UNIVERSIDAD DE CONCEPCIÓN
FACULTAD DE INGENIERÍA
Programa de Magíster en Ciencias de la Computación

SELECCIÓN AUTOMÁTICA DE ALGORITMO A LO LARGO DEL TIEMPO PARA EL PROBLEMA DEL VENDEDOR VIAJERO

Tesis presentada a la Facultad de Ingeniería de la Universidad de Concepción para optar al grado académico de Magíster en Ciencias de la Computación

POR: ISAÍAS IGNACIO HUERTA VARGAS
PROFESOR GUÍA : ROBERTO JAVIER ASÍN ACHÁ

Diciembre, 2020
Concepción, Chile

©

Se autoriza la reproducción total o parcial, con fines académicos, por cualquier medio o procedimiento, incluyendo la cita bibliográfica del documento

Índice General

1. INTRODUCCIÓN **1**
1.1. Contexto 1
1.2. Hipótesis 4
1.3. Objetivos 4
1.3.1. Objetivo General 4
1.3.2. Objetivos Específicos 5
1.4. Limitaciones 5
1.5. Estructura del Informe 6

2. MARCO TEÓRICO **7**
2.1. Problema del vendedor viajero 7
2.1.1. Definición y condiciones 7
2.1.2. Motivación 8
2.1.3. Algoritmos 8
2.2. Redes neuronales 9
2.3. Selección automática de algoritmos 10
2.4. Selección automática de algoritmos a lo largo del tiempo (anytime) 12

3. ESTADO DEL ARTE **13**
3.1. Solvers en el Problema del vendedor viajero 13
3.2. Selección automática de algoritmos 16
3.3. Discusión 18

4. Selección automática de algoritmos a lo largo del tiempo para TSP **19**
4.1. Conjunto de solvers 19
4.2. Conjunto de instancias 21
4.2.1. Instancias de entrenamiento 22
4.2.2. Instancias de validación 25
4.3. Ejecución de cada par solver-instancia 26
4.4. Caracterización de las instancias 28
4.5. Representación de la resolución de una instancia 29
4.6. Modelos 32

4.7. Ajuste de parámetros 34
4.8. Configuración experimental 35

5. RESULTADOS **36**

6. CONCLUSIONES Y TRABAJO FUTURO **42**
6.1. Trabajo Futuro 43

7. Anexos **50**
7.1. Gráficos por dataset del comportamiento de los solvers 50
7.2. Gráficos descriptivos de las instancias públicas 52

1. INTRODUCCIÓN

1.1. Contexto

Los problemas de optimización combinatoria están presentes en muchas aplicaciones del mundo real, por ejemplo, en el despliegue de recursos en la nube, en la búsqueda de caminos óptimos para empresas de transporte, en el diseño computacional de proteínas, en la construcción de microchips o en la asignación de los mejores horarios posibles para las clases en colegios y universidades. Aunque muchos de estos problemas son NP-hard, necesitan ser resueltos de tal manera de encontrar una solución tan buena y tan rápida como sea posible. En muchos casos, una pequeña mejora en la calidad de la solución se traduce en el ahorro de millones de dólares o en un aumento de la productividad.

Para muchos problemas de optimización combinatoria existe un amplio número de publicaciones que presentan nuevos enfoques de solución y reportan avances en el área. Sin embargo, la mayoría de las mejoras en el rendimiento se relacionan a un conjunto de datos específico o a una caracterización del problema. En la mayoría de los casos, cuando probamos con datos distintos, esta mejora de rendimiento desaparece (Teorema No free lunch [1]). Por lo tanto, en el mundo real se utiliza una combinación de distintas técnicas o algoritmos que resuelven el problema dependiendo de las características de la instancia [2].

La combinación de distintos algoritmos que resuelven un problema de optimización combinatoria se puede llevar a cabo en la práctica mediante un *oráculo predictor* que nos indique a partir de un portafolio de algoritmos cuál de ellos es el mejor para resolver una instancia dada, por lo tanto, lo que llamaremos un *metasolver* corresponde a un algoritmo compuesto de dos etapas: primero una etapa predictora en donde se escoge el mejor algoritmo y luego la resolución de la instancia con el algoritmo seleccionado.

El problema de escoger el mejor algoritmo posible para una instancia específica de un problema ha sido establecido en la comunidad de *machine learning* y es conocido como Automatic Algorithm Selection Problem (AASP) [3]. En general, este problema es visto como un problema de clasificación [4] cuyo objetivo es aprender el comportamiento de distintos solvers sobre un conjunto de instancias y generalizar sobre instancias desconocidas. Dada una instancia específica, la predicción del modelo entrega el algoritmo (desde ahora *solver*) que se espera que obtenga la mejor solución.

Usualmente, para el modelo se considera sólo la mejor solución encontrada dentro de un límite de tiempo definido. Si tenemos un tiempo acotado para la resolución de una instancia, este enfoque no es apropiado porque la predicción podría retornar soluciones con un tiempo de resolución superior al deseado. Por ejemplo, si buscamos una solución aproximada en un tiempo muy corto, un algoritmo *greedy* puede ser la mejor opción, mientras que si podemos esperar el tiempo suficiente, es posible obtener el óptimo a través de un algoritmo exacto.

En la Figura 1 se muestra el comportamiento a lo largo del tiempo de cinco solvers para una instancia del problema de la mochila (*knapsack problem*). Si nos posicionamos en distintos puntos de la línea temporal veremos que nuestro mejor solver varía dependiendo del momento.

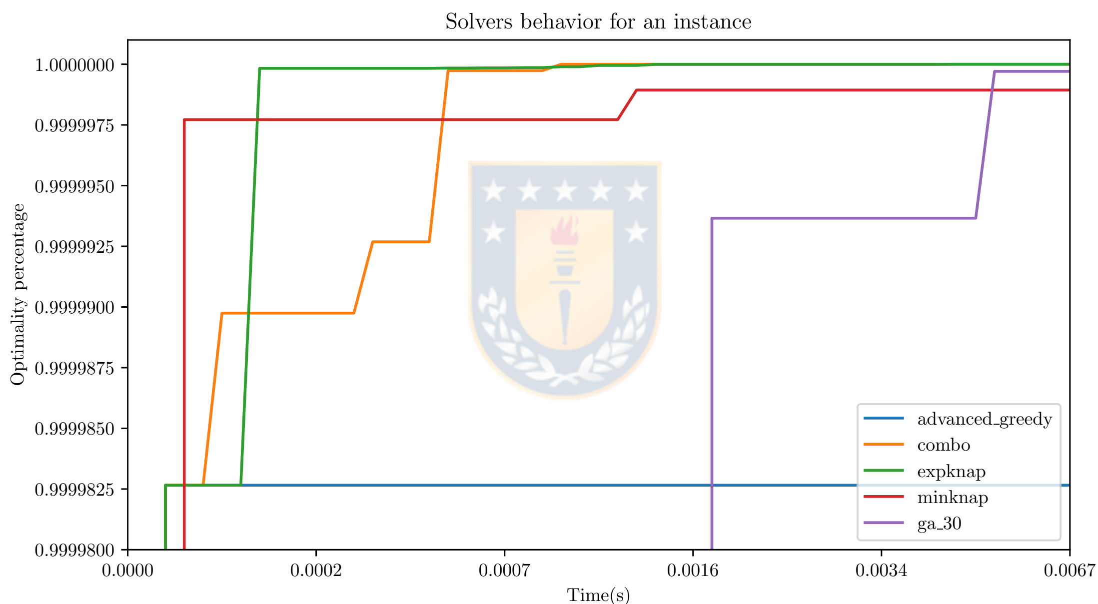


Fig. 1: Ejemplo del comportamiento a lo largo del tiempo de cinco solvers de knapsack para una instancia [5].

En un trabajo anterior realizado sobre el problema de la mochila [5] se proponen diferentes enfoques para abordar la selección automática de algoritmos considerando el comportamiento a lo largo del tiempo. Los resultados muestran que las técnicas de machine learning utilizadas son capaces de predecir el mejor solver dada una instancia y un instante de tiempo específicos. Si bien los resultados son interesantes, la mayoría de los solvers eran capaces de resolver las instancias en un tiempo muy bajo (menor a 1 segundo), por lo tanto, el tiempo de predecir el mejor solver era significativo en el tiempo total (tiempo de selección de algoritmo más tiempo

de ejecución del algoritmo) debido al tiempo de cálculo de las características como la media, mediana o el coeficiente de pearson. El bajo tiempo de resolución para los experimentos en Knapsack se deben a la naturaleza *Weakly NP-complete* del problema y a que las instancias estudiadas de este problema suelen considerar números enteros de tamaño fijo (usualmente 32 bits), no así números enteros de precisión arbitraria, con los que es posible configurar instancias complicadas para el problema.

En este trabajo se propone realizar una selección automática de algoritmos a lo largo del tiempo (o *anytime*) sobre el conocido y muy estudiado problema del vendedor viajero (TSP) euclidiano. Este problema es NP-hard y, a diferencia de Knapsack, no existen algoritmos exactos pseudo-polinomiales o de aproximación arbitraria con respecto a la dimensión de los datos. Esta propiedad lo hace más interesante al abordarlo en un contexto *anytime* debido a que los tiempos de ejecución son altos y, por tanto, la decisión sobre el mejor algoritmo es más relevante.

1.2. Hipótesis

Es posible construir un metasolver capaz de superar el rendimiento promedio de los solvers estado del arte de TSP en datasets públicos por medio de la selección automática de algoritmos.

1.3. Objetivos

1.3.1. Objetivo General

El objetivo general de este trabajo es construir una nueva metaheurística para el TSP euclidiano basada en la selección automática de algoritmos dentro de un portafolio de solvers estado del arte. El modelo utilizado para la selección automática de algoritmos tendrá conocimiento del comportamiento a lo largo del tiempo de la resolución, por lo tanto, puede responder consultas basadas en una condicionalidad temporal. La consulta principal a abordar en este trabajo es: Dada una instancia i y un tiempo de espera máximo t, ¿Qué solver se debería seleccionar para resolverlo?

El algoritmo se puede dividir en dos etapas:

- Selección automática de algoritmo en base al tiempo disponible. Dado el tiempo disponible y la instancia, se decide por el solver que se espera que obtenga un mejor valor objetivo hasta el tiempo dado.
- Ejecución del solver. Se ejecuta el solver obtenido de la etapa anterior con la instancia dada.

El tiempo total de ejecución del metasolver corresponderá a la suma del tiempo de las dos etapas.

1.3.2. Objetivos Específicos

- Realizar una revisión literaria y seleccionar un conjunto de solvers estado del arte en TSP euclidiano.
- Generar un conjunto de datos de entrenamiento y validación a partir de la ejecución de solvers estado del arte sobre instancias de TSP tanto generadas como públicas.
- Proponer, implementar y refinar un modelo de Redes Neuronales para la selección automática de algoritmo.
- Realizar un ajuste de parámetros sobre el modelo y a la caracterización.
- Comparar el tiempo y el grado de suboptimalidad promedio de los solvers estado del arte en las instancias públicas con respecto a la metaheurística.
- Comunicar los resultados de la investigación.

1.4. Limitaciones

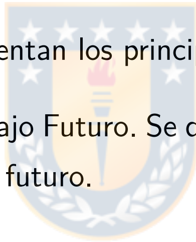
Para realizar este trabajo se consideraron las siguientes limitaciones:

- Los códigos utilizados para cada solver corresponden a los publicados por sus autores. Los parámetros y las optimizaciones son las que se encuentran por defecto y su calibración queda fuera de este estudio.
- El tiempo máximo definido para la ejecución de las instancias se fijó en dos horas. Una consulta mayor a ese tiempo se limitará a basarse en la última mejor solución.
- La ejecución de los experimentos se limitó a utilizar solamente un hilo de ejecución para evitar algún tipo de paralelismo de los solvers que haría desigual la competencia.

1.5. Estructura del Informe

La investigación se estructura en seis capítulos:

- Capítulo 1. Introducción. Se describe el contexto del problema y se presenta la motivación para realizar este trabajo.
- Capítulo 2. Marco Teórico. Se describen conceptos teóricos relevantes para el desarrollo y el entendimiento de la investigación.
- Capítulo 3. Revisión del Estado del Arte. Se describe la investigación del estado del arte realizada en solvers de TSP y en trabajos de Automatic Algorithm Selection.
- Capítulo 4. Solución Propuesta. Se describe la metodología, los solvers, las instancias y los modelos utilizados.
- Capítulo 5. Resultados. Se presentan los principales resultados obtenidos en el trabajo.
- Capítulo 6. Conclusiones y Trabajo Futuro. Se describen los resultados de la investigación y se proponen líneas de trabajo futuro.



2. MARCO TEÓRICO

2.1. Problema del vendedor viajero

2.1.1. Definición y condiciones

El problema del vendedor viajero (TSP) es un problema clásico NP-hard de optimización combinatoria que ha sido ampliamente estudiado debido a que es computacionalmente desafiante de resolver, es fácil de instanciar y de describir. Se encuentra presente en diversas aplicaciones prácticas o en simplificaciones de problemas multidisciplinarios complejos.

La formulación clásica es: dado un conjunto de n ciudades, un vendedor quiere visitar cada una de las ciudades una única vez, comenzando desde cualquier ciudad y volviendo a la misma ciudad al terminar el recorrido, TSP responde a la pregunta: ¿Qué camino debería seguir el vendedor para minimizar la distancia total recorrida? Las distancias entre cualquier par de ciudades se asumen conocidas por el vendedor. La distancia puede ser reemplazada por otra idea como dinero o tiempo. Desde ahora el término costo es utilizado para representar cualquier dominio y el término *ciudad* se utilizará para hablar de los nodos del problema.

Matemáticamente el problema puede ser establecido como: Dada una matriz de costo $C = c_{ij}$, donde c_{ij} representa el costo de ir de una ciudad i a una ciudad j, con $i, j = (1, 2, ..., n)$, encuentre una permutación de ciudades $(i_1, i_2, i_3, ..., i_n)$ que minimice la sumatoria $x_{i_1 i_2} + x_{i_2 i_3} + x_{i_3 i_4} + ... + x_{i_{n-1} i_n} + x_{i_n i_1}$.

Las propiedades de la matriz de costo se pueden utilizar para caracterizar el problema:

- Si $c_{ij} = c_{ji}$ para todo i,j, se dice que el problema es **simétrico**, de otra forma es **asimétrico**.
- Si la desigualdad triangular se mantiene ($c_{ik} \leq c_{ij} + c_{jk}$ para todo i,j y k), entonces el problema se dice **métrico**.
- Si c_{ij} son distancias euclidianas entre puntos en el plano, el problema es **Euclidiano**. Este problema es simétrico, métrico y euclidiano.

2.1.2. Motivación

La importancia directa de TSP recae en su aplicación sobre problemas comunes como la búsqueda de mejores rutas posibles. Sin embargo, también se puede aplicar a problemas que no tienen que ver con la noción de distancias físicas. Por ejemplo, podemos considerar el problema de planificación de procesos. Un número de tareas tienen que procesarse en una máquina de un solo procesador. Antes de que una tarea comience a procesarse, la máquina debe prepararse (limpiar, ajustar, *setear*) para la tarea. Dado el tiempo de procesamiento de cada tarea y el tiempo para cambiar entre cada par de tareas, el objetivo es encontrar una secuencia de tareas de tal forma que el tiempo total de procesamiento sea mínimo. Es fácil ver que este problema es una instancia de TSP. Aquí c_{ij} representa el tiempo de completar la tarea j después de i (tiempo entre cambiar de la tarea i a la tarea j más el tiempo de procesamiento de j).

Muchos problemas del mundo real pueden ser caracterizados como instancias de TSP. Su versatilidad se evidencia ejemplificando algunas de sus aplicaciones: construcción de microprocesadores, cristalografía de rayos X para el estudio de materiales, ruteo de vehículos, control de robots, secuencias cronológicas, entre otras.

2.1.3. Algoritmos

Para una instancia de TSP simétrico con n ciudades, existen $(n-1)!/2$ rutas posibles. Si tenemos 30 ciudades, existen aproximadamente $4 * 10^{30}$ rutas posibles. Si aumentamos el número de ciudades se vuelve impracticable utilizar algoritmos exactos ingenuos, aún considerando una capacidad de cómputo alta. A la fecha, la mayor instancia que se ha logrado resolver de manera exacta posee 85.900 ciudades [6]. Por otro lado, los algoritmos aproximados han logrado resolver instancias con millones de ciudades garantizando estar a un 2 o 3 % del tour óptimo.

Se ha probado que TSP es miembro del selecto grupo de problemas NP-hard. Esta es una clase de problemas difíciles cuya complejidad computacional temporal para el mejor algoritmo es probablemente exponencial. Los miembros de esta clase están relacionados de forma tal que si se encuentra un algoritmo que resuelva en tiempo polinomial un problema, existirían algoritmos de tiempo polinomial para todos estos problemas. Sin embargo, en Teoría de la Computación, la mayoría de los investigadores tiene la conjetura de que dicho algoritmo no

existe ($P \neq NP$). Así, cualquier intento para construir un algoritmo general que encuentre soluciones óptimas para el TSP en tiempo polinomial probablemente fallará.

