Aproximando Soluciones al Problema del Agente Viajero

Juan Antonio López Rivera

March 23, 2018

1 Introducción

El objeto de estudio de las Ciencias de la Computación son los algoritmos, sin embargo informalmente podría decirse que a lo que nos dedicamos es a resolver problemas (mediante automatización). Proponemos soluciones a éstos, y no solo interesa mostrar que funcionen correctamente - interesa saber qué tan buenas son.

Para ello se diseñó la métrica de la O grandota (o análisis de complejidad). Ésta nos permite estudiar cómo crece la cantidad de operaciones de un algoritmo conforme crece su entrada; más aún, induce una clasificación de los problemas según su complejidad.

Decimos que un problema está en la categoría P si una máquina de Turing determinista los resuelve en tiempo polinomial. Informalmente, son los problemas que podemos solucionar rápido (relativamente). Ésta clase se encuentra contenida en la de problemas NP, aquellos que son solventables por una maquina de Turing no-determinista en tiempo polinomial. Éstos no pueden solventarse 'rápidamente' (o más bien no se conoce tal algoritmo) pero al menos se puede encontrar una solución.

Conocemos que $P \subset NP$, sin embargo la contención opuesta (que efectivamente significaría que P=NP) sigue siendo una de las incógnitas más grandes de nuestra ciencia. Es uno de los siete Problemas del Milenio que pueden lucrar $1,000,000 de dólares a quien lo resuelva; una respuesta – afirmativa o negativa – tendría implicaciones profundas.

Los problemas más difíciles de la categoría NP son los NP-completos. Informalmente, son aquellos cuya solución es verificable en tiempo polinomial, pero en sí calcular la solución ocupa tiempo no polinomial (formalmente, comprende a aquellos problemas que se clasifican tanto en NP como en NP-duros). Los NP-duros son aquellos problemas que son *al menos* tan difíciles como los problemas más difíciles en NP (H está en los NP-duros si cualquier problema L en NP se puede reducir en tiempo polinomial a H).

Sin embargo, que un problema sea NP-completo sólo significa que es imposible hallar un algoritmo que resuelva el problema para cualquier entrada en un tiempo 'corto' – no significa que no podamos acercarnos. Es aquí donde entran las heurísticas.

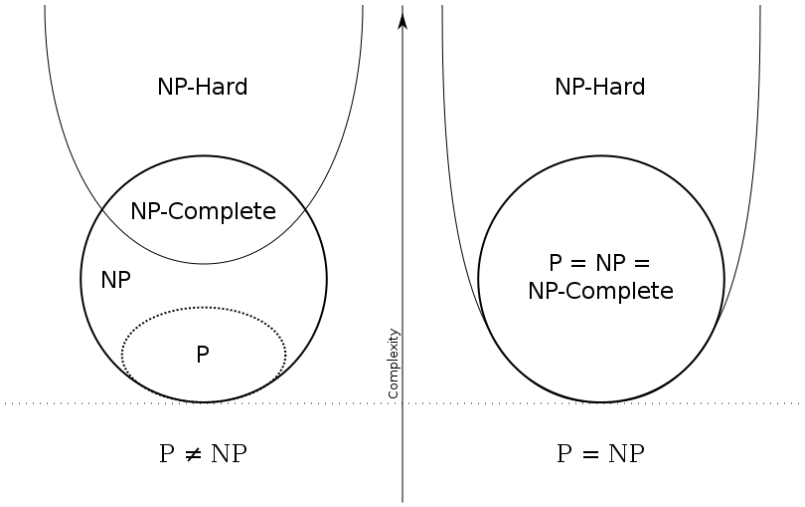


Figure 1: Diagrama de Venn de ambas posibilidades.

Las heurísticas son atajos, reglas empíricas que podemos tomar para resolver un problema cuando de otra manera sería muy lento o no hallaría soluciones exactas. Intercambian precisión u optimalidad por tiempo de ejecución. En éste trabajo experimentamos con la heurística de aceptación por umbrales para hallar soluciones al problema del agente viajero.

