DEPARTAMENTO DE COMPUTACION

Facultad de Ciencias Exactas y Naturales - UBA

TP 2: Problema del Viajante de Comercio (TSP)

2021-1C Algoritmos y estructuras de datos III

Integrante	LU	Correo electrónico
Dacunda Ratti, Jerónimo	710/18	jero.d.r22@gmail.com
Alonso Rehor, Ignacio	195/18	arehor.ignacio@gmail.com
Martinez, Juan Ignacio	558/18	nacho9martinez@hotmail.com

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires
Ciudad Universitaria - (Pabellón I/Planta Baja)
Intendente Güiraldes 2610 - C1428EGA
Ciudad Autónoma de Buenos Aires - Rep. Argentina
Tel/Fax: (++54 +11) 4576-3300
https://exactas.uba.ar

1. Introducción

El problema del Viajante de Comercio (**TSP** por sus siglas en inglés) consiste de lo siguiente. Un viajante debe realizar un recorrido de un conjunto determinado de ciudades, visitando exactamente una vez cada una y finalizando de vuelta en la ciudad de origen. De las posibilidades existentes para ordenar el recorrido, se desea obtener la **mejor**. La condición de **mejor** depende del contexto particular de cada problema. Podría ser el más corto, el más rápido, el que tenga menos peajes, etcétera. En términos de grafos, el objetivo es hallar un ciclo hamiltoniano de longitud mínima en un grafo completo con longitudes asociadas a sus aristas. Formalmente, dado un grafo $G = (V, X)$ con longitudes asignadas a las aristas, $l\colon X \to R^{\geq 0}$, queremos determinar un ciclo hamiltoniano de longitud mínima, es decir, encontrar C^0 tal que

$$l(C^0) = \text{mín}\{l(C) \ : \ C \text{ es un ciclo hamiltoniano de } G\}.$$

El problema en sí mismo es una generalización del problema del ciclo hamiltoniano, el cual consiste en hallar, para algún grafo G, un ciclo que pase por cada vértice de G exactamente una vez. Si quisiéramos averiguar si un grafo G es hamiltoniano podríamos transformarlo en una instancia del problema **TSP** definiendo un grafo $G' = (V, X')$ completo y $l\colon X \to R^{\geq 0}$ como $l(e) = 1$ para $e \in X$ y $l(e) = n + 1$ para $e \in X' \backslash X$. La respuesta de **TSP** en G' será un ciclo hamiltoniano de longitud n si y sólo si G tiene un ciclo hamiltoniano. Utilizando **TSP** de esta forma, resolvemos el problema de saber si G es hamiltoniano.

Existen muchas posibles situaciones de la vida real que podrían modelarse para ser resueltas usando el **TSP**. A continuación se describen algunos ejemplos:

- **Recolección de basura**: podríamos representar con los vértices a los distintos puntos de recolección, con las aristas a las distancias entre estos y luego buscar minimizar la distancia que recorre el camión para pasar por todos. Hay varios problemas equivalentes muy similares, como reparto de paquetes o distintas logísticas de transporte;
- **Schedulling de procesos**: podríamos tomar a las distintas tareas que debe realizar un procesador como los vértices, asignarle a las aristas el tiempo que toma realizar cada tarea, y finalmente intentar minimizar el tiempo total que toma completarlas;
- **Fabricación de circuitos electrónicos**: existen problemas como el de la conexión de chips en placas de circuitos, donde la idea es minimizar la cantidad de cable necesaria para unir todos los puntos de una placa sin que haya interferencias. De esta forma, podríamos tomar como vértices a los pins donde se colocan los cables y asignarles a las aristas la cantidad de cable necesaria para unir estos pins.

En cuanto a la complejidad del problema, no se conocen algoritmos polinomiales que resuelvan el **TSP**. Este pertenece a la clase $\mathcal{NP} - C$ de problemas de optimización combinatoria.

En la Figura 1 se exhibe un ejemplo de una instancia del problema **TSP** y una solución óptima para esta.

(a) Instancia de TSP de 5 vertices

(b) Solución óptima de la instancia. Costo : 55

Figura 1: Solución de **TSP**

Si considerásemos grafos de 4 vértices, tendríamos 3 soluciones posibles correspondientes a los distintos circuitos hamiltonianos. Luego considerando grafos de 5 vértices, la cantidad de soluciones posibles aumenta a 12. Generalizando esta idea, si nuestro grafo tiene n vértices entonces tenemos $\Theta((n-1)!)$ posibilidades de recorridos distintos, ya que, como se trata de grafos completos, cualquier camino resulta ser una permutación del orden en que se recorren los vértices. Si quisiéramos aplicar fuerza bruta para hallar cuál de ellos es el **mejor**, los tiempos de ejecución se volverían prohibitivos para el tipo de instancias que nos interesan evaluar. Por este motivo, se debe recurrir a los algoritmos **heurísticos**.

El trabajo va a estar ordenado de la siguiente manera: primero en la Sección 2, se detallan las distintas **heurísticas** y **metaheurísticas** implementadas junto a un análisis de su **complejidad**. Luego, en la Sección 3 se presentan los **experimentos computacionales** con sus respectivas discusiones. Finalmente, en la Sección 4, daremos el cierre con las **conclusiones** a las que llegamos.

2. Algoritmos y Estructuras

Para poder operar cómodamente con los grafos definimos las siguientes estructuras en el archivo `types.h`[1]:

- un **Grafo** G está compuesto por un número $n \in \mathbb{N}$ que representa la cantidad de vértices de este; un número $m \in \mathbb{N}$ que representa la cantidad de aristas del grafo; y por ultimo una matriz $C \in \mathbb{R}^{n \times n}$, tal que $c_{ij} = c(v_i, v_j)$ si la arista $(v_i, v_j) \in E(G)$, y $c_{ij} = \perp$ caso contrario.
- un **Circuito** está compuesto por un vector de vértices que identifican el orden en el que se recorre el camino; y un número $c \in \mathbb{N}$ que representa el costo de recorrer el mismo.

2.1. Heurística del vecino más cercano

La heurística del vecino más cercano o *Nearest Neighbour*, resulta la más intuitiva de las heurísticas implementadas. Esta consiste en ir agregando el vértice más cercano al último agregado, en otras palabras, ir agregando el vecino más cercano.

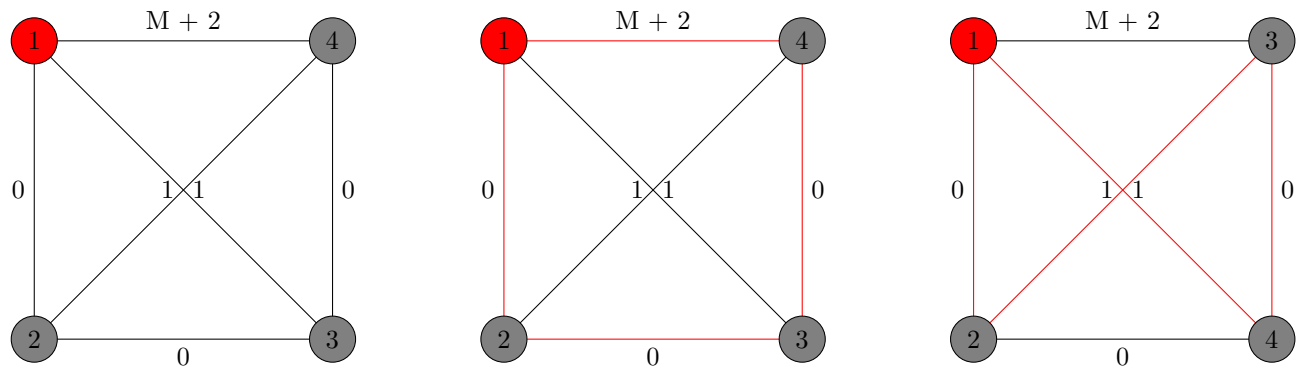
En nuestra implementación definimos el vértice 1 como el vértice inicial para todos los caminos. En cada paso del algoritmo agregamos al circuito el vértice que se corresponde al vecino más cercano. Repetimos esto hasta que la longitud del circuito sea igual a la cantidad de vértices. El pseudocódigo de la heurística es el siguiente:

Algorithm 1 Nearest Neighbour

```
1:  function NEARESTNEIGHBOUR(Grafo G)
2:      H ← circuito vacío de costo 0
3:      H ← H ∪ {v_1}
4:      while |H| < n do
5:          u ← último vértice de H
6:          nn = arg mín(c(u, v) : v ∈ G \ H)
7:          H ← H ∪ {nn}
8:          H.c ← H.c + c(u, nn)
9:      u ← último vértice de H
10:     H.c ← H.c + c(v_1, u)
11:     return H
```

Complejidad: El algoritmo tiene una complejidad de $\Theta(n^2)$ ya que en cada iteración, se debe buscar al vecino del ultimo vértice agregado, cuyo costo sea menor. En la iteración i la cantidad de vértices en $G \setminus H$ es $n - 1 - i$, y obtener el costo de la arista entre dos vértices es $\Theta(1)$. Al ejecutarse $n - 1$ veces el ciclo, la cantidad de veces que se ejecuta el cuerpo del ciclo es exactamente $\sum_{i=1}^{n-1} n - 1 - i$, que es $\Theta(n^2)$. Así mismo, las Lineas 9 y 10 son a lo sumo $\Theta(n)$. Juntando todo esto podemos concluir que la complejidad de la heurística del vecino más cercano es $\Theta(n^2)$.