Para cualquier algoritmo exacto, es posible encontrar instancias para las cuales los tiempos de ejecución crecen al menos exponencialmente con el tamaño de las ciudades. Notar, sin embargo, que la complejidad corresponde al comportamiento del peor caso. No se puede sentenciar que no existan algoritmos cuyo tiempo de ejecución sea polinomial en caso promedio, sin embargo, la existencia de tales algoritmos es todavía una pregunta abierta.

2.2. Redes neuronales

Las redes neuronales son un modelo computacional inspirado por el comportamiento de las neuronas en sistemas biológicos. Consisten de un conjunto de nodos (o neuronas artificiales) que se conectan entre sí a través de una estructura formada por capas. El objetivo de un modelo de redes neuronales supervisado es aprender a resolver una tarea a partir de ejemplos que son entregados a la red en una fase de entrenamiento. En esta fase, se actualizan los parámetros (o pesos) que conectan cada par de neuronas a través del algoritmo *backpropagation* [7] con el propósito de reducir la pérdida o función de costo. Dependiendo de la naturaleza de los datos a predecir, el tipo de red neuronal puede cambiar. Si queremos predecir clases discretas, entonces la red corresponde a un clasificador. Si queremos predecir un valor continuo, entonces la red corresponde a un regresor.

Las redes neuronales se han utilizado para resolver diversos problemas como la visión por computador, el reconocimiento de voz, la generación de texto y sistemas de recomendación, alcanzando el estado del arte en muchas de estas aplicaciones. Existen diferentes arquitecturas en redes neuronales que han permitido avances en la precisión en distintas aplicaciones, entre ellas la arquitectura Convolucional (CNN) [8], las redes recurrentes (RNN) [9], los Transformers [10] han mejorado los resultados en tareas con imágenes, datos secuenciales y texto respectivamente. Así, la arquitectura utilizada depende de la tarea requerida.

2.3. Selección automática de algoritmos

Consideremos la selección de algoritmos para un problema de decisión u optimización P. Específicamente, la selección de algoritmos por instancia puede ser formulada como sigue [3]: Dado un conjunto I de instancias de un problema P, un conjunto $A = \{A_1, ..., A_n\}$ de algoritmos para resolver P y una métrica $m : A \times I \rightarrow \mathbb{R}$ que mide el rendimiento de cualquier algoritmo $A_j \in A$ sobre el conjunto de instancias I, el objetivo es construir un selector S que mapee cualquier instancia $i \in I$ a un algoritmo $S(i) \in A$ tal que el rendimiento promedio de S sobre I es óptimo de acuerdo a la métrica m.

En general, el selector S corresponde a un clasificador de machine learning que requiere como entrada una caracterización de las instancias de tamaño fijo. Para ello se define un conjunto de N_f características $F(i) = \{f_1(i), ..., f_{N_f}(i)\}$ que permiten representar cada una de las instancias $i \in I$ en una representación fija de tamaño N_f. Las funciones $F(i)$ deben ser escogidas cuidadosamente para que sean lo más informativas y fáciles de computar posible.

En la Figura 2 se resumen las tareas involucradas en un trabajo de selección automática de algoritmos. Una vez se decide el problema de optimización sobre el cual se desea realizar un trabajo de este tipo se deben tomar una serie de decisiones. Las primeras decisiones se relacionan con los solvers seleccionados y las instancias a utilizar para el entrenamiento y la validación. Luego, para la caracterización, se debe definir qué conjunto de características utilizar. Finalmente se deben decidir las técnicas de machine learning y el método de validación. Otras decisiones importantes tienen relación con el tiempo y los recursos computacionales disponibles. En principio, es conveniente utilizar tantos solvers como sea posible para considerar comportamientos variados sobre dominios específicos y también es deseable utilizar una cantidad alta de instancias. Sin embargo, si el problema de optimización es NP-hard, la ejecución de cada par instancia-solver puede ser muy costosa.

A continuación dividimos el proceso de trabajo en selección automática de algoritmos según sus componentes:

- **Solvers**. Se seleccionan solvers del problema de optimización definido. Cada uno de los solvers seleccionados se espera que sean de una naturaleza distinta para que el conjunto de solvers presenten comportamientos variados. En la literatura podremos encontrar muchas variaciones sobre solvers conocidos; sin embargo, considerar todas estas variaciones

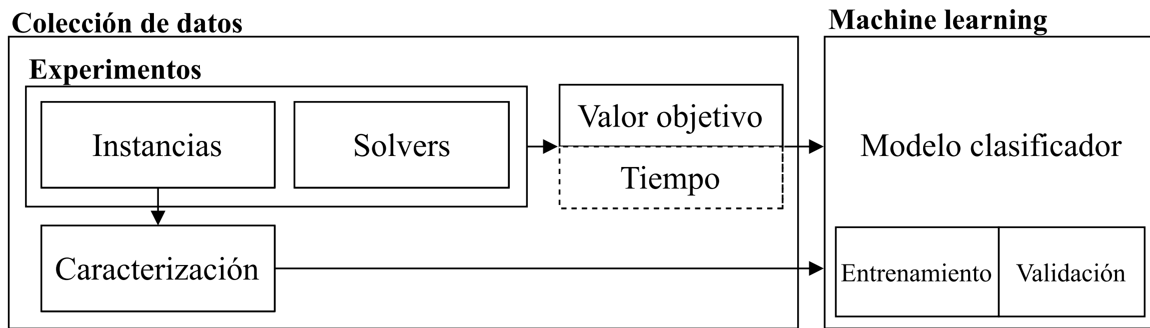


Fig. 2: Esquema general de un trabajo de AASP. El tiempo se muestra con líneas punteadas porque puede utilizarse para trabajos con enfoque *anytime*, como también puede fijarse únicamente como límite de ejecución.

involucra ejecutar todas las instancias sobre cada uno de ellos, lo cual puede ser muy costoso. Una forma más inteligente de seleccionar los solvers es escoger un representante dentro de una familia algorítmica, como pueden ser: algoritmos exactos, algoritmos greedy, algoritmos genéticos, sistemas multi-agentes, entre otros. Algunos trabajos [11] se han enfocado en utilizar la selección automática sobre distintos parámetros de un mismo solver.

- **Instancias**. Se crea o se selecciona un conjunto de instancias del problema de optimización definido. Esta etapa no es trivial debido a que las instancias deben ser lo suficientemente distintas entre sí para que produzcan comportamientos distintos en los solvers. Además, se debe decidir un número de instancias lo suficientemente alto para que se represente la mayor variación posible de las características, como también debe ser lo suficientemente bajo como para poder ejecutar todos los experimentos en un tiempo acotado.

- **Experimentos**. Una vez se tienen definidos los solvers y las instancias, se deberá registrar el valor objetivo de la resolución de la instancia-solver. Dado que para las instancias más desafiantes el tiempo de ejecución para obtener el óptimo puede ser inviable en un tiempo razonable, se define un límite de detención de la ejecución. La mayoría de los trabajos almacenan solamente el **valor objetivo** final obtenido. Sin embargo, esfuerzos recientes [5] almacenan también cada par tiempo-valor objetivo con el propósito de estudiar los comportamientos a lo largo del tiempo.

- **Caracterización**. Dado que el entrenamiento de modelos de *machine learning* requiere una entrada de tamaño fijo, es necesario realizar una caracterización de las instancias.

La caracterización consiste en seleccionar un conjunto fijo de características, tales como métricas o estadísticos, que entreguen información relevante de las instancias. Una mayor precisión de la caracterización se obtendrá con un conjunto mayor de características, sin embargo, aumenta el costo computacional y, por ende, el tiempo de predicción. Algunos trabajos [12] se han encargado de realizar un análisis de sensibilidad para decidir cuántas y cuáles características utilizar, de tal forma que se reduzca la cantidad de características irrelevantes a calcular y con ello se minimice el tiempo de cómputo de la caracterización.

- **Modelo clasificador**. El modelo clasificador es el encargado de mapear cada caracterización a un solver en particular. Un buen modelo debe ser entrenado con instancias y características apropiadas que permitan generalizar a instancias desconocidas.

2.4. Selección automática de algoritmos a lo largo del tiempo (anytime)

Extendiendo la definición de Rice [3]podemos definir la selección automática de algoritmos a lo largo del tiempo como:

Dado un conjunto I de instancias de un problema P, un conjunto $A = \{A_1, ..., A_n\}$ de algoritmos para resolver P, un conjunto de k instantes de tiempo $T \in \{t_1, t_2, ..., t_k\}$ y una métrica $m : (A \times I) \times T \to \mathbb{R}$ que mide el rendimiento de cualquier algoritmo $A_j \in A$ sobre el conjunto de instancias I en los instantes de tiempo T, el objetivo es construir un selector S que mapee cualquier par $(i \in I, t_j \in T)$ a un algoritmo $S(i) \in A$ tal que el rendimiento promedio de S sobre el par (I, T) es óptimo de acuerdo a la métrica m.

La variable temporal adicional $T \in \{t_1, t_2, ..., t_k\}$ que surge en esta definición debe ser descrita dependiendo de la naturaleza del problema a tratar, considerando la ventana de tiempo $[t_1, t_k]$ sobre la cual se trabajará, la granularidad de las divisiones (valor de k) y la distancia entre cada división t_i, t_{i+1}.

3. ESTADO DEL ARTE

3.1. Solvers en el Problema del vendedor viajero

Los **algoritmos exactos** garantizan encontrar la solución óptima de cualquier instancia y muchos pueden entregar una prueba de optimalidad al terminar. Los algoritmos exactos más efectivos son los algoritmos de *cutting-plane* y de *facet-finding*, propuestos en [13, 14, 15]. Estos algoritmos son altamente complejos y demandan mucha capacidad de cómputo. El más conocido es Concorde [16], basado en una sofisticada técnica de Branch & Cut. Debido a su peor caso de tiempo de ejecución exponencial, no es apropiado para utilizar en instancias grandes.

Por otro lado, los **algoritmos aproximados** entregan rápidamente una buena solución, sin embargo, esta solución puede no ser óptima. Algunos algoritmos son capaces de entregar soluciones muy buenas que difieren sólo en un pequeño porcentaje del óptimo. Dependiendo del propósito para el cual se requiera una solución, podrían ser aceptadas pequeñas desviaciones del óptimo.

Se pueden dividir los algoritmos aproximados en tres grupos: Algoritmos de construcción de tour, Algoritmos de mejora de tour y Algoritmos compuestos. Los algoritmos de construcción de tour construyen un tour agregando gradualmente una ciudad por cada paso. Los algoritmos de mejora de tour realizan cambios sobre un tour ya existente con el objetivo de mejorarlo. Los algoritmos compuestos combinan ambas características.

El ejemplo más simple para un algoritmo de construcción de tour es el algoritmo de vecino más cercano. Este algoritmo comienza en una ciudad cualquiera. Desde ahí continúa por la ciudad más cercana hasta que ya no quede ciudad por visitar y vuelve a la ciudad de origen. Este enfoque es simple, pero presenta un algoritmo tipo *greedy* demasiado sencillo, con resultados generalmente muy por debajo del óptimo, en donde las primeras ciudades tendrán una distancia corta entre sí, pero las últimas tenderán a ser más largas, sin embargo, la solución se obtiene en un tiempo bajo. Muchos otros algoritmos han trabajado en mejorar este algoritmo [17, 18, 19].

Los algoritmos de mejora de tour han sido más exitosos. Un ejemplo de este tipo es el algoritmo 2-opt. El algoritmo comienza con un tour dado, luego reemplaza dos enlaces del

tour con otros dos enlaces, de tal forma que la longitud del tour obtenido sea menor al original y continúa hasta que no hayan mejoras posibles. Una generalización de este algoritmo es el λ-opt, basado en el concepto de λ-optimality:

> *Se dice que un tour es λ-optimal (o simplemente λ-opt) si es imposible obtener un tour más corto reemplazando cualquier λ de sus enlaces por cualquier otro conjunto de λ enlaces.*

De esta definición se desprende que cualquier tour λ-optimal es también λ'-optimal para $1 \leq \lambda' \leq \lambda$. Una desventaja de este algoritmo es que es necesario definir con anterioridad el valor de λ, sabiendo que este valor afecta en el tiempo de ejecución y en la calidad de la solución.

El algoritmo Lin-Kernighan [20] viene a eliminar esta desventaja permitiendo que el valor λ sea una variable que puede cambiar durante la ejecución. Posteriormente, Keld Helsgaun incorporó una serie de cambios en su implementación (LKH [21]), principalmente en la estrategia de búsqueda, permitiendo pasos más largos y más complejos en la búsqueda a una nueva solución. Además se incluyó el uso de análisis de sensibilidad para dirigir o restringir la búsqueda. Los tiempos de ejecución de ambos algoritmos crecen a una razón de $n^{2,2}$, sin embargo, la mejora es más efectiva para encontrar soluciones óptimas en problemas más grandes. La última mejora realizada a este algoritmo (POPMUSIC [22]) acelera la generación de conjuntos candidatos para problemas de gran tamaño, logrando obtener conjuntos de candidatos de alta calidad en un tiempo casi lineal.

Otro algoritmo que toma como base la idea de Lin-Kernighan es el algoritmo Chained Lin-Kernighan (CLK) [23]. Este algoritmo sacrifica en parte la calidad de LK para mejorar los tiempos de ejecución. Otra modificación consiste en el intercambio de 4 aristas (llamado *double-bridge*) que permite encontrar otras soluciones que LK no habría alcanzado. Por último, una mejora de CLK es el tour inicial, que para instancias grandes permite comenzar con una buena solución de forma muy rápida. CLK es utilizado para generar una solución inicial en Concorde.

Se han propuesto varias metaheurísticas de búsqueda local que utilizan técnicas inspiradas en la naturaleza para obtener soluciones aproximadas de TSP. Entre ellas está Ant colony optimization (ACO) [24, 25], particle swarm optimization (PSO) [26, 27] y Algoritmos Genéticos (GA) [28, 29, 30, 31]. Éstos últimos han sido los más exitosos cuando integran variantes

de edge assembly crossover [32]: un operador de recombinación que combina las aristas de dos soluciones padre y agrega aristas cortas que los padres no consideraron. El primer solver en utilizar este operador fue EAX [33], alcanzando un rendimiento similar al de LKH.

Otro algoritmo inspirado en la naturaleza es MAOS [34], que en contraste con los algoritmos anteriores, está basado en un framework multi-agente, donde cada agente tiene conocimiento limitado de la instancia de TSP y exploran posibles soluciones de forma paralela. Los agentes actualizan el ambiente compartido y comunican su conocimiento a otros agentes para mejorar la eficiencia del proceso de búsqueda. En principio, cada agente comienza con una estructura definida, que puede ser un subgrafo con los vecinos más cercanos. Luego utiliza cadenas de Markov para generar un conjunto de estados.

En los últimos años también se ha explorado la utilización de redes neuronales para resolver TSP [35]. Algunas de las redes utilizadas corresponden a Elastic Net [36], Self-Organizing Map (SOM) [37] y la red Hopfield-Tank [38]. Como se menciona en algunos artículos, los resultados obtenidos aún no son comparables con algoritmos estado del arte [35, 39]. Con el aumento de la capacidad de cómputo en paralelo de las GPUs, también han surgido algoritmos que aprovechan estas tecnologías y reducen en al menos un orden de magnitud los tiempos de resolución obtenidos por algoritmos tradicionales [40, 41].

3.2. Selección automática de algoritmos

Uno de primeros sistemas de selección automática de algoritmos exitosos fue SATzilla [42], que por muchos años definió el estado del arte en la resolución del problema de satisfacibilidad (SAT), uno de los problemas NP-completo más estudiados. Gracias a este éxito, otros problemas relacionados como MAXSAT también fueron abordados con este enfoque [43]. También se ha aplicado a otros problemas computacionalmente desafiantes como *constraint programming* [44], *continuous black-box optimisation* [45], *mixed integer programming* [46] y *AI planning* [47].

En un trabajo reciente [5] se abordó la selección automática de algoritmo con enfoque anytime en el problema de la mochila (knapsack problem). Se propusieron tres formas para plantear este problema utilizando machine learning dependiendo de cómo se representa la resolución de la instancia:

- Modelo clasificador (Mejor solver): por cada instante de tiempo sólo se considera la etiqueta del solver que mejor resolvió la instancia.
- Modelo clasificador (Ranking): por cada instante de tiempo se considera el orden de los solvers de acuerdo al valor objetivo.
- Modelo regresor (Optimalidad): por cada instante de tiempo se considera directamente el valor objetivo obtenido por cada solver, normalizado entre 0 y 1, donde el máximo corresponde al mejor valor encontrado al término de la ejecución.

Los modelos se utilizaron en dos conjuntos de solvers, por un lado solvers *naive* clásicos y por otro solvers estado del arte. Los mejores resultados se obtuvieron utilizando el modelo clasificador con el enfoque de mejor solver, sin embargo, no se presenta una comparación de rendimiento directa entre la predicción del clasificador y la ejecución del solver predicho, versus la ejecución individual de los solvers.

