Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Estructura de datos para acelerar la búsqueda de subconjuntos en el labeling para TSP Time-Dependent

Tesis de Licenciatura en Ciencias de la Computación

Gonzalo Paradelo

Director: Francisco Soulignac

Buenos Aires, 2023

ESTRUCTURA DE DATOS PARA ACELERAR LA BÚSQUEDA DE SUBCONJUNTOS EN EL LABELING PARA TSP TIME-DEPENDENT

El problema del viajante de comercio con ventanas de tiempo y dependencia temporal (TDTSPTW) es una versión del TSP donde los tiempos de viaje varían a lo largo del horizonte de planificación para simular los efectos de la congestión. Este problema tiene aplicaciones directas a la planificación de entregas directo al consumidor en grandes ciudades, donde la congestión de tránsito es un aspecto predominante. En esta tesis, analizamos una solución exacta con programación dinámica del estado del arte para este problema y proponemos una mejora en su tiempo de ejecución basada en una nueva estructura de datos. La ventaja de esta estructura de datos es que permite buscar eficientemente elementos indexados por conjuntos, cuando se busca aquellos índices contenidos por un conjunto de búsqueda. Se realizaron experimentos computacionales para determinar su rendimiento y se logró demostrar la efectividad de la mejora.

Palabras claves: TSP, dependencia temporal, estructura de datos, labeling, subconjuntos.

ESTRUCTURA DE DATOS PARA ACELERAR LA BÚSQUEDA DE SUBCONJUNTOS EN EL LABELING PARA TSP TIME-DEPENDENT

The time-dependent traveling salesman problem with time windows (TDTSPTW) is a version of the TSP that simulates the effects of traffic congestion with non-constant travel times. This problem has direct application in scheduling direct deliveries to consumers in large cities where traffic congestion is a prevalent problem. In this paper, we analyze an exact state-of-the-art solution using Dynamic Programming for the TDTSPTW and propose an improvement in execution time based on a custom data structure. The advantage of this data structure is that it enables efficient search for elements indexed by a set when we want to retrieve the indices that are a subset of a given search set. Computational experiments show the performance improvements achieved by the data structure in the TDTSPTW.

Keywords: TSP, time-dependent travel times, data structure, labeling, subset.

CONTENTS

1. INTRODUCCIÓN

En los problemas de distribución de mercadería se busca determinar la mejor estrategia para repartir bienes materiales a un conjunto de destinos. Esta mejor estrategia minimiza alguna medida de costo como el tiempo total de viaje o el tiempo de espera de los clientes. El trabajo de esta tesis tiene como objetivo acelerar la resolución de problemas de optimización de distribución y tiene su impacto más directamente en la logística aplicada al transporte de bienes materiales, que es de gran interés por su impacto en industrias de todo tipo.

Por lo general, la parte más compleja y costosa de una cadena de suministros, considerando desde que se comienza con materia prima hasta que el producto final llega a las manos del cliente final, es el denominado transporte de "última milla". Es decir, el último tramo hasta que un producto llega a su destino final. Esta parte presenta mayor complejidad que el resto del viaje debido al incremento exponencial del número de destinos y, por lo tanto, rutas posibles para recorrerlos. Además suele involucrar el recorrido de áreas urbanas, lidiando con posibles problemas de congestión de tránsito. Por estos motivos, en muchos casos, el costo de transportar un producto por su "última milla" puede superar la mitad del costo total del envío (Joerss et al., 2016). Este ambiente es cada vez más relevante debido al aumento de la cantidad de envíos directo al consumidor gracias al crecimiento del comercio electrónico. En esos casos, en vez de abastecer a un negocio que cubre las necesidades de miles de consumidores, se debe hacer una entrega particular para cada uno. En estas circunstancias, uno de los problemas fundamentales para minimizar el costo total de los viajes de los repartidores es elegir correctamente el orden en el que se visitan los destinos, pero, por la gran cantidad de maneras posibles de hacerlo, es muy costoso calcular este orden ideal.

Esta familia de problemas, donde se busca encontrar el camino óptimo para una flota de vehículos que deben satisfacer las demanda de un conjunto dado de consumidores, es conocida como *problema de enrutamiento de vehículos* (VRP por sus siglas en inglés). En ella, cada camino que conecta un par de destinos tiene un costo determinado. El objetivo es encontrar la ruta para cada vehículo que minimiza la suma de los costos del recorrido de cada uno.

El problema dentro de VRP donde se cuenta con un único vehículo es conocido como *problema de viajante de comercio* (TSP por sus siglas en inglés). Además de las aplicaciones obvias en planificación y logística, esta versión también tiene aplicaciones en otros ambientes como la fabricación de circuitos integrados (Grötschel et al., 1991). Dicho esto, en esta tesis vamos a considerar principalmente las aplicaciones en la distribución de mercadería.

Existen varias maneras de definir el costo de viaje de un camino, dependiendo de qué se busque minimizar. Por ejemplo, se podría usar la distancia recorrida o el tiempo de viaje. En esta tesis vamos utilizar el tiempo de viaje como el costo de un camino ya que es una métrica que no solo esta correlacionada con la distancia recorrida, que cuando aumenta aumenta el gasto en combustibles y desgaste del camión, sino que también incluye otros costos como el sueldo del conductor del camión. Además, minimizar el tiempo del recorrido permite hacer más envíos por hora y maximizar la cantidad de paquetes entregados.

En una versión simple del problema los costos de recorrer un camino son valores con-

stantes, es decir que no van a cambiar al cambiar el momento de salida. Este modelo tiene la ventaja de ser más simple de conceptualizar y resolver. Sin embargo, no refleja bien las situaciones del mundo real donde los tiempos de viaje entre un mismo par de puntos varía ampliamente a lo largo del dia, ya que están fuertemente influenciados por las condiciones del tránsito. Esto es especialmente cierto en ciudades, que son el entorno donde la mayoría de los problemas de entrega de última milla tienen lugar. Si bien no es posible conocer, anticipadamente y con exactitud, el tiempo real de viaje entre dos destinos, para cada instante posible de salida, sí es posible predecirlos ya que los cambios en las condiciones del transito suelen seguir los mismos patrones cada dia.

Por este motivo, en esta tesis trabajaremos sobre el TSP con dependencia temporal, donde los tiempos de viaje entre dos destinos varían dependiendo del momento de salida. Es decir, dejan de ser un número constante y pasan a estar dados por una función que depende del momento en el que se comienza el recorrido. De esta manera, podemos modelar los efectos de la congestión variante a lo largo del día, lo que resulta en un modelo donde los viajes planificados tienen una duración más cercana a cuando se realizan en la realidad. Además, permite planificar viajes que pueden aprovechar los momentos de menor duración de cada tramo y por lo tanto, pueden obtener un tiempo total de viaje menor.

Utilizaremos una segunda modificación al problema que consiste en agregarle una ventana de tiempo a cada destino, que establece un lapso de tiempo durante el cual dicho destino debe ser visitado. Esto restringe las posibles rutas, ya que la solución debe visitar cada uno durante su correspondiente ventana de tiempo. Por ejemplo, continuando con la aplicación a un camión que debe entregarle sus pedidos a un conjunto de clientes, cada cliente podría especificar un rango horario en el que va a estar en su domicilio para poder recibirlo.

Luego, la versión del problema con la que vamos a trabajar es denominada problema del viajante de comercio con ventanas de tiempo y con dependencia temporal (TDTSPTW por sus siglas en ingles).

1.1 Definición del problema

Dado un digrafo $G = (V, A)$ con vértices $V = \{0, ..., n + 1\}$, al resolver el TDTSPTW, encontramos un camino de 0 a $n + 1$ que pasa por cada vértice exactamente una vez. Los vértices 0 y $n + 1$ representan el depósito inicial y final respectivamente, mientras que los n vértices restantes representan a los clientes.

El camino es atravesado por un único vehículo y puede ser recorrido únicamente dentro de un horizonte temporal $[0, T]$. Hay un intervalo $[a_v, b_v] \subseteq [0, T]$ asociado a cada vértice $v \in V$ que determina la *ventana de tiempo* durante la cual debe comenzar el servicio de v. El vehículo puede llegar a un vértice v antes de su apertura a_v, pero debe esperar hasta a_v para realizar el servicio en ese caso. Luego de completar el servicio, el vehículo se dirige al próximo vértice en el camino. Para cada arista $(v, w) \in A$, el tiempo de viaje para ir de v a w está dado por una *función de tiempos de viaje* τ_{vw}, donde, para cada $t \in [0, T]$, $\tau_{vw}(t)$ es el tiempo que demora el vehículo en viajar de v a w, partiendo de v en el instante t. En este modelo, propuesto originalmente por Ichoua et al. (2003), cada τ_{vw} es continua, lineal a trozos y satisface la propiedad *first-in, first-out* (FIFO). La propiedad FIFO garantiza que siempre que un vehículo parta de un nodo v hacia otro w en un determinado momento, va a llegar antes de que si partiese en un instante posterior.

Si bien esta definición establece un tiempo de servicio cuando se visita cada vértice v,

vamos a asumir que este tiempo está implícitamente incluido en τ_{vw} y debe ser completado antes de b_w. Además, asumimos que se cumple la *propiedad sin esperas*, que establece que $t + \tau_{vw}(t) \geq a_w$ para todo $t \in [0, T]$. Para garantizar estas suposiciones en cualquier instancia posible, se aplican técnicas de preprocesamiento estudiadas por Lera-Romero y Miranda-Bront (2021).

Además de τ_{vw}, que es la función del tiempo que toma recorrer una arista (v, w), definimos, para cualquier camino $p = (v_0, ..., v_k)$ y cualquier $0 \leq i \leq k$, la función $\delta_p^i(t)$ que establece el primer instante en el que el vehículo puede completar el servicio de v_i, partiendo de v_0 en el instante $t \in [0, T]$ y siguiendo el recorrido establecido por p. Trivialmente, $\delta_p^0(t) = t$. Por inducción, la propiedad FIFO y la propiedad sin esperas, deducimos que $\delta_p^{i+1}(t) = \delta_p^i(t) + \tau_{v_i v_{i+1}}(\delta_p^i(t))$. Para simplificar la sintaxis, definimos $\delta_p = \delta_p^k$ como la *función de arribo* de p.

Como todas las funciones de tiempo de viaje y $\delta_p^0(t)$ son continuas y lineales a trozos, podemos deducir por inducción que δ_p también lo es. Además, por la propiedad FIFO, δ_p es creciente.

De todos los instantes $t \in [0, T]$ en el dominio de δ_p solo nos interesan aquellos en los que se respetan las ventanas de tiempo de todos los vértices al recorrer p partiendo en t. Por la propiedad sin esperas, $\delta_p^i(t) \geq a_{v_i}$ para todo $0 < i \leq k$. Por lo tanto, definimos $t \in [0, T]$ como un *tiempo de partida factible* de p si $t \geq a_{v_0}$ y $\delta_p^i(t) \leq b_{v_i}$ para todo $0 < i \leq k$. Llamamos $\text{TPF}(p)$ al conjunto de instantes de partida factibles de p.

Con esta definición, surge una definición de la duración de un camino $p = (v_0, ..., v_k)$. Si $\delta_p(t)$ es el primer instante en el que v_k puede ser visitado cuando se recorre p partiendo en el instante $t \in \text{TPF}(p)$, luego, $\Gamma_p(t) = \delta_p(t) - t$ es el tiempo mínimo requerido para atravesar p al partir en el instante t. Como tanto δ_p y $t \to t$ son continuas y lineales a trozos, se deduce que Γ también lo es.