2 El Problema del Vendedor Ambulante o TSP

Hay diversas enunciaciones de este problema; la que se consideró para éste trabajo es la siguiente. Dado un grafo no dirigido ponderado $G = \{V, E\}$ y un subconjunto ordenado de sus vértices S, ¿cuál es el camino hamiltoniano sobre S de menor peso?

Éste problema cae dentro de la categoría de NP-duros. Sin usar una heurística la única manera de resolverlo es explorando todas las posibles soluciones, las cuales crecen factorialmente (para N nodos existen $\frac{(N-1)!}{2}$ posibilidades).

2.1 Soluciones vecinas y costos

Éstos dos conceptos serán de utilidad para definir la heurística.

S' es una solución vecina de S si es una permutación de la misma, y ellas difieren a lo mas en la posición de 2 nodos. Por ejemplo, para $S = \{1, 2, 3, 4, 5\}$ una solución vecina es $S' = \{2, 1, 3, 4, 5\}$.

Asimismo definimos el costo de una solución S como la suma de los pesos de las aristas entre todo par de vértices consecutivos de S. Un par de detalles aquí. Es posible que no exista arista entre algún par de vértices. En cualquier caso es necesario poder asignarle un costo a S; por otro lado, es deseable que dicho costo refleje en cierta manera la factibilidad de la solución – por ejemplo, quisiéramos que una solución S_1 que tiene n pares de vértices no conexos tenga un costo menor que alguna S_2 con $m > n$ pares de vértices no conexos. Por ello si dos vértices no son adyacentes le asignaremos un peso grande o castigo. Así definimos la función de costo f de una permutación $S = \{v_1, ..., v_k\}$:

$$f(S) = \sum_{i=2}^{k} \frac{w(v_{i-1}, v_i)}{A_S(|S| - 1)},$$

dónde $w(u, v)$ es la función de peso aumentado que devuelve el peso de la arista entre u y v si son adyacentes o bien el valor de castigo si no lo son, y A_S es el promedio de peso de todos los pares conectados de S.

Ésta definición induce una gráfica donde los vértices son todas las posibles permutaciones de un subconjunto de vértices, y dos soluciones son adyacentes si son vecinas. Lo que haremos en la heurística es explorar ésta gráfica de una manera semialeatoria.

3 La heurística de Aceptación por Umbrales

Ésta heurística es aplicable no solo al vendedor ambulante, sino a cualquier problema donde podemos definir un costo y vecinos para cada solución.

Dada una solución S con un costo C_S, una manera de mejorarla (encontrar una con menor costo) es buscar en sus vecinos alguna S' con un $C'_S < C_S$. Asimismo, podemos buscar en los vecinos de S' alguna solución mejor, y seguir haciéndolo hasta que ya no podamos mejorarla (que haya alguna C_n donde ningún vecino suyo tenga menor costo). Ésta técnica la conocemos como barrido, o descenso por gradiente. Una desventaja clara de éste método es que solamente llegará a costos mínimos locales – no hay manera de garantizar que la solución donde se detenga sea la mejor posible (de costo mínimo absoluto), y efectivamente en la mayoría de los casos no lo será.

Para solventar ésto y poder escapar de mínimos locales (explorando más posibles soluciones) podemos emplear la aceptación por umbrales. Funciona de la siguiente manera.

Se tiene un valor T (por temperatura, el nombre se debe a la inspiración original para éste algoritmo) y una solución S. Una solución vecina S' es aceptada si $C'_S < C_S + T$. De esta manera introducimos cierta aleatoriedad a las soluciones que aceptamos (ya que generamos las soluciones vecinas aleatoriamente), y a la vez abrimos la posibilidad de escapar de mínimos locales. Asimismo, la temperatura T irá decrementándose paulatinamente, con el fin de centrarse en soluciones más y más óptimas. Ésto se hace actualizando el valor de T cada que se completa un lote, con la

asignación $T = \theta T$ (donde $0 < \theta < 1$ es el factor de enfriamiento). La ejecución termina una vez que $T < \epsilon$ (pasamos el límite inferior de temperatura).

