

**Seminario de Ciencias de la Computación B**  
**Heurísticas de Optimización Combinatoria**  
**Problema del Ajente Viajero**  
**con Recocido Simulado**

Profesor: Canek Peláez Valdés  
Autor: Xin Wen Zhang Liu

Viernes, 17 de Marzo, 2023.

### **El problema del agente viajero**

El problema del agente viajero es uno de los problemas de Optimización combinatoria más estudiados. Introducido por primera vez en 1930

Este problema plantea una pregunta fácil de hacer pero difícil de responder:

Dado un conjunto de ciudades y sus coordenadas en el plano cartesiano, ¿cuál es el camino más corto que visite cada ciudad exactamente una vez?

La manera más directa de resolver este problema sería listar todas las posibles combinaciones de caminos que pasen por las ciudades deseadas y comparar sus costos. Sin embargo el número de combinaciones diferentes con  $n$  ciudades crece a la par de  $n!$ , lo que hace que la cantidad de posibles combinaciones crezca a un paso colosal cuando incrementamos  $n$ .

Imaginemos que tenemos 5 ciudades, entonces  $5! = 5 \times 4 \times 3 \times 2 \times 1 = 120$ , ahora dupliquemos el número de ciudades, entonces  $10! = 10 \times 9 \cdots \times 1 = 3,628,800$ . Una vez que tengamos una cantidad considerable de ciudades el tiempo de comparación se vuelve no factible, es por esto que se elaboran métodos alternativos para resolver este tipo de problemas, tal como el recocido simulado.

### **Recocido Simulado**

El recocido simulado (simulated annealing) es un método probabilístico propuesto por Kirkpatrick, Gelett y Vecchi (1983) y Cerny (1985) para encontrar el mínimo global de una función de costo que puede poseer múltiples mínimos. Emulando el proceso físico por el cual un sólido es lentamente enfriado para que al llegar a un estado de congelación, este lo haya hecho con una configuración de energía mínima. [1]

Este método se basa en obtener vecinos randomizados, donde se define una temperatura que decremente lentamente, lo que nos da una cota superior que permite "subidas" que incrementan el costo, esperando que estas "subidas" lleven a un mínimo local. Este proceso nos da una función que converge hacia la solución óptima.

### **Aceptación por umbrales**

### **Diseño**

El diseño de este programa usa el paradigma orientado a objetos, con una estructura bastante simple. Consiste de 5 clases structs

- City el cual es un struct que solamente guarda el id de la ciudad, su longitud y latitud.
- Path contiene funciones de todas las operaciones que puedes realizar sobre estas. Este struct contiene como atributo igual una instancia de todas las ciudades guardadas en un vector, así como el vector bidimensional que guarda todas las distancias.
- SimAnn, como su nombre lo indica, este struct contiene los algoritmos del recocido simulado. Dentro de este, se encuentra la clase path como atributo para realizar las operaciones de costo y swapeo.

- TSI se encarga de generar hilos con instancias de SimAnn, para correr varias semillas al mismo tiempo, lo que facilita mucho la experimentación, ya que correr grupos de cualquier cantidad sin necesidad de atención constante.
- Reader el cual se encarga de leer de la base de datos.

## Implementación

El lenguaje usado para este proyecto fue Rust. Una de las cuantas razones fue la rapidez y su confiabilidad. La rapidez es un factor muy influyente en la experimentación, ya que facilita la rápida obtención de resultados para poder mejorar los parámetros usados.

Al comienzo del proyecto la principal dificultad fue el conocimiento del lenguaje. En la primera etapa del proyecto la inversión de tiempo en comprender estructura, sintaxis y flujo de código dentro de un nuevo lenguaje es considerable. Se necesita primero brechar la cuenca de conocimiento para poder comenzar a programar, e incluso después el implementar cosas de una manera correcta lleva una serie de ensayos y errores.