Similarmente, podemos establecer definiciones simétricas a las anteriores, pero desde el punto de vista de v_k. En este caso, nos interesa el conjunto de instantes en los que el servicio de p puede ser completado. Es decir, aquellos instantes en los que se puede llegar a v_k, habiendo respetado todas las ventanas de tiempo. Nos referimos a este conjunto como *tiempo de arribo factible* de p y usamos $\text{TAF}(p)$ para referirnos a él. Notemos que el primer momento que un vehículo puede llegar a v_k cuando atraviesa p es $\delta_p(a_{v_0})$. Y por lo tanto, $\text{TAF}(p) = [\delta_p(a_{v_0}), b_{v_k}]$.

Además, si $\delta_p(t)$ es el primer momento que un vehículo puede completar el recorrido de p si parte en el momento t, luego $\lambda_p(t) = \max\{t' \in \text{TPF}(p) | \delta_p(t') = t\}$ es el último momento en el que el vehículo puede partir de v_0 para llegar a v_k en el tiempo t. Con λ_p surge otra definición de duración de p equivalente a $\Gamma_p(t)$, la función $\Delta_p(t) = t - \lambda_p(t)$ con dominio $TAF(p)$ es la *duración de p en v_k*, es decir, $\Delta_p(t)$ es el tiempo mínimo requerido para atravesar p llegando a v_k en el instante t.

Cualquier camino p que parte del depósito 0, termina en el depósito $n + 1$ y visita exactamente $n + 2$ vértices es conocido como un *tour* y un camino es *elemental* si no visita ningún vértice más de una vez. La *duración* c_p es el tiempo mínimo requerido en viajar de v_0 a v_k respetando todas las ventanas de tiempo.

El objetivo del TDTSPTW es encontrar un tour elemental p de duración mínima c_p.

1.2 Revisión de literatura

La mayor parte de los trabajos de investigación existentes sobre esta familia de problemas se enfocan en versiones donde los tiempos de viaje son constantes, como el TSP y el TSPTW. Los algoritmos exactos convencionales utilizan técnicas de optimización poliédrica y algoritmos de *branch-and-cut* (por ejemplo, Ascheuer et al. (2000, 2001)). Los algoritmos de programación dinámica también fueron estudiados. Christofides et al. (1981) propusieron técnicas para calcular rápidamente cotas inferiores para el TSPTW y Mingozzi et al. (1997) propusieron un algoritmo de programación dinámica distinto, construido sobre las ideas de Christofides et al. (1981) para el TSPTW con restricción de precedencia (TSPTW-PC). Este algoritmo utiliza la técnica de *bounding* que consiste en usar cotas inferiores para reducir el número de estados explorados.

Algunos de los primeros en estudiar la dependencia temporal fueron Malandraki y Daskin (1992), quienes proponen un modelo donde los tiempos de viaje entre cada par de ciudades se definen como funciones constantes a trozos, sobre el horizonte de momentos de salida. Esto permite reflejar el efecto de la congestion en distintos momentos del día. Una desventaja de este tipo de modelos basados en funciones contantes a trozos es que no garantizan la propiedad "First-In, First-Out" (FIFO), que dice que no es posible llegar más temprano al destino habiendo saliendo más tarde. Al no cumplirse esta condición, podemos obtener soluciones donde se le indica a un vehículo que debe esperar para partir hacia un destino porque, según el algoritmo, llegaría antes. Para evitar este problema, Ichoua et al. (2003), basándose en el modelo de Hill y Benton (1992), proponen un modelo donde los tiempos de viaje están basados en la velocidad promedio en cada par de destinos. Este modelo logra garantizar la propiedad FIFO y fue utilizado por varios autores para resolver el problema.

Una contribución importante, y en la que se va a enfocar el aporte de esta tesis, es la de Baldacci et al. (2011), que proponen la relajación *ng* para acelerar la resolución del VRPTW y CVRP. Estos mismos autores expanden su trabajo anterior (Baldacci et al., 2012) con una nueva relajación, aplicada esta vez al TSPTW. Tilk y Irnich (2017) adaptaron la idea para el MTSP y la mejoraron con aplicando una técnica que llamaron *aumento dinámico de vecindarios*.

Lera-Romero et al. (2022) implementan el primer algoritmo eficiente para el TDTSPTW con programación dinámica. Para ello, utilizan una técnica llamada *dominación parcial*, para tomar ventaja de la dependencia temporal.

Finalmente, Fontaine et al. (2023) proponen una solución para el TDTSPTW basándose en la extensión exacta de A*.

1.3 Contribuciones

Para esta tesis se tomó como punto de partida el artículo *Dynamic Programming for the Time-Dependent Traveling Salesman Problem with Time Windows* (Lera-Romero et al., 2022), donde se desarrolla un algoritmo exacto que define el estado del arte para el TDTSPTW. En base a este algoritmo, proponemos una mejora de performance. Elegimos este paper y el TDTSPTW en particular ya que su código está disponible. Sobre este código, implementamos una nueva estructura de datos para acelerar su tiempo de ejecución.

Recalcamos que el objetivo de acelerar los algoritmos para este tipo de problemas no es principalmente para ahorrar tiempo durante su ejecución, sino que un algoritmo más

rápido puede llegar a resolver instancias que anteriormente no podían ser resueltas en un tiempo práctico debido a su complejidad. Es decir que, para este problema, el tiempo de encontrar su solución aumenta muy rápidamente cuando las instancias incrementan en complejidad por lo que, en algunos casos, los mejores algoritmos actuales no pueden resolver instancias moderadamente complejas.

Si bien seleccionamos la variante del problema de viajante de comercio con ventanas de tiempo y dependencia temporal para el trabajo de esta tesis porque teníamos el código de un algoritmo del estado del arte como punto de partida, consideramos que esta versión es muy relevante para la mayoría de los casos reales que se buscan resolver. La inclusión de dependencia temporal es de especial importancia porque el efecto de la congestion es el que mayor impacto tiene en la duración de los viajes en la mayoría de las ciudades del mundo, y por lo tanto, debe ser considerado en prácticamente cualquier situación de planificación logística.

Dicho esto, la estructura de datos implementada es agnóstica al TDTSPTW (o a cualquier otro problema). Más precisamente, esta enfocada en acelerar la búsqueda en una colección de conjuntos de elementos, cuando estamos interesados en encontrar aquellos conjuntos que están contenidos por un conjunto de búsqueda. Es decir, cuando tenemos una colección de conjuntos (o un diccionario con conjuntos como claves) y queremos encontrar aquellos elementos que son subconjunto del conjunto de búsqueda. Sin esta estructura, la forma usual (y usada por Lera-Romero et al. (2022)) de realizar esta búsqueda es comparar todos los conjuntos con el de búsqueda, uno por uno, y ver si están contenidos. La estructura permitirá almacenar y ordenar los conjuntos de tal manera que, a la hora de hacer una búsqueda, podremos saltear grandes ramas de conjuntos, sin tener que realizar una comparación. Vamos a trabajar su implementación y aplicación dentro del TDTSPTW, pero puede ser utilizada en otros problemas donde una contribución importante al tiempo de ejecución venga de realizar búsquedas de este estilo.

Finalmente, se realizaron experimentos para determinar su impacto en el tiempo de ejecución de la parte del algoritmo modificada y también en el algoritmo total.

1.4 Estructura de la tesis

La Sección 2 describe en profundidad el algoritmo actual que define el estado del arte para el TDTSPTW y su implementación. Comienza por la versión básica que resuelve el TSP, para luego adaptarlo para el TDTSPTW. La Sección 2.3 trata uno de los componentes del algoritmo, la relajación NG, ya que es de especial importancia para el trabajo de esta tesis.

En la Sección 3 se describe la contribución realizada. Se detalla la nueva estructura de datos, con sus propiedades, invariantes y operaciones, y se modifica el algoritmo de la Sección 2 para aprovecharla.

En la Sección 4 se muestran los experimentos realizados para comprobar la efectividad de esta nueva estructura.

Finalmente, la Sección 5 contiene las conclusiones.

2. ALGORITMO

En este capítulo repasaremos el algoritmo base que vamos a trabajar, que fue propuesto por Lera-Romero et al. (2022) en el artículo mencionado anteriormente. Comenzaremos por su versión en el TSP elemental, para ganar una intuición del problema y luego lo adaptaremos para el TDTSPTW.

El problema del viajante de comercio pertenece a la clase de complejidad NP-hard, lo que significa que no se conocen algoritmos eficientes para resolverlo (y posiblemente no existan). Luego, los mejores algoritmos cuentan con numerosas mejoras para lograr mejores tiempos de ejecución posibles a pesar de tener complejidad exponencial.

En su esencia, el algoritmo con el que vamos a trabajar encuentra la solución óptima utilizando el método de *programación dinámica*. Se construyen iterativamente soluciones parciales óptimas hasta encontrar la solución del problema entero. Sobre esta base, existen podas, preprocesamientos y relajaciones, entre otras técnicas para mejorar la eficiencia. Si bien no mejoran la complejidad teórica del peor caso, tienen un gran impacto en los tiempos de ejecución en la práctica.

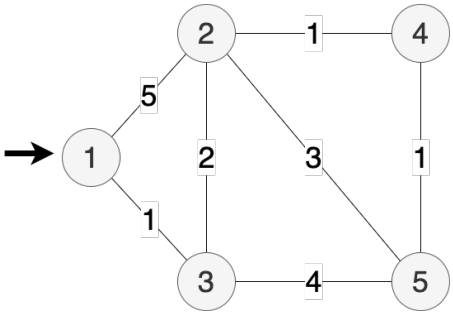
2.1 Algoritmo de Labeling para TSP

Consideremos primero TSP sin ventanas de tiempo ni dependencia temporal. Recordemos que el objetivo del problema es encontrar, en un grafo con costos asociados a cada arista, un tour elemental de costo mínimo. Como mencionamos, el problema se resuelve comenzando con una solución parcial inicial y extendiéndola. Como lo que estamos buscando es un camino que recorre todo el grafo, las soluciones parciales son caminos que visitan un subconjunto de los vertices del grafo y se comienza con el camino que visita solo el depósito inicial.

Para representar un camino, utilizamos una estructura de datos que llamamos *label*, que describe el conjunto de vértices visitados y el tiempo total de recorrerlos. Dado un camino elemental $p = (v_0, ..., v_k)$ desde $v_0 = 0$, un label de p es una tupla $\ell = (\text{prev}, v_k, S, c)$, donde prev es un puntero a un label de $(v_0, ..., v_{k-1})$, $S = \{v_0, ..., v_k\}$ es el conjunto de vértices visitados y c es el costo de recorrer p. Recordemos que en el TSP, los costos de las aristas son valores constantes y el costo de un camino es la suma de los costos de cada una de sus aristas.

Se comienza con un label inicial $\ell_0 = (\perp, 0, \{0\}, 0)$ que representa el camino que solamente visita al deposito. Luego, en cada iteración, se extienden todos los labels enumerados, para considerar los caminos que visitan un vértice adicional. Es decir, cada label se extiende a traves de todas de las aristas que van desde su último vértice visitado hasta algún otro vértice que no haya sido visitado por el camino actual, generando un nuevo label por cada arista por la cual fue extendido. De esta manera, se terminan considerando todos los caminos elementales posibles en el grafo.

Si bien este algoritmo es suficiente para encontrar la solución al problema, es altamente ineficiente ya que extiende labels redundantes que se podrían descartar sin tener que enumerarlos hasta el final. Por ejemplo, consideremos el siguiente grafo:



Al extender las soluciones parciales, comenzando por el label que visita solamente al vértice 1, se construirán los siguientes dos labels:

$$L = (prev_L, 5, \{1, 2, 3, 5\}, 11) \text{ y } M = (prev_M, 5, \{1, 3, 2, 5\}, 6)$$

Como podemos ver, la única diferencia entre ambos es sus costo, ya que terminan en el mismo vértice, y visitaron el mismo conjunto. Por lo tanto, cualquier extensión posible de M también es una extensión posible de L (y viceversa). Como L tiene mayor costo que M, cualquier extensión a partir de L también tendrá mayor costo que esa misma extensión realizada a partir de M. Por este motivo, no es necesario seguir extendiendo ambos labels ya que extender L siempre va a resultar en un camino de mayor costo. Esto implica que esa extensión no va a ser la solución óptima del problema del problema (aplicando la misma extensión a M tenemos una solución de menor costo). Luego, podemos descartar a L sin tener que extenderlo hasta el final. Decimos que el label L está *dominado* por M.