En TSP existen tres trabajos que han abordado la selección automática de algoritmo. En el primero [11] se utiliza un enfoque *anytime* para un portafolio de parámetros de un solver genético. En este caso la utilidad de la selección automática recae en la selección de los mejores parámetros de un algoritmo genético sobre diferentes instancias. El tiempo de ejecución se considera una característica importante al momento de elegir el mejor algoritmo. Se generaron 10 instancias uniformemente aleatorias con 40 ciudades y se utilizaron 54 configuraciones de

parámetros diferentes de un algoritmo genético. Para 6 de las 10 instancias, el modelo supera al rendimiento del mejor solver.

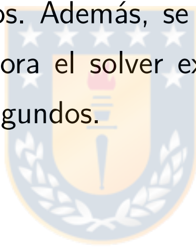
En el segundo trabajo [48] se enfocan principalmente en la selección de características apropiadas para un trabajo de selección automática de algoritmos sobre TSP. Se utilizan dos solvers: LKH y MAOS, y un dataset de 2163 instancias. Se entrenan 5 clasificadores de machine learning, siendo *Decision Trees* y *Bayesian networks* los que obtuvieron mejores resultados. Se utilizaron dos instantes de tiempo para comparar el rendimiento: a los 60 segundos y a los 1800 segundos, siendo exitosa la selección de algoritmos sin considerar el tiempo necesario para las características. Se propone como trabajo futuro seleccionar menos características para reducir el costo computacional.

El tercer trabajo [12] aborda la selección automática de algoritmos con el objetivo de mejorar el estado del arte. Consideran 5 solvers estado del arte: LKH, LKH+restart, EAX, EAX+restart y MAOS, y se utilizan 6 conjuntos de instancias: TSPLIB, National, VLSI, RUE, Netgen y Morphed. El límite utilizado para la resolución de los solvers fue de 1 hora. Se abordan tres enfoques para el modelo de selección automática de algoritmo: clasificación, regresión y *regresión pareada*, con varias técnicas de machine learning por cada uno. En el enfoque de clasificación se etiquetó cada instancia con el solver de mejor rendimiento. En el enfoque de regresión se entrenan 5 modelos (uno por solver), se predice el tiempo de ejecución de cada solver y se escoge el solver con el menor tiempo de ejecución. En el último enfoque de *regresión pareada* se predice la diferencia de rendimiento entre cada par de solvers. Se escoge como solver ganador el que posea mejor diferencia de rendimiento con todos los otros solvers. Un trabajo importante que se realiza es el análisis de sensibilidad en la selección de características, logrando obtener buenos resultados con el cómputo de pocas características. Los resultados muestran que si no se considera el tiempo del cómputo de las características se mejora el estado del arte en algoritmos inexactos de TSP por medio de la selección automática en todos los datasets utilizados, sin embargo, al incluír los tiempos de cómputo de características, el mejor selector es capaz de ganar sólo en 3 de 6 datasets.

3.3. Discusión

Los trabajos de selección automática sobre TSP descritos anteriormente presentan grandes diferencias entre sí porque centran su objetivo en distintas características: por un lado el primer trabajo utiliza la selección automática para definir los mejores parámetros de un algoritmo dependiendo de la instancia. Este tipo de trabajos se conoce también como Configuración Automática de Algoritmos[1]. El segundo trabajo centra la importancia en una extensa selección de características que maximizan la información descriptiva de instancias para TSP, mientras que el último trabajo pretende mejorar el estado del arte a través de una selección compacta de características y de modelos de machine learning.

La metodología utilizada en este trabajo comparte más relación con el último trabajo [12] por los solvers y las instancias utilizadas, sin embargo, el enfoque *anytime* y la utilización de redes neuronales para el modelo divergen de la metodología clásica para trabajar en proyectos de selección automática de algoritmos. Además, se aumenta la cantidad total de instancias utilizadas de 1845 a 6689, se incorpora el solver exacto Concorde y se aumenta el tiempo límite de ejecución de 3600 a 7200 segundos.



[1]https://cran.r-project.org/web/packages/irace/index.html

4. Selección automática de algoritmos a lo largo del tiempo para TSP

En esta sección se describen los solvers y las instancias seleccionadas, el proceso de caracterización de las instancias, los modelos de redes neuronales utilizados, el entrenamiento y el ajuste de parámetros.

4.1. Conjunto de solvers

En este trabajo se utilizan cinco solvers considerados como estado del arte para TSP hasta el año 2019. En un principio se propusieron otros solvers interesantes como Self-Organizing Maps[2] [37], ACO[3] [49] y OR-Tools[4]; sin embargo, ensayos experimentales mostraron que el rendimiento en tiempo y calidad no es comparable con los seleccionados. Existen también otros solvers estado del arte que utilizan paralelismo en su algoritmo que no fueron considerados para realizar una comparación justa entre los solvers.

- Concorde: Es el estado del arte en los algoritmos exactos para resolver TSP [16]. Concorde utiliza la estrategia de búsqueda Branch & Bound, así como cortes de planos (Branch & Cut) para reducir el espacio de búsqueda. Una buena solución inicial es obtenida utilizando CLK. Además de generar y explotar límites, Concorde también utiliza planos de corte para estrechar y enfocar el espacio de búsqueda. El código utilizado[5] está implementado en ANSI C y fue actualizado por última vez en 2003.

- Chained Lin Kernighan (CLK): Es una forma de búsqueda local iterada. Cuando Lin Kernighan se ejecuta en su forma no iterada encuentra un óptimo local cuya calidad es imprecisa definir debido al uso de la vecindad k-opt. Después de que CLK no encuentra una mejora en una solución, es decir, se queda estancado en un óptimo local, reinicia a través de una perturbación de la solución actual. La perturbación se basa en la eliminación de aristas que se intersectan, conocidas como *double-bridge*. El código utilizado

[2] https://diego.codes/post/som-tsp/
[3] https://github.com/cptanalatriste/isula
[4] https://developers.google.com/optimization/routing/tsp
[5] http://www.math.uwaterloo.ca/tsp/concorde/downloads/downloads.htm

se encuentra disponible como parte de la biblioteca de Concorde. Está implementado en ANSI C y fue actualizado por última vez en 2003.

- Lin Kernighan Helsgaun (LKH): LKH es una mejora al algoritmo Lin Kernighan, propuesta por Helsgaun [21]. Se ha demostrado que con estas mejoras el algoritmo se acerca más al óptimo en algunas instancias, sin emgargo, esto también aumenta los tiempos de ejecución. Entre las innovaciones propuestas por Helsgaun se encuentran movimientos 5-opt secuenciales y análisis de sensibilidad a las búsquedas directas. La versión del algoritmo utilizada corresponde a la última propuesta que incluye una nueva heurística para generar el conjunto de candidatos llamada POPMUSIC [22]. El código utilizado[6] está implementado en C y fue actualizado por última vez en 2019.

- Edge Assembly Crossover (EAX): EAX es un algoritmo genético basado en Edge Asseembly Crossover (EAX) [33], considerado uno de los operadores de recombinación más efectivos para TSP. La utilización de EAX permite mejorar en aspectos como la localización, logrando una implementación más efectiva a través del uso de mecanismos de búsqueda local para generar una buena construcción de soluciones padre y, por lo tanto, obtener mejores soluciones hijas. EAX genera sus soluciones hijas a través de la combinación de aristas de dos soluciones padres y agregando algunas aristas cortas, que son determinadas por un procedimiento simple de búsqueda. El código utilizado[7] está implementado en C++ y fue actualizado por última vez en 2018.

- Multiagent optimization system (MAOS): MAOS [34] es un método que se basa en la organización propia de los agentes, trabajando con conocimiento declarativo limitado y procedimientos que funcionan bajo una racionalidad ecológica. Específicamente, los agentes exploran en paralelo, basados en el aprendizaje individual socialmente sesgado (SBIL) e interactúan indirectamente con otros agentes mediante el intercambio de información pública organizada en el ambiente. El código utilizado[8] está implementado en Java y fue actualizado por última vez en 2017.

Las implementaciones de los solvers sólo fueron modificadas para estandarizar la entrada de las instancias y para generar una salida cada vez que encuentra una solución mejor, mostrando el valor objetivo alcanzado y el tiempo necesario para obtenerlo. En el caso de

[6]http://akira.ruc.dk/ keld/research/LKH/
[7]https://github.com/british-sense/Edge-Assembly-Crossover
[8]https://github.com/wiomax/MAOS-TSP

Concorde, la mejor solución encontrada es definida por la cota superior del algoritmo Branch & Cut. Los parámetros utilizados en cada solver corresponden a los definidos por defecto en las implementaciones.

4.2. Conjunto de instancias

Las instancias utilizadas en este trabajo se clasifican en dos grupos: Instancias de entrenamiento e instancias de validación. Las instancias de entrenamiento corresponden a instancias generadas por generadores existentes en la literatura, mientras que las instancias de validación corresponden a instancias públicas conocidas. Remarcar que todas las instancias utilizadas son instancias Euclidianas (2D). En la tabla 1 se resumen las principales características de los conjuntos.

	Dataset	**N instancias**	**Media N cities**	**Min N cities**	**Max N cities**	**Desv. std cities**	**25 % cities**	**50 % cities**	**75 % cities**
Validación	TSPLIB	88	3041	14	85900	10185	134	428	1468
	NATIONAL	27	11088	29	71009	14558	1800	8079	12412
	VLSI	102	25654	131	744710	90865	1926	3660	19373
	TNM	141	4432	52	100000	16041	157	262	367
Entrenamiento	RUE	3150	1452	500	2000	451	1000	1500	2000
	NETGEN	600	1250	500	2000	559	875	1250	1625
	NETGENM	600	1250	500	2000	559	875	1250	1625
	TSPGEN	1981	10976	11	32304	7373	4969	10004	15783

Tabla 1: Resumen de los datasets utilizados.

4.2.1. Instancias de entrenamiento

Las instancias de entrenamiento en total corresponden a 6.331 instancias divididas en cuatro conjuntos: RUE, NETGEN, NETGENM y TSPGEN. Las instancias de RUE, NETGEN y NETGENM son las mismas que se utilizaron en un trabajo previo de selección automática de algoritmo sobre TSP [12] y se encuentran disponibles públicamente[9].

A continuación se describen en detalle cada uno de los conjuntos:

- **RUE** (Random Uniform Euclidean): Son 3150 instancias obtenidas ubicando n coordenadas aleatorias en un cuadrado. El rango de valores enteros que pueden tomar las coordenadas van de 1 a 1.000.000. Estas instancias fueron generadas para el trabajo [12] utilizando el generador *portgen* de la 8va competición de implementación DIMACS[10].

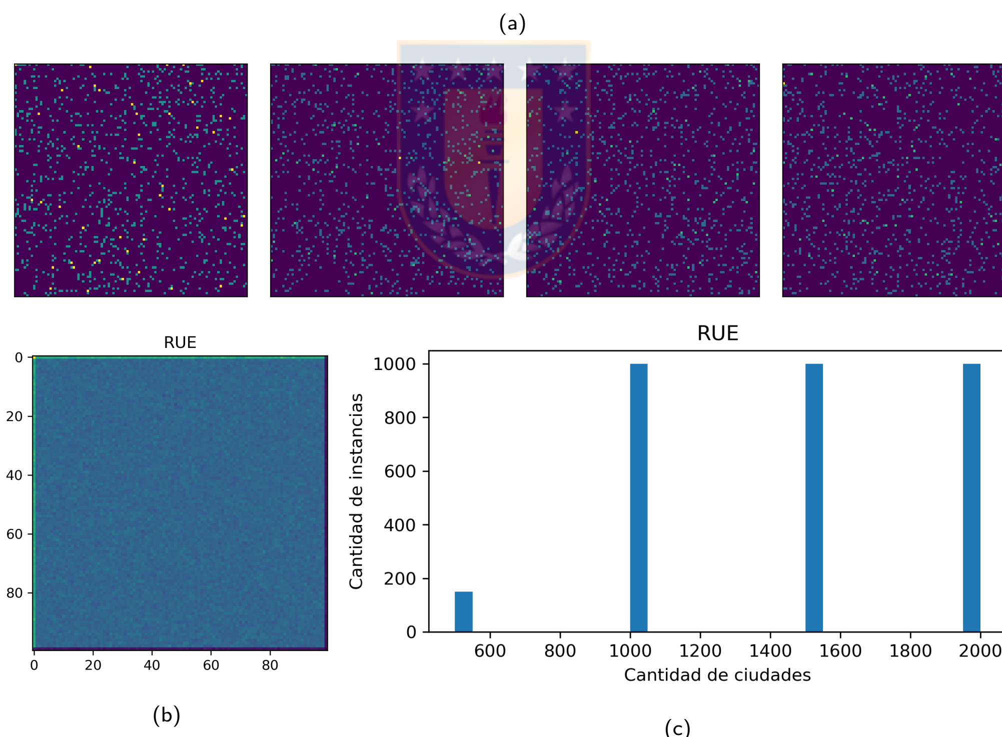


Fig. 3: En **(a)** se presentan 4 instancias de ejemplo de RUE coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de RUE. En **(c)** se muestra un histograma de la cantidad de ciudades de RUE.

[9]https://tspalgsel.github.io/
[10]http://dimacs.rutgers.edu/archive/Challenges/TSP/download.html

- **NETGEN**: Son 600 instancias cuyas ciudades estan dispuestas en 2, 5 o 10 *clusters*. Se puede dividir en 4 grupos de 150 instancias que contienen 500, 1000, 1500 y 2000 ciudades. Fueron generadas a partir del generador netgen, disponible como biblioteca de R[11], utilizando la función *generateClusteredNetwork*. Esta función primero ubica 2, 5 o 10 puntos distribuídos a lo largo del espacio utilizando un muestreo LHS [50], luego, por cada punto, se muestrean otros puntos a partir de una distribución normal multivariada en donde los puntos iniciales corresponden a la media de la distribución.

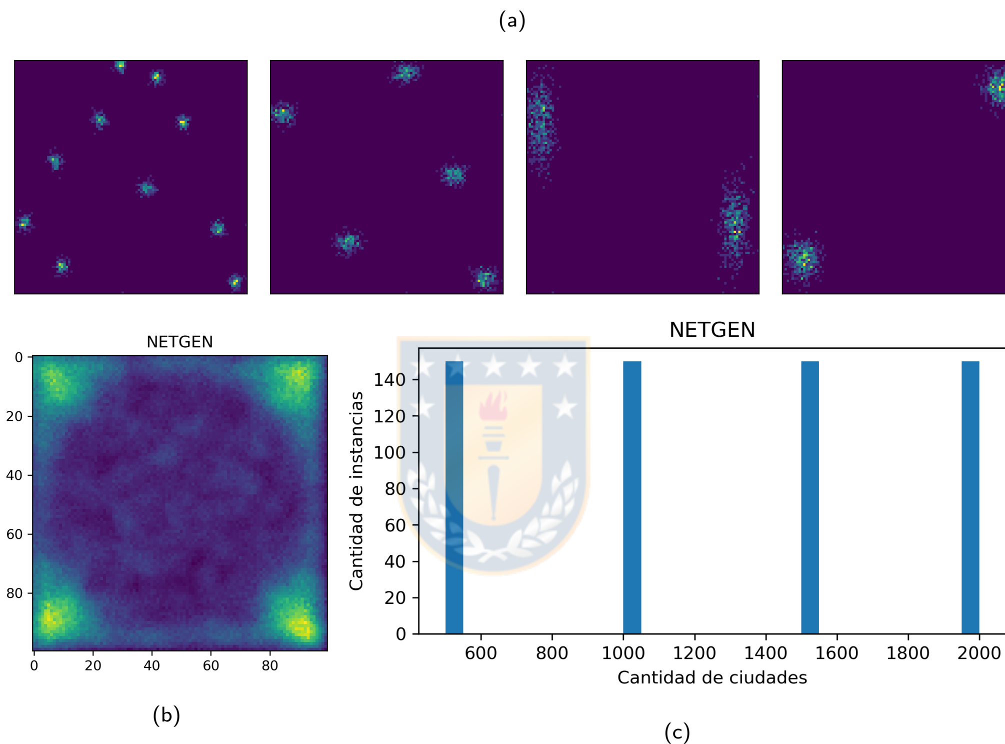


Fig. 4: En **(a)** se presentan 4 instancias de ejemplo de NETGEN coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de NETGEN. En **(c)** se muestra un histograma de la cantidad de ciudades de NETGEN.

- **NETGENM**: Son 600 instancias que combinan características de RUE y NETGEN. Fueron generadas a partir del generador netgen, disponible como biblioteca de R[11], utilizando la función *morphInstances* con un factor de *morphing* $\alpha = 0{,}5$. El algoritmo de esta función se puede dividir en dos etapas: primero, dadas dos instancias, una de RUE y otra de NETGEN, se enlaza cada punto p_1 de una instancia con el punto p_2 más cercano de la otra instancia. Luego, se reemplaza cada par de puntos enlazados con un nuevo

[11]https://rdrr.io/cran/netgen/

punto $p_3 = (\alpha p_1 + (1 - \alpha) p_2)$. Notar que $\alpha = 1$ mantendría la instancia RUE y $\alpha = 0$ mantendría la instancia NETGEN.