En resumen, el flujo es el siguiente. Partiendo de una solución inicial exploramos sus vecinos, aceptando una solución si a lo más empeora por T. Ésto lo hacemos hasta que la cantidad de soluciones aceptadas exceda el tamaño del lote definido. En este punto, si el promedio de soluciones aceptadas ha mejorado, decrementamos la temperatura T, de lo contrario hacemos otro lote. Éste proceso se repite hasta que la temperatura T sea menor a ϵ.

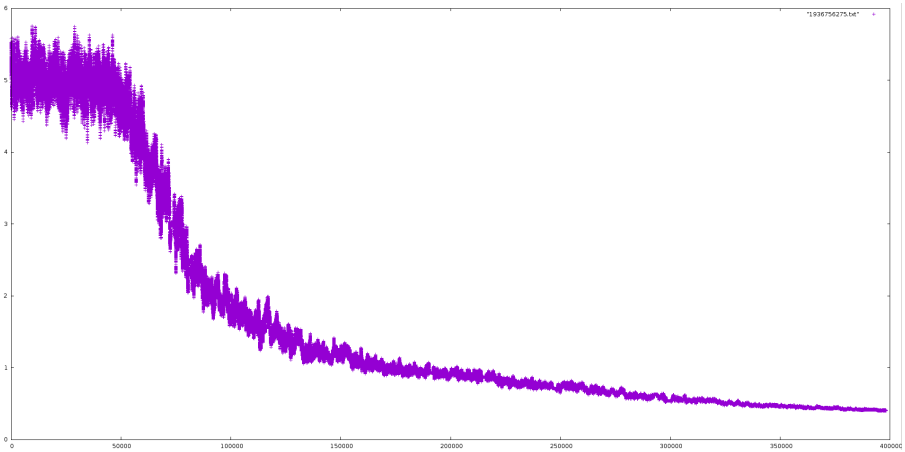


Figure 2: Gráfica de costos aceptados a lo largo de una ejecución de la heurística (para 150 ciudades con la semilla 1936756275).

El algoritmo ocupa varios valores iniciales que determinan la calidad de soluciones halladas, así como qué tanto tiempo correrá una instancia dada. La dificultad principal del mismo se encuentra en sintonizar éstos parámetros para encontrar un balance apropiado entre calidad de los resultados y tiempo de ejecución. Una ϵ muy baja, un tamaño de lote grande, y/o un factor de enfriamiento muy cercano a 1, nos darán muy buenas soluciones pero tardarán un tiempo excesivo en completar cada corrida, disminuyendo el número total de corridas que podemos realizar en un tiempo razonable. Sin embargo si empeoramos demasiado éstos valores las soluciones halladas tienen costos no muy buenos, o de plano no son factibles. Tener que hallar los valores adecuados para todos los parámetros es la mayor desventaja de ésta heurística.

Existen unas cuantas variaciones del algoritmo. Una sencilla de implementar es el tabú – no es posible aceptar una solución si implica regresarse a una solución aceptada con anterioridad. En la práctica ésto no tuvo gran influencia en el desempeño del programa (la probabilidad de regresarse es pequeña dada la cantidad enorme de vecinos existentes). Una que sí mejoró las soluciones halladas fue incorporar el barrido (mencionado anteriormente, para una solución S se explora su vecindad hasta no poder mejorar más el costo) cada vez que se halla una mejor solución. Su desventaja es que incrementa notoriamente el tiempo de ejecución de cada corrida.

4 Implementación en Julia

Julia es un lenguaje de programación dinámico de alto nivel diseñado para análisis numérico. Por omisión es compilado JIT (Just In Time), pero es posible precompilarlo nativamente (o generar ejecutables mediante el paquete PackageCompiler). Parte de la motivación de usar Julia era ver cómo se compara en eficiencia contra otros lenguajes; mi impresión es que es adecuado.

4.1 Insumos

Se partió de una base de datos con 1092 ciudades, donde cada una cuenta con nombre, país, población, y coordenadas geográficas reales. Además, existen 123,403 caminos no dirigidos entre pares de éstas ciudades. A partir de éstos se generó una matriz de adyacencias (con -1 en los elementos donde no existen conexiones). Éste código y otras funciones relacionadas se encuentran en el archivo graphs.jl.

