El problema del agente viajero
Recocido Simulado

Seminario de Ciencias de la Computación B

Canek Peláez Valdés

Universidad Nacional Autónoma de México

Sánchez Correa Diego Sebastián

Índice general

1. Introducción **2**
- 1.1. El problema . . . 3
- 1.2. La Heurística . . . 3

2. Diseño **4**
- 2.1. La base de datos . . . 4
- 2.2. Las ciudades . . . 4
- 2.3. Los caminos . . . 5
- 2.4. TSP . . . 5
- 2.5. Recocido Simulado . . . 6

3. Implementación **7**
- 3.1. Hilos de ejecución . . . 7
  - 3.1.1. Generador de números aleatorios . . . 7
- 3.2. Problemas . . . 8
  - 3.2.1. Fugas de memoria . . . 8
- 3.3. Registros . . . 9

4. Experimentación **10**
- 4.1. Instancia de 40 ciudades . . . 10
- 4.2. Instancia de 150 ciudades . . . 13

5. Conclusiones **14**

Capítulo 1

Introducción

Podríamos pensar que el mundo actual, dada la tecnología con la que contamos, cuenta con computadoras capaces de hasta la más grande de las computaciones. Vaya que es cierto, que, desde el surgimento de la computadora, el progreso tecnológico ha sido inmenso, sin embargo, sin importar el poder computacional al que hemos llegado, siguien existiendo problemas que no son resolubles en un tiempo racionalmente corto, unos problemas que, incluso, podrían definirse como irresolubles por el tiempo que conllevaría computarlos para obtener la mejor solución.

Los tiempos de computación de cualquier algoritmo caerán en dos grupos. El primero, consiste de solo los problemas cuyas soluciones están acotadas por un polinomio de un grado bajo. El segundo, por los problemas cuyos mejores algoritmos son no polinomiales.

Estos últimos, son los problemas NP. Los problemas *NP* (Non Deterministic Polynimial Time) pueden ser subdivididos en *NP-Completo* y *NP-Duro*.

Un problema que es *NP-Completo* tiene la propiedad de que podrá ser resulto en tiempo polinomial si y solo si todos los problemas en el conjunto *NP-Completo* pueden ser resueltos en tiempo polinomial. Además, existe un algoritmo que verifica que una solución dada es la óptima. Esta última propiedad hace destacar a los problemas *NP-duro* y este el conjunto donde se encuentra el Problema del Agente Viajero.

1.1. El problema

...la pregunta es encontrar, para un conjunto finito de puntos de los cuales se conocen las distancias entre cada par, el camino más corto entre estos puntos. Por supuesto, el problema es resuelto por un conjunto finito de intentos. ***Schrijver (2005)*** [1]

Dados n puntos y todas las distancias entre las parejas ordenadas de distintos puntos.
El problema es hallar un sistema de circuitos ordenados tal que:
1. Cada punto permanezca en exactamente un circuito.
2. Cada circuito contenga al menos 2 puntos.
3. Ningún circuito pase por a través del mismo punto más de una vez.
4. La distancia total de los circuitos es un mínimo. [5]

1.2. La Heurística

El recocido simulado *es un método probabilístico propuesto por Kirkpatrick, Gelett, y Vecchi y Cerny para encontar el mínimo global de una función de costo que puede poseer varios mínimos locales [3].*

Esta fue formulada a partir de la simulación de un fenómeno físico usado comúnmente en la metalurgia, se trata del proceso que se da, una vez calentado un metal y enfriado, posteriormente, si este no es vuelto a calentar, entonces, este se hará muy duro y frágil, en cambio, el recocido pretende dar maleabilidad y ductilidad al material enfriando en pequeños umbrales el material.

[1] Alexander (Lex) Schrijver (4 de Mayo 1948, Ámsterdam) es un matemático y científico en computación holandés, profesor de matemáticas discretas y optimización en la Universidad de Ámsterdam

Capítulo 2

Diseño

El diseño siguió una estructura orientada a objetos. Contar con la información de las ciudades en una base de datos relacional, sugirió la implementación de un objeto **`database_loader`**. De la msima manera, el contar con soluciones compuestas por permutaciones de ciudades, hizo claro el uso de las clases **`city`** y **`path`** (ciudad y camino, respectivamente). De la misma manera, se ha planteado el modelado del problema y de la heurística como clases compuestas a partir las unidades más fundamentales y con atributos y comportamiento asociado.