Definición 1. *Decimos que un label L, que representa el camino p_L, está* dominado *por otro label M, que representa el camino p_M, cuando todos los tours elementales $p_L + p$ que podemos extender desde L, $p_M + p$ también es elemental y tiene un costo no mayor al de $p_L + p$.*

Observamos que para que un label pueda dominar a otro, ambos deben haber visitado los mismos vértices, ya que la extensión debe formar un tour elemental a partir de ambos labels.

Esta definición de dominación esta basada en que se calculen todas las extensiones posibles desde un label. Pero esto es justamente lo que queremos evitar, ya que calcularlo de esta manera no resultaría en una mejora en el tiempo de ejecución del algoritmo. Por eso, vamos a utilizar una regla que implica dominación y es más rápida de calcular.

Regla 1 (Dominación)**.** *Si un label L tiene mayor costo, visita los mismos vértices y termina en el mismo vértice que otro label M, entonces L está dominado por M.*

Como mencionamos, cualquier label dominado, puede ser descartado inmediatamente sin considerar sus posibles extensiones. En este pequeño ejemplo no hay una gran diferencia en cuanto a la cantidad de operaciones realizadas cuando se aplica la regla de dominación. Pero, en casos reales, considerando que cada label descartado elimina toda una rama de extensiones futuras, la dominación es fundamental para que el algoritmo sea eficiente.

Además de la dominación, utilizaremos un segundo concepto para descartar labels antes de terminar de extenderlos.

Definición 2. *Un label L, que representa el camino p_L, está* acotado *cuando el costo de todos los tours elementales $p_L + p$ que se pueden extender a partir de p_L es mayor a alguna cota superior ub dada.*

Llamamos el *costo del tour de L* como el mínimo costo de un tour que comienza con p_L. Luego, L está acotado cuando la duración de su tour es mayor a ub. Cualquier label que está acotado puede ser descartado porque no va a resultar en una solución óptima.

Al igual que con la dominación, el mecanismo para calcular si un label está acotado por definición requiere que lo extendamos de todas las maneras posibles hasta completar todos los tours posibles, para encontrar el costo de su tour y poder compararlo con ub. Nuevamente, esto es lo que queremos evitar para acelerar la ejecución del algoritmo. Por este motivo, calcularemos una cota inferior de este costo, que llamaremos *cota de terminación*. Luego, si la cota de terminación es mayor a ub, el costo del tour de L también lo será. Utilizaremos una función ct que retorna, para cada label, su cota de terminación.

Regla 2 (Bounding)**.** *Sea* ct *una cota de terminación y ub una cota superior a la duración del tour óptimo. Si* $\text{ct}(L) > ub$ *para un label L, entonces L puede ser descartado.*

Es importante que la cota de terminación sea cercana al costo real para maximizar la cantidad de labels descartados. Cuanto menos ajustada es esta cota de terminación, hay más chances de que sea menor que ub por más que el costo real de completar el tour sea mayor que ub, y no se pueda descartar el label. Hablaremos sobre como se obtiene la cota de terminación en la Sección 2.3.

Para que una cota de terminación sea a *aceptable* es necesario que $\text{ct}(L)$ sea mayor o igual al costo de L para cada label L.

Regla 3 (Terminación)**.** *Sea L un label representando un tour elemental, y* ct *una cota de terminación aceptable. Si* $\text{ct}(L) \leq \text{ct}(M)$ *para cada label M que representa un tour elemental, luego L es óptimo y tiene costo* $\text{ct}(L)$.

Obtener la cota superior ub relevante también podría presentar un problema. Como dijimos, para descartar la mayor cantidad de labels posibles, debe ser lo más baja posible pero no menor que la solución del problema, que no se conoce a priori. Sin embargo, el algoritmo procesa y extiende los labels en orden de cota de terminación de menor a mayor y, por la Regla 3, nunca vamos a extender un label con costo mayor al óptimo. Por este motivo, descartar labels por la regla de bounding no tiene impacto en el tiempo de ejecución y su único propósito es el de reducir el uso de memoria. Luego, la cota de terminación es utilizada principalmente para ayudar al algoritmo a elegir los labels más relevantes para extender.

2.2 Algoritmo de Labeling para TDTSPTW

A la hora de extender el algoritmo para el TDTSPTW, existen distintas maneras de extender la estructura de label al contexto de dependencia temporal. Lera-Romero et al. (2022) realizaron un análisis de las distintas opciones. Haremos un repaso de este análisis porque consideramos que es importante para entender el algoritmo final.

Recordemos que en TSP, dado un camino elemental $p = (v_0, ..., v_k)$, un label de p es una tupla $\ell = (\text{prev}, v_k, S, c)$, donde prev es un puntero a un label de $(v_0, ..., v_{k-1})$,

$S = \{v_0, ..., v_k\}$ es el conjunto de vértices visitados y c es el costo de recorrer p. Para TDTSPTW, definimos, para un camino p_L, un *label total* L, que es una tupla como ℓ, con la diferencia que, en vez de un costo constante c, L almacena la función de duración Δ_{p_L} en dominio de los tiempos de llegada factibles $\text{TAF}(p_L)$.

Si bien, computacionalmente, L es una estructura, es mejor pensarlo como una función cuyo dominio es $\text{TAF}(p_L)$, tal que $L(t)$ es el label sin dependencia temporal que representa a p cuando v_k es visitado en el instante t.

Esta nueva definición de label implica que también debamos adaptar la definición de dominación. Un label total L está dominado por otro M en un instante $t \in \text{TAF}(p_L)$ si $t \in \text{TAF}(p_M)$ y $L(t)$ está dominado por $M(t)$ según la Definición 1. Además, L está *totalmente dominado* por M si L está dominado por M para cada $t \in \text{TAF}(p_L)$.

Esta definición de dominación requiere que para poder descartar un label total L, exista algún otro M que lo domine por su cuenta en todo su dominio. Por lo tanto, no permite que un algoritmo de labeling para TDTSPTW pueda descartar un label total L en caso de que esté dominado por un conjunto de otros labels totales que entre ellos dominen a L en cada instante de su dominio. Una mejor alternativa es aplicar *dominación parcial*, en la cual L está dominado cuando, para cada $t \in \text{TAF}(p)$, existe un label total M tal que $M(t)$ domina a $L(t)$. Observemos que, para cada $t \in \text{TAF}(p)$, el label total M puede ser distinto al label total que domina a L en un punto $t' \neq t$. La dominación parcial es superior a la total ya que se descarta un mayor número de labels. Sin embargo, el uso de labels totales implica que, mientras L no esté dominado en algún instante t', $\delta_p(t)$ debe ser mantenido por más que $L(t)$ este dominado. En consecuencia, muchas porciones dominadas de δ_{p+w} ($w \in V$) deben ser computadas cuando L se extiende al label representando el camino $p + w$, lo que consume tiempo y memoria innecesarios. Una mejor alternativa es mantener *labels parciales*.

Un *label parcial* es una estructura L que codifica a un camino p_L y mantiene la función de duración δ_{p_L} para un subconjunto no vacío de $\text{TAF}(p_L)$.

Al aplicar dominación entre labels parciales, un tiempo de llegada factible $t \in \text{dom}(L)$ puede ser descartado de L cuando $L(t)$ está dominado, independientemente de si $L(t')$ esta dominado para $t' \in \text{dom}(L) \setminus t$. Luego, el dominio $\text{dom}(L)$ de L es una familia de intervalos disjuntos y L puede ser considerada una función parcial sobre el dominio $\text{TAF}(p)$. Podemos ver un ejemplo de dominación con labels parciales y totales en la Figura 2.1.

En su implementación del algoritmo para resolver el TDTSPTW, Lera-Romero et al. (2022), utilizaron una estructura de que contiene todos los intervalos no dominados de δ_{p_L}. Cada intervalo es representado por un *label lineal* ℓ, que es una estructura que codifica un camino p_L y almacena δ_{p_L} únicamente para una porción lineal de $\text{TAF}(p_L)$

Luego, cada label parcial L está implementado como una secuencia de labels lineales $\ell_1, ...\ell_k$ cuyas duraciones tienen dominios disjuntos. En rigor, los labels parciales son solo conceptuales y no están almacenados explícitamente en el algoritmo. Además, no existe una manera rápida de determinar si dos labels lineales corresponden al mismo camino, por más que pertenezcan a la misma secuencia $\ell_1, ..., \ell_k$. Esta representación, permite mezclar labels lineales de distintos caminos en un mismo bucket para procesarlos de manera más eficiente.

En este caso también utilizaremos una regla eficiente para determinar dominación entre labels lineales.

Regla 4 (Dominación Parcial)**.** *Un label lineal ℓ está dominado por otro m en el tiempo $t \in \text{TAF}(\ell)$ si $t \in \text{TAF}(m), v(m) = v(\ell), S(m) = S(\ell)$, y $\Delta_m(t) \leq \Delta_\ell(t)$.*

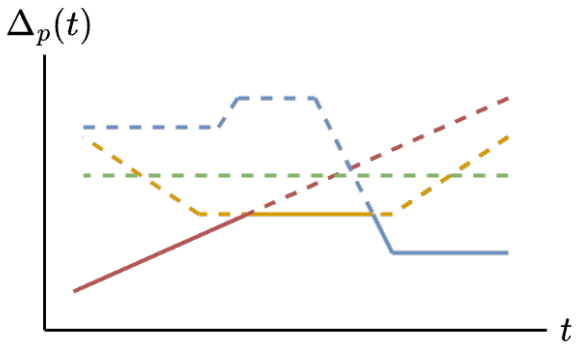


Fig. 2.1: Cada color representa la función de costo de un label. Observemos que ningún label domina a otro en la totalidad de su dominio. Por lo tanto, si se utilizan labels totales y dominación total, ningún label puede ser descartado. Con labels totales y dominación parcial, se puede descartar al label verde. Por último, al utilizar labels parciales, todas las secciones punteadas pueden ser descartadas.

Todo label lineal ℓ mantenido por el algoritmo no está dominado para ningún $t \in \text{dom}(\ell)$. Luego, si un label extendido ℓ es dominado en el momento t, entonces t es eliminado de $\text{dom}(\ell)$ y ℓ es descartado cuando su $\text{dom}(\ell)$ es vacío.

De acá en adelante, cuando decimos label, nos referimos a un label lineal.

2.2.1 Implementación

En esta sección contaremos como está implementado el algoritmo de labeling en el código provisto por Lera-Romero et al. (2022).

Este algoritmo (Algoritmo 1) recibe la instancia a resolver, compuesta por un digrafo $D = (V, A)$, un vértice inicial o, un vértice final d, una ventana de tiempo $[a(v), b(v)]$ para cada $v \in V$, un horizonte temporal T y la función $\tau_{vw}(t)$ que devuelve el tiempo de viaje entre v y w, partiendo en el instante $t \in [0, T]$, para cada $(v, w) \in A$. Además, recibe una función ct que retorna una cota de terminación aceptable de un label y una cota superior de la duración del tour óptimo ub.