Una vez ya empezada la implementación el resolver errores lógicos tomó una gran parte del tiempo, y es donde las pruebas unitarias fueron de mayor utilidad, de esta manera se excluye la necesidad de probar funciones mano. Aunque la cantidad de funciones no era grande, esta etapa fue la que más tiempo tomó.

Casos como implementar la función de distancia natural, que puede llevar a pequeños errores difíciles de encontrar. Así como el implementar el swap de tiempo constante y tener en cuenta todos los posibles casos a salir.

Esto sin considerar los problemas de optimización, ya que en una sola instancia del recocido simulado hay operaciones que se ejecutan hasta millones de veces, el problema de optimización se hace más pertinente. Del mismo modo el poder optimizar alguna parte dentro de esa iteración decremente de manera considerable el tiempo de ejecución.

Igual la falta de una planeación de antemano llevó a la necesidad de refactorizar código. Como en el caso de que la implementación de la heurística de la temperatura inicial y del barrido se realizó después de que el proyecto ya había avanzado bastante, lo que llevó a hacer nuevas refactorizaciones.

Incluso después ya teniendo las funciones el conectar su uso con los diferentes procesos es

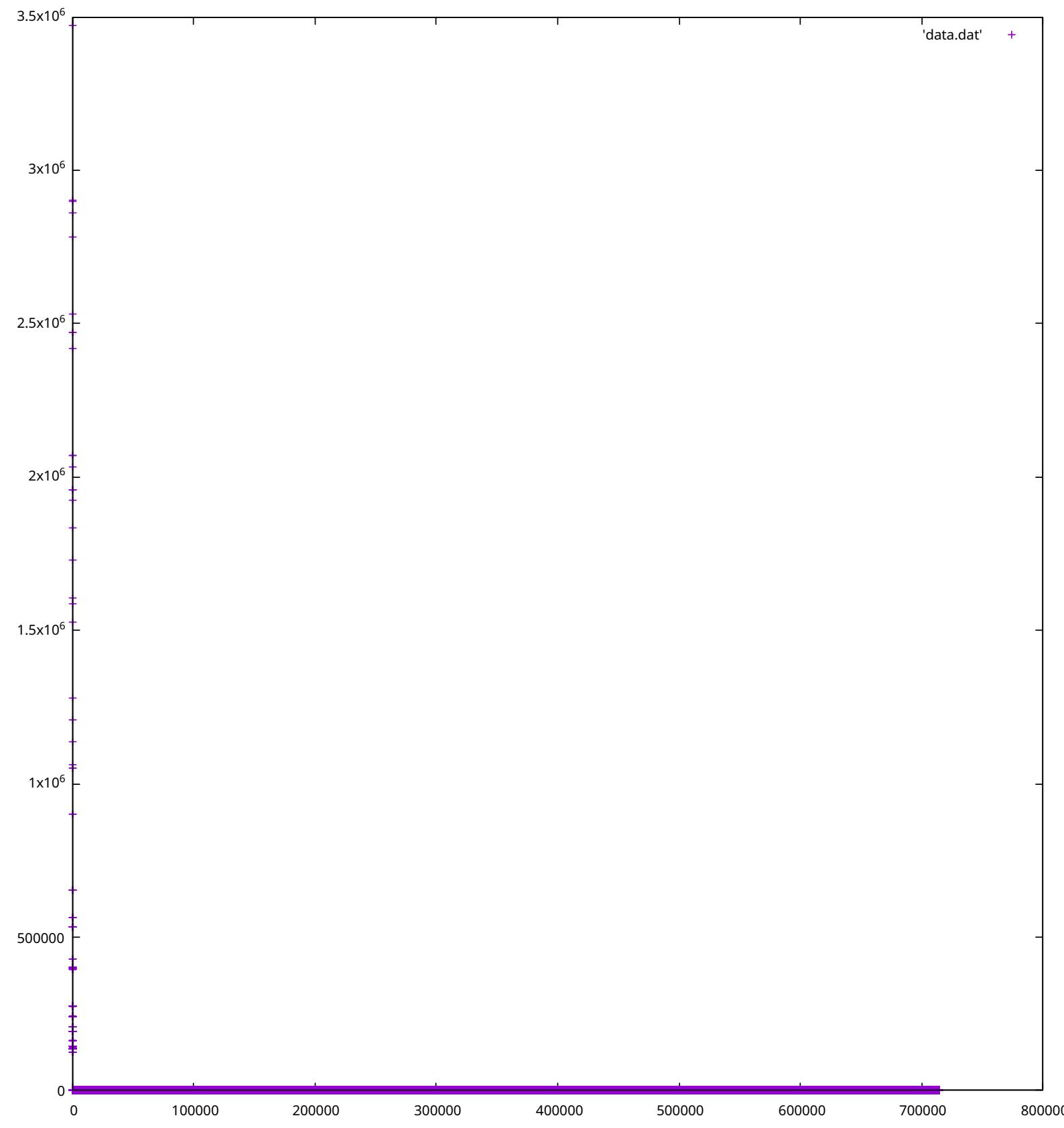
## Experimentación y resultados

La etapa de la experimentación para la cual se había previsto una mayor inversión de tiempo terminó siendo recorrida por la implementación.

La primera corrida después de haber terminado la implementación del algoritmo llevó a un resultado favorable, lo que igual influyó en la tardanza de experimentar con cambios más grandes de los parámetros , los valores usados fueron

- Epsilon : 0.002
- Phi : 0.95
- Tamaño de lote : 2000
- Temperatura inicial : 8