2.1. La base de datos

El problema se planteó como una base de datos relacional cuyo contenido está compuesto de las ciudades (1092 en total) y de sus conexiones (que representan la gráfica del problema).

Es importante mencionar que la matriz de adyacencias represnetada por la tabla de conexiones no es cuadrada, sin embargo, el objeto **`database_loader`** la cargará a una matriz de enteros (**`int`** de C) haciéndola cuadrada.

La tabla de ciudades se carga a un arreglo de apuntadores a objetos de tipo **`city`**. Normalmente se usaría un puntero a un objeto de tipo **`city`**, pero el constructor de la clase, naturalmente, aloja los objetos en el heap, devolviendo, por lo tanto, un puntero a un objeto; por ello, el arreglo solo reúne los punteros de los objetos que ya se encuentran guardados dinámicamente; evitando, en conjunto, una posible ambigüedad al tratar con ambos tipos que, en esencia, no comparten características adicionales a ser un apuntador a una ciudad.

2.2. Las ciudades

El objeto **`city`** tiene como propiedades los campos incluidos en la base de datos, con excepción de la población. La clase abstrae el concepto de una ciudad, omitiendo una representación total a partir de atributos asociados (como lo hace la base de datos), sino involucrando en la composición un comportamiento particular.

Esta abstracción se da al decidir que una ciudad proporcione el comportamiento para calcular la distancia con otra ciudad en conjunto con su uso como unidad fundamental del problema.

Cabe destacar que la clase **`city`** también es la encargada de crear los arreglos de ciudades, resultado de una "modularización"del diseño : ninguna clase conoce el tamaño de la estructura **`city`**, por lo tanto, ninguna clase (a excepción de city) es capaz de crear dinámicamente un arreglo de ciudades; esto podría evitarse si se incluyera la declaración de la estructura dentro del encabezado (header) de la clase, pero considero se trataría de una violación a la encapsulación de los datos.

2.3. Los caminos

Al tener la idea de una ciudad ya abstraída en una clase, era natural que los caminos consitieran de, al menos, un arreglo de ciudades. El primer diseño que realicé se trataba de una representación sin atributos, ni comportamiento; solo un arreglo de ciudades. Poco tiempo después escribí la clase **`path`**. Esta ahora no solo se trataba de un arreglo de ciudades, sino de una estructura con atributos y comportamiendo asociados: esto facilitó (desde un punto de vista del diseñño) la implementaciones de funciones como el intercambio de ciudades dentro de un camino, el cálculo del normalizador y la función de costo.

La creación de un camino implica el cálculo de la distancia máxima de esa instancia, así, como el normalizador y la suma de los costos de sus ciudades. Estos atributos se calculan a partir de operaciones lineales, por ello, son valores que incializo dentro del constructor de la clase y, si se quiere acceder a ellos una vez creado al objeto, la operación se reducirá a una consulta lineal del atributo deseado del objeto. Esta solución implicaba la disminución de la cantidad de invocaciones de funciones para crear una camino válido (dejó de ser necesario invocar la función, seguido de usar un setter del objeto para guardar el resultado), sin embargo, me impedía utilizar el mismo constructor para hacer copias de un objeto (usadas en la heurística) si pretendía mantener el tiempo de ejecución en un estado óptimo, por supuesto.

La implementación de la función copia de la clase **`path`**, entonces, no invoca al constructor de la clase, sino que accede a las propiedades del objeto parámetro y aquellas que eran computadas linealmente, se asociarían como copias al nuevo objeto.

2.4. TSP

El problema del agente viajero fue, como lo han sido todas las estructuras del proyecto, modelado siguiendo un diseño orientado a objetos, sin embargo, no sería un error afirmar que este no era su propósito inicial. La clase comenzó siendo un objeto central de ejecución que administraría la invocación y liberación de todas las clases necesarias, incluso, sería la encargada de ejecutar la heurística.

La clase desempeña el papel de marcar el inicio de la ejecución del problema, creando un objeto **`database_loader`**, proveyendo la información computada en este. Tiene como atributo un camino (el único creado en la ejecución del programa; con

el constructor de la clase, al menos) que también será usado para la heurística. Por ello, una ejeución, no estará completa sin una invocación a la clase **`sa`**.

