significa que el código se puede interpretar mucho más rápido. Esto además acompañado de la interpretación automática de los tipos de datos, esto nos permitiría una interpretación más rápida y eficaz del código.

Dada la naturaleza de Hill Climbing consideramos que al implementar una paralelización resultaría en un aceleramiento del proceso iterativo inherente al algoritmo. Ya que nuestro algoritmo busca encontrar la ruta más adecuada. Entonces con ayuda de *Numba* y su paralelización nos ayudaría a correr simultáneamente soluciones propuestas al algoritmo.

A continuación se muestran los resultados de la paralelización con y sin *Numba* en un ejercicio realizdo con un dataset de prueba que contiene 17 ciudades, ejecutando 100 reinicios del punto inicial y una tolerancia de $1e-7$.

```
In [1]: %%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):                          8
On-line CPU(s) list:             0-7
Thread(s) per core:              2
Core(s) per socket:              4
Socket(s):                       1
NUMA node(s):                    1
Vendor ID:                       GenuineIntel
CPU family:                      6
Model:                           85
Model name:                      Intel(R) Xeon(R) Platinum 8175M CPU @ 2.50GHz
Stepping:                        4
CPU MHz:                         3120.058
BogoMIPS:                        5000.00
Hypervisor vendor:               KVM
Virtualization type:             full
L1d cache:                       128 KiB
L1i cache:                       128 KiB
L2 cache:                        4 MiB
L3 cache:                        33 MiB
NUMA node0 CPU(s):               0-7
Vulnerability Itlb multihit:     KVM: Vulnerable
Vulnerability L1tf:              Mitigation; PTE Inversion
Vulnerability Mds:               Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
Vulnerability Meltdown:          Mitigation; PTI
Vulnerability Spec store bypass: Vulnerable
Vulnerability Spectre v1:        Mitigation; usercopy/swapgs barriers and __user pointer sanitization
Vulnerability Spectre v2:        Mitigation; Full generic retpoline, STIBP disabled, RSB filling
Vulnerability Srbds:             Not affected
Vulnerability Tsx async abort:   Vulnerable: Clear CPU buffers attempted, no microcode; SMT Host state unknown
Flags:                           fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush mmx fxsr s
se sse2 ss ht syscall nx pdpe1gb rdtscp lm constant_tsc rep_good nopl xtopology nonstop_tsc cpuid aperfmperf tsc_known_freq
pni pclmulqdq ssse3 fma cx16 pcid sse4_1 sse4_2 x2apic movbe popcnt tsc_deadline_timer aes xsave avx f16c rdrand hypervisor
lahf_lm abm 3dnowprefetch invpcid_single pti fsgsbase tsc_adjust bmi1 hle avx2 smep bmi2 erms invpcid rtm mpx avx512f avx51
2dq rdseed adx smap clflushopt clwb avx512cd avx512bw avx512vl xsaveopt xsavec xgetbv1 xsaves ida arat pku ospke
```

Figura 2: Caracteristísticas de la instancia utilizada.

```
In [42]: %timeit -n 7 -r 11 hc.best_solution(tsp_cities, 0 , 1e-9, 300)
         2.6 s ± 12.2 ms per loop (mean ± std. dev. of 11 runs, 7 loops each)
```

En primer lugar correremos la función `best_solution` sin *Numba*:

```
In [5]: %timeit -n 3 -r 7 best_solution(dat2, 0, 1e-7, 100)
        888 ms ± 19.2 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

Ahora mostramos la forma de correr la función $best_solutionconNumbayaconelcódigocompilado$:

```
In [8]: %timeit -n 3 -r 7 nb_best_solution(dat2, 0, 1e-7, 100)
        5.33 s ± 83.6 ms per loop (mean ± std. dev. of 7 runs, 3 loops each)
```

Comparando ambos resultados notamos que se multiplica el tiempo por un factor de 5 nuestro código con *Numba*.

A continuación de este ejercicio comparamos los resultados de la ruta óptima calculada por ambos algoritmos, corriendo tres ejemplo para cada caso, con los mismos parámetros establecidos previamente.

Rutas óptimas calculadas sin *Numba*:

```
In [11]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3337.113634493085
[0, 1, 5, 6, 12, 13, 11, 15, 9, 16, 14, 7, 2, 4, 10, 8, 3, 0]
0.9038689136505127

In [20]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3353.4536058912754
[0, 3, 2, 4, 9, 16, 14, 7, 10, 8, 15, 11, 13, 12, 6, 5, 1, 0]
5.282973051071167