Instancias patológicas: Si bien esta heurística puede resultar intuitiva, no es cierto que las soluciones que retorna sean óptimas, lo que es más, el costo de las soluciones que devuelve esta heurística pueden distar tanto como uno quiera del de la solución óptima. Un ejemplo de esto es el grafo de la Figura 2. Notemos que al aplicar la heurística del vecino más cercano, la diferencia entre el costo del circuito óptimo y el del circuito resultante es M, para un M arbitrariamente grande. En otras palabras, no existe una cota para que tan "mala" puede ser una solución retornada por esta heurística.



(a) Grafo original (b) Solución de `NN`; costo = $M + 2$ (c) Solución óptima; costo: 2

Figura 2: Instancia de TSP en la que la heurística *Nearest Neighbour* no retorna la solución óptima.

[1] El archivo se encuentra en el directorio `tp2/include`.

2.2. Heurística de Farthest Insertion

La heurística de inserción del más lejano (o *Farthest Insertion*) comienza por agregar un vértice inicial al circuito. Luego a cada paso, el algoritmo se encarga de extender el circuito según el siguiente criterio. Primero, se calcula el menor costo que realizan todos los vértices que todavía no están en el circuito, a algún vértice de este. De todos estos costos asociados a vértices fuera del circuito, se elige el máximo. A este vértice se lo conoce como el más lejano o *farthest*. Esta selección está resumida en la Linea 5 del Algoritmo 2. Si bien tomar el vértice más lejano podría parecer contra-intuitivo, en promedio suele retornar mejores soluciones que otras heurísticas de inserción como *Nearest Insertion* o *Cheapest Insertion*[1]. Una vez seleccionado el vértice con el cual extender el circuito, se debe elegir en que parte de este insertarlo. Formalmente, una vez elegido el vértice u, se buscan dos vértices v_i y v_{i+1} en nuestro circuito H de manera tal que se minimice

$$l(v_i, u) + l(u, v_{i+1}) \text{ - } l(v_i, v_{i+1}).$$

Luego, se intercala a u entre v_i y v_{i+1}. Esta inserción esta resumida en la Linea 6.

Algorithm 2 Farthest Insertion

1: **function** FARTHESTINSERTION(Grafo G)
2: $H \leftarrow$ circuito vacío de costo 0
3: $H \leftarrow H \cup \{v_1\}$
4: **while** $|H| < n$ **do**
5: $v_F \leftarrow \arg\max\Big\{c(v, H) \ : \ v \in G \setminus H\Big\}$
6: $H \leftarrow \min\Big\{H_i \cup v_F \cup H_{i+1} \ : \ 1 \leq i < |H|\Big\}$
7: **return** H

Complejidad: el costo de seleccionar el próximo vértice a insertar es $O(n^2)$ ya que hay que comparar todas las aristas disponibles para cada vértice aún no visitado, esto ocurre para todos los vértices, resultando en un costo de $O(n^3)$; luego la inserción en el circuito siguiendo el criterio previamente explicado cuesta $O(n)$ ya que recorremos una vez todo el recorrido buscando lugar más óptimo, por lo tanto la complejidad total del algoritmo es $O(n^3)$.

Optimalidad: Según [1], se ha probado que para toda heurística de inserción la solución obtenida es a lo sumo $\lceil \log_2(n) \rceil + 1$ veces la óptima. Esto quiere decir que Particularmente, *Farthest Insertion* tiende a obtener mejores soluciones que otros métodos de inserción. Se pudo observar que aplicada a problemas de `TSPLIB`, la heurística retorna soluciones con costos 16 % mayores que el óptimo. Sin embargo, no se ha probado que esta heurística posea cotas de peor caso mejores que otras heurísticas de inserción.

2.3. Heurística de Árbol Generador Mínimo

Lo primero que hace este algoritmo es buscar un AGM T del grafo que le pasamos. Luego recorre los vértices de T usando **DFS**. Por último, arma un circuito hamiltoniano recorriendo los vértices en este orden y agregando la arista desde el último vértice al primero.

Algorithm 3 Algoritmo de Christofides simplificado

1: **function** CHRISTOFIDESSIMPLIFICADO(Grafo G)
2: $H \leftarrow$ circuito vacío de costo 0
3: $T \leftarrow$ AGM de G
4: $H \leftarrow$ DFS de T, con raíz en v_1
5: $H.c \leftarrow$ calcularCosto(H)

Complejidad: En la Linea 3 obtenemos el AGM del grafo G, la complejidad de esto, mediante nuestra implementación del algoritmo de **Prim**, es $O(n^2)$. También tenemos que la complejidad de hacer **DFS** es $O(m + n)$, pero como se trata de un grafo completo esto también es $O(n^2)$ ya que $m \in \Theta(n^2)$. Finalmente calcular el costo del circuito cuesta $\Theta(n)$ ya que se trata simplemente de recorrer las aristas sumando el costo de estas. Podemos concluir que la complejidad de la heurística basada en AGM es $O(n^2)$.

Instancias patológicas: Al igual que en el caso de la heurística 2.1, *Nearest Neighbour*, es arbitrario lo lejanas que pueden resultar las soluciones obtenidas respecto de las óptimas. Los ejemplos observados en la Figura 2 pueden aplicarse de igual forma para la heurística de árbol generador mínimo.

2.4. Meta-heurística de búsqueda Tabú

La meta-heurística búsqueda Tabú (TS) se puede resumir en, dada una solución al TSP, explorar nuevas soluciones relativamente *cercanas* a esta con la esperanza de acercarse más a la solución óptima. Estas soluciones cercanas se las

denominan soluciones *vecinas*, y en este caso definimos el conjunto de soluciones vecinas a explorar como el conjunto 2-opt. Sea H un circuito hamiltoniano, el conjunto 2-opt es el conjunto de todos los circuitos que resultan al aplicar un *swap de aristas* a H. Un *swap de aristas* consiste en intercambiar los extremos de dos aristas de un circuito H. Además, para evitar cálculos repetitivos o ineficientes y orientar la exploración hacia mejores soluciones, se introduce el concepto de *memoria*, que consiste en mantener una estructura con cierta información acerca de las últimas soluciones visitadas. La elección de dicha información resulta determinante, y en nuestro caso nos vamos a enfocar en dos variantes: una memoria basada en recordar *circuitos* y otra basada en mantener los *swap de aristas*, que a la hora de la implementación los vamos a interpretar como pares de índices. En el algoritmo 4 se puede observar la diferencia entre ambas opciones en la Linea 4. También se puede observar que se tienen otros parámetros de entrada además del grafo:

- **M** es el tamaño máximo asignado a la memoria. En la Linea 12 podemos ver como, cuando llegamos a la capacidad establecida, eliminamos de esta el circuito (o swap de aristas) más antiguo (esto es se debe a que implementamos la memoria tabú como una cola);
- **K** es la cantidad máxima de iteraciones realizará nuestro algoritmo, lo que se conoce como *criterio de parada*;
- **p** es un porcentaje que indica qué proporción del vecindario 2-opt se considera a la hora de explorar nuevas soluciones. Cabe mencionar que el vecindario se elige de manera aleatoria. Esto último tiene por objetivo evitar caer en soluciones denominadas *óptimos locales*, que podrían alejarse del *óptimo global*.

Algorithm 4 Tabu Search

```
1:  function tabuSearch(Grafo G, uint M, uint K, float p)
2:      H ← NearestNeighbour(G)
3:      S ← H
4:      memoriaTabu ← cola ⟨Circuitos o swaps⟩
5:      vecindario ← pila⟨Circuitos⟩ de prioridades.
6:      for i = 0 to K do
7:          vecindario ← 2opt(H, G, p)
8:          while vecindario ≠ ∅ and vecindario.top() ∉ memoriaTabu do
9:              vecindario.pop()
10:         if vecindario = ∅ then continuar
11:         H ← vecindario.top()
12:         if |memoriaTabu| = M then memoriaTabu.pop()
13:         if H.c < S.c then
14:             S ← H
15:     return S
```

Complejidad: Para calcular la complejidad, primero veremos la complejidad de realizar los swaps de aristas para un circuito H a la hora de obtener el vecindario `2-opt`. Dado que, para cada swap de aristas se recalcula el costo del circuito resultante, la complejidad de este algoritmo es $O(n)$. Para obtener el vecindario 2-opt completo, debemos realizar $\Theta(n^2)$ swaps ya que la cantidad de pares de aristas de un circuito de longitud n es $\binom{n}{2}$. Por lo tanto, obtener el vecindario 2-opt es $\Theta(n^3)$. Como la cantidad de soluciones del vecindario es $O(n^2)$, y verificar si un circuito pertenece a la memoria tabú es $O(n \cdot M)$, la complejidad de obtener la próxima solución a explorar es $O(n^3 \cdot M)$. Por último, como se exploran a lo sumo K soluciones, la complejidad total de la meta-heurística tabú search resulta $O(n^3 \cdot M \cdot K)$.

3. Experimentación

3.1. Metodología

El objetivo de esta experimentación es evaluar el rendimiento de las heurísticas implementadas, y cómo se comportan estas frente a distintas instancias. Para esto hay que definir aspectos como las instancias a evaluar, el espacio de exploración de los parámetros de la meta-heurística TabuSearch, y cómo medir este rendimiento.

3.1.1. Heurísticas a evaluar

A continuación presentamos las heurísticas a evaluar en la experimentación.

- NearestNeighbour (`NN`)
- FarthestInsertion (`FI`)

- ChristofidesSimplificado (`AGM`)

Así mismo, se busca evaluar el rendimiento de la meta-heurística TabuSearch en sus dos variantes:

- TabuSearch - Memoria de soluciones (`TS-C`)
- TabuSearch - Memoria de estructura (`TS-S`)

3.1.2. Instancias a evaluar

Decidimos evaluar las instancias del dataset `MP-TESTDATA-TSPLIB`[2] que contiene instancias simétricas del **TSP**. Mas específicamente, para medir la calidad de las soluciones encontradas, utilizamos aquellas instancias para las que se conoce una solución óptima, o alguna cota inferior de la misma. Por cuestiones de rendimiento, decidimos experimentar únicamente con aquellas instancias cuya dimensión (entiéndase como la cantidad de vértices del grafo) no sea mayor a 100. Son 28 las instancias que cumplen estos criterios[3].