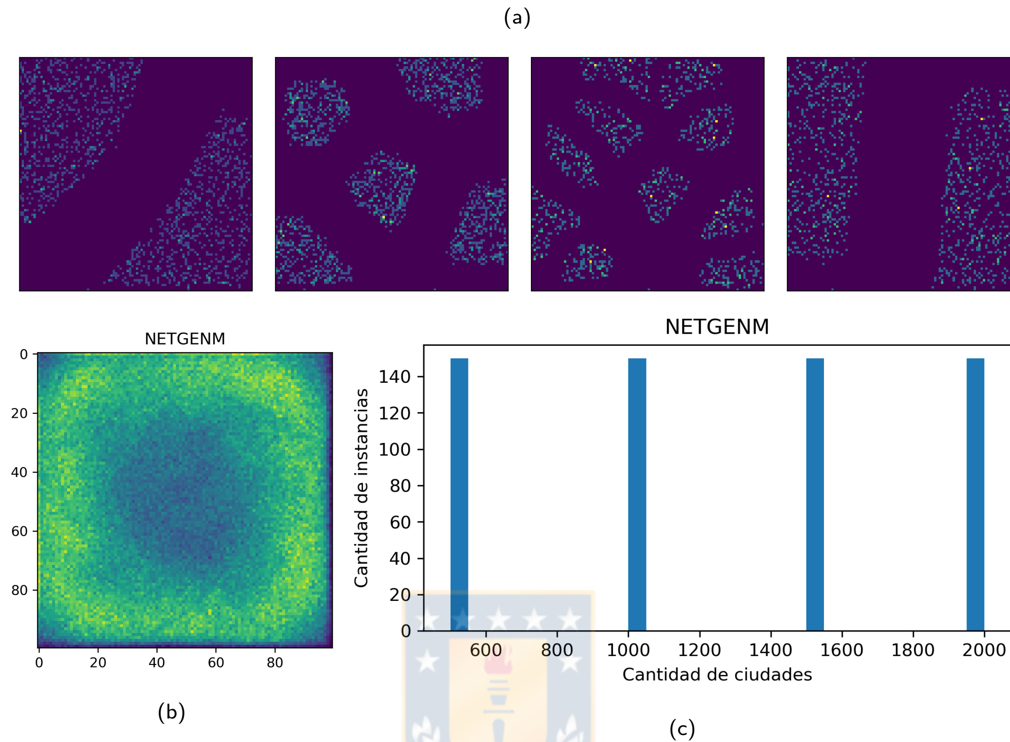


Fig. 5: En **(a)** se presentan 4 instancias de ejemplo de NETGENM coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de NETGENM. En **(c)** se muestra un histograma de la cantidad de ciudades de NETGENM.

- **TSPGEN**: Son 1981 instancias creadas a partir del generador TSPGEN[12][51], una evolución del generador netgen utilizado en los datasets anteriores. Este nuevo generador está diseñado especialmente para generar instancias diversas que permitan realizar trabajos de selección automática de algoritmos sobre TSP. Para generar las instancias se utiliza un framework de algoritmo evolutivo para combinar diversos operadores de mutación: *Normal mutation*, *Uniform mutation*, *Explosion mutation*, *Implosion mutation*, *Cluster mutation*, *Rotation mutation*, *Linear projection mutation*, *Expansion mutation*, *Compression mutation*, *Axis projection mutation* y *Grid mutation*. Estos operadores se aplican aleatoriamente y de manera iterativa sobre una instancia inicial RUE.

[12]https://github.com/jakobbossek/tspgen

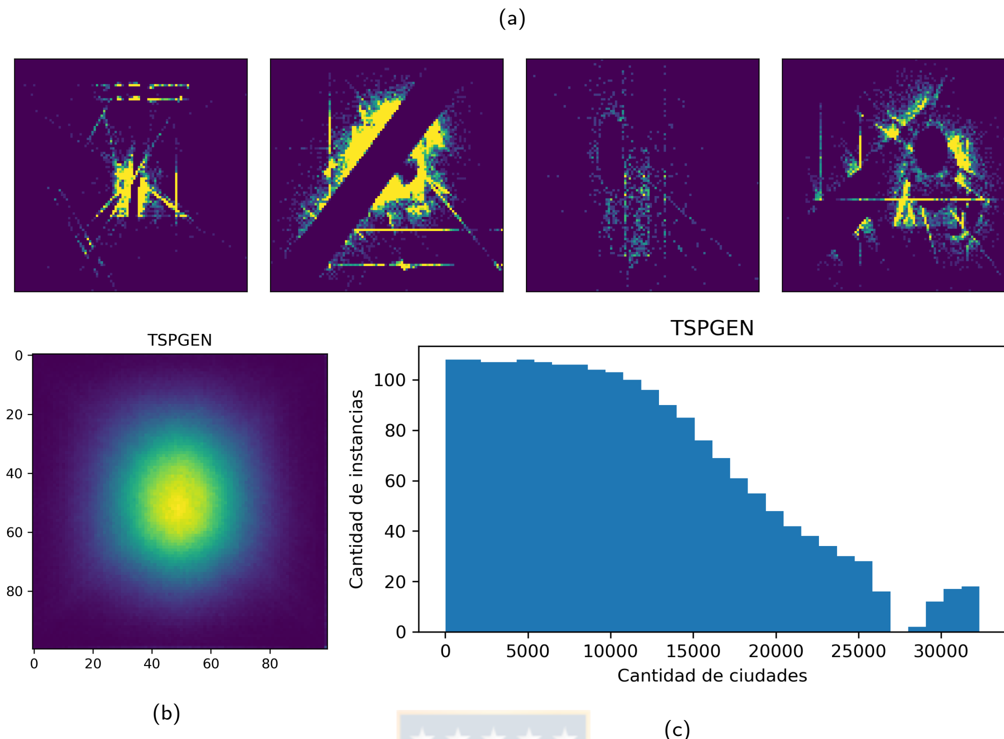


Fig. 6: En **(a)** se presentan 4 instancias de ejemplo de TSPGEN coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de TSPGEN. En **(c)** se muestra un histograma de la cantidad de ciudades de TSPGEN.

4.2.2. Instancias de validación

Las instancias de validación corresponden a instancias que se encuentran disponibles de forma pública a modo de competición para probar y comparar distintos algoritmos que resuelven TSP. En muchos casos, el óptimo es conocido debido a que ha podido ser resuelto por algoritmos exactos; sin embargo, en otros casos se desconoce el óptimo y sólo se dispone de la mejor solución que ha podido ser encontrada.

- TSPLIB: Conjunto de 88 instancias de distintos orígenes y de distintos tipos. Es el dataset más popular en comparaciones entre solvers debido a su diversidad de instancias. Las instancias se obtuvieron del sitio web de la universidad Heidelberg[13]. Para ver gráficos descriptivos ir a Anexo 7.2 Figura 23.
- TNM: Conjunto de 141 instancias obtenidas a partir de un generador de instancias de

[13]http://comopt.ifi.uni-heidelberg.de/software/TSPLIB95/tsp/

tetraedros, cuya dificultad de resolución es mayor para algoritmos exactos. Las instancias fueron descargadas del sitio web del creador del generador[14]. Para ver gráficos descriptivos ir a Anexo 7.2 Figura 24.

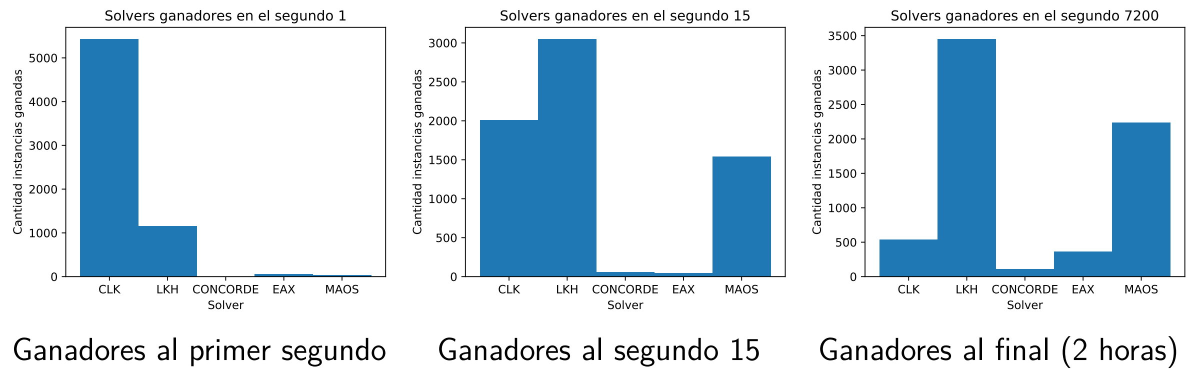
- NATIONAL: Corresponden a 27 subconjuntos de la instancia World TSP agrupados por país. Cada instancia se relaciona con un país y los nodos son las ciudades más importantes de ese país. Sólo tres instancias no han sido resueltas óptimamente. El dataset fue descargado del sitio web de TSP[15]. Para ver gráficos descriptivos ir a Anexo 7.2 Figura 25.
- VLSI: Es un conjunto de 102 instancias de integración a gran escala (VLSI) de circuitos integrados creado en la Universidad de Bonn, Alemania[16]. La importancia de TSP en este tipo de datos influye en la construcción de chips y microprocesadores con mejor rendimiento. Para ver gráficos descriptivos ir a Anexo 7.2 Figura 26.

4.3. Ejecución de cada par solver-instancia

Cada una de las instancias fue resuelta por cada solver. Los códigos de los solvers se modificaron solamente para estandarizar la forma en que se leen las instancias y para generar el archivo de salida. Por cada par solver-instancia, se generó un archivo de salida que contiene una secuencia de tuplas con la información del mejor costo y el tiempo necesario para llegar a esa solución. Cada vez que el solver encuentra una nueva mejor solución, se genera una nueva tupla.

Realizando gráficos con los resultados obtenidos (Figura 7) es fácil ver que en la mayoría de las instancias, CLK es el que obtiene los primeros resultados, LKH domina en tiempos medios y MAOS aumenta sus mejores soluciones en tiempos avanzados. En la Figura 8 se muestra el ejemplo del comportamiento a lo largo del tiempo de los solvers al resolver una instancia en particular. En la Figura 9 se muestran con distintos colores los solvers que obtuvieron primero la mejor solución encontrada. Todos los gráficos para cada dataset se encuentran en el Anexo 7.1.

[14]http://www.or.uni-bonn.de/ hougardy/HardTSPInstances.html
[15]http://www.math.uwaterloo.ca/tsp/index.html
[16]http://www.math.uwaterloo.ca/tsp/vlsi/index.html



Ganadores al primer segundo | Ganadores al segundo 15 | Ganadores al final (2 horas)

Fig. 7

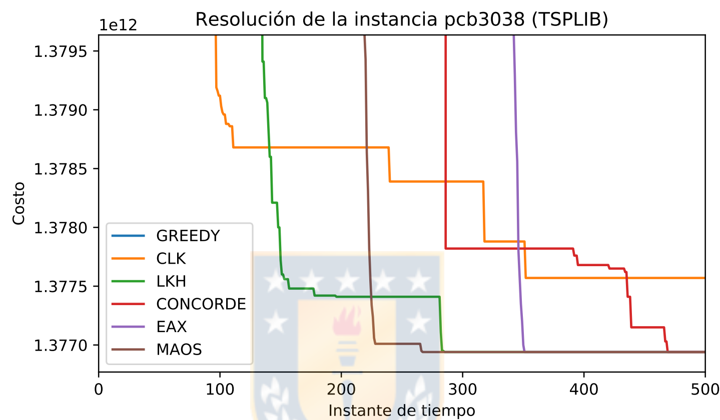


Fig. 8: Zoom de la resolución de una instancia con los 5 solvers. El solver Greedy corresponde a la primera solución entregada por CLK.

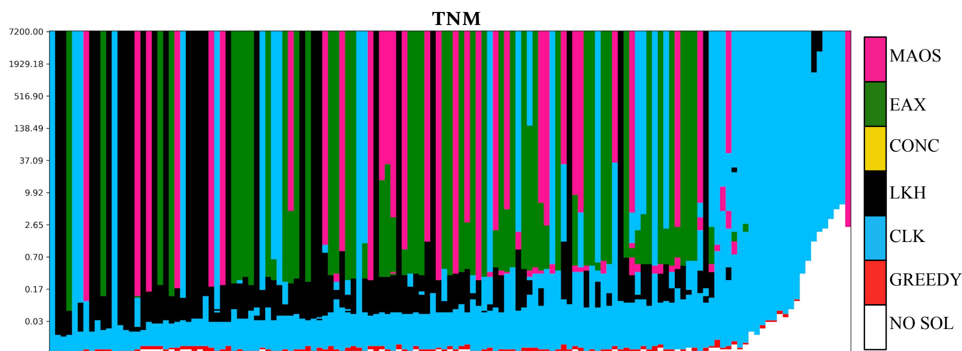


Fig. 9: Comportamiento de los mejores solvers a lo largo del tiempo. En el *eje y* se muestra el tiempo en segundos. En el *eje x* están las instancias ordenadas en forma ascendente respecto a su cantidad de ciudades. Las instancias del dataset TNM son difíciles de resolver para Concorde, en consecuencia, no se distingue una victoria de este solver en la gráfica

4.4. Caracterización de las instancias

La búsqueda de características informativas y de bajo costo computacional para la caracterización no es una tarea trivial. En muchos casos las características seleccionadas no son decisivas en el proceso de selección de algoritmo, desperdiciando capacidad de cómputo. Más aún, pueden existir características más apropiadas que no están siendo consideradas. Esto se debe principalmente a la arbitrariedad involucrada en la selección de las características. Una alternativa para evitar este problema es utilizar redes neuronales cuyos parámetros son ajustados automáticamente para considerar las características relevantes.

La caracterización debe ser un proceso rápido y que conserve la información de la instancia. Para ello, la caracterización propuesta consiste en transformar la información de las coordenadas de las ciudades en una matriz cuadrada que posea representación espacial. Así, los límites de la matriz coinciden con los límites de las coordenadas y las celdas contienen la cantidad de ciudades que se ubican en esa posición. En la Figura 10 se muestra un ejemplo en el que dada una instancia euclidiana, podemos enmarcar el conjunto de ciudades en un cuadrado considerando los puntos extremos para las dimensiones. Luego se contruye una matriz de grado N_grid en donde cada celda tiene la cantidad de ciudades que se encuentran en ese lugar. El tamaño N_grid será ajustado en la subsección 4.7.

La normalización de la representación matricial a valores entre 0 y 1 requiere conocer el valor máximo que puede tener una celda. Dado que este valor puede ser muy alto en algunas instancias, se define un límite max_cities que indica la cantidad máxima de ciudades que tendrá cada celda. Si una celda contiene más de max_cities ciudades, su valor se restringirá a max_cities. Aumentar arbitrariamente este parámetro podría incidir en que el valor de una celda con una sola ciudad sea despreciable, mientras que definir $max_cities = 1$ es una pérdida de información en la densidad de las celdas. Este límite será ajustado en la subsección 4.7.

Para aumentar la cantidad de datos en el entrenamiento, se realizaron tres rotaciones de 90, 180 y 270 grados a cada matriz de una instancia, pasando de 6.331 a 25.324 instancias en el entrenamiento. Dado que la resolución de un par de instancias rotadas es idéntico, se repite la salida para cada instancia rotada.

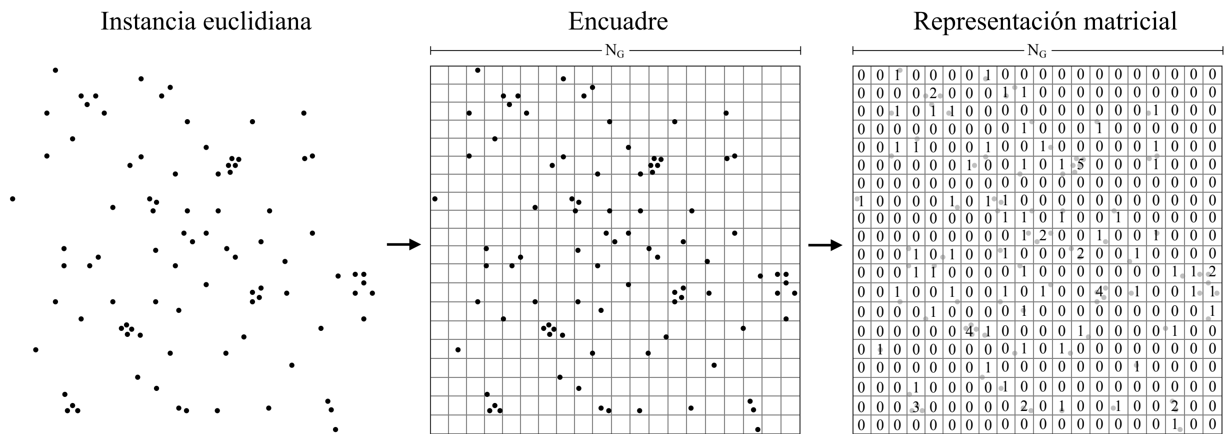


Fig. 10: Ejemplo de una caracterización.

4.5. Representación de la resolución de una instancia

La salida del modelo corresponde a una representación del grado de suboptimalidad de cada solver a lo largo de instantes de tiempo discretizados. La cantidad de instantes de tiempo se definió en 500, que contemplan desde el segundo 0 hasta el segundo 7200 (dos horas). La forma en que se distribuye el tiempo no es lineal, sino que se corresponde con una curva exponencial (Figura 11), de manera que contemple más instantes de tiempos bajos. La motivacion para considerar más tiempos bajos se debe a que en un principio se producen más cruzamientos de soluciones, mientras que en tiempos altos se estabilizan los surgimientos de nuevas soluciones.