In [22]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3266.456452336076
[0, 1, 5, 6, 12, 13, 11, 15, 8, 10, 9, 16, 14, 7, 2, 4, 3, 0]
0.8811595439910889
```

Rutas óptimas calculadas con *Numba*:

```
In [14]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3332.348304772912
[0, 1, 5, 6, 12, 13, 11, 15, 10, 8, 9, 14, 16, 7, 2, 4, 3, 0]
5.391537666320801

In [16]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3341.316828737772
[0, 3, 4, 2, 7, 9, 14, 16, 10, 8, 15, 11, 13, 12, 6, 5, 1, 0]
5.332757234573364

In [18]: print(distancia)
         print(ruta)
         print(tiempo_ejec)

3269.2653147263823
[0, 1, 5, 6, 12, 13, 11, 15, 8, 10, 9, 14, 16, 7, 2, 4, 3, 0]
5.427420616149902
```

Las mejoras que se hicieron a *Numba* aumentaron el tiempo de ejecución y la distancia mínima calculada se mantiene en el mismo rango de valores que sin *Numba*. Por tanto implementar nuestro algoritmo con *Dask*.

6. Resultados

A continuación mostramos comparaciones de resultados entre nuestra antigua y nueva implementación del paquete. Encontrando que en la implementación en paralelo se reducen a menos de la mitad el tiempo entre nuestra implementación paralelizada y no paralelizada.

Por otro parte la comparación de nuestro paquete con OR-Tools los resultados para 20 ciudades encontramos que nuestro paquete comienza hacer más grandes los resultados, alejándose del valor encontrado por el paquete de Google OR Tools. Cómo en el ejemplo con 50 ciudades, que se muestra más adelante, el resultado muestra una 13 % mayor con respecto a OR Tools y esta diferencia aumenta con el número de ciudades.

Primer ejemplo donde se muestran los tiempos con el paquete paralelizado y sin paralelizar:

Ejemplo 20 ciudades

Los primeros tres parámetros que se ingresan en cada función son idénticos equivalentes, mientras que el cuarto parámetro, que para `parallel_hc` es 100 y para `best_solution` es 800, estas son equivalentes. Ya que está instancia tiene, como se muestra en la descripción, ocho workers posibles, en cambio `best_solution` solo corre un worker a la vez.

```
[10]: %%time
      parallel_hc(tsp_cities, 0, 1e-1, 20)

      CPU times: user 3.39 ms, sys: 36 ms, total: 39.4 ms
      Wall time: 1.84 s
[10]: (3420.477836065084,
       [0, 3, 8, 10, 9, 4, 2, 7, 16, 14, 19, 18, 17, 15, 11, 13, 12, 6, 1, 5, 0],
       1.6029994487762451)

 [9]: %%time
      best_solution(tsp_cities, 0, 1e-1, 160)

      CPU times: user 4.24 s, sys: 3.99 ms, total: 4.25 s
      Wall time: 4.24 s
 [9]: (3493.4940349361523,
       [0, 3, 8, 17, 18, 19, 16, 14, 7, 9, 2, 4, 10, 15, 11, 13, 12, 6, 5, 1, 0],
       4.237102746963501)
```

Ejemplo 25 ciudades

```
[14]: tsp_cities_dos = dat1[0:25,:]

 [6]: %%time
      parallel_hc(tsp_cities_dos, 0, 1e-1, 20)

      CPU times: user 20.6 ms, sys: 12.3 ms, total: 32.9 ms
      Wall time: 5.45 s
 [6]: (3688.2628362386577,
```

Comparación de paquete antes y después de paralelizar para 25 ciudades:

```
 [7]: %%time
      best_solution(tsp_cities_dos, 0, 1e-1, 160)

      CPU times: user 12.1 s, sys: 0 ns, total: 12.1 s
      Wall time: 12.1 s
 [7]: (3668.3573639350216,
```

Ejemplo 20 ciudades: comparación OR Tools vs parallel_hc)

```
[10]: %%time
      tsp_sol = tsp.main(tsp_cities)

      Objective: 3317
      Route:
       0 -> 1 -> 5 -> 6 -> 12 -> 13 -> 11 -> 15 -> 17 -> 18 -> 19 -> 16 -> 14 -> 7 -> 9 -> 2 -> 4 -> 10 -> 8 -> 3 -> 0

      CPU times: user 14.5 ms, sys: 0 ns, total: 14.5 ms
      Wall time: 13.5 ms

[11]: %%time
      parallel_hc(tsp_cities, 0, 1e-1, 100)