4.2 Costos

Se implementó la función de costo descrita en la sección 2.1. Aquí cabe mencionar un par de detalles que optimizan el tiempo de ejecución. En primera, para una solución dada tanto el promedio de pesos como el valor de castigo son constantes. Solo se calculan una vez y se pasan como parámetro a las funciones que los ocupan.

Otro detalle es que no siempre es necesario calcular el costo mediante esa función. Si ya se conoce el costo de una solución, para todos los vecinos se puede calcular su costo en tiempo constante – sólamente hay que recalcular los pesos de los caminos afectados por el intercambio de los dos índices (a lo más son 4, hay que tener cuidado de los casos en que los índices se encuentran al inicio de la solución, al final, o son consecutivos). Su implementación en el sistema es la siguiente:

```
function cost_fast(g::Graph, s::Array{Int,1}, pun::Float64,
         avg::Float64, u::Int, v::Int, c::Float64)
    w = c * (avg * (length(s)-1))
    if u > 1
        w -= augmented_weight(g, s[u-1], s[u], pun)
        w += augmented_weight(g, s[u-1], s[v], pun)
    end
    if u < length(s)
        w -= augmented_weight(g, s[u], s[u+1], pun)
        w += augmented_weight(g, s[v], s[u+1], pun)
    end
    if v > 1
        w -= augmented_weight(g, s[v-1], s[v], pun)
        w += augmented_weight(g, s[v-1], s[u], pun)
    end
    if v < length(s)
        w -= augmented_weight(g, s[v], s[v+1], pun)
        w += augmented_weight(g, s[u], s[v+1], pun)
    end
    w / (avg*(length(s)-1))
end
```

Es importante mencionar que ésta función no devuelve exactamente el valor de costo: puede desviarse casi infinitesimalmente (después de 17 decimales para ser exactos) por cuestiones de aritmética de punto flotante de las que adolecen todas las computadoras. Por ello no podemos usar ésta función indiscriminadamente, aunque parezca pequeña la imprecisión a lo largo de una corrida sí se llega a desviar significativamente el costo si solamente ocupamos ésta. El punto medio al que llegamos es usarla para explorar la vecindad de una solución, y si se acepta alguna, calculamos su costo con la función completa antes de seguirnos sobre ella.

4.3 El Recocido Simulado

El código relativo a éste algoritmo se encuentra en el archivo *simulated_annealing.jl*. Consta de 4 funciones, *calc_lot* (calcula un lote, ie. busca vecinos partiendo de una solución inicial hasta aceptar L soluciones), *acceptance_by_thresholds* (contiene el ciclo principal del archivo), y dos relativas al método del barrido (*sweep_once* y *sweep* – la última manda a llamar a la anterior hasta que deje de devolver una mejor solución).

Los métodos del recocido simulado se implementaron como describe la sección 3, con un par de addendas. En primera, en la función *calc_lot* se introdujo otro contador a (de 'attempts') y un *max_value* (al que le asignamos arbitrariamente el valor de L^2); incrementamos a cada que generamos un vecino (sea aceptado o no) y si $a > max\_value$ terminamos la corrida. Ésto con el fin de que no se quede trabado si no halla soluciones aceptables (suele pasar cuando la temperatura es muy baja). Además, en la misma función se agregó código que guarda en un archivo de texto los costos aceptados a lo largo de una corrida – de ésta manera se generó la figura 2, que fue escogida al azar de las múltiples que se generaron.

Se incorporó el barrido de la siguiente manera. En el método principal (*acceptance_by_thresholds*) se va guardando la mejor solución hallada en toda la corrida; cada que se mejora, hacemos barrido

sobre esa solución para hallar el mínimo local. No ocupamos ésta solución hallada en el recocido, ya que como es un mínimo usualmente solo atora el sistema en esa vecindad, especialmente si la temperatura ya está algo baja. Básicamente el barrido existe independiente del sistema de recocido, solamente lo ocupamos para garantizar que hallamos la mejor solución en cierto vecindario.