La siguiente es la gráfica de esa primera solución



Como se puede ver, hay un completa falta de exploración al principio que nos lleva de manera precipitada a una solución. Esto es por el valor tan bajo que tenía la temperatura inicial. El incrementar la temperatura lleva a un mayor porcentaje de aceptación y a una mayor exploración pero incrementa el tiempo de ejecución. Del mismo modo con el tamaño de lote/

Después de varios intentos fallidos estas fueron algunas de las observaciones concluidas.

La primera etapa de la experimentación consiste en afinar los parámetros. Fue mejor correr un número reducido de semillas en esta etapa, de esta manera se obtienen resultados rápidos y hay exploración constante, tal como la heurística, el principio es el momento de intentar parámetros muy grandes so muy chicos. Y observando los resultados uno converge a parámetros más óptimos para cada caso.

Una vez que los resultados parecen ser favorables es entonces cuando llega el momento de correr una gran cantidad de semillas para experimentar con la probabilidad de encontrar mejores soluciones.

Durante la experimentación una de las observaciones encontradas es que una epsilon menor parece encontrar mejores soluciones para el caso con un mayor número de ciudades, en cambio el usar una epsilon muy reducida para el caso con menos ciudades parecía hacer que el algoritmo tardara mucho o no terminara, además de que los resultados dados no eran tan favorables. Por lo que usar  $\epsilon = 0.0001$  para 150 ciudad y  $\epsilon = 0.002$  para 40 ciudades, parece ser la manera de encontrar soluciones más óptimas.

Como el uso de la herística de temperatura incial fue implementada ya por el final, llevó a que una parte del tiempo de experimentación se haya usado para manualmente calcular una temperatura indicada para el caso general.