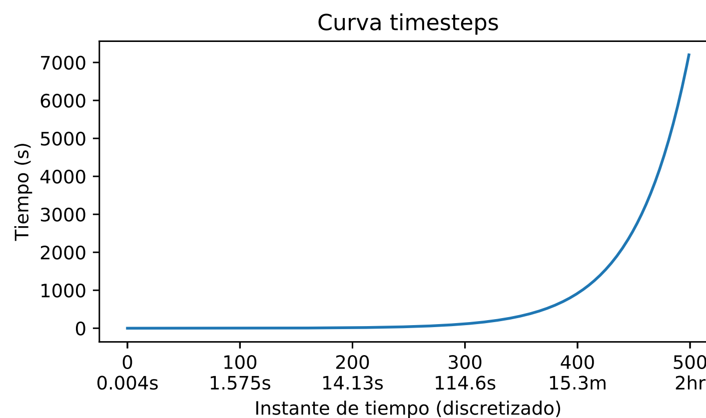


Fig. 11: Curva utilizada para la definición de los timesteps

En cada instante de tiempo, tendremos la información del grado de suboptimalidad de los cinco solvers, sin embargo, hay cuatro problemas al representar la información con el grado de suboptimalidad directamente, es decir, con el valor objetivo obtenido por cada solver:

- Diferentes instancias, dependiendo de la escala de distancias entre las ciudades, tendrán distintos órdenes de magnitud en su valor objetivo.
- En los primeros instantes de tiempo, el valor objetivo es infinito cuando no se han encontrado soluciones, luego, los valores encontrados por los solvers pasarían a ser ínfimos.
- En los instantes de tiempo avanzados, las diferencias entre las mejores soluciones aproximadas son mínimas. Pequeñas diferencias de valores harán la diferencia entre la mejor y la segunda mejor solución. Estos valores pequeños podrían ser despreciables al lado de las primeras soluciones encontradas.
- Si dos o más solvers comparten el mismo grado de suboptimalidad en un instante dado, no sabemos qué solver obtuvo la solución antes.

Una representación que no presenta estos problemas es la representación escalonada. Aquí todos los solvers comienzan en cero y todos los valores que toma posteriormente son enteros. Para construír la línea temporal con las posiciones $P_s(i,t) \in \mathbb{Z}$, se debe evaluar el grado de suboptimalidad $O_s(i,t) \in \mathbb{R}$ de cada solver $s \in \{1, ..., 5\}$ y de cada instancia $i \in I$ en el instante t respecto al instante $t-1$, donde $t \in \{1, ..., 500\}$. Si entre $t-1$ y t un solver a supera el grado de suboptimalidad de otro solver b, entonces su posición se define como $P_a(i,t) = P_b(i,t-1)+1$, y si otro solver c ya tiene ese entero, se aumenta en uno su posición $P_c(i,t) += 1$. Una excepción es el caso en que a tiene una posición adyacente a b, entonces se hace *swap* con sus posiciones. Para ser más justos con los solvers que encontraron primero una solución, se define una condición de desempate: si $O_a(i,t) = O_b(i,t)$, pero a encontró la solución primero, entonces $P_a(i,t) > P_b(i,t)$. Notar que bajo estas reglas, dos solvers sólo pueden compartir la posición 0 (estado inicial).

En el Algoritmo 1 se muestra el pseudocódigo para la construcción de la representación. Un ejemplo de esta representación se encuentra en la Figura 12, que comparte relación con la Figura 8 porque se realiza sobre la misma instancia para fines comparativos.

Algoritmo 1: Representación escalonada de una resolución para una instancia i

Input: $O_s(i,t), s \in \{1,2,3,4,5\}$

```
1  timeline ← [ ]
2  prev_positions ← [0, 0, 0, 0, 0]
3  for t = 1, ..., 500 do
4      argsorted ← argsort([O1(i,t), O2(i,t), O3(i,t), O4(i,t), O5(i,t)])
5      curr_positions ← prev_positions
6      for s = 4, ..., 0 do
7          if argsorted[s] ≠ prev_argsorted[s] then
8              if argsorted[s] = prev_argsorted[s − 1]
9              and argsorted[s − 1] = prev_argsorted[s] then
10                 Swap positions
                   curr_positions[argsorted[s]] ← curr_positions[argsorted[s − 1]]
11                 curr_positions[argsorted[s − 1]] ← curr_positions[argsorted[s]]
12             else
13                 Find the index of change and increase its position and the upper
                   positions by 1
14                 new_index ← argsorted.index(prev_argsorted[s])
15                 for ix = 0, ..., new_index do
16                     curr_positions[prev_argsorted[ix]]+ = 1
17                 end
18             end
19         end
20     end
21     timeline[t] ← curr_positions
22     prev_positions ← curr_positions
23     prev_argsorted ← argsorted
24 end
25 return timeline
```

Con esta representación, cada instante que miremos individualmente nos entregará la información del *ranking* entre los solvers (el solver con el valor más alto se encuentra en primer lugar y va disminuyendo a medida que se reduce su posición), a la vez que mantiene la información de mejora de la solución y los cruzamientos entre solvers a lo largo del tiempo. Con los experimentos realizados en este trabajo, el valor más alto que tomó la posición es 12.

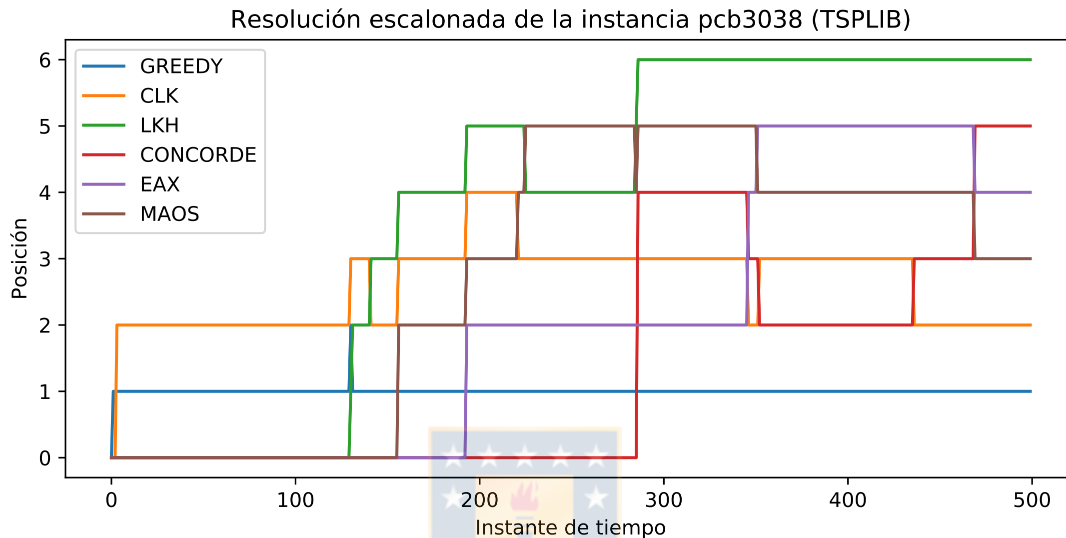


Fig. 12: Ejemplo de una representación escalonada. El sover Greedy corresponde a la primera solución entregada por CLK.

4.6. Modelos

La utilización de una caracterización matricial ofrece la posibilidad de utilizar redes neuronales convolucionales, asi como también redes Fully Connected con la matriz aplanada. En la caracterización utilizada la información de la cantidad de ciudades no es directa, menos aún cuando se utiliza un parámetro max_cities bajo porque se pierde la información de densidad en la matriz, por lo tanto, este valor será entregado a la red como un dato adicional. Notar que al generar la matriz de la caracterización se puede obtener la cantidad de ciudades N_cities sin cómputo adicional. En el caso de la red Fully Connected, la cantidad de ciudades será entregada como un valor concatenado a la matriz aplanada, por lo que la entrada será de tamaño $(N_grid \cdot N_grid + N_cities)$, mientras que en la red Convolucional, la cantidad de ciudades será entregada con un segundo canal cuya matriz tendrá el valor repetido en cada celda, por lo que la entrada será de tamaño $2(N_grid \cdot N_grid)$.

Como se vió en la Subsección 4.5, la representación de la resolución corresponderá a una lista para cada solver con valores enteros que representen su posición de tipo ranking y además aumenta a medida que existen más cruzamientos. La dimensión de estos datos es (5 solvers, 500 instantes de tiempo). Este dato será entregado a ambas redes en formato aplanado, es decir, una lista de tamaño 2500 con valores enteros. Con estos datos se pueden entrenar las redes con un enfoque regresivo y una salida lineal de la red. Sin embargo, un problema de este enfoque es que la salida puede tener valores continuos que no tienen sentido con la representación utilizada. Para evitar este problema se realiza una codificación one-hot [52] sobre la lista de tamaño 2500. El mayor valor entero que toma esta lista es 12, por lo que la codificación one-hot nos entrega una salida con dimension (2500, 12). Así, al utilizar una salida con función de activación *sigmoide*, estaremos trabajando con un modelo de clasificación.

En las figuras 13 y 14 se muestra en detalle las arquitecturas utilizadas para cada modelo. La cantidad de parámetros entrenables son 5.162.720 y 9.866.864 para el modelo Fully Connected y Convolucional respectivamente. Para el entrenamiento se utilizó un optimizador *Adam* [53] con *learning rate* 0.0001 y una función de pérdida *binary cross-entropy* [54].

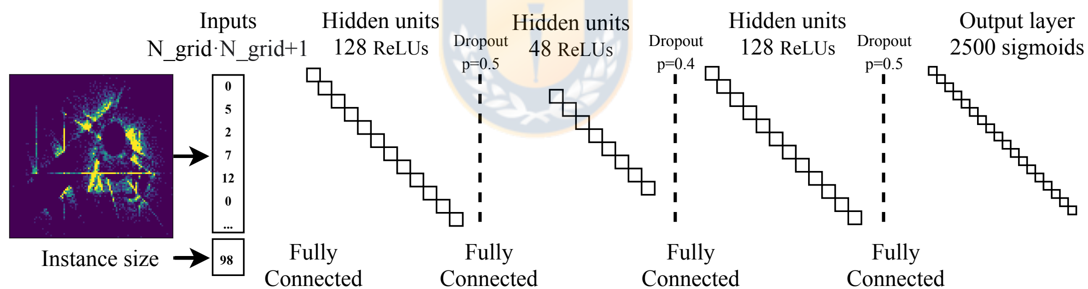


Fig. 13: Arquitectura Fully Connected

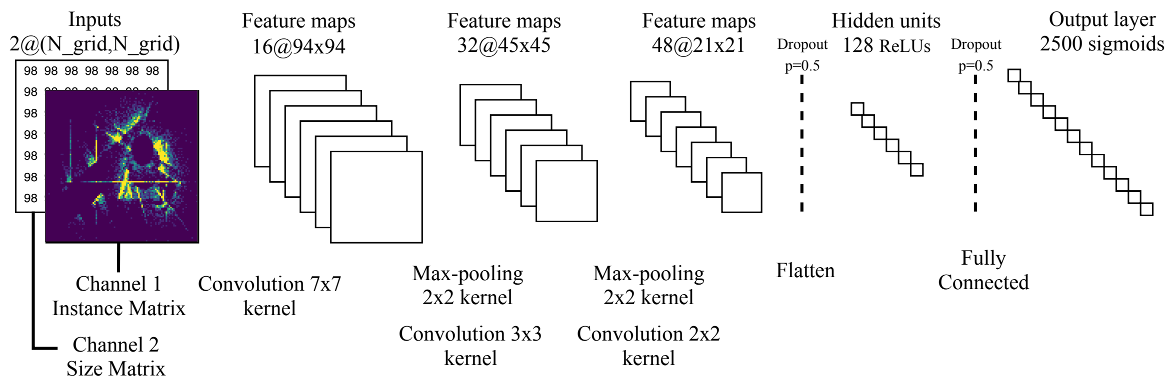


Fig. 14: Arquitectura Convolucional

4.7. Ajuste de parámetros

Hay dos parámetros que se deben ajustar:

- N_grid: es el grado de la matriz cuadrada con la que se caracterizan las instancias. Se realizarán pruebas con $N_grid \in \{100, 200, 300\}$.
- max_cities: es el límite de la cantidad de ciudades que contiene una celda en la matriz. Se realizarán pruebas con $max_cities \in \{1, 5, 10, 20\}$.

Además se debe definir el modelo definitivo entre los dos modelos propuestos (*fully connected* o convolucional), por lo tanto, la cantidad de entrenamientos realizados corresponden a 24 (2 modelos, 3 N_grid y 4 max_cities).

En la Figura 15 se muestran los resultados de las pruebas realizadas. La métrica utilizada para comparar los entrenamientos es *categorical accuracy*. Esta métrica calcula el porcentaje de valores predichos que coinciden con los valores reales en su forma one-hot [52]. Dada una predicción, se obtienen los índices de los valores más altos de cada valor de la predicción (función *argmax*) para realizar el inverso de la codifcación one-hot y se compara con las etiquetas reales. La cantidad de aciertos, es decir, la cantidad de valores cuya etiqueta predicha es igual a la etiqueta real, se divide por la cantidad total de valores (en este caso, 2500). Este porcentaje se obtiene para cada instancia y luego se promedia sobre todo el *batch*. Por ejemplo, si la lista original es [[0,1,0,0],[0,0,0,1],[1,0,0,0]] y los valores predichos son [[0.2,0.7,0.1,0],[0.05,0.05,0,0.9],[0.3,0.4,0.2,0.1]], el *argmax* respectivo de cada lista es [1,3,0] y [1,3,1], donde dos etiquetas de tres fueron acertadas, por tanto el *categorical accuracy* corresponde a $2/3 = 66\,\%$.

De los resultados podemos ver que el modelo con red Convolucional sobrepasa la calidad del modelo Fully Connected. La mejor exactitud se obtuvo con $N_grid = 100$ y respecto a max_cities es difícil discernir el mejor modelo, sin embargo, el seleccionado será $max_cities = 20$ debido a que es más informativo para instancias con mayor densidad.

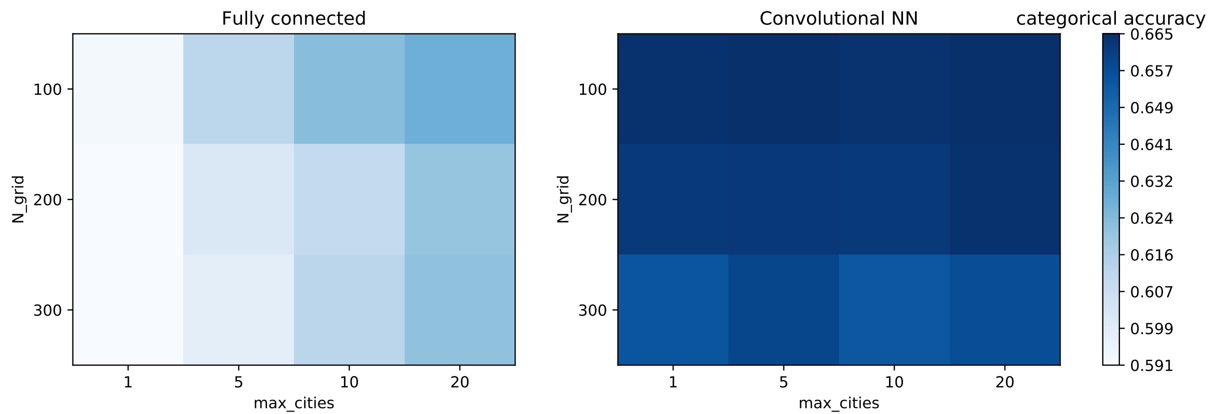


Fig. 15: Ajuste de parámetros

4.8. Configuración experimental

La ejecución de los experimentos, es decir, la ejecución de las instancias con cada solver, se realizó sobre un clúster administrado con *slurm*[17]. Cada nodo del clúster tiene un procesador Intel Xeon E3-1270 v6 de 3.80GHz con 64GB de memoria RAM y distribución Ubuntu 18.04. Las instancias se ejecutaron una vez por cada solver y se definió un límite de tiempo de dos horas y una condición de término cuando el solver no encuentra una solución nueva en un rango de 10 minutos. Por otro lado, el entrenamiento y las pruebas con redes neuronales fueron ejecutadas en un procesador Intel Core i5-10400 de 2.9Ghz con 16GB de RAM y una GPU Nvidia GeForce GTX 1650 SUPER.

[17]https://slurm.schedmd.com/documentation.html

5. RESULTADOS

Con el modelo convolucional descrito y los parámetros $N_grid = 100$ y $max_cities = 20$, se reentrenó el modelo utilizando un learning rate más bajo de $1 \cdot 10^{-5}$, obteniendo un $categorical\ accuracy$ de 0.66782 al final del entrenamiento. Con este modelo se realizó la predicción sobre las instancias de validación y se calculó, por cada instancia, el tiempo necesario para realizar la predicción, es decir, la lectura de la instancia, la creación de la matriz y la predicción sobre el modelo. Los resultados posteriores se muestran con dos enfoques de evaluación: por un lado contemplando un tiempo de predicción nulo y por otro lado añadiendo el tiempo de predicción al tiempo de ejecución total. En la Figura 20 se muestra cuánto tiempo, en promedio, requiere el proceso de predicción y qué tanto aporta en este valor a cada subproceso.