4.4 El ciclo principal

Se encuentra en el archivo *run.jl*. Básicamente lo que hace es guardar los valores especificados en el archivo de settings (sección 4.5) en un diccionario. Utiliza la semilla dada como maestra para generar una semilla aleatoria para cada corrida. En un punto hubo problemas con el generador de números aleatorios de Julia: después de un cierto número de semillas generadas (como 500) se empezaban a repetir en grupos de 4. Por éste motivo se implementó la siguiente forma de generar semillas:

```
seeds = randperm(num_runs*3)
for i in 1:num_runs
    current_seed =  10000seeds[i]^2 + 100seeds[i+num_runs]^2 + seeds[i+2num_runs]^2
                    + master_seed
```

La función $randperm(N)$ devuelve un arreglo con elementos del 1 al N permutados aleatoriamente. Luego para cada corrida, se agarran 3 elementos del arreglo y se suman – después de multiplicarse por constantes distintas – lo cual debería garantizar que no haya repeticiones (o por lo menos, la probabilidad de que se repitan es muy muy baja). Se le suma el valor de $master\_seed$ para que las subsemillas usadas en diferentes conjuntos de corridas no se repitan (no hay tanto problema allí, ya que si se modificaron los parámetros iniciales la heurística va a tomar un camino distinto rápidamente aunque las semillas ya se hayan usado con anterioridad).
En fin, se ejecuta el método $acceptance\_by\_thresholds$ el número especificado de veces, guardando en un archivo de texto e imprimiendo a la consola los resultados de cada corrida. Más detalles en la siguiente sección.

4.5 Ejecución del Programa

La clase principal es main.jl. Ésta recibe como parámetro desde la consola la ruta al archivo de settings que se desee utilizar. Por ejemplo:

```
> julia main.jl settings/settings.txt
```

En el archivo de settings se pueden modificar los siguientes parámetros iniciales: semilla maestra ($seed$), archivos de insumos (db_cities_file y $db\_connections\_file$), subconjunto de ciudades ($cities$), tamaño de lote (L), temperatura inicial (T), límite inferior de temperatura ($epsilon$), factor de enfriamiento ($theta$), factor de castigo ($punishment\_factor$), y número total de corridas deseadas ($runs$). Se pueden comentar lineas empezándolas con un #. Por ejemplo, el siguiente es el contenido de un archivo de settings válido:

```
seed=1234567890
db_cities_file=cities.txt
db_connections_file=connections.txt
cities=22,74,109,113,117,137,178,180,200,216,272,299,345,447,492,493,498,505,521,572,
607,627,642,679,710,717,747,774,786,829,830,839,853,857,893,921,935,986,1032,1073
L=1000
T=.01
epsilon=0.0005
theta=0.99
punishment_factor=3.5
runs=1000
```

Es posible ejecutar una única corrida de dos maneras, mediante *main.jl* con runs=1, o bien mediante el archivo $run\_once.jl$. Éste último es útil para replicar corridas, ya que no realiza una permutación aleatoria antes de empezar. Inicia el recocido simulado con las ciudades en el orden en que aparecen en el archivo de settings, a diferencia de *main.jl* que hace un shuffle antes de cada

corrida. Ésta permutación inicial se guarda para cada corrida, para facilitar la reproducción de resultados. Dos archivos de settings de los mejores resultados hallados para 40 y 150 se incluyen en el repositorio, se pueden ejecutar de la siguiente manera:

```
> julia run_once.jl settings/settings-40-best.txt
```

y

```
> julia run_once.jl settings/settings-150-best.txt
```

El primero completa su ejecución en segundos, sin embargo el segundo tarda alrededor de 30 segundos.

Adicionalmente, se incluyen algunos scripts para visualizar información de las corridas. El archivo *solution_cost.jl* imprime en consola el costo de una solución dada. Ejemplo de uso:

```
> julia solution_cost.jl 830,117,337,669,39,310,497,178,498,410,583,515,243,478
```

Análogamente, se puede usar *list_to_polyline.jl* para obtener las coordenadas reales de las ciudades en formato polilinea (que es sencillamente una lista de listas). Ésta puede ser visualizada por cualquier sistema de trazado de mapas, un ejemplo usando leaflet se encuentra en la carpeta *polyline* (con éste se generaron algunas figuras de la sección 5).