2.5. Recocido Simulado

La clase implementa el algoritmo del Recocido Simulado guarda los parámetros que usa la heurística (además de definir unos predeterminados) y está asociada con **`tsp`** teniéndola como un atributo; el camino inicial es proporcionado por **`tsp`**.

Además de implementar las funciones que la heurística de Recodio Simulado define, implementa la heurística de cálculo de temperatura incial y el barrido; este último para lograr llegar a un mínimo local, dada la solución final de la heurística.

Como la heurística describe lotes de soluciones, se incluye una estructura interna que tiene como atributos un camino y el promedio del lote.

Capítulo 3

Implementación

C no es un lenguaje de programación orientado a objetos, sin embargo, dada su baja abstracción es fácil simularla. Cada una de las clases descritas en el diseño están definidas a partir de estructuras (objetos) que engloban una serie de variables (atributos) y con una serie de funciones asociadas a ellos (comportamiento).

La implementación de una simulación de la orientación a objetos suele basarse en la implementación de estructuras que, además de contar como miembros variables que se considerarían atributos, tienen como miembro apuntadores a funciones. Sin emabargo, mi implementación consistió en la definición de funciones que recibían, en la mayoría de los casos, un apuntador a un objeto de la clase (estructura) en cuestión.

3.1. Hilos de ejecución

En función de la maximización de ejecuciones del sistema, la idea de la implementación de la creación de hilos de ejecución no podía omitirse. El programa fue ejecutado en una computadora de escritorio con un procesador AMD Ryzen™ 5 1600; este podía ejecutarse un total de 16 veces en paralelo y, usando hasta el 70 % de los 16GB de RAM, era posible crear 1,000 hilos de ejecución.

3.1.1. Generador de números aleatorios

Los hilos de ejecución traían consigo la limitación en el uso de variables o funciones globales que pudieran generar alguna condición de carrera o el establecimiento de una variable como constante para todos los hilos de ejecución. Este último caso fue un problema presente en mi implementación; en un inicio, usé las funciones **`srand()`** y **`rand()`** para generar los números pseudoaleatorios para los algoritmos no deterministas, sin embargo, noté que al introducir los hilos de ejecución, la primera semilla que se pasaba como parámetro a la primera ejecución de **`srand()`**, era la única usada durante la ejecución del programa y, por lo tanto, para todos los hilos creados durante esta.

Por ello, introduje una función nueva: **`rand_r()`**. Esta función recibe una referencia a una entero sin signo y genera números pseudoaleatorios en el rango [0-

RAND_MAX[1]], esto podría explicar el comportamiento inusual de generación de temperaturas iniciales y soluciones casi idénticas y con alta regularidad. Es definida como una función generadora de números pseudoaleatorios débil, porque el rango del tipo de su semilla no es destacable.

3.2. Problemas

Hace un par de meses realicé un proyecto en el lenguaje de programación C++, esto me introdujo a conceptos con los que no estaba familiarizado; en general, aquellos relacionados con la asignación y liberación de memoria. Los punteros (o apuntadores) fueron un concepto destacable entre las fuentes de problemas más comunes durante la depuración. Sin embargo, C++ provee abstracciones (como la clase String) que evitaban estos conceptos y se asumían como buenas prácticas o estándares del lenguaje. Es por esto que, a pesar de ser C++ un superconjunto de C, no me familiaricé con estos conceptos hasta poco antes de comenzar el proyecto.

3.2.1. Fugas de memoria

Al no contar con un recolector de basura, es necesario liberar la memoria dinámica reservada durante la ejecución del programa. Producto de la abstracción del programa en clases (estructuras) la creación de objetos implicaba una liberación análogo: aparear cada llamada de **`malloc()`** a **`free()`**. Esto se torna una tarea complicada al asignar memoria dinámica en ciclos. Por ejemplo, en la implementación de la clase **`database_loader`**, se asigna una porción de memoria a un arreglo de apuntadores a objetos de la clase **`city`** (comportamiento provisto por la clase **`city`**) , y para su llenado se itera sobre la función **`callback`** que usa **`sqlite`** para interactuar con los datos que una consulta devuelve; en este ciclo se asigna memoria para cada ciudad contenida en la base de datos, resulta, por lo tanto, fácil perder de vista cada una de las asignaciones (1092 en total). Se torna una tarea complicada al mezclarla con la ausencia de la asignación dinámica del nombre de las ciudades, que define comportamiento indefinido (guardado en el stack de la función), y que solían desaparecer al intentar imprimirlos en la función **`callback`**: no me fue claro si el error provenía de la liberación, en un principio.