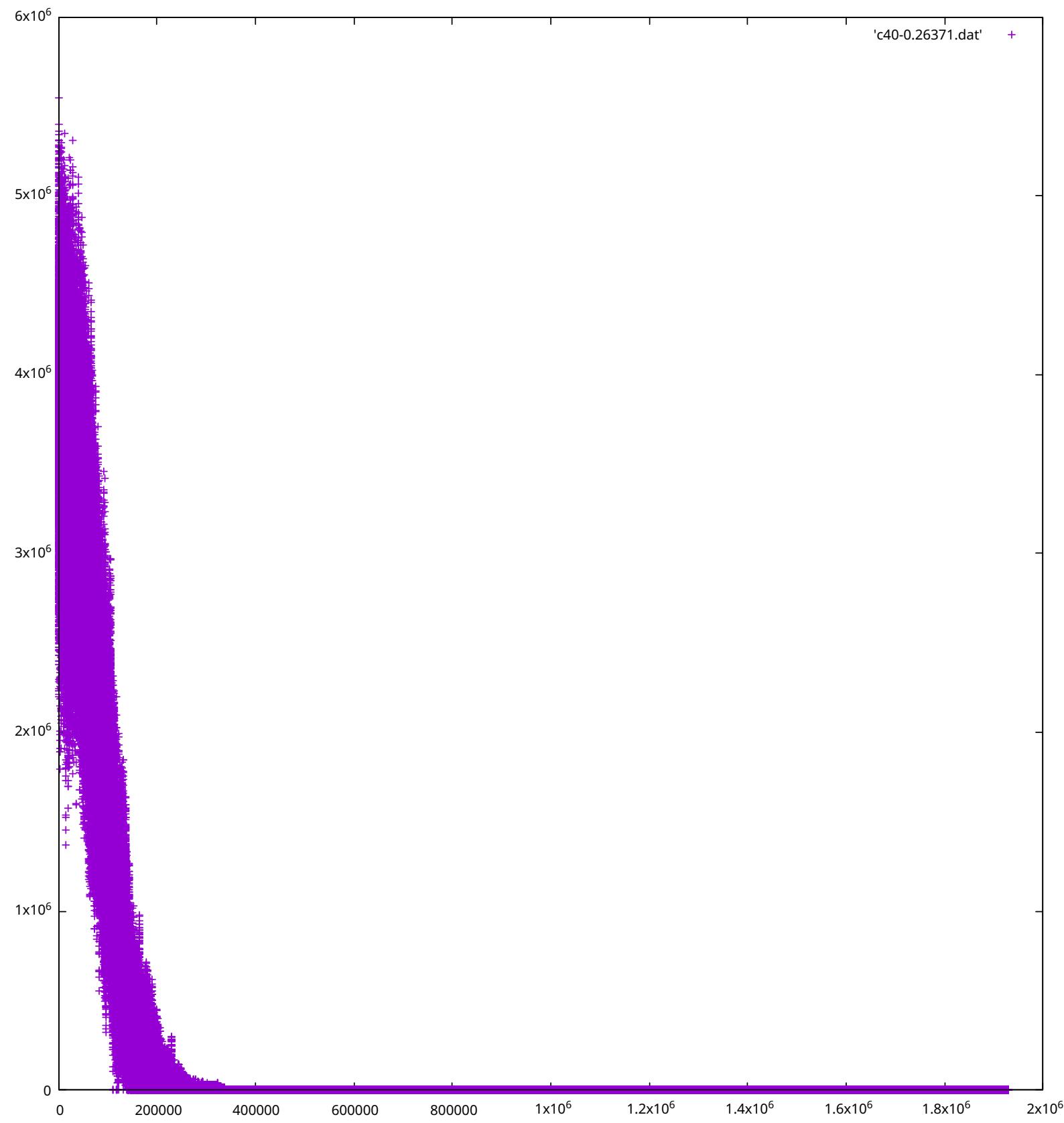
Dentro del algoritmo de temperatura inicial , el cambiar a un porcentaje de aceptación mayor parece permitir una mayor exploración lo que genera una mayor cantidad de resultados factibles sacrificando tiempo de ejecución.