Para calcular el $accuracy$ real de los solvers, incluyendo el Metasolver, se calculó el porcentaje de instantes de tiempo que el solver obtuvo el mejor valor objetivo. Si $cost_s(t)$ es el valor objetivo obtenido por el solver s en el instante t, donde $t \in \{1, 2, ..., 500\}$, y $best_cost(t)$ es el mejor valor objetivo obtenido entre todos los solvers hasta el instante t, entonces el $accuracy$ de un solver s para una instancia será definido como

$$accuracy(s) = \frac{\sum_{t=1}^{500} equal(cost_s(t),\ best_cost(t))}{500}$$

$$\text{donde} \quad equal(a, b) = \begin{matrix} 1 & \text{si } a = b \\ 0 & \text{si } a \neq b \end{matrix}$$

Recordemos que 500 corresponde a la cantidad de instantes de tiempo considerados. Con esta métrica calculamos el promedio sobre cada dataset de validación, cuyos resultados se muestran en la tabla 2.

En las figuras 16, 17, 18 y 19 se muestra el comportamiento a lo largo del tiempo del $accuracy$ de los solvers sobre los datasets públicos. En los gráficos de la izquierda se muestra la línea temporal, donde el *eje x* corresponde a los instantes de tiempo y el *eje y* a la cantidad de veces que un solver ha obtenido el mejor valor objetivo, mientras que en el gráfico de la derecha se muestra el área bajo la curva de la línea temporal.

accuracy (%)	**TSPLIB**	**VLSI**	**NATIONAL**	**TNM**	**TODOS**
Metasolver (sin tiempo de predicción)	90.0	91.3	89.5	90.8	90.7
Metasolver (con tiempo de predicción)	**82.7**	**73.2**	**66.9**	**85.1**	**79.8**
CLK	63.1	57.5	61.8	77.2	67.0
LKH	62.1	55.0	44.0	42.4	51.0
CONCORDE	50.2	15.8	18.5	19.1	25.8
EAX	39.2	32.2	22.2	54.1	41.8
MAOS	46.2	42.2	01.4	67.6	50.1

Tabla 2: Accuracy promedio de cada solver sobre cada dataset y sobre el conjunto de validación completo. Los resultados muestran una clara superioridad para ambas versiones del Metasolver.

Los gráficos del área bajo la curva nos muestran que el metasolver es capaz de ser superior en todos los datasets públicos bajo esta métrica, sin embargo, es necesario hacer notar el umbral desde el cual el metasolver fue capaz de obtener, en promedio, mejores soluciones. En todos los datasets, CLK logra tener soluciones muy rápidas y es superior al metasolver en los tiempos iniciales. En TSPLIB este umbral se encuentra a los 0.578 segundos, en VLSI a los 5.18 segundos, en NATIONAL a los 16.73 segundos y en TNM se supera a LKH a los 0.158 segundos. En TNM, CLK supera por un estrecho margen al metasolver desde los dos minutos. Una hipótesis de porqué sucede este fenómeno en TNM es que el dataset posee instancias muy similares entre sí (ver Anexo 7.2, Figura 24c), por lo que si un solver es bueno resolviéndolo, como es el caso de CLK, entonces será bueno siempre.

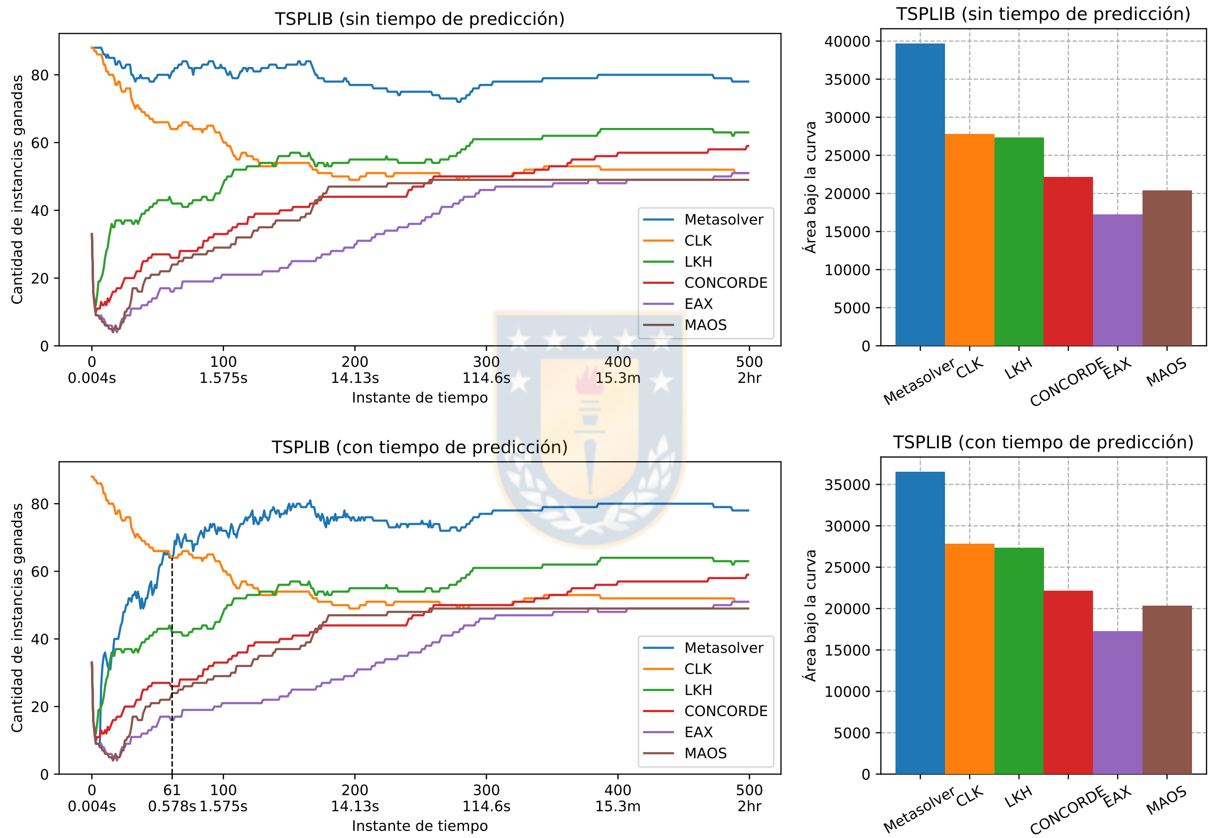


Fig. 16: Resultados con el dataset TSPLIB

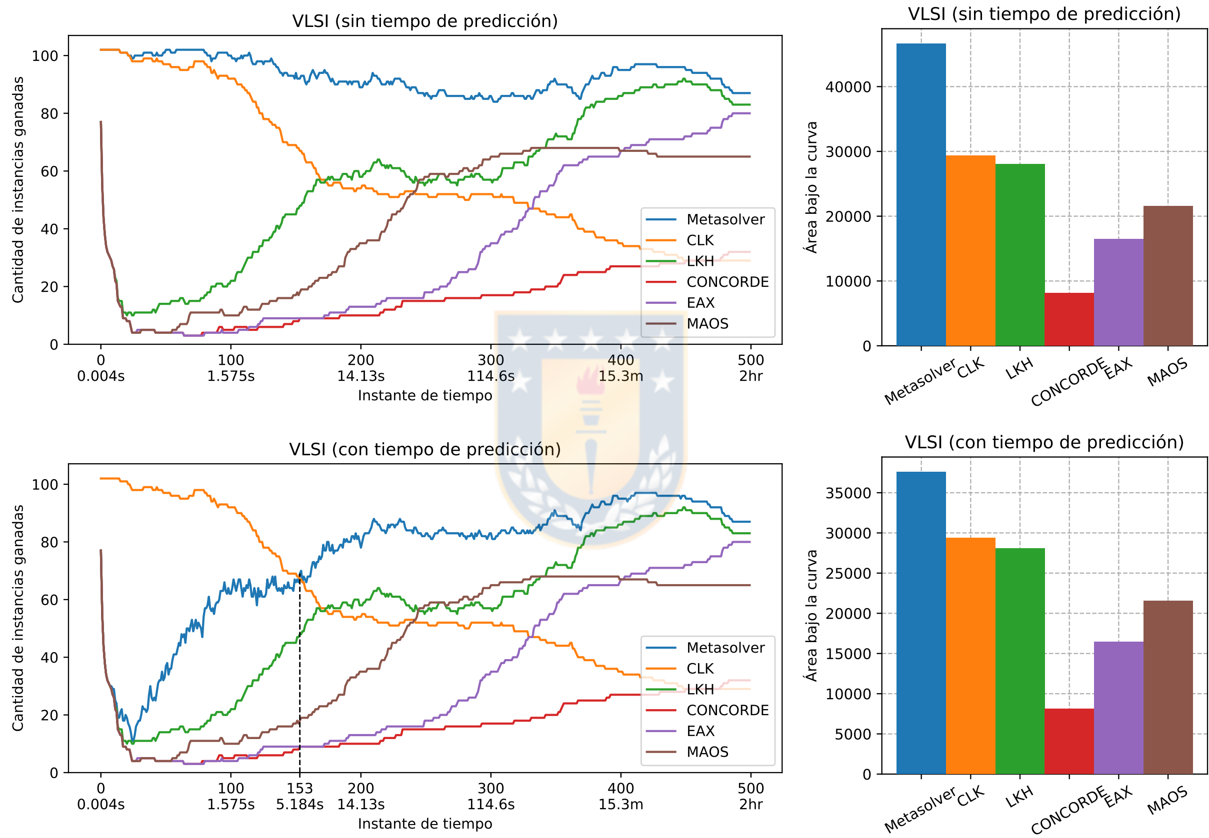


Fig. 17: Resultados con el dataset VLSI

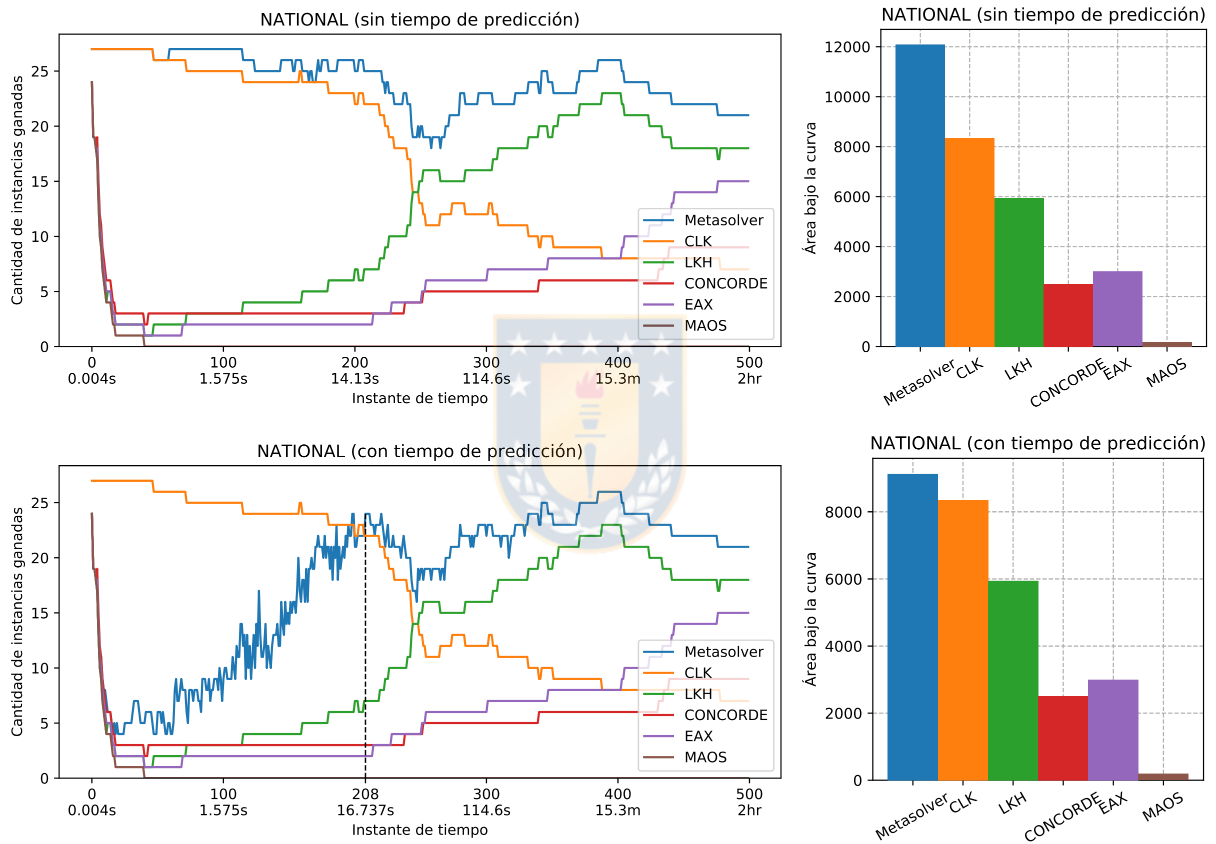


Fig. 18: Resultados con el dataset NATIONAL

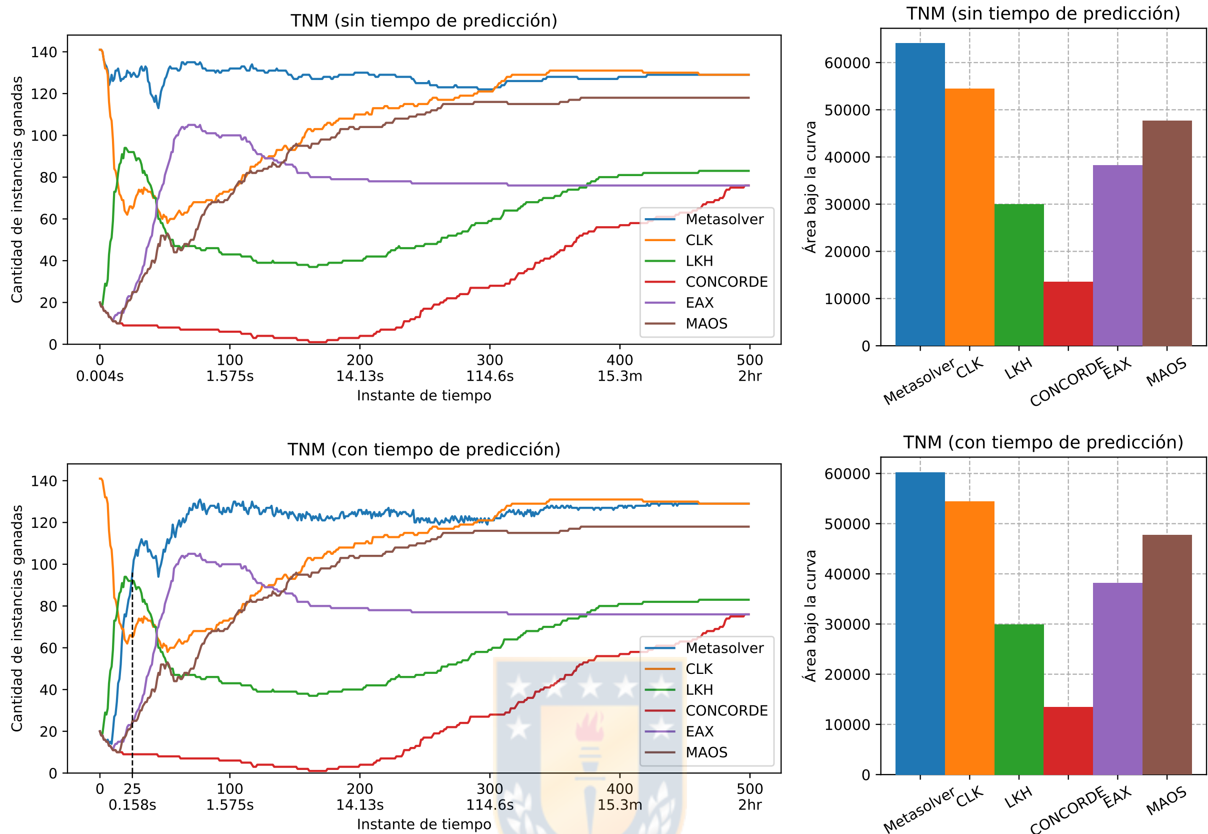


Fig. 19: Resultados con el dataset TNM

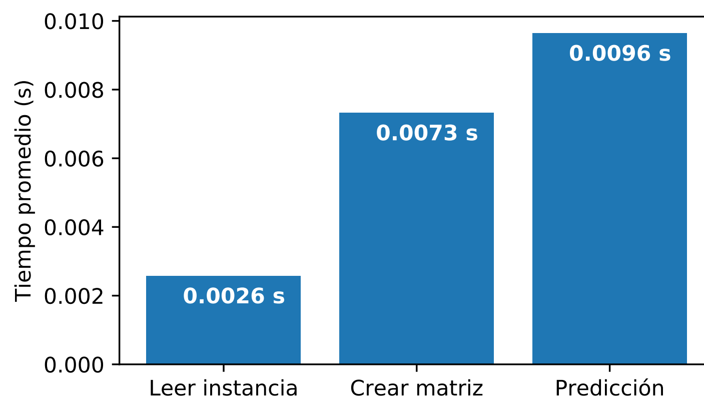


Fig. 20: Medida promedio, dividida en subtareas, del tiempo necesario para realizar la predicción. Para calcularlo se utilizaron todas las instancias de validación. El tiempo promedio total para realizar una predicción es de 0.01954 segundos en promedio.