No mucho tiempo después encontré una herramienta diseñada por Google que pretende ayudar en la localización de las asignaciones que no tuvieron una liberación apareada (*ASAN*; además de hacer mucho más explícitos errores como el acceso a un índice no válido de un arreglo o al intentar liberar memoria ya liberada, entre otros). Si bien no es completamente certero el decir que el programa no cuenta con fugas de memoria[2], la evaluación de *ASAN* (Address Sanitizer) y la ejecución prolongada del programa (con un máximo de 6 horas continuas), sugieren que, en caso de contar con alguna, esta será despreciable.

Durante la implementación de las pruebas unitarias, un error asociado a la asignación de memoria surgió, este era aún más obscuro. Se trataba de un error que sirgía

[1] El valor del macro RAND_MAX será de al menos 32767.

[2] *..las fugas de memoria pueden ser sustituídas por la determinación del alto de un programa con las mismas consecuencias lógicas.* La determinación de la ausencia de fugas de memoria implicaría, en ciertos casos, la solución al problema del paro (al lidiar con programas lo suficientemente complejos). [6]

cuando el directorio **`build`** (usado por el sistema de construcción *Meson*) del programa no era generado con las banderas de *ASAN*; las pruebas parecían no tener éxito en casos previamente comprobados. Era aún más raro el hecho de que *ASAN* no detectara alguna fuga o problema asociado con la memoria dinámica, en general. Este resultó ser causado por el uso de **`malloc()`** en la inicialización de variables (arreglos de números) que requerían un estado inicial, antes de ser usados o de determinar qué valores estarían contenidos en ellas (**`calloc()`** tiene este comportamiento). *Valgrind* (otra herramienta para detección de fugas de memoria) fue primordial en la detección de este error.

3.3. Registros

Las variables de tipo **`register`** son descritas como sigue:

> *Una declaración **`register`** sugiere al compilador que la variable en cuestión será muy utilizada. La idea es que las variables **`register`** serán colocadas en registros máquina, lo que resultaría en programas más pequeños y rápidos. Pero los compiladores son libres de ignorar esta sugerencia. [1]*

Consideré, así, su introducción en la implementación del programa (ya que la eficiencia y la optimización son dos conceptos claves). Parecía una tarea simple, pero considerando que la asignación de registros *tiene como objetivo el asignar una cantidad finita de registros máquina a un número ilimitado de variables temporales tal que las variables temporales con rangos de vida interferentes sean asignados a diferentes registros [4]*, suele resolverse con algoritmos (heurísticas, en realidad; se trata de un problema *NP*) que pretenden encontrar soluciones al problema de coloración de teoría de gráficas[3][4]. El mapeo se hace de vértice a variable temporal, donde las aristas conectan a las variables cuyos rangos de vida interfieren [4].

Que los compiladores eligieran si aceptaban estas sugerencias y que la obtención de una buena asignación fuera reducible a encontrar una solución para un problema *NP* fueron suficientes para evitar introducirlos en la estructura del programa.

[3] El problema de coloración de gráficas puede ser descrito de la siguiente manera: dada una gráfica G y un entero positivo K, asigna un color a cada vértice de G, usando a lo más K colores, tal que dos vértices adyacentes reciban el mismo color. [4]

[4] *En 1982, Chaitin redujo el problema de coloración de gráficas, al de asignación de registros , y con ello, probando que la asignación de registros es un problema NP completo también* [4] [2]

Capítulo 4

Experimentación

El comportamiento del sistema frente a las instancias presentadas no fue el óptimo, el equilibrio buscado en los parámetros no fue encontrado (esto resultata más notable en la segunda instancia). Sin embargo, fue posible alcanzar soluciones óptimas, dada la rapidez del mismo, que involucraba una ligera pérdida en el rendimiento al aumentar ciertos parámetros a valores que, normalmente, serían considerados inmensos.