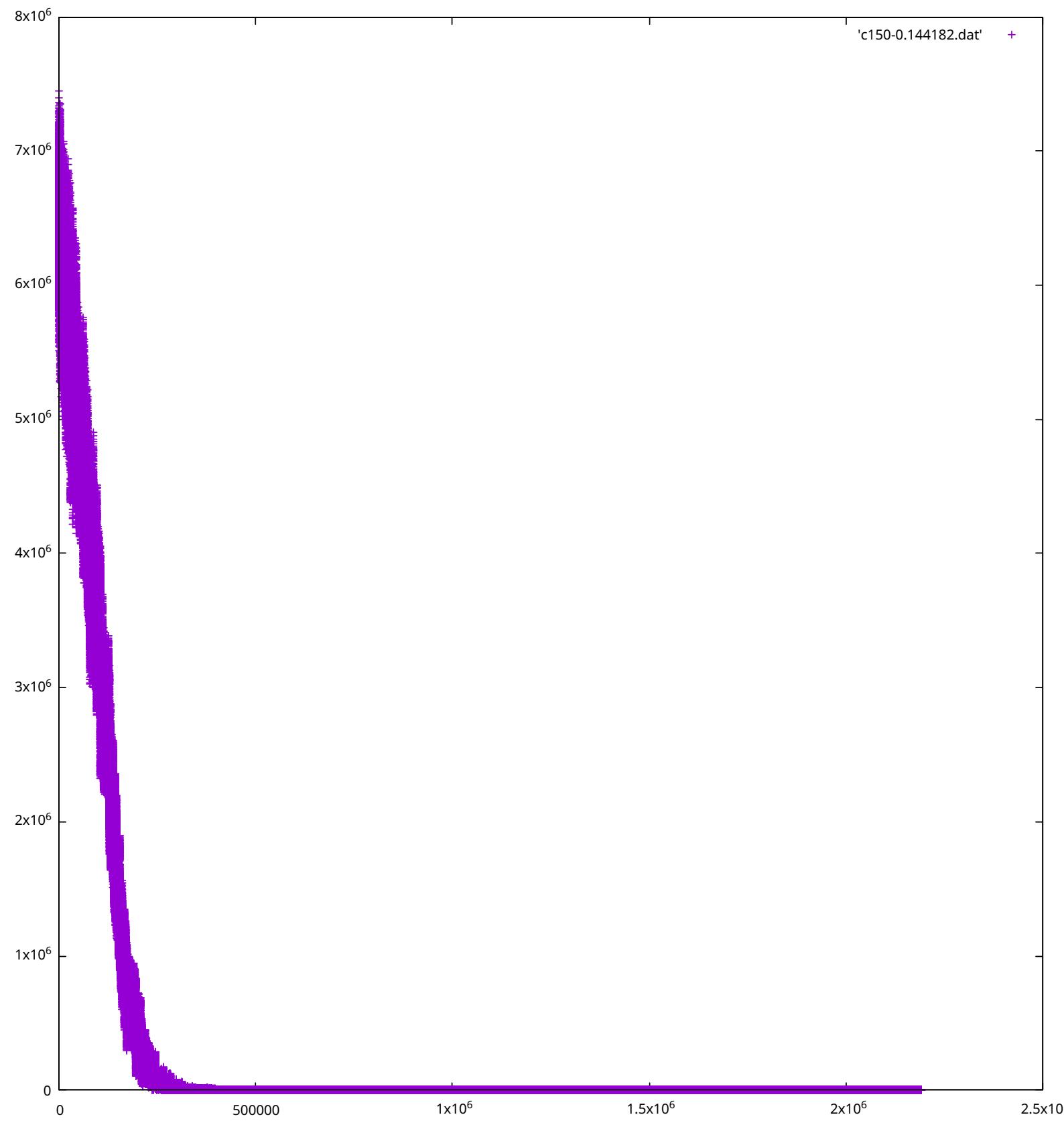
Del mismo modo la temperatura inicial generada por el algoritmo para el caso de 40 ciudades es más grande que la generada para 150 ciudades. En el caso de 40 el rango de tempraturas generadas era de 200,000-300,000 , sin embargo , para el de 150 ciudades el rango era de 120,000-150,000 approximandate. Este patrón igual se puede ver al correr la instancia de 200 ciudades, el cual generaba temperaturas dentro de un rango de 60,000-90,000.