3.1.3. Espacio de exploración de parámetros de tabuSearch

Definimos el espacio a explorar de los parámetros de tabuSearch de la siguiente manera:

- Cantidad de iteraciones: $\{50, 100, 200, 500, 1000\}$.
- Tamaño de la memoria tabú: $\{50, 100, 500, 1000\}$.
- Porcentaje de vecindad **2opt**: $\{10, 20, 50, 80, 100\}$.

3.1.4. Métricas

Para poder evaluar el rendimiento de las distintas heurísticas decidimos utilizar las siguientes métricas.

- tiempo de ejecución de la heurística.
- *gap* relativo de la solución obtenida.

El tiempo de ejecución lo usamos para medir la eficiencia de las distintas heurísticas, mientras que el *gap* relativo sera nuestra manera de medir la calidad de las soluciones obtenidas. Este ultimo se define de la siguiente manera: Sea I una instancia de **TSP**. Sean x^h una solución de I obtenida mediante una heurística, y x^* una solución óptima (o en su ausencia una solución que realice una cota inferior) de I. Sea c la función objetivo de **TSP** (el costo del camino hamiltoniano), entonces

$$\text{gap relativo} = \frac{c(x^h) - c(x^*)}{c(x^*)}.$$

3.1.5. Especificaciones técnicas

Los experimentos se ejecutan en una computadora con CPU AMD(R) FX(TM) 6100 @ 3.3 GHz y 8 GB de memoria RAM. Las implementaciones de las heurísticas se realizan en el lenguaje de programación *C++*.

3.2. Rendimiento de heurísticas constructivas golosas

Se comenzó midiendo el rendimiento de las heurísticas implementadas (`NN`, `FI` y `AGM`) para distintas instancias del problema **TSP**. Como podemos ver en la Figura 3, para el conjunto de instancias evaluado, la soluciones obtenidas mediante la heurística `FI` son las que presentan el menor *gap* relativo, seguidas por las obtenidas mediante heurística `NN`, y por último las obtenidas por la heurística `AGM`. Los *gaps* relativos promedio de las soluciones de estas heurísticas los podemos ver en la Figura 4. En la Figura 5 podemos observar que la heurística `FI` resulta la menos eficiente en términos del tiempo de ejecución, mientras que las heurísticas `NN` y `AGM` poseen tiempos más acotados y muy similares entre sí, siendo esta última la más rápida de las tres.

3.3. Impacto de los parámetros en el rendimiento de tabuSearch

A continuación presentamos distintos experimentos cuyo objeto es determinar cuánto afectan a la calidad de las soluciones obtenidas mediante la meta-heurística búsqueda tabú, los distintos valores que pueden tomar los parámetros M, K y p, es decir, cómo se relacionan estos con el *gap* relativo de la solución obtenida. Asimismo, en cada caso se busca cuantificar la eficiencia en términos de tiempo de ejecución. En los distintos experimentos se fijan dos de los parámetros en cuestión y se evalúan distintos valores para el restante. En cada uno de ellos se evalúan las dos variantes implementadas de la meta-heurística tabuSearch: `TS-C` y `TS-S`.

[2] Zuse Institute Berlin, `http://elib.zib.de/pub/mp-testdata/tsp/tsplib/`.

[3] Para más detalles sobre la obtención y el procesamiento de estas instancias, referirse a los archivos `tp2/exp/descargar_instancias.ipynb` y `tp2/exp/lectura_instancias.ipynb`.

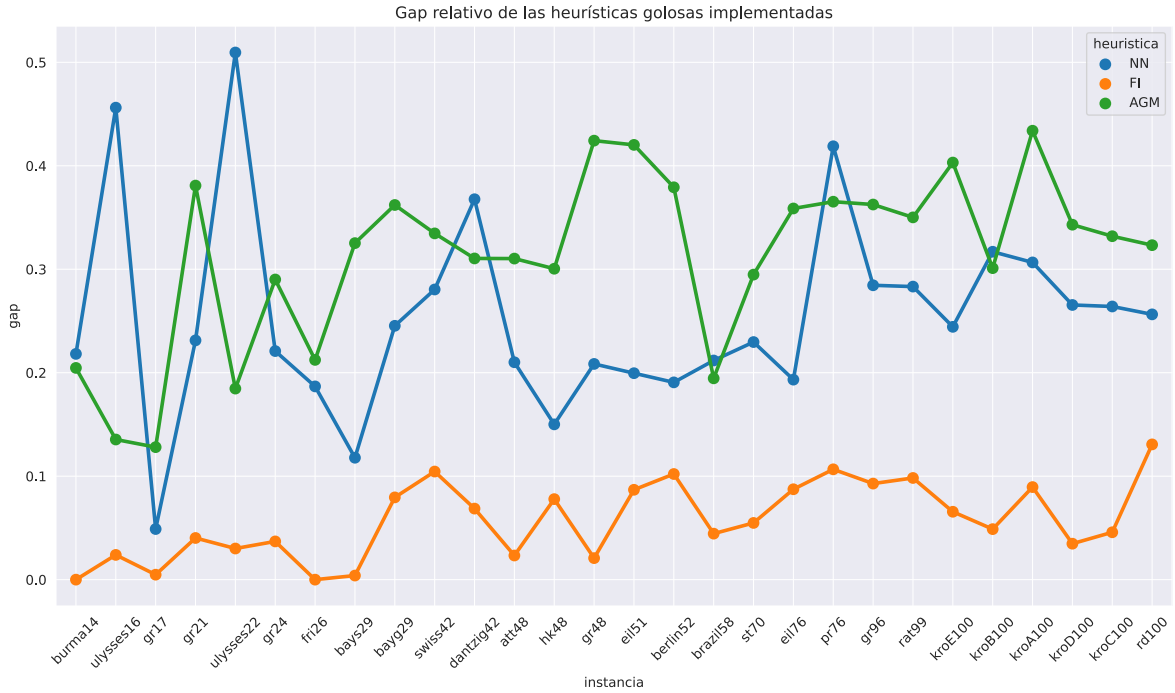


Figura 3: *gap* relativo de las heurísticas golosas implementadas

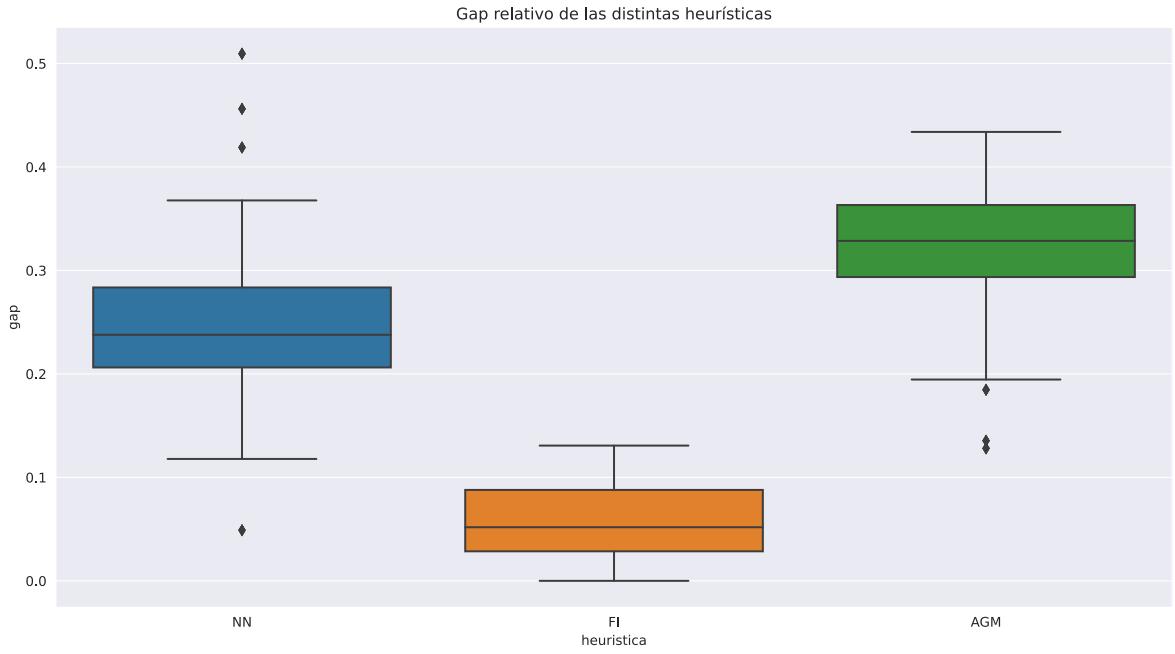


Figura 4: *gap* relativo de las heurísticas golosas implementadas

3.3.1. Variación del tamaño de la memoria tabú

Este experimento consiste en evaluar el impacto de distintas capacidades máximas que puede tomar la estructura de *memoria*. Para ello se fijan los parámetros $K = 1000$ y $p = 100$ y se consideran los siguientes valores de M:

$$\{50, 100, 500, 1000\}.$$

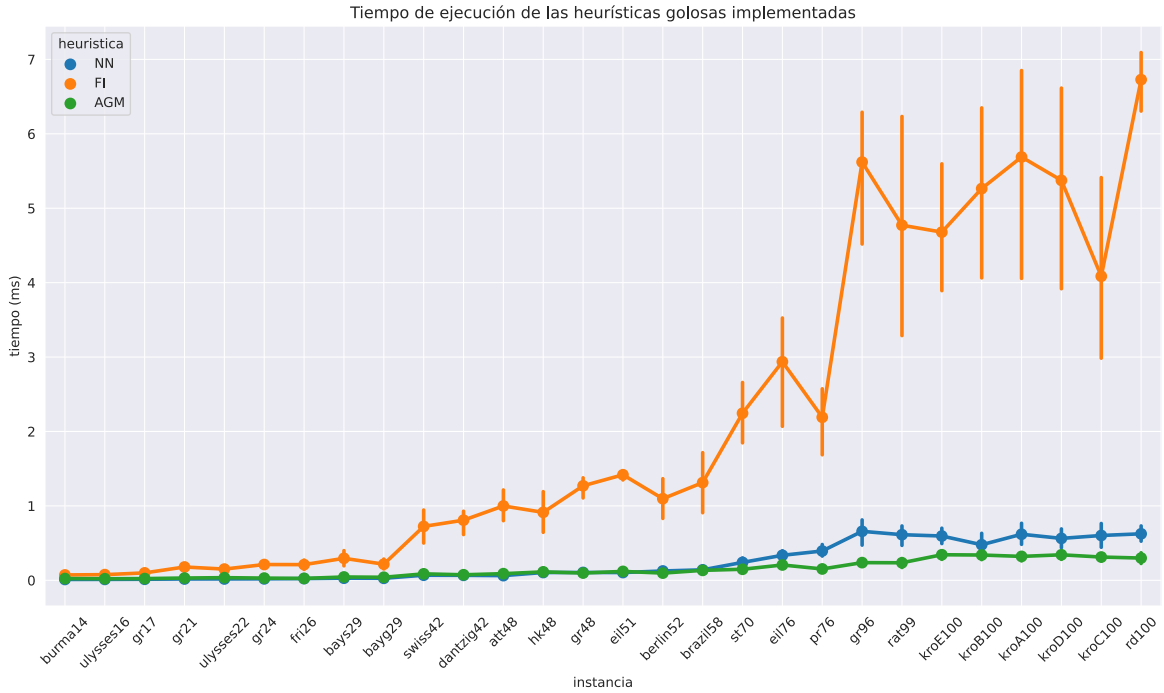


Figura 5: tiempo de ejecución de las heurísticas golosas implementadas

Podemos ver en la Figura 6 que a partir de $M = 100$ el *gap* relativo de las soluciones se mantiene estable. Si bien los tiempos de ejecución no parecen verse alterados, tal como muestra la Figura 7, no ocurre lo mismo con la complejidad espacial que sí se verá incrementada en proporción a M.

3.3.2. Variación de la cantidad de iteraciones

Este experimento consiste en evaluar el impacto del parámetro que limita la cantidad de soluciones exploradas (o iteraciones) que realiza tabuSearch. Para esto, fijamos los parámetros $M = 50$ y $p = 100$ y consideramos los siguientes valores para K:

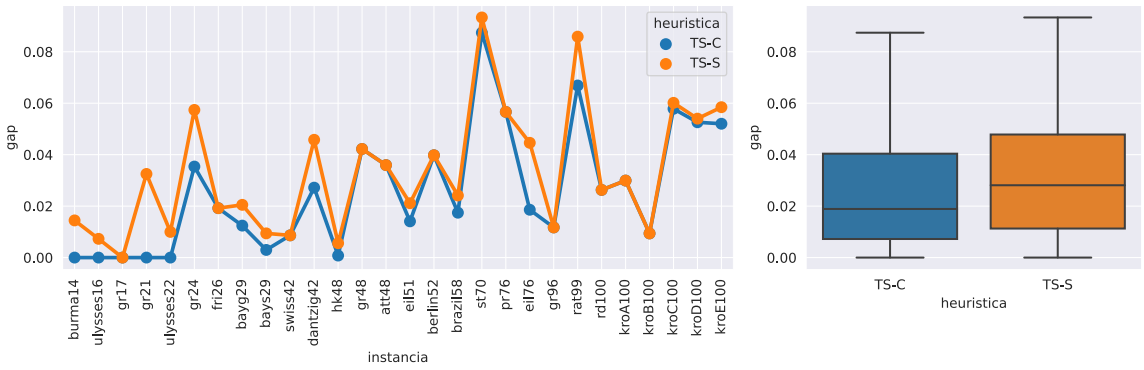
$$\{50, 100, 500, 1000\}.$$

En la Figura 8 podemos observar que, salvo para una cantidad muy reducida de iteraciones realizadas, la calidad de las soluciones obtenidas se mantiene aproximadamente en los mismos valores para todas las instancias evaluadas. Esto no es el caso con los tiempos de ejecución de la heurística: a medida que aumenta la cantidad de iteraciones realizadas, también lo hace el tiempo de ejecución. Esto se puede observar en la Figura 9.

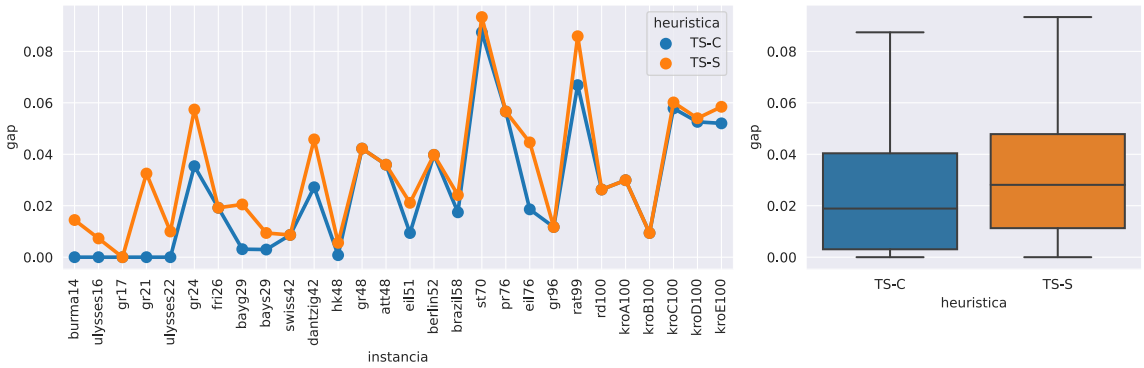
3.3.3. Variación del porcentaje del vecindario 2-opt

En este experimento buscamos evaluar el impacto que tiene el porcentaje del vecindario `2-opt` que consideramos en cada iteración. Con dicho fin, se fijan los parámetros $M = 50$ y $K = 1000$ y se varía a $p \in \{10, 20, 50, 80, 100\}$. Elegimos esos valores para fijar M y K porque consideramos que conforme aumenta el porcentaje de vecinos a considerar, crece la cantidad de colisiones con la memoria tabú, y esto puede impactar en los tiempos de ejecución. Nuestra hipótesis es que al ser la memoria tabú de un tamaño bastante menor al número total de vecinos de una solución, aumentando el porcentaje que consideramos, se producirán más colisiones con dicha memoria, y por lo tanto se deberán visitar numerosos vecinos hasta encontrar uno que no figure en la memoria. Así mismo, buscamos visualizar el impacto en términos de *gap* relativo de la solución obtenida.

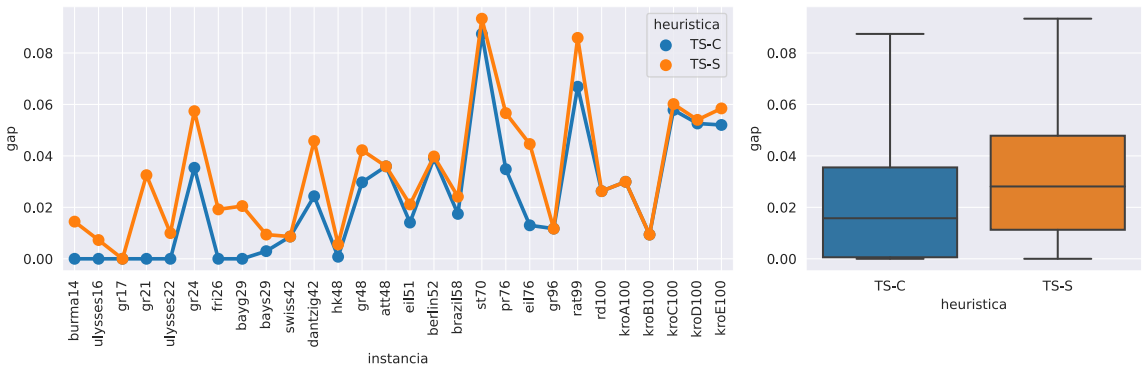
En la Figura 10 podemos observar que, a partir de $p = 20$, el *gap* relativo de las soluciones encontradas empieza a aumentar. Asimismo, en la Figura 11d podemos ver que el *gap* relativo es máximo cuando p es igual a 100, es decir, cuando estamos considerando el vecindario `2-opt` completo. Esto no es ninguna sorpresa ya que considerar todo el vecindario, y de este quedarse con la mejor solución, no es más que una búsqueda local, que está sesgada a soluciones óptimas locales. Por último podemos ver, en la Figura 11, que el tiempo de ejecución crece a medida que aumentamos p, alcanzando su valor más elevado con $p = 100$ en la Figura 11d, lo cual refuerza nuestra hipótesis.