Los resultados se guardan en la carpeta *results*, en un archivo de texto con el nombre de la semilla maestra especificada (solo hay que tener cuidado de no usar la misma semilla y sobreescribir resultados – igual es buena costumbre no ocupar la misma semilla maestra en diferentes corridas, para aleatorizar debidamente y no repetir resultados).

Adicionalmente, en *results/plots* se guardan archivos de texto con los costos de las soluciones aceptadas a lo largo de cada corrida. Éstos se pueden visualizar fácilmente con programas como gnuplot. El nombre de cada archivo corresponde a la subsemilla ocupada en esa corrida.

5 Resultados

Se ejecutó la heurística con dos subconjuntos de ciudades, uno de longitud 40 y uno de 150.

5.1 40

La mejor solución hallada tiene costo 0.6162358649273801. No es posible demostrarlo pero ya que varios llegamos a ésta, creemos que es la óptima (se puede demostrar mediante métodos probabilísticos que tan cerca se encuentra de la cota mínima).

Figure 3: Mapa del mejor camino hallado para el subconjunto de 40 ciudades.

Las ciudades escogidas son: 22,74,109,113,117,137,178,180,200,216,272,299,345,447,492,493,498, 505,521,572,607,627,642,679,710,717,747,774,786,829,830,839,853,857,893,921,935,986,1032, 1073.

Y la permutación hallada es: 830,117,178,498,857,200,747,679,921,109,1073,607,180,786,627,74,299,1032,774,137,853,642,113,572,710,492,272,521,935,493,893,216,22,717,345,447,839,505,986,829.

5.2 150

La mejor solución hallada tiene costo 0.2725278545820132. Creemos que la mejor solución se encuentra cerca del 0.26.

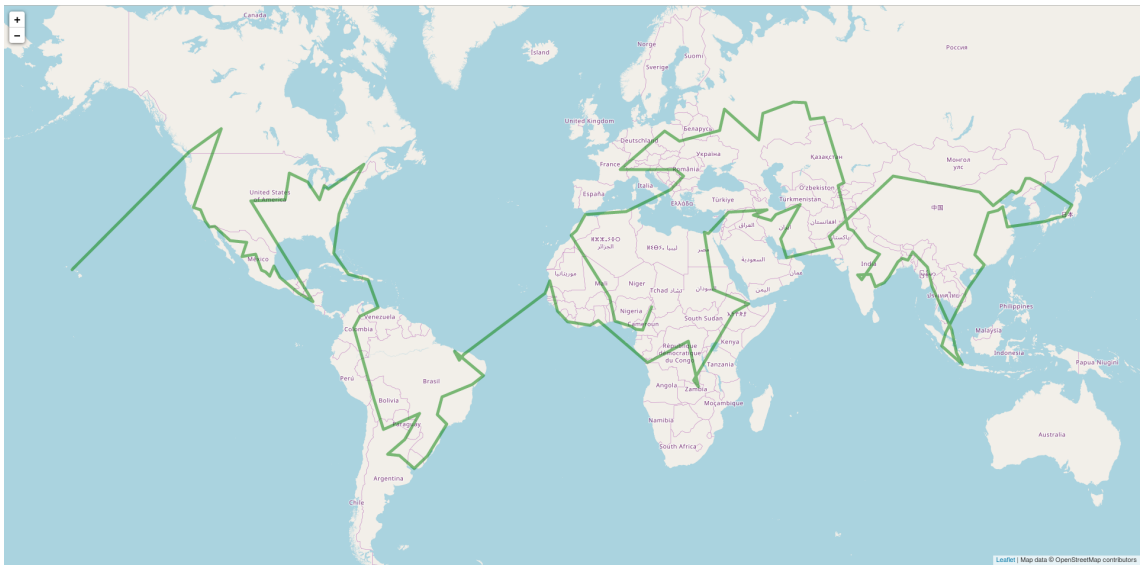
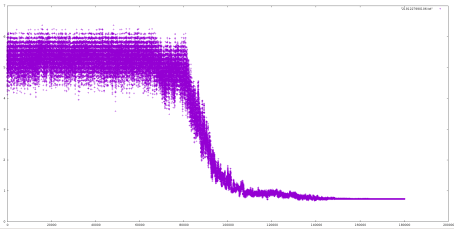


Figure 4: Mapa del mejor camino hallado para el subconjunto de 150 ciudades. Podemos notar que tiene varios cruces y retornos que probablemente no son óptimos.