Esto parece contraintuitivo, pero si consideramos que casos con menores ciudades es más rápido encontrar soluciones factibles por lo que tenemos un mayor rango de exploración, en cambio el encontrar soluciones factibles con un mayor número de ciudades es más difícil, así que una vez cuando el algoritmo encuentra mejores soluciones entonces es mejor permanecer en un rango amenorizado . Del mismo modo el porcentaje de soluciones factibles parece bajar al incrementar el número de ciudades, ya que cada vez se vuelve más difícil encontrar soluciones factibles.

Tal vez el hecho de combinar el generar una temperatura incial con un mayor porcentaje de aceptación y tener una epsilon con un valor bastante bajo, llevó a que durante la exploración encontrara un camino o dirección favorable y se especializara de manera decisiva por el final hasta llegar a la solución más mínima.

## Gráficas de las mejores soluciones





## Reproducir los resultados

Una parte importante y la razón por la que se usan semillas al generar los números aleatorios, es que los resultados encontrados sean reproducibles.

Esta es la lista de variables usadas para obtener estos resultados, y cómo reproducirlos.

### Para el caso con 40 ciudades

Las variables usadas fueron

- Evaluación : 0.2637132755109184
- El camino:  
[979, 493, 329, 163, 172, 496, 815, 657, 168, 1, 656, 2, 653, 490, 654, 7, 816, 982, 332, 820,  
981, 333, 3, 165, 6, 5, 978, 817, 4, 489, 492, 491, 984, 331, 164, 327, 980, 186, 483, 54]
- Epsilon : 0.002
- Phi : 0.98
- Tamaño de lote : 1000
- Semilla de vecinos aleatorios : 16214040050592208640
- Semilla para generar la solución inicial : 16042267250931732492

Para poder correr esta instancia, ejecutar lo siguiente desde la línea de comandos.

```
cd target/release  
. ./simulated_annealing --cities 40 -e 0.002 --neigh 16214040050592208640  
--init 16042267250931732492
```

### Para el caso con 150 ciudades

- Evaluación : 0.14418250861220022196
- El camino:  
[54, 652, 1075, 483, 171, 77, 183, 346, 75, 821, 512, 179, 671, 16, 520, 186, 190, 675, 340, 502,  
151, 828, 12, 1038, 339, 826, 444, 17, 164, 11, 501, 25, 492, 491, 499, 347, 489, 4, 174, 817,  
23, 176, 668, 352, 978, 5, 6, 988, 165, 3, 981, 990, 333, 991, 351, 185, 22, 676, 665, 173, 2,  
656, 184, 815, 172, 182, 496, 19, 505, 168, 508, 986, 9, 1, 829, 657, 663, 661, 832, 667, 507,  
653, 344, 490, 654, 26, 820, 345, 332, 181, 14, 982, 187, 816, 678, 823, 7, 673, 163, 329, 509,  
493, 979, 837, 995, 984, 1003, 349, 331, 662, 8, 999, 674, 334, 343, 510, 660, 20, 680, 825,  
500, 985, 504, 511, 327, 670, 350, 840, 336, 297, 980, 191, 822, 1001, 74, 166, 658, 666, 818,  
655, 819, 330, 1073, 169, 1037, 326, 328, 167, 495, 494]
- Epsilon : 0.0001
- Phi : 0.98
- Tamaño de lote : 1000

- Semilla de vecinos aleatorios : 11613177925935108993
- Semilla para generar la solución inicial : 8480295797754924014

Primero se debe de cambiar la siguiente línea de código dentro del archivo `sa.rs`

```
105 let mut batch_average =
```

después, ejecutar lo siguiente desde la línea de comandos.

```
cd target/release
./simulated_annealing --cities 150 --neigh 11613177925935108993
--init 8480295797754924014
```

## Conclusiones

Incluso para un sistema relativamente chico la importancia de la planeación y el diseño sigue siendo igual de importante, reduce el tiempo de implementación y elimina el tiempo gastado por refactorización, del mismo modo con las pruebas unitarias.

## References

- [1] D. Bertsimas and J. Tsitsiklis. Simulated annealing. *MIT Statistical Science*, 8, 1993.