El algoritmo mantiene un diccionario de *buckets* $\mathcal{L}$ donde se almacenan todos los labels. Cada label se inserta en el *bucket* con índice $(|S(\ell)|, v(\ell), S(\ell))$. Observemos que, por la Regla 4, ℓ solamente puede ser dominado por otro label que corresponda al mismo bucket. Para cada índice $\mathcal{B}$, $\mathcal{L}(\mathcal{B})$ mantiene una secuencia $\ell_0, ..., \ell_j$ de labels donde $b(\ell_i) \leq a(\ell_{i+1})$ para $0 \leq i < j$. Luego, para cada instante $t \in [a(\ell_0), b(\ell_j)]$, el bucket tiene a lo sumo un label ℓ_i no dominado y con t en su dominio. Observemos que $\mathcal{L}(\mathcal{B})$ contiene a todos los labels con índice $\mathcal{B}$, independientemente de que camino representan.

Además, este algoritmo mantiene una cola de prioridad q que usaremos para ordenar los labels a *procesar*. Para ordenar los labels de q, se les agrega un campo adicional con su cota de terminación, como describimos anteriormente. En cada iteración del ciclo principal (comenzando en la línea 4), el algoritmo *procesa* la secuencia L de labels sin procesar con minima cota de terminación y índice de bucket (k, v, S). Como extraemos los labels por mínima cota de terminación, si $k = |V|$, entonces cada label de L es un tour de duración óptima y el algoritmo termina por la Regla 3 y porque ct es una cota aceptable (línea 10).

En el resto de los casos, se extienden los labels de L por todas las aristas (v, w) posibles, resultando en una nueva secuencia M (línea 12). Luego, se restringen los dominios de los labels de M según la ventana de tiempo $b(w)$ (línea 14) y se realiza un merge de M a N, donde N es la secuencia de labels en $\mathcal{L}$ en el mismo bucket correspondiente a M (línea 15). El merge consiste en que, si N esta vacío (en cuyo caso M es la primera secuencia de ese bucket en ser extendida), simplemente se inserta M en $\mathcal{L}$. Si no, se combina M y N y se eliminan los tiempos dominados del resultado (línea 17) (Figura 2.2). Luego, se actualizan los nuevos labels $m \in M \cap N$ con su cota de terminación $\text{ct}(m)$ y se descartan los que superan la cota superior ub (línea 19). Finalmente, se encola cada m en q (línea 20).

Algorithm 1: Algoritmo de Labeling para TDTSPTW

Input:
$D = (V, A)$, grafo con conjunto de vertices V y de aristas A,
$o = 0$, origen,
$d = n + 1$, deposito final,
a, comienzo de ventanas de tiempo,
b, fin de ventanas de tiempo,
τ, función del costo de atravesar una arista en un instante del tiempo,
T, horizonte temporal,
ct, función de cota de terminación para cada label,
ub, cota superior

Result: Un tour elemental de costo mínimo con duración menor a ub

1 Sea $\ell_0 = (\perp, o, \{o\}, [0, T] \to 0)$ con costo igual a 0 $\forall t \in [0, t]$
2 Sea q una cola de prioridad inicializada con $(\text{ct}(\ell_0), 1, \text{o}, \{o\})$
3 Sea $\mathcal{L}$ un diccionario de labels procesados inicializado con $(1, o, \{0\}) \to \{\ell_0\}$
4 **while** $q \neq \emptyset$ **do**
5 Sea (b, k, v, S) el resultado de extraer el primer elemento de q
6 Sea L la secuencia de labels ℓ en $\mathcal{L}(k, v, s)$ con $\text{ct}(\ell) = b$ a ser procesada
7 **if** $L = \emptyset$ **then**
8 Continuar con la próxima iteración
9 **if** $k = n + 1$ **then**
10 **return** *tour representado por un label de* L
11 **for** $(v, w) \in A | w \notin S$ **do**
12 Sea M la secuencia resultante de extender los labels de L por (v, w)
13 **for** $m \in M$ **do**
14 Restringir $\text{dom}(m)$ a los tiempos de llegada menores a $b(w)$, descartar m si $\text{dom}(m) = \emptyset$
15 Se realiza un merge de M en N, donde $N = \mathcal{L}(k + 1, w, S + \{w\})$
16 **for** $l \in N$ **do**
17 Se aplica la Regla 4 para eliminar todos los tiempos dominados de $\text{dom}(l)$
18 **for** $m \in M \cap N$ **do**
19 Actualizar a m con $\text{ct}(m)$ y descartar m si $\text{ct}(m) > ub$ (Regla 2)
20 Encolar $(\text{ct}(m), k + 1, w, S + \{w\})$ en q $\forall m \in M \cap N$

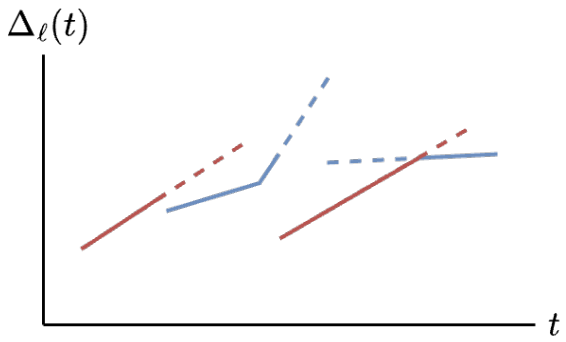


Fig. 2.2: Merge entre dos secuencias de labels (roja y azul). De cada secuencia, se mantienen las secciones con de menor $\Delta_\ell(t)$. Las secciones punteadas son descartadas.

2.3 Relajación NG

Recordemos que la cota de terminación de un label es un valor que nos permite estimar la duración del tour de un label L, que es una continuación del camino representado por el label que visita todos los destinos. Esta cota nos permite descartar labels por estar acotados. En particular esta cota es una cota inferior y por lo tanto debe ser menor o igual a la duración del tour porque queremos descartar labels que estamos seguros que tendrán una duración final mayor a una determinada cota superior. Además, utilizamos esta cota para elegir los labels mas relevantes a procesar primero y nos da la condición de terminación del algoritmo.

Una forma de obtener esta cota podría ser usando el costo del label, ya que nunca puede ser mayor a ningún tour resultante de extenderlo. Sin embargo, esta cota es muy lejana al costo real y no nos permitiría descartar ningún label. Recordemos que cuanto más ajustada es la cota, más posibilidades tenemos de descartar el label. Necesitamos una forma de obtener un valor cercano a la duración del tour pero que no sea tan costosa de calcular como el valor exacto, que implicaría extender el label por todos los caminos posibles.

Para resolver este problema, utilizaremos una relajación del TDTSPTW. Una relajación de un problema es una expansión de su conjunto de soluciones. Es decir, se quitan restricciones del problema original, con el objetivo de simplificarlo para que sea más fácil de resolver. Lógicamente, no necesariamente va a servir la solución de la relajación como solución del original. Pero sí la podemos utilizar para acotar las soluciones del original, ya que necesariamente una solución del problema relajado tiene costo menor o igual al del problema original. Esto ocurre porque, una solución del problema original siempre es solución de la versión relajada, por lo que el costo es a lo sumo igual.

Buscamos, dado un label L, obtener una cota inferior de la duración de su tour. Para lograrlo, queremos resolver el problema relajado comenzando con el camino representado por L. Obtenemos un nuevo tour, que comienza con el camino de L y continua hasta formar una solución del problema relajado. El costo de esta solución será menor o igual al costo del tour de L en el problema original.

Concretamente, relajaremos el problema utilizando la técnica *NG*, propuesta originalmente por Baldacci et al. (2011) para el TSPTW y adaptada por Lera-Romero et al. (2022) para el TDTSPTW. Como dijimos, una relajación es una quita de restricciones so-

bre las soluciones de un problema. Con la relajación NG, vamos a permitir algunos tours no elementales como solución, con el objetivo de reducir la cantidad buckets mantenidas por el algoritmo. Recordemos que el Algoritmo 1 mantiene un bucket por cada par (S, v) donde S es el conjunto de vértices visitados por el label y v es el último vértice visitado. Al reducir la cantidad de buckets, aumentan las "colisiones" entre los labels extendidos y, por lo tanto, las dominaciones. Esto recorta la cantidad de labels extendidos en cada iteración del algoritmo.

En la relajación NG, se modifica el conjunto S asociado a cada label. Recordamos que para un label L, $S(L)$ es el conjunto de vertices que no pueden ser visitados en la proxima extensión del label ya que ya fueron visitados por el camino representado por L. En esta relajación, S no contendrá necesariamente todos los vertices que fueron visitados en algún momento por el camino. Esto permitirá que se vuelvan a visitar vértices en ciertos casos.

Concretamente, requeriremos que los ciclos salgan de un *vecindario* para evitar ciclos cortos. Llamamos vecindario, a un conjunto de vértices asociado a cada vértice del grafo. Este es un conjunto que contiene a v y a otros vertices del grafo, definido antes de comenzar a resolver el problema. Cuando se extiende un label a un nuevo vértice w, lo agregamos al conjunto S de vertices no visitables del label y eliminamos todos los vertices que no pertenezcan al vecindario de w. De esta manera, prohibimos que se vuelva a visitar un vértice mientras se estén recorriendo otros vertices que lo tengan en su vecindario.

Además, debemos redefinir la solución del problema, que en el original era un camino de costo mínimo que, comenzando por el deposito inicial y terminando en el deposito final, visita los $n+2$ vértices del grafo una única vez, es decir, un tour elemental. Similarmente, un *tour NG* es un camino que arranca en el deposito inicial, termina en el final y visita exactamente $n+2$ vertices. Pero, en vez de exigir que el camino sea elemental, solamente vamos a requerir que no visite vertices prohibidos según la definición de la relajación NG. Luego, el objetivo del TDTSPTW-NG es encontrar un *tour NG* de costo mínimo.

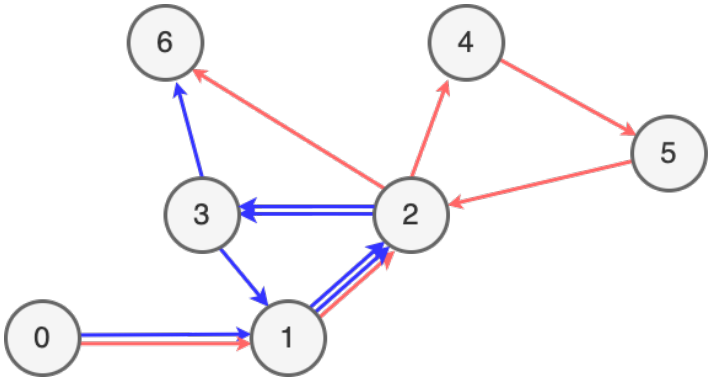
En la Figura 2.3 podemos ver un ejemplo de como funciona la relajación NG en un grafo.

2.3.1 Selección de vecindarios

Como dijimos, el problema con la relajación NG permite caminos no elementales pero prohíbe ciertos ciclos. Intuitivamente, para que las cotas resultantes de resolver el problema con la relajación sean lo más cercanas al original, queremos prohibir ciclos cortos, ya que implican que el camino se queda ciclando por un pequeño conjunto de vértices cercanos sin recorrer muchos vértices distintos del grafo, lo que no es muy similar a una solución del problema original. En cambio, un camino que, si bien no es elemental, tiene un ciclo muy largo, necesariamente va a visitar una gran cantidad de vértices distintos y por lo tanto tendrá un costo más cercano al de la solución elemental.

Para lograr eliminar los ciclos cortos y no cualquier otro, es importante seleccionar los vecinos de cada vértice correctamente. Sin embargo, no es fácil determinar a priori que vertices son los que debemos incluir en cada vecindario para evitar ciclos cortos. Además, tampoco es fácil determinar el tamaño ideal de los vecindarios, cuanto más grandes, más ciclos se prohíben y mejores son las cotas, pero también aumenta la dificultad del problema.