6. CONCLUSIONES Y TRABAJO FUTURO

Se presentó un nuevo modelo de selección automática de algoritmos capaz de superar ampliamente el rendimiento de solvers estado del arte en todos los datasets públicos utilizados, incluso considerando el tiempo de predicción. El modelo es capaz de predecir el mejor solver, de un abanico de 5 solvers, en 500 instantes de tiempo distintos, que comprenden desde 0 hasta 7200 segundos con un accuracy de 79.8 % (ver tabla 2).

En trabajos anteriores similares [12, 48], el principal problema recae en la selección de características apropiadas para caracterizar las instancias. Seleccionar pocas características puede significar una mala representación de las instancias, mientras que seleccionar muchas significa aumentar el tiempo de cómputo de las características, y por tanto, de la predicción. Por ejemplo, calcular el conjunto de características TSPmeta[18] [55](68 características) o full UBC [56](50 características) puede tomar un tiempo entre 9 a 12 segundos (por instancia) en computar. En este trabajo evitamos el cálculo de las características, logrando un tiempo de predicción de 0.02 segundos en promedio por instancia.

Si bien los solvers y las instancias utilizadas en [12] mantienen relación con las de este trabajo, se aumentó la cantidad de instancias totales de 1845 a 6689, y se incorporó el solver exacto Concorde, por lo que el estado del arte contemplado en este trabajo no sólo se reduce a solvers aproximados. Además, se aumentó el tiempo límite de ejecución de los experimentos de 3600 a 7200 segundos.

La utilización del enfoque anytime no sólo ofrece la posibilidad de responder a consultas basadas en un límite temporal, sino que también resulta más informativo para el modelo clasificador, permitiendo aumentar la precisión del modelo y también generalizar mejor para comportamientos desconocidos.

Los logros principales de este trabajo se resumen en:

- Se desarrolló un marco de trabajo apropiado para trabajar con redes neuronales en la selección automática de algoritmos sobre TSP euclidiano, que consiste en una nueva caracterización matricial de las instancias, evitando el costoso cálculo de características

[18] https://github.com/berndbischl/tspmeta

tradicionales y permitiendo la utilización de redes neuronales convolucionales en el modelo clasificador.

- Se diseñó una representación de la resolución de una instancia que, por un lado mantiene la información de tipo ranking a lo largo del tiempo y, por otro, permite ser utilizada en la salida de una red neuronal clasificadora.

6.1. Trabajo Futuro

Si bien los tiempos de predicción alcanzados en este trabajo son muy bajos, es posible optimizar aún más el tiempo cuando sólo queremos responder a la consulta "¿Qué solver me resuelve mejor la instancia i en el tiempo t?". Aquí sólo necesitamos conocer la información de lo que sucede en el instante t y no es necesario predecir todo lo que sucede en los otros 499 instantes de tiempo. Por lo tanto, se pueden *apagar* los nodos irrelevantes en la capa de salida de la red neuronal al momento de realizar la predicción, reduciendo la cantidad de parámetros a calcular.

La naturaleza temporal de la capa de salida en el enfoque anytime puede ser mejor aprendida por arquitecturas especializadas en este tipo de datos como las redes neuronales recurrentes.

Una novedad del trabajo presentado es la utilización del enfoque anytime, sin embargo, la aplicación práctica del modelo generado sólo tiene sentido si se aplica sobre la misma arquitectura (CPU, Memoria, etc.) del sistema en donde se realizaron los experimentos, de otra forma, los tiempos de predicción no coinciden. Se propone explorar la idea de generar un mapeo de los tiempos dependiendo de los recursos disponibles en otros sistemas.

Así como el tiempo de ejecución es considerado una dimensión importante en este trabajo, esta idea también se puede expandir a los recursos computacionales. Por ejemplo si un solver necesita almacenar un volumen alto de datos en memoria en RAM (como podría ser un solver de programación dinámica) pero el sistema utilizado no posee los recursos suficientes, sería apropiado considerar esta variable al momento de seleccionar el mejor solver.

Si para un problema de optimización se pueden escoger solvers complementarios, es decir, que pueden ser capaces de ser interrumpidos y entregar su salida a otro solver, entonces la

selección automática a lo largo del tiempo no sólo es aplicable para escoger, en un principio, el mejor solver, sino que también podría permitir el cambio de solvers en tiempos intermedios de la resolución.

Referencias

[1] D. H. Wolpert and W. G. Macready, "No free lunch theorems for optimization," *IEEE transactions on evolutionary computation*, vol. 1, no. 1, pp. 67–82, 1997.

[2] A. Guerri and M. Milano, "Learning techniques for automatic algorithm portfolio selection," in *ECAI*, vol. 16, 2004, p. 475.

[3] J. R. Rice *et al.*, "The algorithm selection problem," *Advances in computers*, vol. 15, no. 65-118, p. 5, 1976.

[4] L. Kotthoff, "Algorithm selection for combinatorial search problems: A survey," in *Data Mining and Constraint Programming*. Springer, 2016, pp. 149–190.

[5] I. I. Huerta, D. A. Neira, D. A. Ortega, V. Varas, J. Godoy, and R. Asín-Achá, "Anytime automatic algorithm selection for knapsack," *Expert Systems with Applications*, p. 113613, 2020.

[6] W. Cook, D. Espinoza, and M. Goycoolea, "A study of domino-parity and k-parity constraints for the tsp," in *International Conference on Integer Programming and Combinatorial Optimization*. Springer, 2005, pp. 452–467.

[7] H. J. Kelley, "Gradient theory of optimal flight paths," *Ars Journal*, vol. 30, no. 10, pp. 947–954, 1960.

[8] K. Fukushima and S. Miyake, "Neocognitron: A self-organizing neural network model for a mechanism of visual pattern recognition," in *Competition and cooperation in neural nets*. Springer, 1982, pp. 267–285.

[9] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, "Learning representations by back-propagating errors," *nature*, vol. 323, no. 6088, pp. 533–536, 1986.

[10] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in neural information processing systems*, 2017, pp. 5998–6008.

[11] A. S. Fukunaga, "Genetic algorithm portfolios," in *Proceedings of the 2000 Congress on Evolutionary Computation. CEC00 (Cat. No. 00TH8512)*, vol. 2. IEEE, 2000, pp. 1304–1311.

[12] P. Kerschke, L. Kotthoff, J. Bossek, H. H. Hoos, and H. Trautmann, "Leveraging tsp solver complementarity through machine learning," *Evolutionary computation*, vol. 26, no. 4, pp. 597–620, 2018.

[13] M. Padberg and G. Rinaldi, "A branch-and-cut algorithm for the resolution of large-scale symmetric traveling salesman problems," *SIAM review*, vol. 33, no. 1, pp. 60–100, 1991.

[14] M. Grötschel and O. Holland, "Solution of large-scale symmetric travelling salesman problems," *Mathematical Programming*, vol. 51, no. 1-3, pp. 141–202, 1991.

[15] D. Applegate, R. Bixby, V. Chvátal, and W. Cook, *Finding cuts in the TSP (A preliminary report)*. Citeseer, 1995, vol. 95, no. 05.

[16] V. C. W. C. D Applegate, R Bixby, "Concorde tsp solver," 2006.

[17] E. L. Lawler, "The traveling salesman problem: a guided tour of combinatorial optimization," *Wiley-Interscience Series in Discrete Mathematics*, 1985.

[18] G. Laporte, "The traveling salesman problem: An overview of exact and approximate algorithms," *European Journal of Operational Research*, vol. 59, no. 2, pp. 231–247, 1992.

[19] G. Reinelt, *The traveling salesman: computational solutions for TSP applications*. Springer-Verlag, 1994.

[20] S. Lin and B. W. Kernighan, "An effective heuristic algorithm for the traveling-salesman problem," *Operations research*, vol. 21, no. 2, pp. 498–516, 1973.

[21] K. Helsgaun, "An effective implementation of the lin–kernighan traveling salesman heuristic," *European Journal of Operational Research*, vol. 126, no. 1, pp. 106–130, 2000.

[22] ——, "Using popmusic for candidate set generation in the lin-kernighan-helsgaun tsp solver," *Roskilde Universitet*, vol. 7, 2018.

[23] D. Applegate, W. Cook, and A. Rohe, "Chained lin-kernighan for large traveling salesman problems," *INFORMS Journal on Computing*, vol. 15, no. 1, pp. 82–92, 2003.

[24] M. Dorigo, V. Maniezzo, and A. Colorni, "Ant system: optimization by a colony of cooperating agents," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 26, no. 1, pp. 29–41, 1996.

[25] T. Stützle and H. H. Hoos, "Max–min ant system," *Future generation computer systems*, vol. 16, no. 8, pp. 889–914, 2000.

[26] R. C. Eberhart, Y. Shi, and J. Kennedy, *Swarm intelligence*. Elsevier, 2001.

[27] X. H. Shi, Y. C. Liang, H. P. Lee, C. Lu, and Q. Wang, "Particle swarm optimization-based algorithms for tsp and generalized tsp," *Information processing letters*, vol. 103, no. 5, pp. 169–176, 2007.

[28] J. R. Sampson, "Adaptation in natural and artificial systems (john h. holland)," 1976.

[29] P. Merz and B. Freisleben, "Memetic algorithms for the traveling salesman problem," *Complex Systems*, vol. 13, no. 4, pp. 297–346, 2001.

[30] R. Baraglia, J. I. Hidalgo, and R. Perego, "A hybrid heuristic for the traveling salesman problem," *IEEE Transactions on evolutionary computation*, vol. 5, no. 6, pp. 613–622, 2001.

[31] H.-K. Tsai, J.-M. Yang, Y.-F. Tsai, and C.-Y. Kao, "An evolutionary algorithm for large traveling salesman problems," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 34, no. 4, pp. 1718–1729, 2004.

[32] Y. Nagata and S. Kobayashi, "Edge assembly crossover: A high-power genetic algorithm for the travelling salesman problem," *Proceedings of the 7th International Conference on Genetic Algorithms (ICGA)*, pp. 450–457, 1997.

[33] ——, "A powerful genetic algorithm using edge assembly crossover for the traveling salesman problem," *INFORMS Journal on Computing*, vol. 25, no. 2, pp. 346–363, 2013.

[34] X.-F. Xie and J. Liu, "Multiagent optimization system for solving the traveling salesman problem (tsp)," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, vol. 39, no. 2, pp. 489–502, 2008.

[35] J.-Y. Potvin, "State-of-the-art survey—the traveling salesman problem: A neural network perspective," *ORSA Journal on Computing*, vol. 5, no. 4, pp. 328–348, 1993.

[36] J.-S. Chen, X.-Y. Zhang, and J.-J. Chen, "An elastic net method for solving the traveling salesman problem," in *2007 International Conference on Wavelet Analysis and Pattern Recognition*, vol. 2. IEEE, 2007, pp. 608–612.

[37] Ł. Brocki and D. Koržinek, "Kohonen self-organizing map for the traveling salesperson problem," in *Recent Advances in Mechatronics*. Springer, 2007, pp. 116–119.

[38] R. Andresol, M. Gendreau, and J.-Y. Potvin, "A hopfield-tank neural network model for the generalized traveling salesman problem," in *Meta-Heuristics*. Springer, 1999, pp. 393–402.

[39] B. F. La Maire and V. M. Mladenov, "Comparison of neural networks for solving the travelling salesman problem," in *11th Symposium on Neural Network Applications in Electrical Engineering*. IEEE, 2012, pp. 21–24.

[40] N. Fujimoto and S. Tsutsui, "A highly-parallel tsp solver for a gpu computing platform," in *International Conference on Numerical Methods and Applications*. Springer, 2010, pp. 264–271.

[41] K. Rocki and R. Suda, "High performance gpu accelerated local optimization in tsp," in *2013 IEEE International Symposium on Parallel & Distributed Processing, Workshops and Phd Forum*. IEEE, 2013, pp. 1788–1796.

[42] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Satzilla: portfolio-based algorithm selection for sat," *Journal of artificial intelligence research*, vol. 32, pp. 565–606, 2008.

[43] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm portfolios based on cost-sensitive hierarchical clustering," in *Twenty-Third International Joint Conference on Artificial Intelligence*. Citeseer, 2013.

[44] S. Kadioglu, Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellmann, "Algorithm selection and scheduling," in *International Conference on Principles and Practice of Constraint Programming*. Springer, 2011, pp. 454–469.

[45] P. Kerschke and H. Trautmann, "Automated algorithm selection on continuous black-box problems by combining exploratory landscape analysis and machine learning," *Evolutionary computation*, vol. 27, no. 1, pp. 99–127, 2019.

[46] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Hydra-mip: Automated algorithm configuration and selection for mixed integer programming," in *RCRA workshop on experimental evaluation of algorithms for solving problems with combinatorial explosion at the international joint conference on artificial intelligence (IJCAI)*, 2011, pp. 16–30.

[47] M. Vallati, L. Chrpa, and D. Kitchin, "Asap: an automatic algorithm selection approach for planning," *International Journal on Artificial Intelligence Tools*, vol. 23, no. 06, p. 1460032, 2014.

[48] J. Pihera and N. Musliu, "Application of machine learning to algorithm selection for tsp," in *2014 IEEE 26th International Conference on Tools with Artificial Intelligence*. IEEE, 2014, pp. 47–54.

[49] C. Gavidia-Calderon and C. B. Castañon, "Isula: A java framework for ant colony algorithms," *SoftwareX*, vol. 11, p. 100400, 2020.

[50] M. D. McKay, R. J. Beckman, and W. J. Conover, "A comparison of three methods for selecting values of input variables in the analysis of output from a computer code," *Technometrics*, vol. 42, no. 1, pp. 55–61, 2000.

[51] J. Bossek, P. Kerschke, A. Neumann, M. Wagner, F. Neumann, and H. Trautmann, "Evolving diverse tsp instances by means of novel and creative mutation operators," in *Proceedings of the 15th ACM/SIGEVO Conference on Foundations of Genetic Algorithms*, 2019, pp. 58–71.

[52] D. Harris and S. Harris, *Digital design and computer architecture*. Morgan Kaufmann, 2010.

[53] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

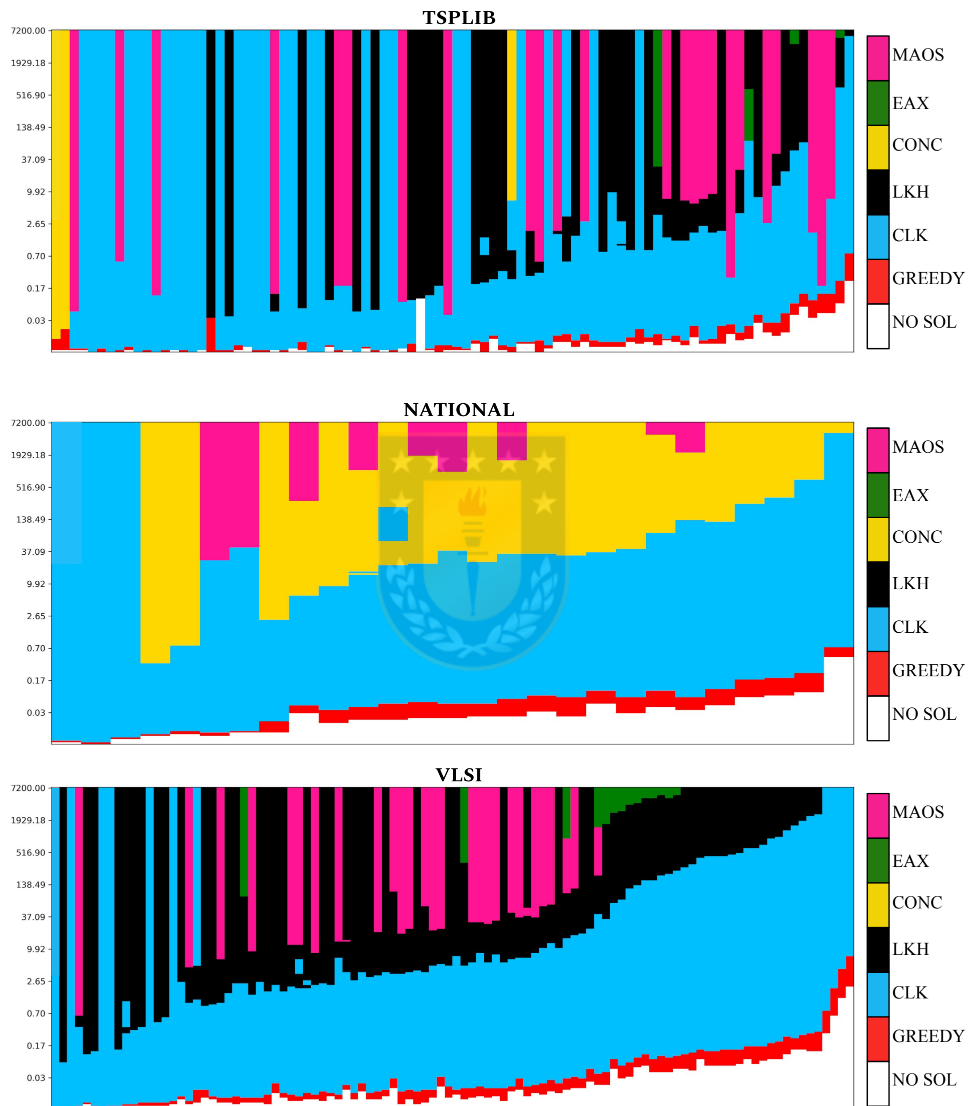
[54] D. R. Cox, "The regression analysis of binary sequences," *Journal of the Royal Statistical Society: Series B (Methodological)*, vol. 20, no. 2, pp. 215–232, 1958.