La calibración de los parámetros no parecía tener una gran repercusión en las ejecuciones de la primera instancia. Fue hasta la segunda instancia en la que noté que el incremento de algunos de ellos hacían del sistema un programa más lento, pero con mucho mejores ejecuciones.

La experimentación fue realizada principalmente de manera concurrente en una computadora de escritorio con especificaciones:

- *Procesador:* AMD Ryzen™ 5 1600 Six-Core @ 3.20GHz
  (8 Cores / 16 Threads) AMD "ZenÇore Architecture
- *Memoria RAM:* 16GB
- *Cachés de los procesadores:* Caché L1 total: 576KB
  Caché L2 total: 3MB
  Caché L3 total: 16MB
- *Almacenamiento:* 960GB KINGSTON SA400S3 SSD
- *OS:* Arch rolling
  Kernel: 6.1.8-zen1-1-zen (x86_64)

4.1. Instancia de 40 ciudades

Esta instancia no necesitó de parámetros elevados, ni de una gran cantidad de ejecuciones para obtener una solución óptima. Esta fue obtenida sin la implementación de la heurística de temperatura inicial y sin el algoritmo del barrido. Su obtención se hizo a través de 3 mil ejecuciones, dividida en 3 ejecuciones de 100 hilos cada una, con una duración aproximada de 3 minutos por ejecución (el tamaño del lote tan reducido provocara que las ejecuciones no tomaran una gran cantidad de tiempo).

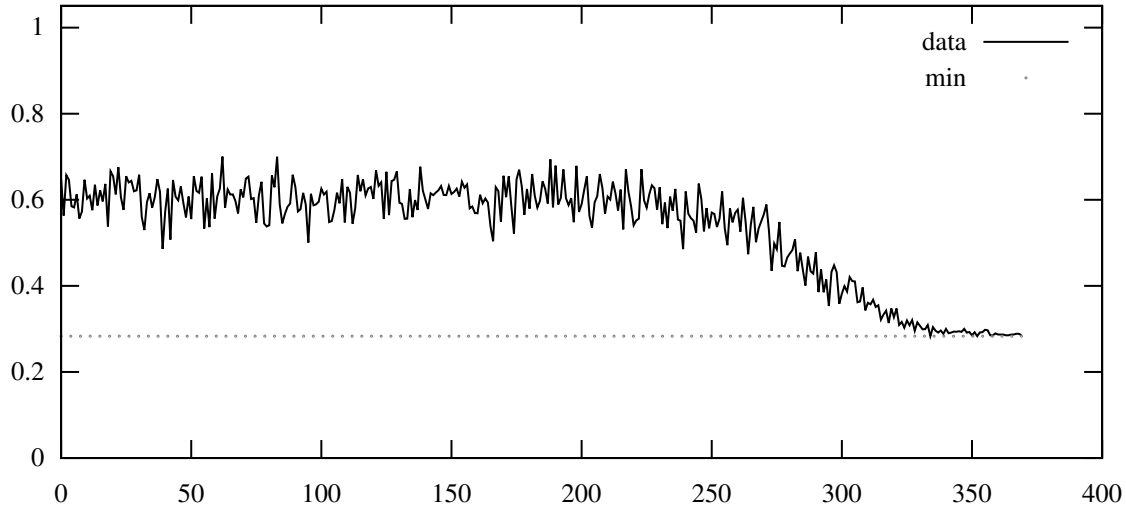

Figura 4.1: Impresión de las evaluaciones computadas por el lote de una solución factible.

Parámetros de la mejor solución encontrada para la instancia de 40 ciudades.

Parámetro	*Valor*
Semilla	102
Temperatura	14
M	122000
L	1200
Épsilon	0.002
Phi	0.95