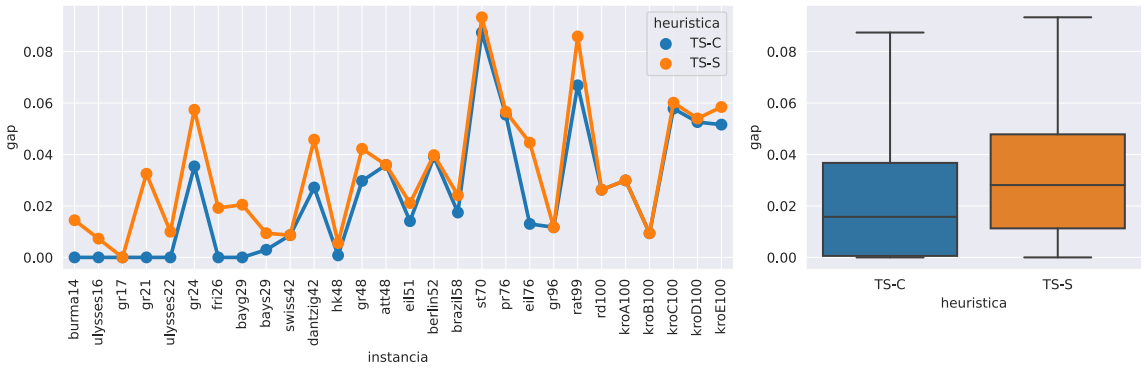
(a) $M = 50$



(b) $M = 100$

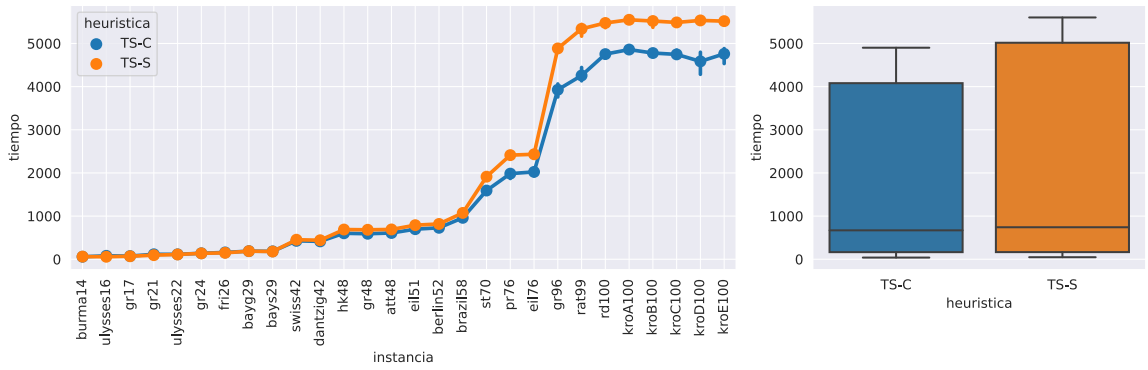


(c) $M = 500$

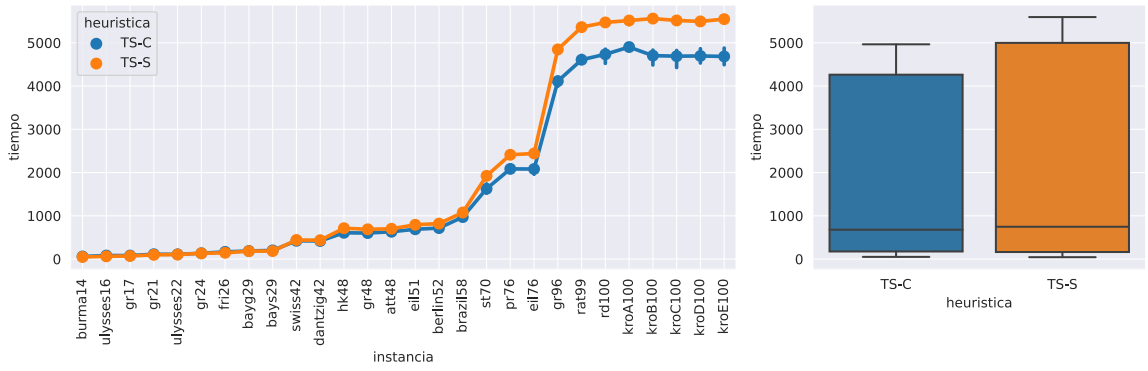


(d) $M = 1000$

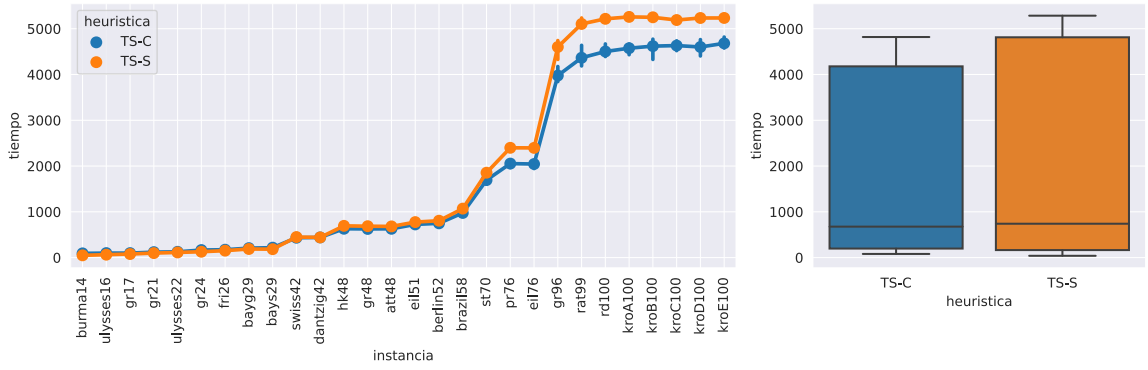
Figura 6: *gap* relativo para memoria de *circuitos* vs *swaps* variando M



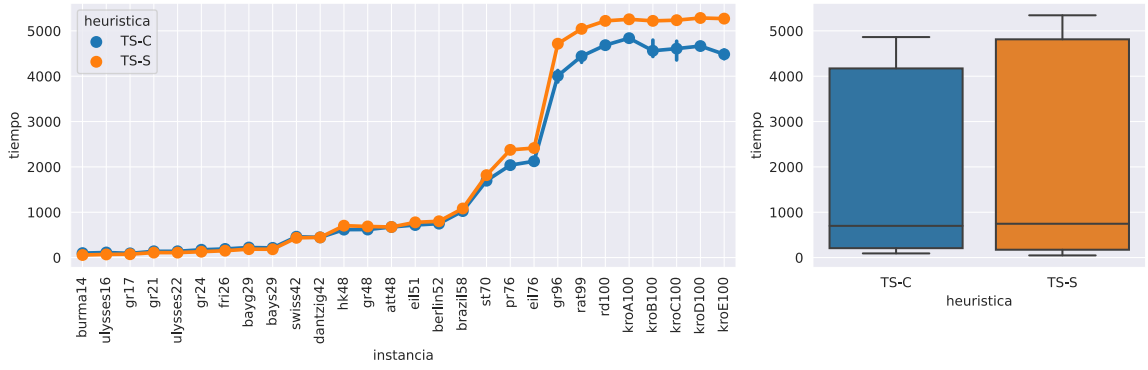
(a) $M = 50$



(b) $M = 100$

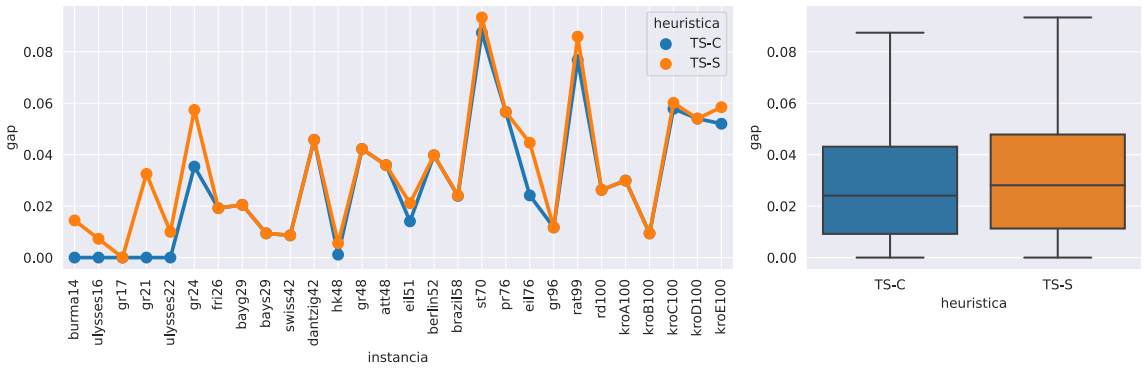


(c) $M = 500$

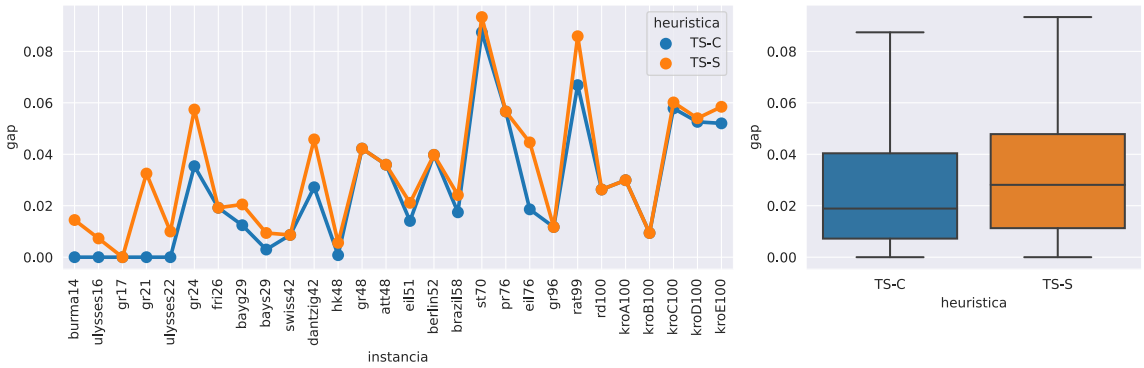


(d) $M = 1000$

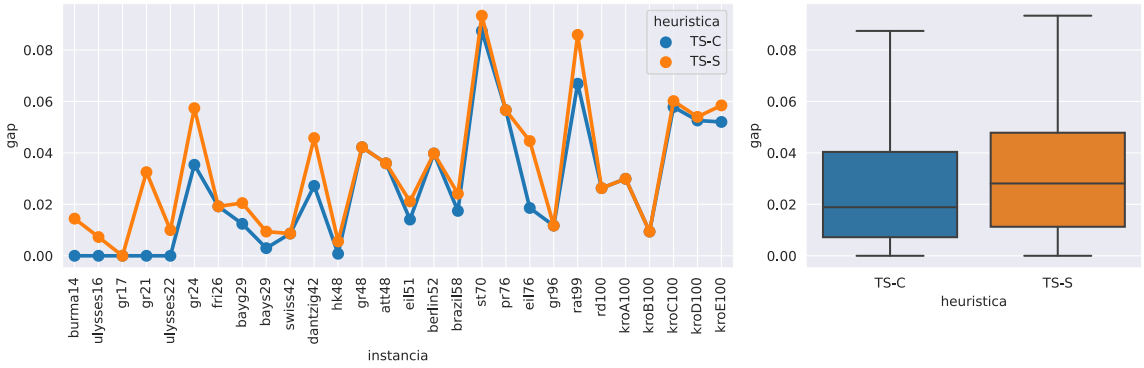
Figura 7: tiempo de ejecución para memoria de *circuitos* vs *swaps* variando M