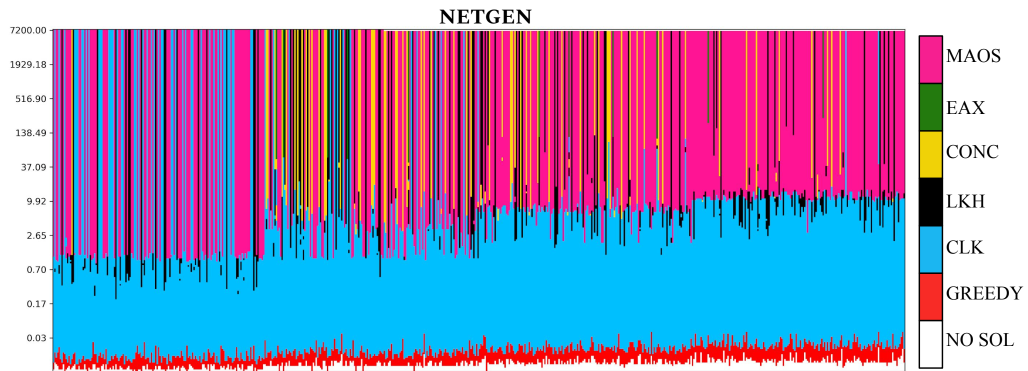
[55] O. Mersmann, B. Bischl, H. Trautmann, M. Wagner, J. Bossek, and F. Neumann, "A novel feature-based approach to characterize algorithm performance for the traveling salesperson problem," *Annals of Mathematics and Artificial Intelligence*, vol. 69, no. 2, pp. 151–182, 2013.

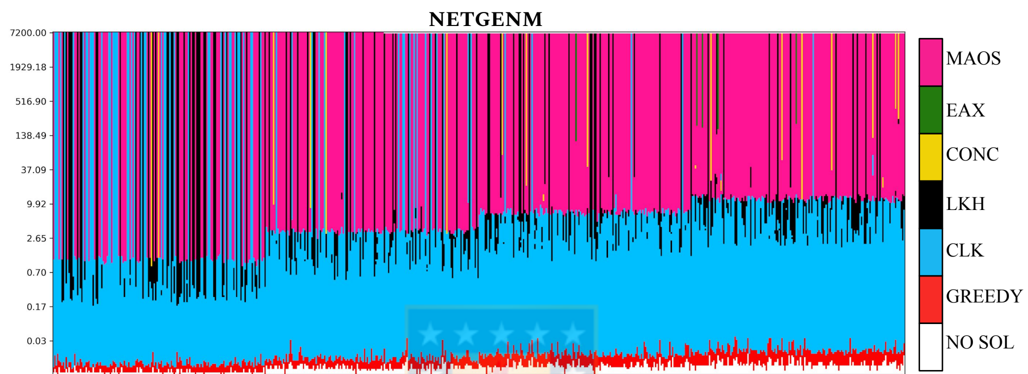
[56] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

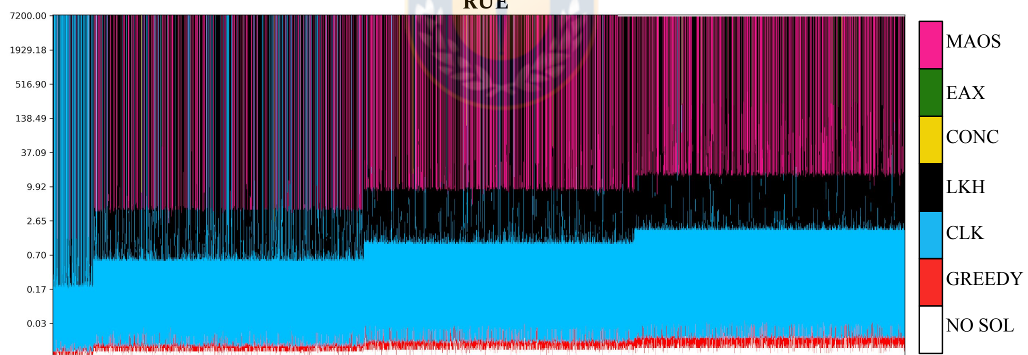
7. Anexos

7.1. Gráficos por dataset del comportamiento de los solvers

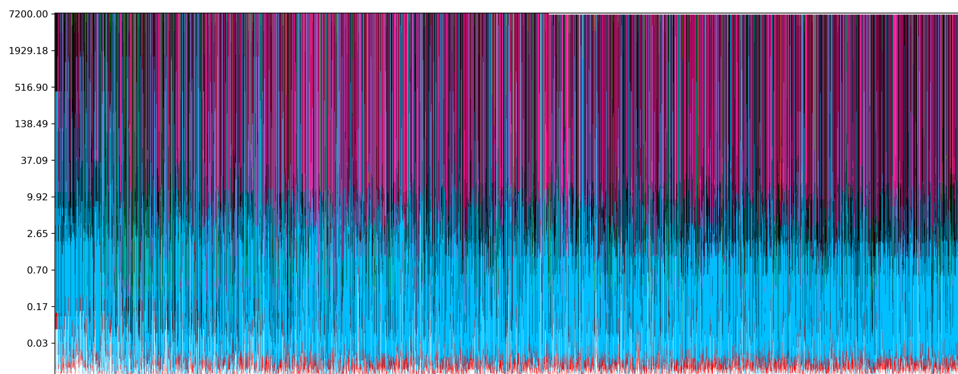








Todas las instancias en orden aleatorio.



7.2. Gráficos descriptivos de las instancias públicas

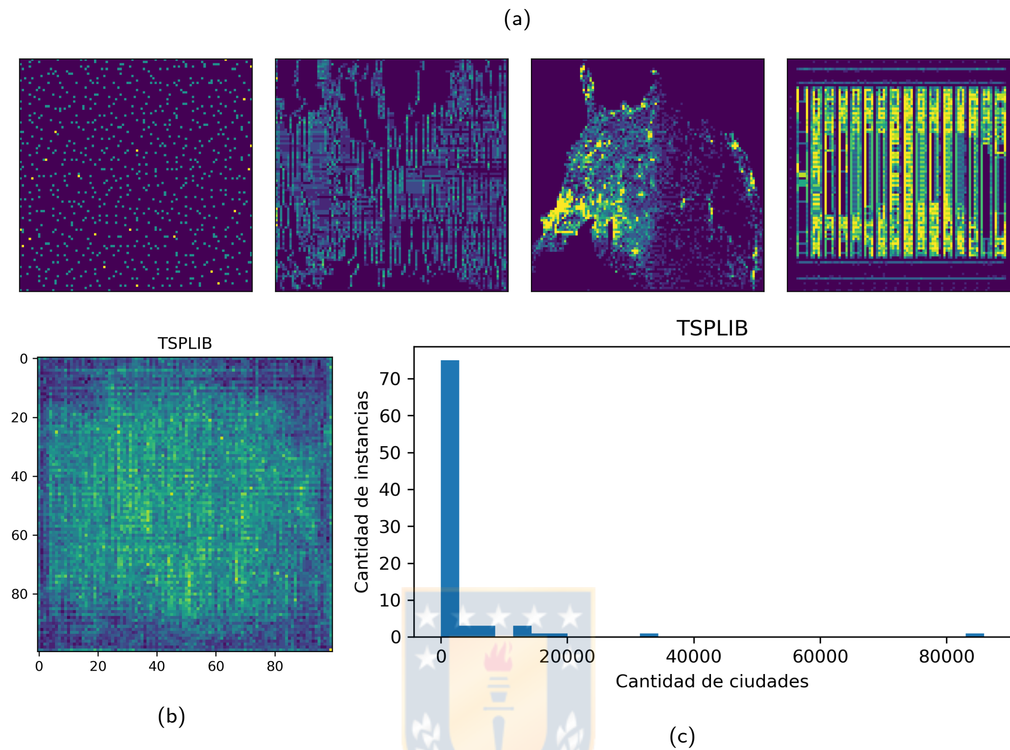


Fig. 23: En **(a)** se presentan 4 instancias de ejemplo de TSPLIB coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de TSPLIB. En **(c)** se muestra un histograma de la cantidad de ciudades de TSPLIB.

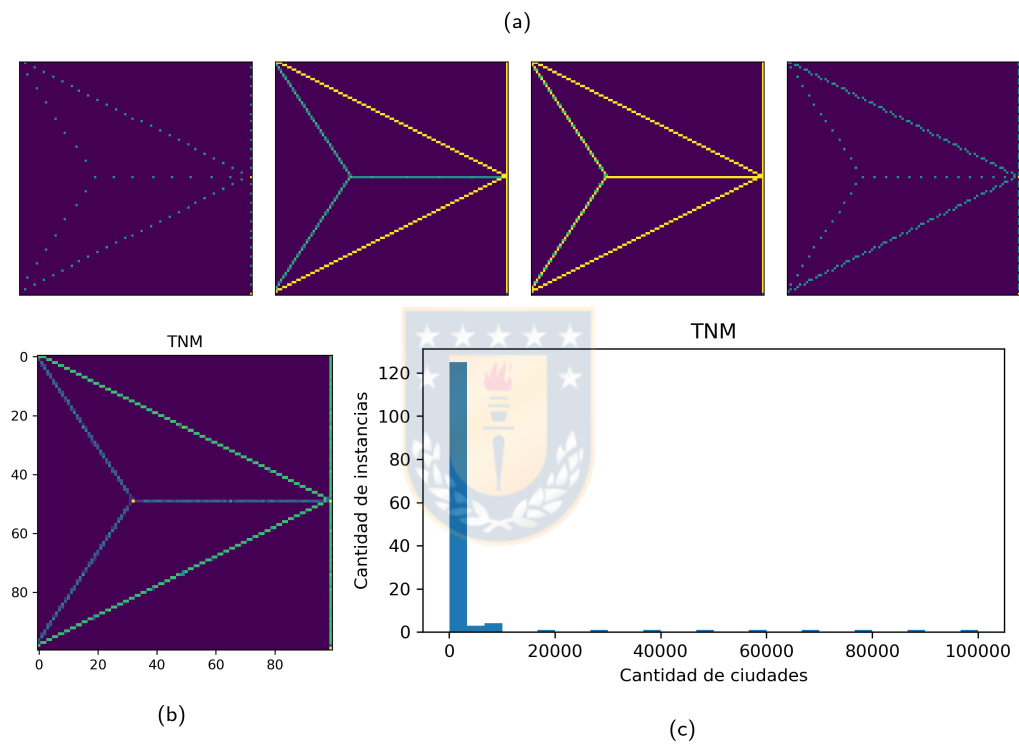


Fig. 24: En **(a)** se presentan 4 instancias de ejemplo de TNM coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de TNM. En **(c)** se muestra un histograma de la cantidad de ciudades de TNM.

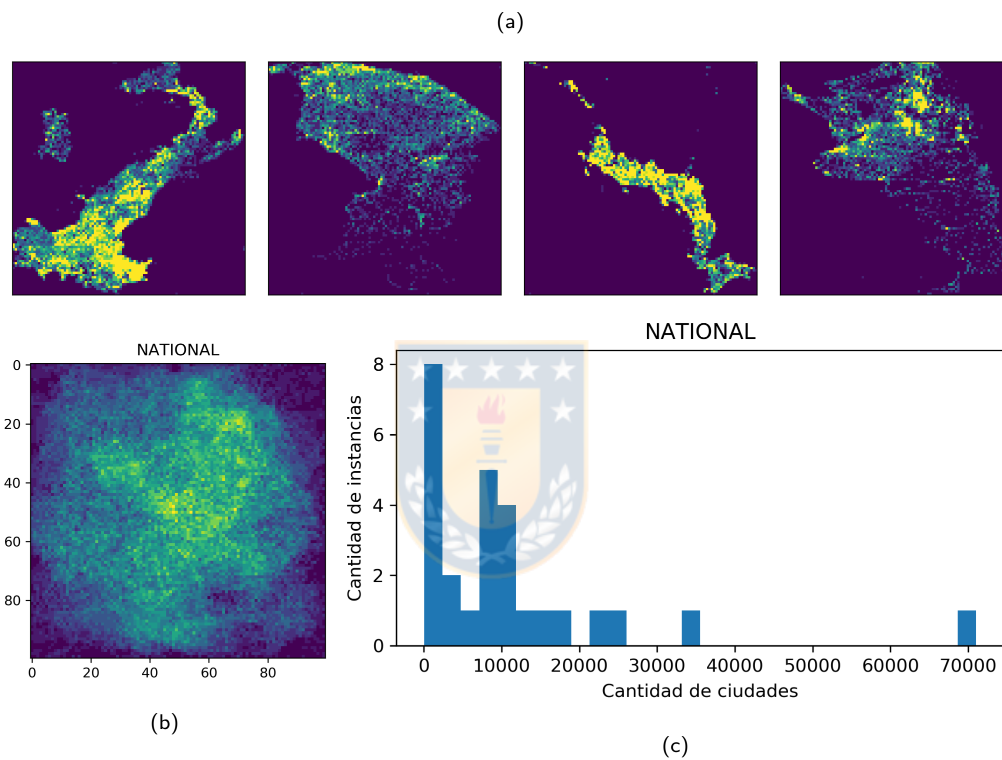


Fig. 25: En **(a)** se presentan 4 instancias de ejemplo de NATIONAL coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de NATIONAL. En **(c)** se muestra un histograma de la cantidad de ciudades de NATIONAL.

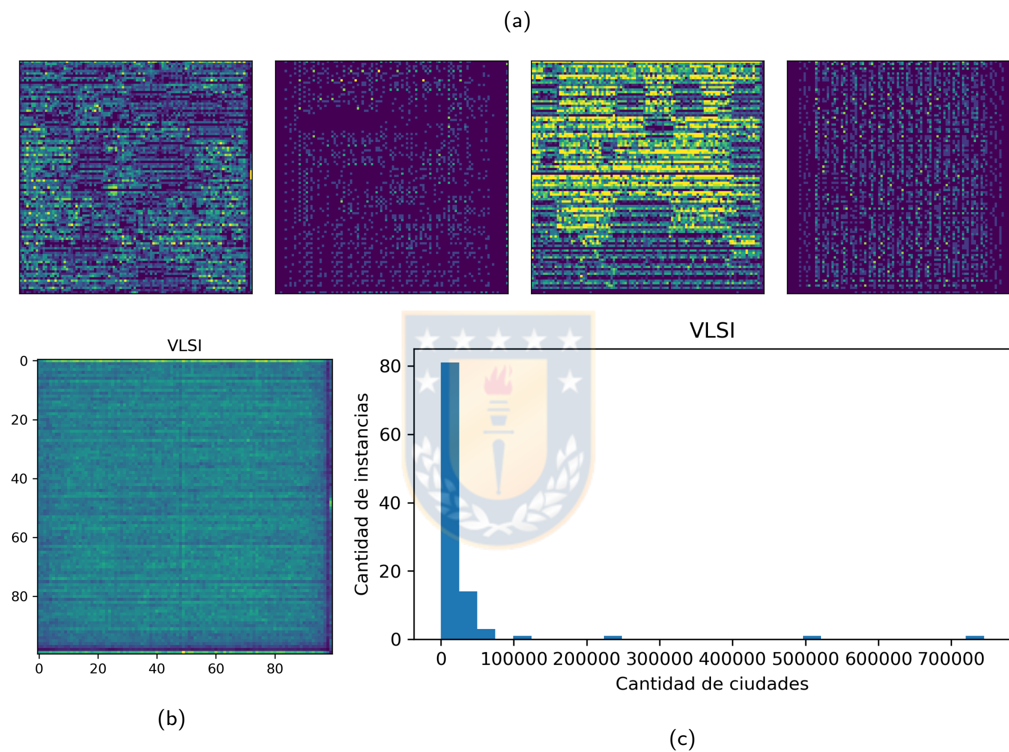


Fig. 26: En **(a)** se presentan 4 instancias de ejemplo de VLSI coloreadas según su densidad (más amarillo, más denso). En **(b)** se muestra la suma de la densidad de todas las instancias de VLSI. En **(c)** se muestra un histograma de la cantidad de ciudades de VLSI.