Para resolver esta dificultad, Lera-Romero et al. (2022) utilizan el método de *aumento dinámico de vecindarios* (DNA por sus siglas en ingles), propuesto originalmente por Tilk y Irnich (2017), que construye los vecindarios de manera iterativa. Se busca una



Nodo	Vecindario
1	1,2,3
2	1,2,3
3	1,2,3
4	2,4
5	5

Fig. 2.3: Ejemplo de un grafo con un tour NG (rojo) comparado con un camino de longitud n que no respeta la restricción NG (azul). El camino rojo tiene el ciclo $[1, 2, 3, 1]$ prohibido ya que, cuando se esta extendiendo el camino $[1, 2, 3]$, el vértice 1 pertenece al conjunto S de vértices no visitables porque esta en el vecindario de 2 y 3. Por lo tanto, el camino $[1, 2, 3]$ no puede ser extendido para visitar nuevamente al vértice 1. Lo mismo ocurre con $[2, 3, 1, 2]$ y $[3, 1, 2, 3]$. En cambio, el ciclo $[2, 4, 5, 2]$ no esta prohibido ya que 2 no pertenece al vecindario de 5.

solución optima de la relajación para luego agrandar los vecindarios y continuar repitiendo el proceso de búsqueda de solución, seguida de ampliación de vecindarios.

Se comienza resolviendo el problema con los tres vecinos más cercanos en el vecindario de cada vértice. Se obtiene un tour NG óptimo. Necesariamente, este tour NG óptimo es elemental o contiene un ciclo. Si es elemental, también es una solución valida para el problema original y no hace falta hacer nada más. Si no lo es, vamos a prohibir sus ciclos y volver a buscar un tour NG óptimo con esos ciclos prohibidos. Para prohibir un ciclo $v_1, v_2, \ldots, v_k, v_1$, se agrega v_1 al vecindario de cada $v_i, 1 < i \leq k$.

Se vuelve a ejecutar el algoritmo con esta modificación en los vecindarios, obteniendo un nuevo tour NG. Se continua iterando de esta manera hasta que: se encuentra una solución elemental; los vecindarios alcanzan algún tamaño máximo K prefijado; o se alcanza el límite de tiempo designado. En resumen, se comienza con vecindarios chicos y al expandirlos iterativamente nos aseguramos que los vertices agregados son significativos, y sólo agregamos el número de vértices que podemos soportar para resolver el problema en el tiempo asignado.

A medida que se prohíben más ciclos y aumenta el tamaño de los vecindarios, también aumenta rápidamente el tiempo de ejecución, hasta que eventualmente se supera el tiempo asignado para la relajación y se toman las cotas del la ultima versión del TDTSPTW-NG que se pudo terminar de resolver.

2.3.2 Dominación NG

Recordemos que un label ℓ está dominado por otro m si cualquier extensión p a partir de ℓ puede aplicarse también a partir de m y con un costo no mayor. En el Algoritmo 1, si ℓ y m están en el mismo vértice final y visitaron a los mismos vértices, entonces cualquier extensión p de ℓ puede aplicarse en m.

Para el problema con la relajación NG, necesitamos adaptar la regla de dominación para que la relajación resulte eficiente. En este caso, no es necesario que ℓ y m hayan

visitado exactamente los mismos vértices, si no que alcanza con que hayan visitado la misma cantidad de vértices y el conjunto de vertices no visitables de m este contenido o sea igual al de de ℓ. Esto implica que m puede extenderse a cualquier vértice de los que ℓ puede.

Regla 5 (Dominación NG)**.** *Un label NG ℓ está dominado por otro m en el tiempo $t \in \mathrm{TAF}(\ell)$ si $t \in \mathrm{TAF}(m), v(m) = v(\ell), S(m) \subseteq S(\ell)$, y $\Delta_m(t) \leq \Delta_\ell(t)$.*

Una diferencia importante entre la Regla 1 y la Regla 5 es que la Regla 1 requiere que $S(\ell) = S(m)$, mientras que la Regla 5 permite que $S(m) \subseteq S(\ell)$ para que m domine a ℓ. Esta diferencia implica que, cuando queremos dominar un label ℓ, con la Regla 1 podemos buscar eficientemente en un diccionario al conjunto labels que visitan los mismos vértices que ℓ. En cambio, con la Regla 5, debemos comparar ℓ con todos los labels cuyo conjunto de no visitables es subconjunto de $S(\ell)$. Si bien con este cambio aumenta drásticamente el número de labels dominados, esta búsqueda es costosa ya que hay $2^{|S(\ell)|}$ posibles buckets de labels que lo podrían dominar.

En el algoritmo de Lera-Romero et al. (2022) se mantienen los labels en la estructura de buckets $(|S(\ell')|, v(\ell'), S(\ell'))$. Para buscar los buckets que contienen labels que pueden dominar a ℓ, se itera por todos los buckets del diccionario con labels existentes tales con $|S| \leq |S(\ell)|$ y se compara cada uno para ver si su conjunto S está incluido en $S(\ell)$. Esta estrategia tiene la desventaja de que se deben enumerar todos los buckets del diccionario con $|S| \leq |S(\ell)|$, por más que no sean subconjunto. Otra alternativa seria buscar en el diccionario el bucket correspondiente a cada posible subconjunto de $S(\ell)$, sin saber de antemano si existen labels en ese bucket. En la práctica, la estrategia utilizada por Lera-Romero et al. (2022) resulta ser la más rápida cuando los vecindarios son grandes. Esto sucede por que la segunda estrategia requiere hacer $2^{|S(\ell)|}$ consultas, incluyendo las consultas redundantes por buckets sin labels asociados y el número de buckets en el diccionario con $|S| \leq |S(\ell)|$ es menor, en general. De todas maneras, esta búsqueda es la mayor parte del costo total de resolver esta relajación especialmente a medida que aumenta el tamaño de los vecindarios. Esto limita fuertemente la cantidad de iteraciones de DNA que se pueden resolver, evitando que se encuentren mejores cotas.

Luego, para resolver el problema con la relajación NG el Algoritmo 1 de Lera-Romero et al. (2022) fue modificado en la líneas 15 a 17 para aplicar la nueva regla de dominación. A la hora de querer dominar una secuencia de labels M, anteriormente accedíamos a la secuencia existente capaz de dominarla N directamente desde $\mathcal{L}$ (Algoritmo 1, línea 15). Para el TDTSPTW-NG, como mencionamos, debemos buscar todos los $N = \mathcal{L}(k+1, w, S')$ que existen en $\mathcal{L}$, para cualquier S' con $|S'| \leq |S(M)|$(línea 15). Para cada N, comprobamos si se cumple que $S(N) \subseteq S(M)$, donde $S(M) = (S + \{w\}) \cap N_v$, para poder aplicar la Regla 5 y dominar los labels de M (línea 16). En caso de cumplirse la condición, se dominan los labels de M (línea 17). Finalmente, si al menos un label de M no fue dominado totalmente, se inserta M en $\mathcal{L}$ (línea 19).

2.3.3 Algoritmo backward

Como dijimos, el objetivo de resolver la relajación es utilizarla para obtener la cota de terminación de un label ℓ extendido en el Algoritmo 1. En otras palabras, lo que buscamos es, de cierta manera, correr el algoritmo con la relajación utilizando como punto de partida el camino parcial definido por ℓ y usar el resultado como su cota. Dado ℓ, podemos resolver

Algorithm 2: Algoritmo para TDTSPTW-NG

Input:
...,
N_v, vecindario, para cada $v \in V$
Result: Un tour NG de costo mínimo con duración menor a ub

...
for $N = \mathcal{L}(k+1, w, S')$ *para cualquier* S' *tal que* N *esta definido y* $|S'| \leq |S(M)|$ **do**
  if $S' \subseteq (S + \{w\}) \cap N_v$ **then**
    Se aplica la Regla 5 para eliminar todos los tiempos dominados de $l \in M$ por los labels de N
if $M \neq \emptyset$ **then**
  Se inserta M en $\mathcal{L}(k+1, w, (S + \{w\}) \cap N_v)$
...

el TDTSPTW-NG restringiendo las soluciones a los caminos que comienzan con el camino representado por ℓ.

Sin embargo, ejecutar el algoritmo NG para cada label demasiado costoso, ya que implica pagar el costo entero de resolverlo para cada label que extendemos. Lo que vamos a buscar entonces es resolverlo una única vez, antes de ejecutar el Algoritmo 1, y generar de esta manera todas las cotas para todos los labels posibles del problema original.

Para lograrlo, comencemos observando que el diccionario de labels $\mathcal{L}$ luego de finalizar la ejecución del algoritmo para resolver el TDTSPTW, contiene todos los labels parciales óptimos para el problema, es decir, si tomamos algún label ℓ de $\mathcal{L}$, que finaliza en v y visitó los vértices del conjunto S, ℓ representa el mejor camino que comienza en el origen y finaliza en v, visitando los vértices de S. Algo similar es cierto para el algoritmo NG, donde un label ℓ' del diccionario de labels NG $\mathcal{L}'$, que finaliza en w y visito n vértices representa el mejor camino NG que va del origen a w y visita exactamente n vértices. Además, observemos que si $v = w$ y $|S| = n$, entonces el costo de ℓ es mayor o igual al de ℓ', ya que con la relajación NG, el conjunto de soluciones es mayor.

Queremos aprovechar el hecho de que ℓ' es una cota inferior de ℓ, pero ℓ' es un camino parcial que comienza desde el origen y finaliza en un vértice del grafo. Para obtener nuestra cota de terminación necesitamos un camino NG que comienza desde un vértice del grafo y finaliza en el depósito final, para poder combinarlo con ℓ y así obtener el tour NG de todo el grafo.

Necesitamos generar labels NG *backward* que representan un camino que parte de algún vértice y finalizan en el depósito final. Con esto, podríamos buscar el label NG *backward* que comienza en v y recorre $n - |S|$ vértices y su costo, sumado al costo de ℓ, seria la cota de terminación de ℓ.

Si se ejecuta el algoritmo NG en la dirección inversa, comenzando por el depósito final e invirtiendo las direcciones de las aristas, lo que se obtienen son labels NG que van desde el depósito final hasta uno de los vértices. Críticamente, estos labels tienen el mismo costo que los que recorren el mismo camino sin invertir la dirección de las aristas.

Luego, combinado un label NG backward con un label de TDTSPTW, obtenemos un camino completo de origen a fin que comienza siendo elemental y finaliza como un tour

NG. El costo de recorrer este camino es menor o igual al del tour del label TDTSPTW, por lo que funciona como la cota de terminación que estamos buscando.

En resumen, antes de comenzar a resolver el TDTSPTW, ejecutamos el algoritmo NG con el proceso DNA para el problema invertido, obteniendo como resultado el diccionario de labels generado. Luego, mientras ejecutamos el algoritmo para el problema original, cuando necesitemos obtener la cota de terminación de un label, simplemente debemos sumarle el costo del label NG que comienza en su vértice final y visita el número de vértices que le falta para completar el tour.

3. LABEL TREE

En este capítulo vamos a introducir una nueva estructura de datos: el *Label Tree*. Como su nombre lo indica, esta estructura es un árbol que representa un conjunto labels y su propósito es resolver el problema con la relajación NG más eficientemente. Mencionamos en la Sección 2.3.2 que, en el algoritmo para el TDTSPTW-NG, cuando queremos ver si un label ℓ está dominado, debemos comprobar si algún otro label ya extendido lo domina. Por la Regla 5, cualquier label m que visita la misma cantidad de vértices que ℓ y que $S(m) \subseteq S(\ell)$, puede llegar a dominar a ℓ. Sin embargo, no existe una forma eficiente de encontrar los labels m que cumplen la condición $S(m) \subseteq S(\ell)$. Actualmente, la forma en que lo resuelven Lera-Romero et al. (2022) es iterando por todo el diccionario de labels existentes con el mismo vértice final que ℓ y comparando los conjuntos de vértices no visitables de cada uno con $S(\ell)$. Algunos de estos labels tendrán su conjunto de vértices no visitables contenido por $S(\ell)$ y podrán dominar a ℓ, pero la gran mayoría no.