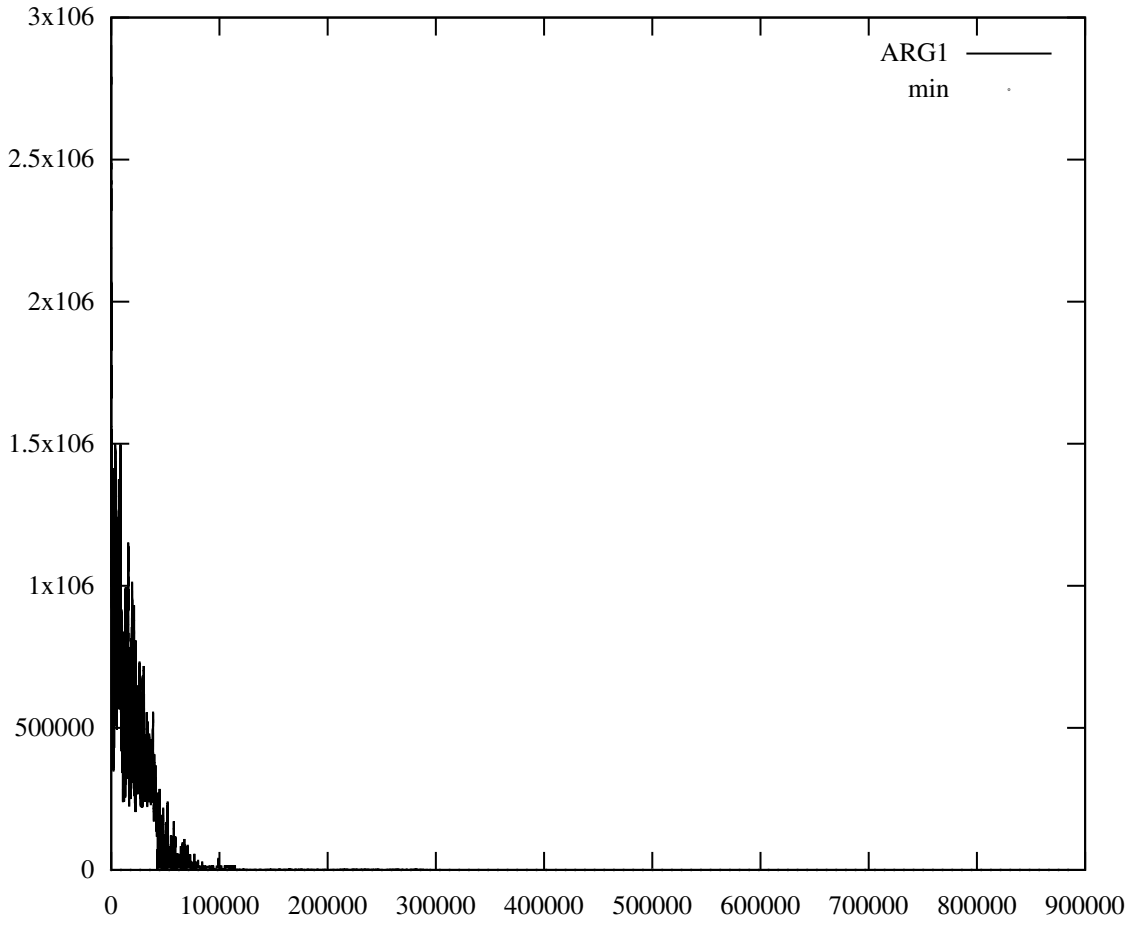

Figura 4.2: Evaluaciones de la mejor solución encontrada en la instancia de 40 ciudades.

4.2. Instancia de 150 ciudades

Parámetros de la mejor solución encontrada para la instancia de 150 ciudades.

Parámetro	*Valor*
Semilla	263
Temperatura	8
M	260000
L	12000
Épsilon	0.000016
Phi	0.98
P	0.98
N	900