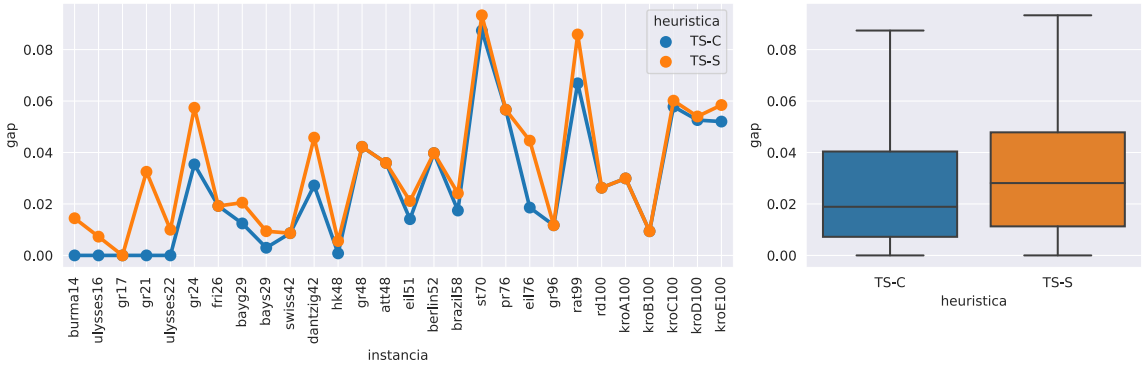
(a) $K = 50$



(b) $K = 200$

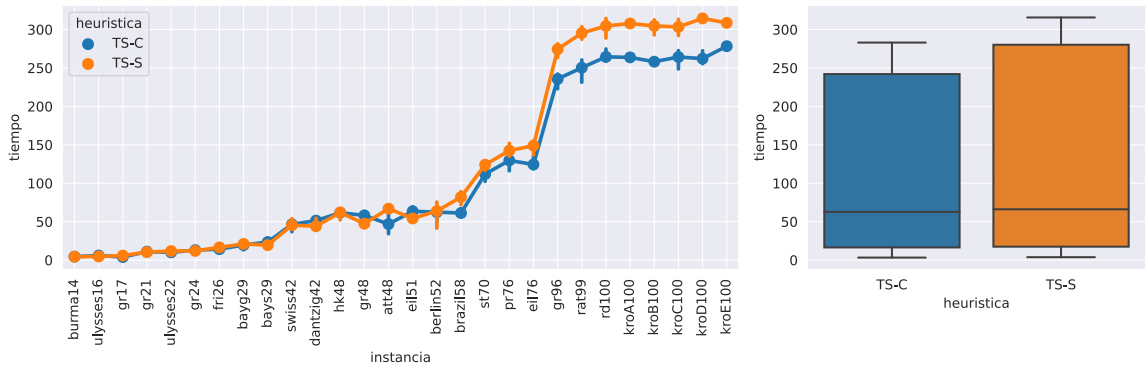


(c) $K = 500$

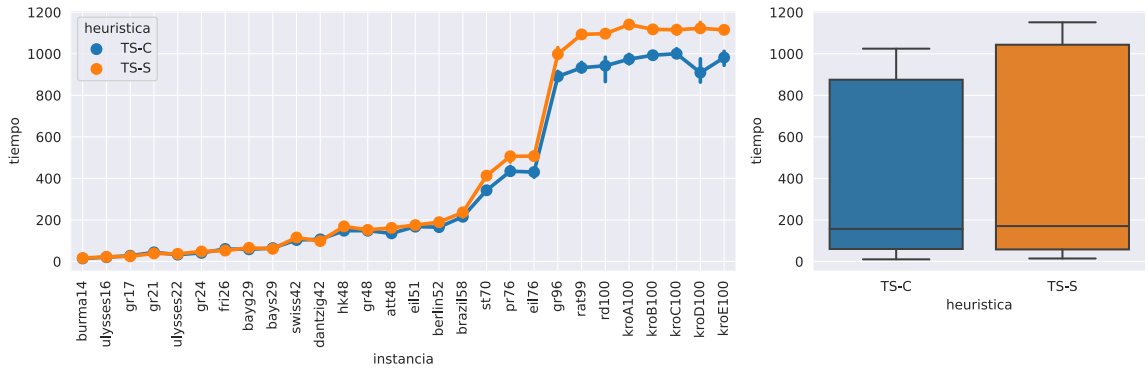


(d) $K = 1000$

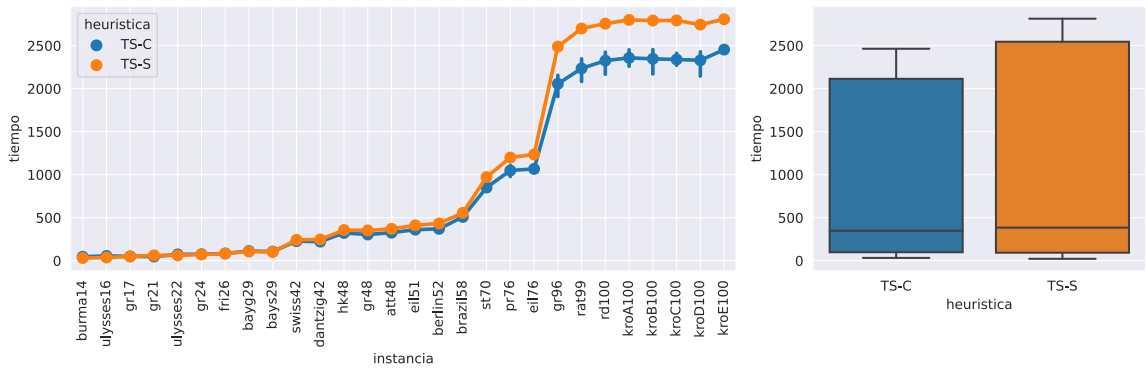
Figura 8: *gap* relativo para memoria de *circuitos* vs *swaps* variando K