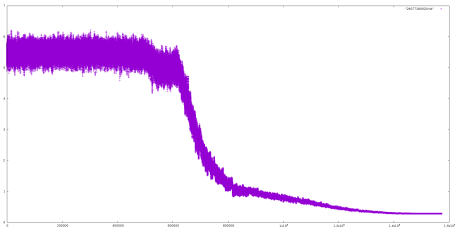
Las ciudades escogidas son: 10,12,13,39,40,44,67,69,74,79,86,93,94,95,96,98,107,111,115,117,119,151,156,159,178,186,189,190,203,206,213,219,233,234,243,254,256,257,271,278,281,284,293,297,299,302,308,310,313,319,323,332,337,341,347,355,358,359,385,393,395,398,408,410,413,414,438,447,454,470,477,478,486,489,492,497,498,505,515,517,528,535,547,552,557,563,571,576,582,583,587,591,607,611,614,627,637,639,654,669,670,675,679,688,690,697,700,705,712,717,735,741,748,757,766,785,795,796,830,833,837,853,854,857,865,871,872,876,877,885,890,893,896,919,933,935,955,957,981,986,991,1002,1007,1016,1034,1037,1057,1069,1072,1080

Y la permutación hallada es: 830,117,337,310,669,39,178,497,498,410,515,583,243,478,13,1057,535,679,582,234,857,712,865,517,1072,313,700,454,552,955,833,547,94,119,398,1037,341,557,395,607,785,156,690,189,1002,10,854,93,637,627,293,302,385,86,74,408,115,299,44,151,107,933,571,795,748,69,470,67,40,735,96,896,492,614,1016,347,203,489,877,393,528,358,688,486,505,281,79,111,986,611,95,639,233,278,477,1069,98,355,447,576,654,587,332,219,206,872,717,213,308,957,413,591,991,256,319,981,271,919,254,159,885,563,766,893,935,257,837,284,705,323,796,741,890,871,670,414,1034,1080,853,697,876,757,297,438,359,12,675,186,190,1007

Los archivos con los parámetros iniciales usados para ambas corridas se encuentran en el repositorio, como se mencionó en la sección 4.5.



(a) Serie de tiempo de soluciones aceptadas en la mejor corrida de 40.



(b) Serie de tiempo de soluciones aceptadas en la mejor de 150.

Figure 5: En ambas podemos notar que pasó un tiempo considerable antes de que empezara a mejorar el costo. Ésto es debido a una temperatura inicial demasiado alta.

6 Conclusiones y Trabajo Futuro

Me parece que la heurística de aceptación por umbrales tiene potencial. Como ya se mencionó sí tiene la desventaja de ser algo 'artesanal', en el sentido de que se deben modificar los parámetros de manera experimental para encontrar los que funcionan para cada instancia del problema. Personalmente me agrada, creo que en muchos casos – especialmente en problemas NP-duros quizás – el algoritmo que optimiza resultados y tiempo de ejecución debe incorporar tanto determinismo como aleatoriedad de una manera inteligente, y creo que el recocido simulado hace precisamente eso.

Julia aún se encuentra en su versión 0.7 (ni han llegado a la 1.0), y se nota en varios aspectos. Los errores no son muy descriptivos, y la documentación me pareció dispareja e insuficiente. Además los arreglos empiezan en 1. Sin embargo, una vez que se logró echar a andar el sistema se ejecuta de maravilla. Me agradó la sintaxis limpia (y que se puedan usar caracteres unicode para nombrar variables y funciones – parece una bobada pero se ve bonito), y que el gestor de paquetes es muy simple de usar. Y el sistema de compilado JIT es prometedor, si se implementa correctamente podría ser lo mejor de ambos mundos (entre interpretado y compilado). Probablemente no ocupe el lenguaje en un buen rato pero me interesará ver cómo ha avanzado en unos años.

Me gustaría experimentar con la misma heurística aplicada a algún otro problema de optimización combinatoria.