El costo elevado de recorrer todos los labels ya extendidos limita fuertemente el tamaño de los vecindarios ya que a mayor tamaño, más grandes serán los conjuntos de vértices no visitables por los labels, lo que resulta en una mayor cantidad de labels distintos almacenados en el diccionario. Como vimos anteriormente, el objetivo es aumentar el tamaño de los vecindarios lo más posible para que las soluciones del problema con la relajación NG sean más similares a las soluciones del problema original y, por lo tanto, se obtengan cotas de terminación más ajustadas. Para poder maximizar el valor obtenido de resolver el problema NG, proponemos la estructura Label Tree para acelerar el proceso de dominación.

Esta será la contribución de este trabajo: acelerar la búsqueda de labels m que cumplen la condición $S(m) \subseteq S(\ell)$ para el proceso de dominación de un label ℓ en el TDTSPTW-NG. Utilizaremos el Label Tree, una estructura de datos especializada que permite recorrer de manera más eficiente un conjunto de labels. En esta estructura, se usan los conjuntos de vértices no visitables para indexar a cada label y la función de inclusión entre conjuntos para ordenarlos.

Como los labels están ordenados por la función de inclusión entre los conjuntos S, podremos encontrar aquellos que cumplen la condición $S(m) \subseteq S(\ell)$, sin tener que recorrer todos los labels conocidos. Esto permite saltear grandes ramas de labels que no podrán ser candidatos para dominación, lo que se traduce a un recorte importante en la cantidad de operaciones necesarias para resolver una instancia del problema.

3.1 Propiedades del Label Tree

El Label Tree es una estructura de datos en forma de árbol que almacena un conjunto de labels. Cada nodo n tiene un conjunto de vértices $S(n)$ asociado y puede contener una secuencia de labels $L(n)$. Si n contiene labels, el conjunto de vértices no visitables por ellos es $S(n)$. Como dijimos, es posible que n no contenga labels, pero ese caso sólo puede ocurrir cuando n no es una hoja.

Los labels de un Label Tree se mantienen ordenados en nodos de manera que se respete la *propiedad de ordenamiento vertical*.

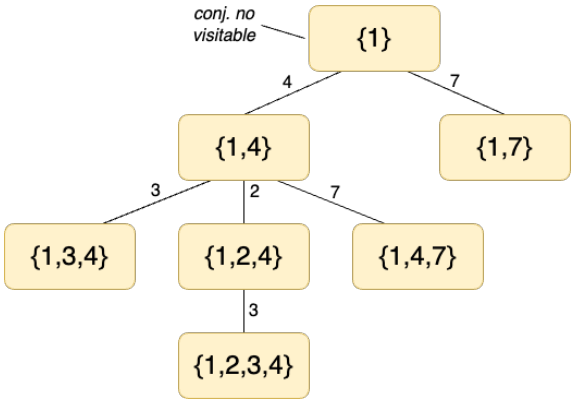


Fig. 3.1: Ejemplo de organización de nodos de un Label Tree con el conjunto S de cada nodo. Notemos que para cada nodo n, hijo de m, $S(m) \subset S(n)$. En el TDTSPTW-NG, donde S representa el conjunto de vértices no visitables por los labels de $L(n)$, cualquier label de $L(m)$ puede visitar a todos los vértices que pueden visitar los de n. Supongamos que queremos buscar el nodo con $S = \{1, 4, 7\}$, comenzando desde la raíz vemos que 4 es el primer vértice que no pertenece a $\{1\}$ pero si a S. Luego, nos dirigimos por esa arista al nodo $\{1, 4\}$. Nuevamente, nos dirigimos por el primer vértice 7 para encontrar el nodo buscado.

Propiedad. *En todo subárbol con raíz r que representa un conjunto de labels L, $S(r) = \bigcap_{\ell \in L} S(\ell)$. (Figura 3.1).*

En otras palabras, para un nodo n y un label ℓ_i, si $S(n) \subset S(\ell_i)$, podemos decir que ℓ_i no puede visitar a todos los vértices que no pueden visitar los labels de $L(n)$ y que además hay al menos un vértice más que no puede visitar. Por esta propiedad, el árbol queda ordenado de manera que los nodos cercanos a la raíz tienen labels que pueden visitar a más vértices que los nodos descendientes.

Para cada nodo n con padre p, vamos a destacar un indice, llamado $unique(n)$ que corresponde al mínimo de $\{v \mid v \in S(n) \setminus S(p)\}$. Es decir, $unique(n)$ es el "primer" vértice de $S(n)$ que no pertenece a $S(p)$ si ordenamos los vértices por su número. Éste atributo permitirá recorrer el árbol eficientemente al momento de buscar el nodo de un S determinado. Como expresamos, la idea es que *unique* funcione como un índice, por lo cual nunca hay dos nodos hermanos que tengan el mismo valor de *unique* (Figura 3.2).

Propiedad. *Sean n y m dos nodos de un Label Tree. Si n y m son hermanos, entonces $unique(n) \neq unique(m)$.*

Como vamos a usar el Label Tree para resolver el TDTSPTW-NG, vamos a requerir que no contenga labels dominados entre sí. Por lo tanto, cada label en una secuencia de labels de un nodo, debe tener dominio disjunto.

Propiedad. *Sean ℓ_1 y ℓ_2, dos labels de $L(n)$,* $\mathrm{TAF}(\ell_1) \cap \mathrm{TAF}(\ell_2) = \emptyset$

Además, dados dos nodos n y m, si m es descendiente de n, sabemos por la propiedad de ordenamiento vertical que $S(n) \subset S(m)$. Es decir que los labels de $L(n)$ pueden visitar a todos los vértices que pueden visitar los de $L(m)$. Esto implica que los vértices de m deben tener costo menor o, de lo contrario, estarían dominados por los de $L(n)$. Luego, a

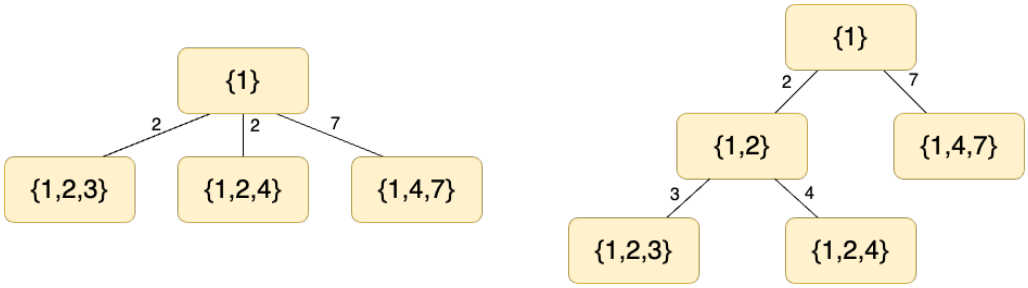


Fig. 3.2: Ejemplo de un Label Tree inválido (izquierda), por tener dos hijos con el mismo *unique*. Y su representación corregida (derecha).

medida que se recorre un Label Tree y se visitan nodos más lejanos a la raíz, se encuentran labels que tienen costo menor, además de tener conjuntos de vértices no visitables más grandes.

Con estas propiedades, podemos tener una intuición de como usar Label Trees para los labels del TDTSPTW-NG ayuda a acelerar su ejecución. Al querer ver si un label ℓ que visitó k vértices está dominado, anteriormente debíamos buscar todos los labels m existentes que también visitaron k o menos vértices y comparar el conjunto S de cada uno con $S(\ell)$ para encontrar los que cumplen que $S(m) \subseteq S(\ell)$. En cambio, si los labels están almacenados en un Label Tree, supongamos que estamos recorriendo un nodo n tal que $S(n) \nsubseteq S(\ell)$. En este caso, no es cierto que los labels de $L(n)$ pueden dominar a ℓ por la Regla 5, por lo que n puede ser descartado. Además, por la propiedad de ordenamiento vertical, los labels de cualquier nodo descendiente de n también pueden ser descartados. En resumen, si al recorrer un Label Tree para dominar un label ℓ, encontramos un nodo n tal que $S(n) \nsubseteq S(\ell)$, se puede descartar todo el subárbol de labels que tiene a n como raíz.

Comparado con el método anterior, que no permitía saltear ningún label, esto es una gran mejora, pero podemos hacer aún más. Para ello, cada nodo n mantiene, además de una secuencia de labels, un costo $c(n)$, que es el mínimo costo de todos los dominios de las funciones de los labels del subárbol con raíz en n. Con $c(n)$, requeriremos que los nodos de un Label Tree respeten una segunda propiedad que establecerá un ordenamiento entre los nodos hermanos: la *propiedad de ordenamiento horizontal*.

Propiedad. *Si dos nodos n y m de un Label Tree son hermanos, están ordenados de menor a mayor por el costo mínimo de su subárbol. (Figura 3.3).*

Esta propiedad establece que, al recorrer un Label Tree, se visitarán los nodos de una secuencia de hermanos en orden de sus costos. Si al intentar dominar un label ℓ, encontramos un nodo n tal que el costo máximo de ℓ es menor a $c(n)$, ningún label del subárbol de n podrá dominar a ℓ por tener costo mayor, por lo que n puede ser descartado. Lo que es más, por la propiedad de ordenamiento horizontal todos los subárboles de los hermanos mayores de n (que todavía no han sido visitados), también pueden ser descartados. Saltear nodos del Label Tree por la comparación de costos permite saltear aún más nodos del Label Tree que al comparar sus conjuntos de no visitables, por lo que es conveniente comparar primero por costo.

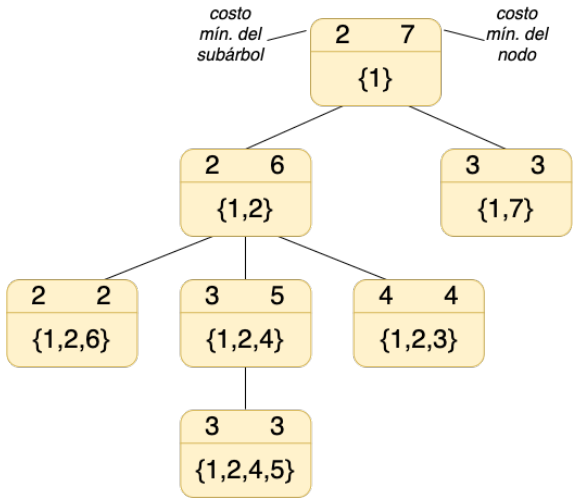


Fig. 3.3: Estructura de un Label Tree con el conjunto S de cada nodo y sus costos. Observemos que los nodos hermanos están ordenados, de izquierda a derecha, por el costo mínimo de su subárbol c, y no por el costo mínimo del nodo.

3.2 Representación

Un Label Tree esta representado por un conjunto de nodos que contienen una secuencia de labels y un costo. Como vemos en la Figura 3.4, cada nodo n contiene, además, dos punteros *next* y *skip*. *next* apunta a su primer hijo y *skip* apunta a su próximo hermano. Recordemos que la secuencia de hijos de un nodo esta ordenada por costos de manera ascendente. Por lo tanto, *next* apunta al hijo de costo mínimo y *skip* apunta al primer hermano con costo mayor a n.

3.3 Uso en el TDTSPTW-NG

Como mencionamos al principio de éste capítulo, el objetivo es usar Label Trees para el proceso de dominación en el TDTSPTW-NG. Este proceso está descrito en el Algoritmo 2 donde, dada una secuencia de labels M, recorremos el conjunto de secuencias de labels N tales que visitaron a lo sumo la misma cantidad de vértices y terminan en el mismo (línea 15). Para cada una, comparamos su conjunto de vértices no visitables $S(N)$ con $S(M)$ (línea 16) y, si esta contenido, dominamos los labels de M con los de N (línea 17). Al finalizar este proceso, si quedan labels en M sin dominar, se insertan en el diccionario de labels $\mathcal{L}$.