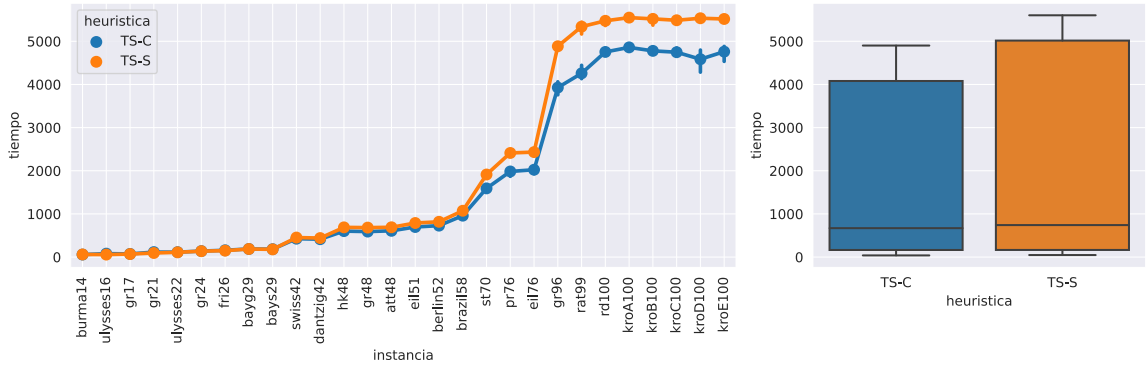
(a) $K = 50$



(b) $K = 200$

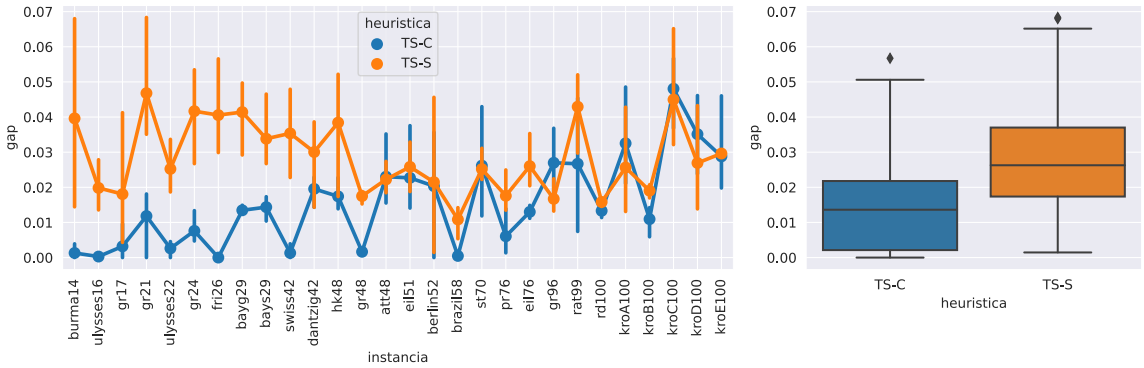


(c) $K = 500$

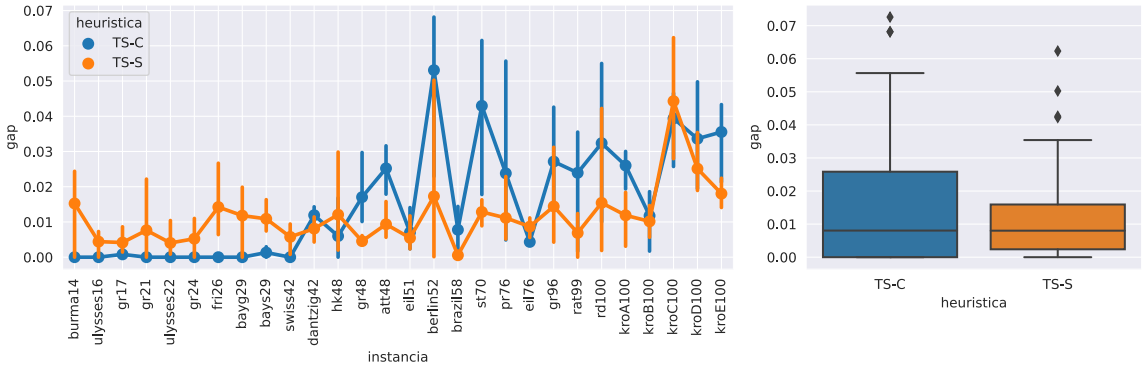


(d) $K = 1000$

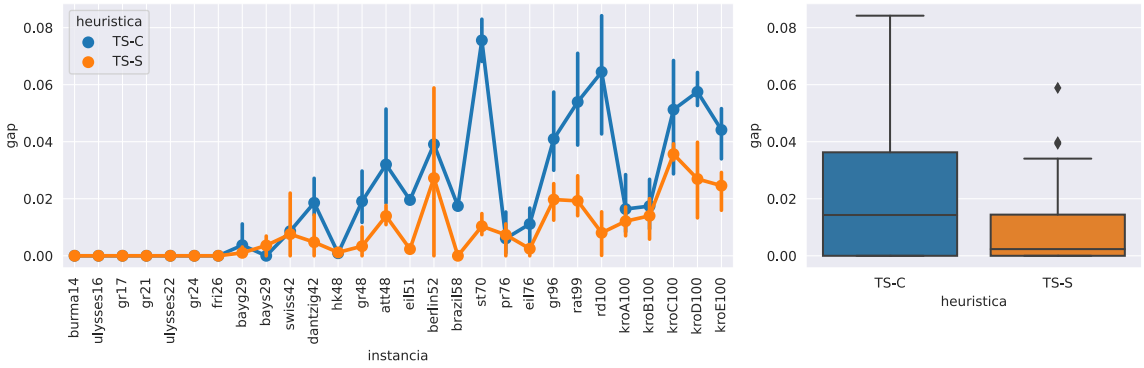
Figura 9: tiempos de ejecución para memoria de *circuitos* vs *swaps* variando K



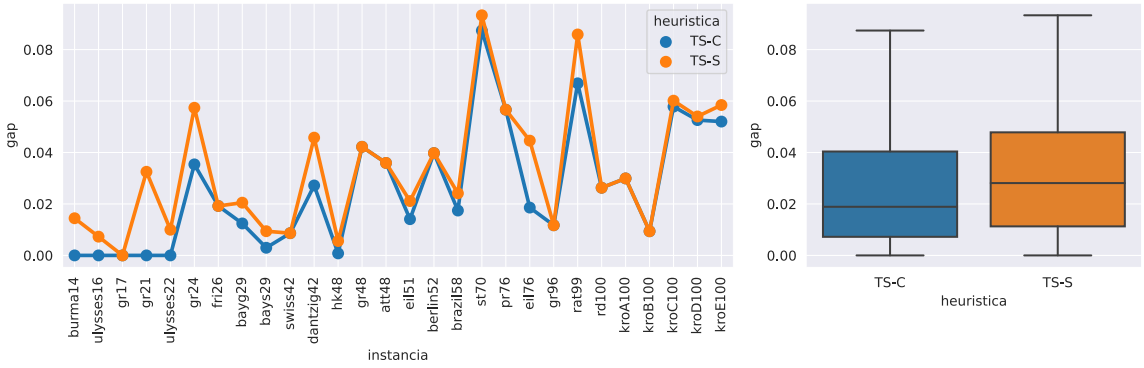
(a) $p = 10$



(b) $p = 20$

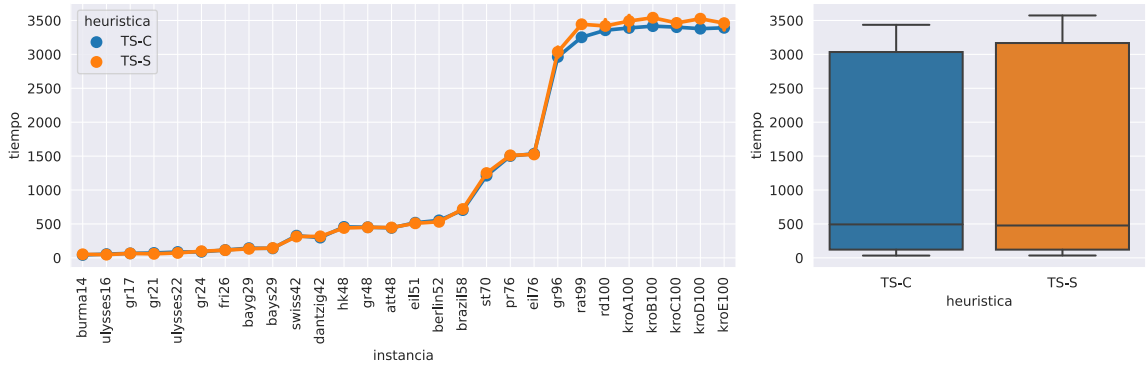


(c) $p = 50$

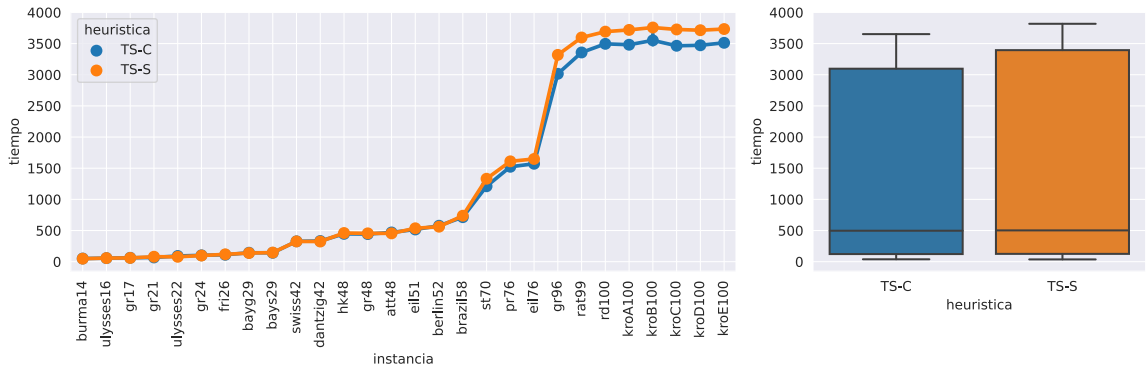


(d) $p = 100$

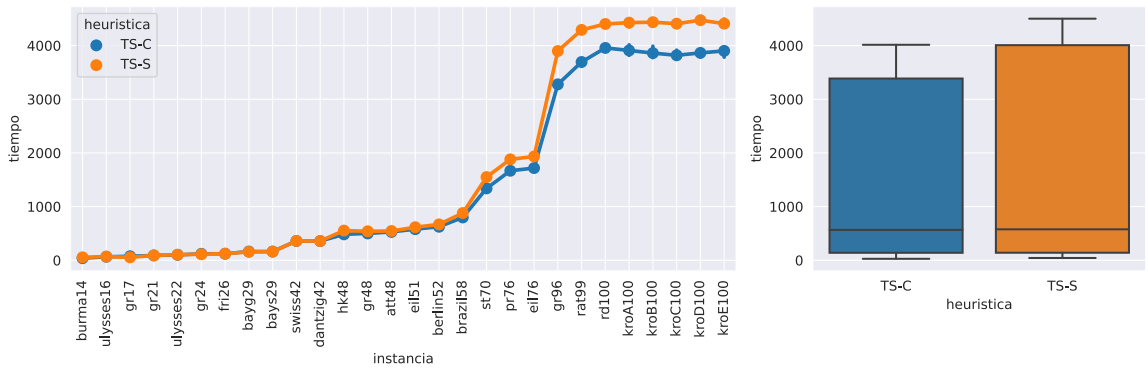
Figura 10: *gap* relativo para memoria de *circuitos* vs *swaps* variando p