      CPU times: user 21.1 ms, sys: 22.4 ms, total: 43.6 ms
      Wall time: 9.2 s
[11]: (3382.97987100559,
       [0, 3, 8, 10, 2, 4, 7, 9, 14, 16, 19, 18, 17, 15, 11, 13, 12, 6, 5, 1, 0],
       9.042816638946533)
```

Resultados tanto de OR Tools como de parallel_hc para 50 ciudades, mostrando la distancia encontrada por ambas como factible:

Ejemplo 50 ciudades: comparación OR Tools vs parallel_hc)

```
[12]: tsp_sol = tsp.main(tsp_cities)

      Objective: 9943
      Route:
       0 -> 1 -> 5 -> 6 -> 11 -> 13 -> 24 -> 21 -> 12 -> 25 -> 27 -> 30 -> 35 -> 33 -> 34 -> 32 -> 31 -> 49 -> 47 -> 41 -> 38 ->
      44 -> 45 -> 48 -> 39 -> 40 -> 42 -> 37 -> 26 -> 28 -> 29 -> 46 -> 43 -> 36 -> 15 -> 17 -> 18 -> 22 -> 19 -> 20 -> 23 -> 16
      -> 14 -> 7 -> 9 -> 2 -> 4 -> 10 -> 8 -> 3 -> 0

 [8]: prueba = parallel_hc(tsp_cities)

 [9]: prueba

 [9]: (11246.084161235227,
```

7. Conclusiones

Uno de los más aspectos importantes a mencionar del problema TSP que buscamos resolver con nuestra implementación del método de Hill Climbing, es que el espacio de posibles soluciones tiene un crecimiento factorial, esto es, para un conjunto de n ciudades se tiene un total de $n!$ rutas posibles. Esto hace que no sea factible explorar todo el espacio de soluciones y sea necesario utilizar métodos que lo hagan de forma aleatoria pero suficientemente amplia, como lo hace Hill Climbing. Invariablemente, conforme crece el número de ciudades, es necesario que crezca también el espacio a explorar para que se garantice que la mejor ruta es suficientemente cercana al mínimo global.

Por lo anterior, fue necesario realizar nuestro perfilamiento de código restringiendo el problema y la implementación del método a condiciones que nos diera una solución óptima (mejor ruta, más corta) con una exploración suficiente del espacio de soluciones (rutas posibles). Para ello consideramos un subconjunto de 17 ciudades de nuestro dataset y se utilizaron 300 puntos iniciales (*random-restart*).

Durante el perfilamiento de nuestro método, nos permitió aplicar las funciones que nos ayudaron a conocer la anatomía de nuestro método durante el proceso de ejecución y ser capaces de identificar en qué partes de nuestro código se concentra el tiempo de ejecución y el incremento en el uso de memoria RAM.

Por lo anterior, particularmente los puntos primero y último, hemos establecido que nuestro principal enfoque sería reducir los tiempos de ejecución realizando reimplementaciones, con lo cual incrementaríamos la exploración del espacio de soluciones lo que nos permitiría obtener una ruta suficientemente cercana a la más corta.

Basado en estos datos del perfilamiento, implementamos mejoras en las parte del código con las que se tenia mayor interacción. Se observó en el ejemplo que los tiempos son más pequeños sin el uso de *Numba*, en dicho ejemplo aumentaron los tiempos de ejecución en más de 5x. A pesar de haber tomando dicha acción, ni los tiempos de ejecución y la solución dada por el algoritmo, cómo la ruta más corta, mostraron mejoras sustanciales con esta implementación.

Dado estos resultados se tomó la decisión de modificar nuestro código para la práctica final, con el objetivo de utilizar *frameworks* cómo el módulo de `python_multiprocessing`, para implementar el cómputo en paralelo. En esta nueva implementación nos ayudo a tener mejores resultados disminuyendo a la mitad el tiempo de ejecución, pero ya que el espacio de posibles soluciones crece exponencialmente con el aumento de ciudades y que *Hill Climbing* es un algoritmo meta-heurístico la soluciones encontradas se alejan del resultado encontrado por el paquete de *Google* conforme aumenta el número de ciudades tomadas en cuenta en el problema.

8. Referencias

1. *Hill Climbing*
2. *How to Implement the Hill Climbing Algorithm in Python (towards data science)*
3. Erick Palacios Moreno (2021), Optimización, Nota 5.3
4. OR tools TSP
5. Russell, Stuart J.; Norvig, Peter (2003), *Artificial Intelligence: A Modern Approach* (2nd ed.), Upper Saddle River, New Jersey: Prentice Hall.