Veamos cómo adaptar este algoritmo para incluir Label Trees (Algoritmo 3). Anteriormente, comenzábamos definiendo un diccionario de labels $\mathcal{L}$, que indexaba secuencias de labels M por $(k(M), v(M), S(M))$, donde $k(M)$ es la cantidad de vértices visitados por los labels de M, $v(M)$ es el último vértice visitado y $S(M)$ es su conjunto de vértices no visitables (Algoritmo 1, línea 3). Este es el diccionario que reemplazaremos para usar Label Trees. En el Algoritmo 3, definimos un diccionario $\mathcal{T}$ que, dada la tupla (k, v), devuelve un Label Tree T que contiene secuencias de labels M, tales que $k(M) = k$ y $v(M) = v$ (línea 3).

Una vez que definimos el diccionario $\mathcal{T}$ que almacena Label Trees con los labels proce-

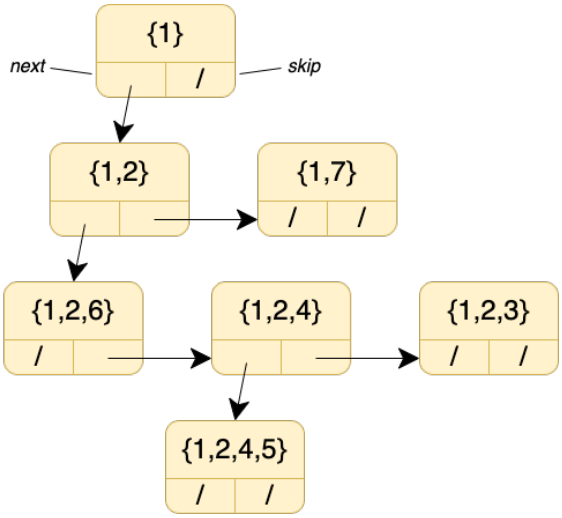


Fig. 3.4: Representación del Label Tree en 3.3 con el conjunto S de cada nodo y sus punteros *next* y *skip*. Podemos ver que un nodo no tiene una referencia directa a cada uno de sus hijos, si no que se acceden a través de la secuencia que comienza por su primer hijo y esta definida por los punteros *skip*.

sados, veamos cómo lo usamos en la dominación. Un Label Tree se recorre mediante un *cursor* que apunta a uno de sus nodos. El orden en el que se recorre es: primero nos desplazamos al primer hijo; si no hay hijos nos desplazamos al siguiente hermano; cuando un nodo no tiene hijos ni hermanos mayores, nos desplazamos al primer hermano del predecesor más próximo que los tenga.

Luego, cuando queremos dominar un label M, comenzamos buscando el Label Tree T que contiene labels candidatos a dominarlo (línea 15), movemos su cursor a la raíz (línea 16) y comenzamos a recorrerlo mediante dicho cursor. Primero comparamos el costo del nodo apuntado por el cursor t con el costo máximo de un label de M (línea 19). Recordemos que el costo c de un nodo está definido por el costo mínimo de un label en su subárbol. Si $c(t)$ es mayor al de M, ningún label del subárbol de t puede dominar a los labels de M. Además, por la propiedad de ordenamiento horizontal, ningún hermano no visitado de t tiene un label en su subárbol que puede dominar a un label de M. En este caso, queremos avanzar el cursor al siguiente nodo fuera del subárbol del padre de t. Llamamos *tio derecho* a éste nodo, que sería el hermano e inmediatamente a la derecha del padre. Utilizamos la operación *cut* que avanza el cursor de esta manera (línea 20). Luego, continuamos con la siguiente iteración.

En caso de que el costo no sea mayor, comparamos los conjuntos no visitables asociados a t y M. Si $S(t) \nsubseteq S(M)$, donde $S(M) = (S + \{w\}) \cap N_v$ (línea 21), no podremos aplicar aplicar la Regla 5 para dominar los labels de M por los de t. Además, por la propiedad de ordenamiento vertical, ningún label de un nodo del subárbol de t podrá dominar a un label de M. Luego, debemos avanzar el cursor al siguiente hermano de t. Para ello, utilizamos la operación *skip* (línea 22) y continuamos con la siguiente iteración.

Si no se cumple ninguna de las condiciones anteriores, entonces $S(t) \subseteq S(M)$ y se dominan los labels de M por los labels de t (línea 24). Además, los labels del subárbol de t siguen siendo candidatos a la dominación, por lo que el cursor se avanza al siguiente

nodo, sin saltear a ningún otro, utilizando la operación *advance* (línea 25).

Finalmente, cuando terminamos de recorrer el árbol, si existe al menos un label de M con dominio no-vacío, se inserta M en su correspondiente Label Tree (línea 27).

En las siguientes secciones, hablaremos más en detalle sobre el funcionamiento y la implementación de las operaciones de movimiento del cursor, sobre la operación de *insert* y cómo se mantienen los invariantes de la estructura.

Algorithm 3: Algoritmo para TDTSPTW-NG con Label Tree

Input:
...,
Result: Un tour NG de costo mínimo con duración menor a ub

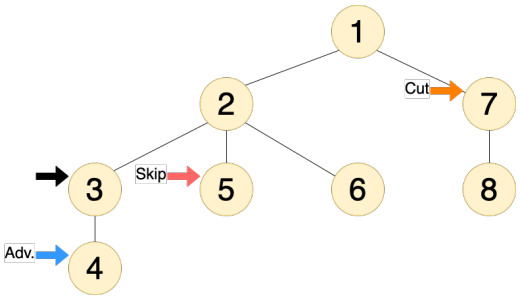
...
3 Sea $\mathcal{T}$ un diccionario de tal que $\forall k \in [0, n], v \in V, (k, v) \to T$ donde T es un Label Tree vacío
...
15 Sea T el Label Tree en $\mathcal{T}(k, v)$
16 Se mueve el cursor de T a la raíz
17 **while** *No se terminó de recorrer* T **do**
18 Sea t el nodo apuntado por el cursor de T
19 **if** $t.costo \geq M.costo$ **then**
20 $T.cut()$
21 **else if** $S(t) \nsubseteq (S + \{w\}) \cap N_v$ **then**
22 $T.skip()$
23 **else**
24 Se aplica la Regla 5 para eliminar todos los tiempos dominados de $l \in M$ por los labels de t
25 $T.advance()$
26 **if** $M \neq \emptyset$ **then**
27 Se inserta M en $\mathcal{T}(k + 1, w, (S + \{w\}) \cap N_v)$

3.4 Cursor

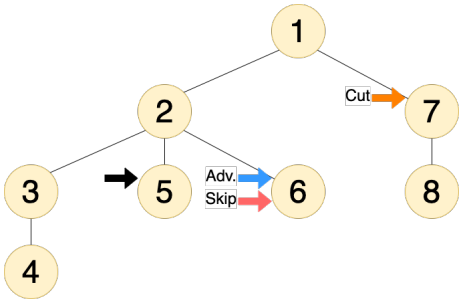
Como mencionamos, para acceder a los labels de un Label Tree, se utiliza un cursor que recorre los nodos. El cursor se inicializa en la raíz del árbol con la operación *begin*. Para desplazar el cursor usamos las operaciones *advance*, *skip* y *cut*. En la Figura 3.5, podemos ver el movimiento del cursor con cada una de las operaciones para cada caso posible.

En la mayoría de los casos, la implementación de las operaciones *advance* (Algoritmo 4) y *skip* (Algoritmo 5) es simple, ya que mueven el cursor al nodo apuntado por los punteros *next* y *skip*, respectivamente. Recordemos que, para cada nodo del Label Tree, *next* apunta a su primer hijo y *skip* apunta a su próximo hermano. En caso de que se haga un *advance* desde un nodo sin hijos, se utiliza también el puntero *skip* (Figura 3.5b).

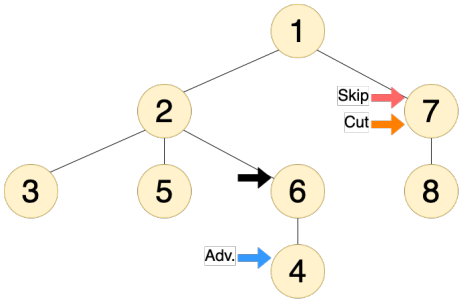
Para *cut* (Algoritmo 6), no se puede seguir directamente ningún puntero de los que definen la estructura. Como mencionamos anteriormente, *cut* saltea a los hijos del nodo actual y a sus hermanos. Es decir, saltea al subárbol del padre del nodo actual. Luego, para implementarla, el cursor mantiene una *pila*. En todo momento, la pila debe contener



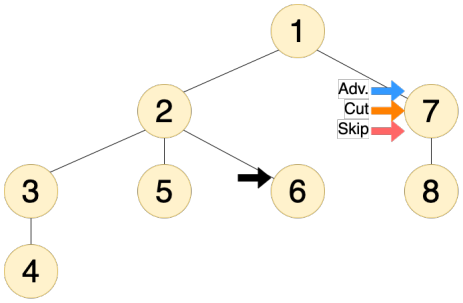
(a) Movimiento del cursor cuando apunta a un nodo con hijos y con hermanos mayores.



(b) Movimiento del cursor cuando apunta a un nodo sin hijos y con hermanos mayores.



(c) Movimiento del cursor cuando apunta a un nodo con hijos y sin hermanos mayores.



(d) Movimiento del cursor cuando apunta a un nodo sin hijos y sin hermanos mayores.

Fig. 3.5: Movimiento del cursor de un Label Tree

Algorithm 4: Advance

Input:
T, Label Tree,

```
1 if T.cursor.skip ≠ NIL then
2     T.pila.push(T.cursor.skip)
3 T.cursor ← T.cursor.next
4 if cursor = NIL then
5     cut(T)
```

Algorithm 5: Skip

Input:
T, Label Tree,

```
1 T.cursor ← T.cursor.skip
2 if cursor = NIL then
3     cut(T)
```

en su tope al próximo nodo al que se debe mover el cursor al realizar un *cut*. Para ello, cuando se hace un *advance* hacia el hijo de un nodo, debemos poner a su *skip* en el tope de la pila.

La pila, además, se utiliza para los casos en los que se realiza una operación *skip* con un nodo que no tiene hermanos (Figura 3.5c y 3.5d) o cuando se hace *advance* en un nodo sin hijos ni hermanos (Figura 3.5d). En estos casos, el puntero *skip* es nulo y el cursor se mueve igual que haciendo un *cut*.

Siempre que se inicializa el cursor, moviéndolo a la raíz del árbol, también debe vaciarse la pila. Si en algún momento se realiza un *cut* con la pila vacía (directamente o a través de *advance* o *skip*), quiere decir que se terminó de recorrer el Label Tree.

Algorithm 6: Cut

Input:
T, Label Tree,

```
1 if T.pila.top = NIL then
2     T.cursor ← NIL
3 else
4     T.cursor ← pila.pop()
```

3.4.1 insert

Como vimos, en la línea 27 del Algoritmo 3 necesitamos agregar secuencias de labels que no pudieron ser dominados al Label Tree. Por este motivo, el Label Tree debe tener una operación para realizar esta tarea que respete los invariantes de la estructura. La operación *insert* permite agregar una secuencia de labels L con un conjunto de vértices no visitables S al árbol. Al insertar una nueva secuencia de labels, debemos reordenar la estructura para mantener la propiedad de ordenamiento vertical y la propiedad de ordenamiento horizontal. Recordemos que, cada nodo contiene un *unique*, que es el primer vértice que pertenece a su conjunto no visitable y no pertenece al de su padre. Luego, cada nodo contiene *labels*, *costo*, *next*, *skip*, S y *unique*. *unique* sirve para navegar eficientemente el árbol cuando estamos buscando el nodo que corresponde a un determinado S.