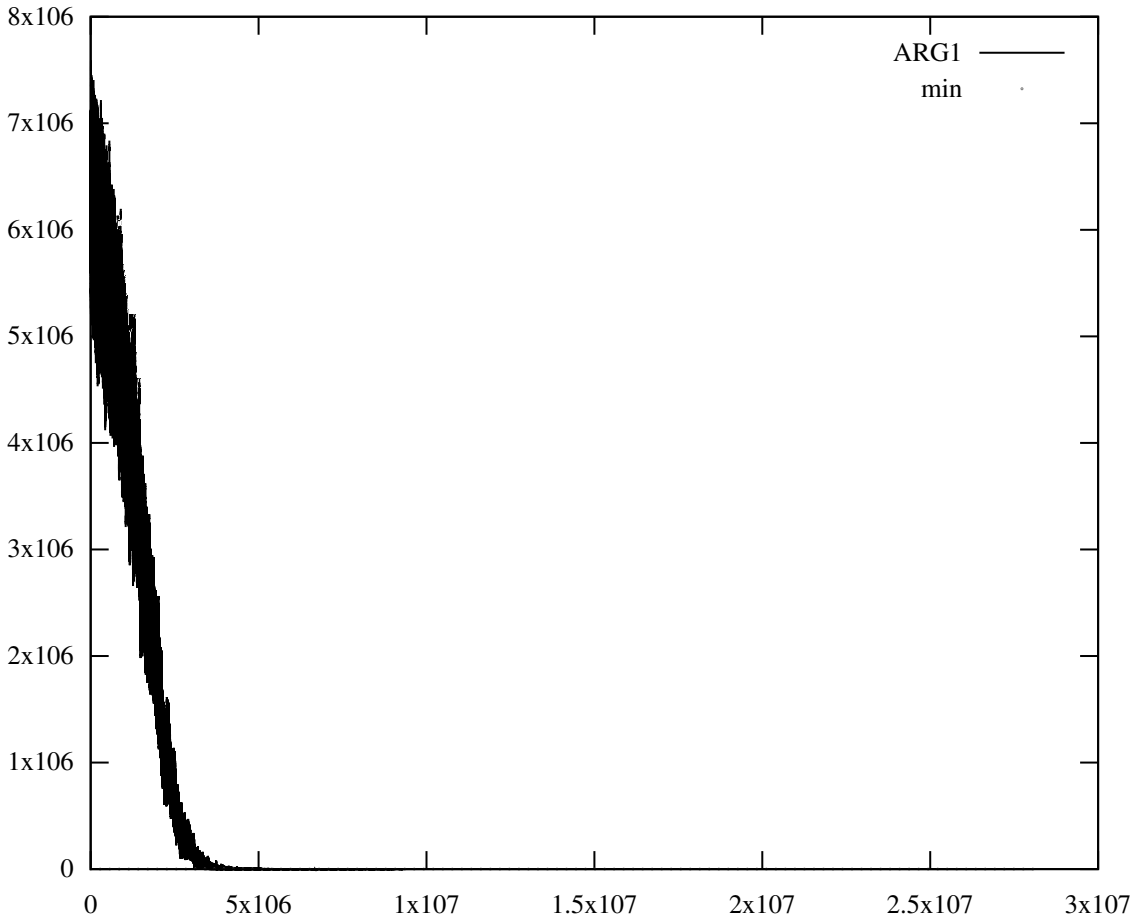

Figura 4.3: Evaluaciones de la mejor solución encontrada en la instancia de 150 ciudades.

El tamño del lote fue un factor muy relevante en la computación de las soluciones; dejar 64,000 ejecuciones (con un tiempo aproximado de ejecución de 6 horas) no rindió grandes frutos, en esta ejecución se obtuvo como resultado más óptimo 0.154. Posteriormente, se ejecutó el sistema con una tamaño de lote de 12000 y se logró superar la solución previa, en 3 minutos y ejecutando 200 semillas concurrentes.

Capítulo 5

Conclusiones

Logré observar que mi sistema se comportaba de manera óptima incluso con parámetros grandes he ahí la justificación de los parámetros de la mejor solución encontrada para la instancia de 150 ciudades. Por otro lado, un tamaño de lote reducido (que, aparentemente, funcionaba mejor con otros sistemas) parecía solo funcionar con la intancia 40 ciudades. La instancia de 150 ciudades tardaba, aproximadamente, 20 veces más en la ejecución de una semilla.

Bibliografía

[1] Dennis M. Ritchie Brian W. Kernighan. The c programming language, 1978.

[2] G J Chaitin. Register allocation and spilling via graph coloring. Symposium on Compiler Construction, 1982.

[3] John Tsitsiklis Dimitris Bertsimas. Simulated annealing. Statistical Science, 1993.

[4] Jens P. Fernando M. Q. Pereira. Register allocation via coloring of chordal graphs. https://homepages.dcc.ufmg.br/ fernando/publications/papers/APLAS05.pdf.

[5] Julia Robinson. On the hamiltoninan game (a traveling salesman problem). Research Memorandum, 1949.

[6] Samsai. Memory leaks are memory safe, and there isn't much to do about it. https://samsai.eu/post/memory-leaks-are-memory-safe/, 2020.