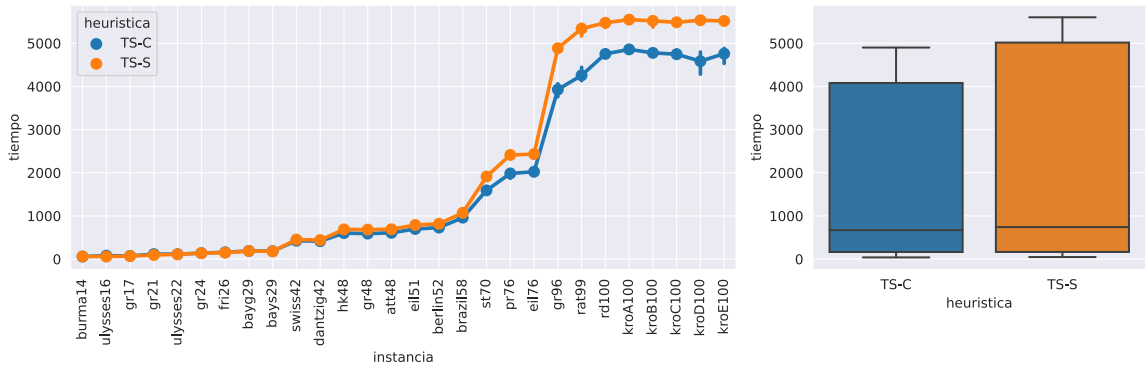
(a) $p = 10$



(b) $p = 20$



(c) $p = 50$



(d) $p = 100$

Figura 11: tiempos de ejecución para memoria de *circuitos* vs *swaps* variando p

3.4. Parámetros óptimos

Al cabo de la experimentación, definimos los parámetros óptimos de la meta-heurística tabuSearch como $p = 20$, $M = 100$ y $K = 200$. Elegimos estos parámetros ya que consideramos que minimizan el *gap* relativo de las soluciones encontradas, sin resultar en tiempos de ejecución muy altos. Usando estos valores, realizamos un experimento final donde contrastamos el *gap* relativo y el tiempo de ejecución para todas las heurísticas implementadas.

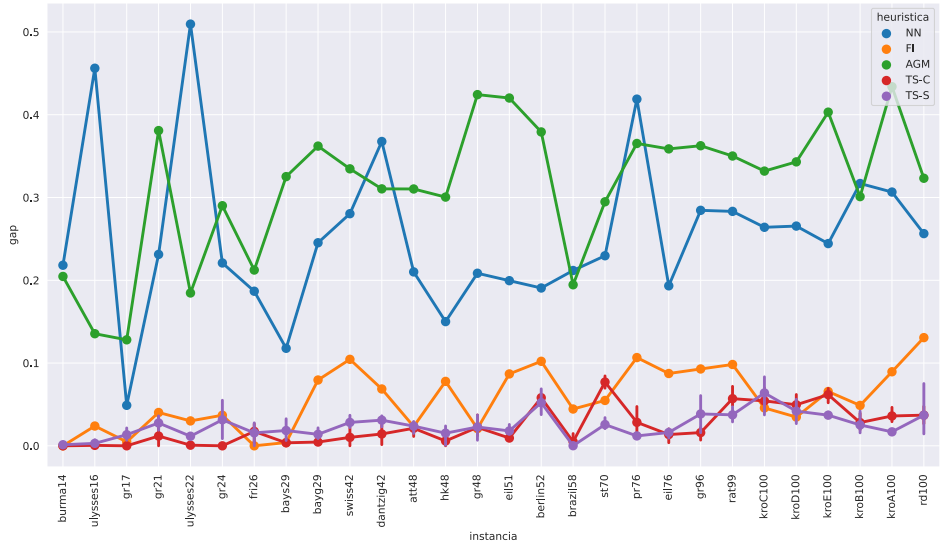


Figura 12: *gap* relativo entre todas las heurísticas

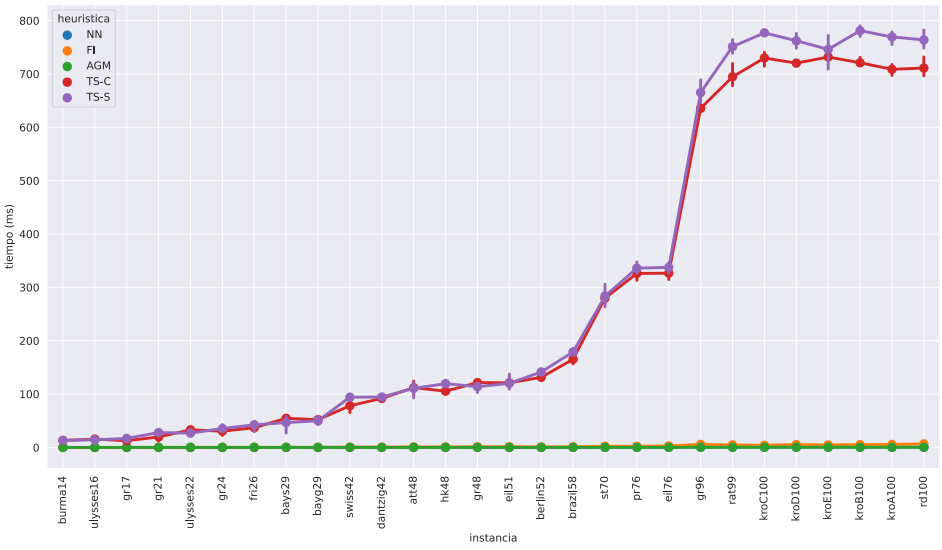


Figura 13: tiempos de ejecución para todas las heurísticas

3.5. Testing de nuevas instancias

Por último, decidimos evaluar el rendimiento de las heurísticas golosas implementadas, y la meta-heurística tabuSearch con parámetros optimizados, para tres nuevas instancias del dataset explorado:

- `kroA150`. La dimensión de esta instancia es 150, y el costo de la solución óptima es 26524.
- `u574`. La dimensión de esta instancia es 574, y el costo de la solución óptima es 36905.

- `pcb1173`. La dimensión de esta instancia es 1173, y el costo de la solución óptima es 56892.

El objetivo de este análisis es ver qué tan bien *fitteados* están los parámetros definidos, y cómo se comparan las otras heurísticas a instancias más grandes. En el Cuadro 1 se puede observar que, a pesar de haber ajustado los parámetros de tabuSearch con instancias mucho menores a estas, ambas variantes de la meta-heurística siguen retornando soluciones de menor costo que las soluciones obtenidas mediante las heurísticas golosas. Por otro lado, los tiempos de ejecución de la meta-heurística incrementan considerablemente a medida que la dimensión de la instancia lo hace. En el Cuadro 2 podemos ver que para la instancia de mayor dimensión, ambas variantes de tabuSearch toman más de 2400 segundos (40 minutos) en retornar una solución.

Gap relativo			
Heurística	`kroA150`	`u574`	`pcb1173`
`NN`	0.268021	0.367267	0.265169
`FI`	0.081436	0.101721	0.148088
`AGM`	0.461431	0.351524	0.435404
`TS-C`	0.081096	0.060751	0.072049
`TS-S`	0.076534	0.058610	0.071662

Cuadro 1: *gap* relativo de las soluciones obtenidas para las instancias de *testing*

Tiempo de ejecución (s)			
Heurística	`kroA150`	`u574`	`pcb1173`
`NN`	0.001065	0.039197	0.375024
`FI`	0.008166	0.401921	5.437850
`AGM`	0.000384	0.002784	0.012333
`TS-C`	3.231590	177	2656
`TS-S`	3.254460	182	2496

Cuadro 2: tiempos de ejecución de las soluciones obtenidas para las instancias de *testing*

4. Conclusiones

Finalmente, tras analizar los resultados de los experimentos realizados, llegamos a distintas conclusiones que discutimos a continuación.

- Desde un principio viendo la clase del laboratorio habíamos intuido que la heurística basada en AGM iba a darnos un *gap* relativo mucho mejor al de las otras heurísticas golosas implementadas. Una vez realizados los experimentos pudimos visualizar que este no era el caso. Esto puede deberse a que el armado del circuito mediante el algoritmo de DFS es indiferente a las aristas que inserta al conectar dos hojas del árbol.
- Otro punto a notar es que, a pesar de ser la heurística golosa cuyas soluciones poseen el menor *gap* relativo, los tiempos de ejecución de *Farthest Insertion* son superiores a los de las otras heurísticas golosas.
- Considerando que el incremento del parámetro M de tabuSearch en valores por encima de 100 genera soluciones cuyo *gap* relativo no mejora cuantitativamente, pero sí aumenta considerablemente la complejidad espacial, es posible que no valga la pena elegir valores muy superiores a este.
- Finalmente, a partir de la experimentación exhibida en la Figura 12 concluimos que ambas variantes de la meta-heurística tabuSearch, como así también la heurística de *FarthestInsertion* retornan soluciones que no distan significativamente del óptimo, siendo la primera de estas la más cercana. Sin embargo, cuando miramos la Figura 13 podemos ver que el tiempo de ejecución de tabuSearch es órdenes de magnitud mayor al de la heurística *FarthestInsertion*. Lo que es más, como se puede ver en el Cuadro 2, este factor es determinante a la hora de evaluar instancias de mayor tamaño.

Un área donde se podría profundizar a futuro es considerar iteraciones efectivas, es decir donde efectivamente se encuentre una solución mejor a las todas las anteriores exploradas. Así mismo, sería interesante evaluar las soluciones encontradas por la meta-heurística tabuSearch partiendo de soluciones de otras heurísticas distintas a *NearestNeighbour*. Por último, creemos que vale la pena explorar la generalización de la heurística basada en AGM, que es es algoritmo de **Christofides**. Consideramos que sería interesante explorar como este algoritmo mitiga las falencias que ocurren al acircuitar el árbol mediante DFS.

Referencias

[1] W. Cook - W. Cunningham - W. Pulleyblank - A. Schrijver. *Combinatorial Optimization. Chapter 7: TSP.* URL: http://math.mit.edu/~goemans/18433S15/TSP-CookCPS.pdf.