La operación *insert* (Algoritmo 7) recorre los nodos del árbol con un puntero n hasta encontrar un lugar para insertar a L. Se comienza inicializando a n para que apunte a la raíz del árbol (línea 1). Como el costo de cada nodo del árbol esta definido por el costo mínimo de uno de sus descendientes, debemos comenzar actualizando el costo de la raíz para que sea el mínimo entre su costo actual y el costo mínimo de L (línea 2). Recordemos que, como L es una secuencia de labels, su costo puede depender del momento de salida, por eso, tomamos el costo mínimo saliendo de cualquier instante en su dominio.

Luego, comenzamos a recorrer el árbol. En cada iteración del ciclo en la línea 3 se avanza el puntero a uno de los hijos del nodo apuntado, hasta encontrar el lugar donde insertar a L. El recorrido por el árbol no retrocede y no visita a más de un hijo del mismo padre. Por este motivo, se cumple la condición de que L será insertado en el subárbol que tiene a n en su raíz en todo momento. Para poder insertar en el subárbol de n, que contiene un conjunto no visitable $n.S$, es necesario que $n.S \subseteq S$, para mantener la propiedad de ordenamiento vertical. Si esto no se cumple (línea 4), debemos modificar a $n.S$, lo que implica un reordenamiento de sus hijos y reubicar su secuencia de labels en

otro nodo, si no estuviese vacía. El resultado deseado es que se cumpla $n.S \subseteq S$ y que se mantenga la propiedad de ordenamiento vertical entre n y cada uno de sus hijos. Para esto, vamos a utilizar la operación *split* (línea 5), que describiremos en la Sección 3.4.2.

De ahí en adelante, sabemos que se cumple $n.S \subseteq S$, hasta que volvamos a avanzar a n. Si $n.S = S$, encontramos el nodo donde insertar a L y finaliza la ejecución (líneas6-8). Para insertar L, realizamos un merge entre los nodos de n y L.

Si no se cumple la condición anterior, debemos avanzar el cursor a uno de los hijos de n o crear un nuevo hijo para mover el cursor. Calculamos el primer vértice u que esta en S y no en $n.S$ (línea 9). u será el *unique* del próximo nodo al que moveremos a n. Si existe algún nodo m entre los hijos de n tal que $m.unique = u$, lo eliminamos de la lista de hijos de n (línea 11) y creamos un nuevo puntero *next* que apunta a m (línea 12). Si no, *next* es un nodo nuevo con $next.S = S$ y $next.unique = u$ (línea 14). Al igual que con la raíz, debemos actualizar el costo de *next* ya que, con L en su subárbol puede ser menor (línea 15). Luego, insertamos a *next* entre los hijos de n (línea 16). En caso de que *next* apunte a un nodo m que ya era hijo de n, es necesario eliminarlo y reinsertarlo, cuando se modifica el costo, para quede bien ordenado. Finalmente, se avanza n a *next* (línea 17).

Algorithm 7: insert en un Label Tree

Input:
T, un Label Tree,
L, una secuencia de labels,
S, el conjunto de vértices que no pueden visitar los labels de L,
c, el costo mínimo de los labels de L,
Result: Modifica a T para incluir a L

1 $n = T.root$
2 $n.costo = min\{n.costo, c\}$
3 **while** *no hayamos insertado a* L **do**
4 **if** $n.S \not\subseteq S$ **then**
5 split(T, n, S)
6 **if** $n.S = S$ **then**
7 Se realiza un merge de L en $n.labels$
8 **return**
9 Sea u el primer vértice en S y no en $n.S$
10 **if** $\exists m \in$ *hijos de* $n \mid m.unique = u$ **then**
11 Se elimina a m de los hijos n
12 $next \leftarrow m$
13 **else**
14 $next$ es un nuevo nodo con $next.S = S$ y $next.unique = u$
15 $next.costo \leftarrow$ mínimo entre su costo y el costo mínimo de L
16 Se agrega $next$ a la lista de hijos de n, ordenado por costo
17 $n \leftarrow next$

3.4.2 split

Como mencionamos, para insertar en el Label Tree, buscaremos en el árbol un lugar para agregar la nueva secuencia de labels. Para ello, contamos con un puntero n para recorrer

el árbol. Cuando queremos insertar una secuencia de labels L con un conjunto no visitable S en el subárbol que tiene a n en su raíz, si no se cumple que $n.S \subseteq S$, hay que modificar $n.S$ y reordenar a sus hijos, para mantener la propiedad de ordenamiento vertical.

La operación *split* (Algoritmo 8) es la encargada de realizar esta modificación, dando como resultado un nodo n que cumple $n.S \subseteq S$. Este cambio se realiza manteniendo la propiedad de ordenamiento vertical entre n y cada uno de sus hijos y manteniendo la singularidad de los *unique* de los hijos de n. La Figura 3.6 muestra un ejemplo de los casos posibles.

Dados un Label Tree T, un puntero a un nodo de T n y un conjunto de vértices S, el procedimiento (Algoritmo 8) comienza por buscar al primer vértice u que no pertenece a S pero si pertenece a $n.S$ (línea 1). Separamos aquellos hijos de n con $unique > u$ y los extraemos del árbol (línea 2).

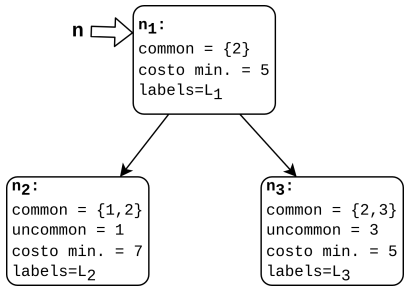
Luego se divide en casos. En el primer caso (Figura 3.6a), n es un nodo con labels (línea 3). Se crea un nuevo nodo nu tal que $nu.S = n.S, nu.unique = u$, nu pasará a tener los labels de n y tendrá como hijos a la secuencia *mayores* (líneas 4 a 6). Esto último se realiza para evitar que n contenga hijos con el mismo *unique*.

En el segundo caso (Fiugra 3.6b), n no contiene labels. Definimos S_{nu} igual a la intersección de todos los labels en *mayores* (línea 8). Si existe algún nodo nu en mayores con $nu.S = S_{nu}$ (línea 9), se actualiza su *unique* con u (línea 10) y se agregan los nodos de *mayores* a su secuencia de hijos, ordenados por costo (11). Luego, debemos recalcular el *unique* de los hijos de nu ya que, al haber movido los de *mayores*, cambió el conjunto S de su padre (línea 12).

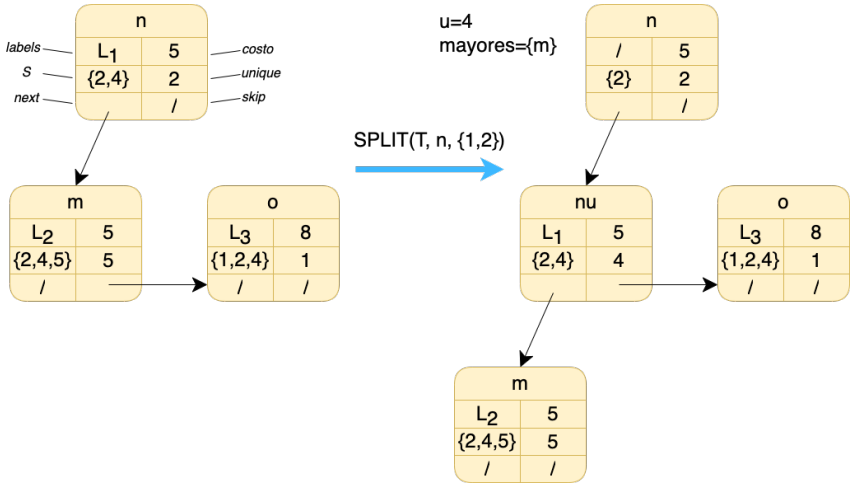
En el tercer caso (Figura 3.6c), n tampoco tiene labels, pero, a diferencia del anterior, no existe ningún nodo en mayores con $S = S_{nu}$. Aquí, nu es un nodo nuevo tal que $nu.S = n.S, nu.unique = u$ y se inserta *mayores* en su secuencia de hijos (líneas 14 y 15).

Finalmente, luego de cada uno de estos casos, debemos asignar el costo de nu, como el mínimo de su subárbol (línea 16), agregarlo a la secuencia de hijos de n (línea 17) y actualizar $n.S$ con su intersección con S (línea 18).

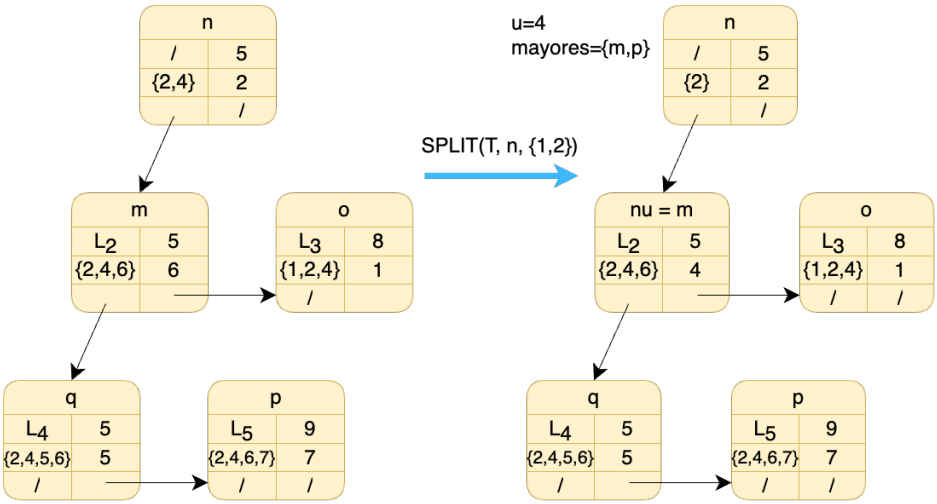
Veamos un ejemplo. Queremos insertar la secuencia de labels L, que no puede visitar a los vértices $S = \{1, 4\}$ y su costo mínimo es 6, en el siguiente Label Tree T:



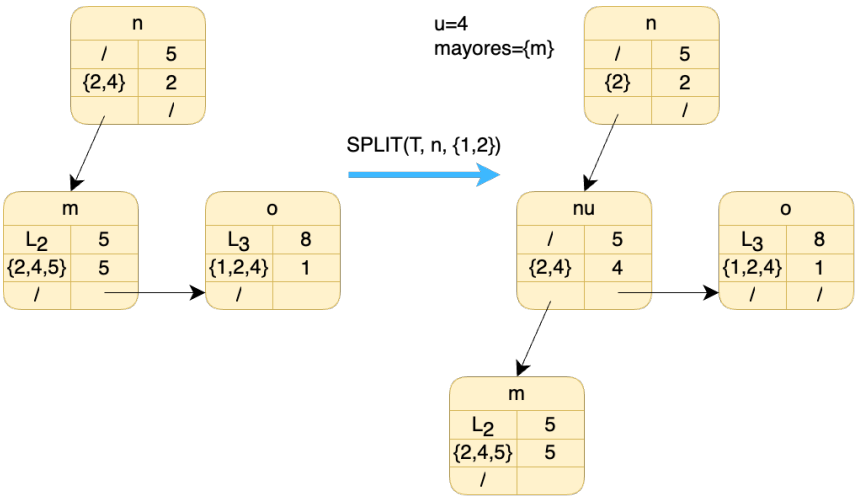
Se recorre T a través del puntero n comenzando por su raíz. $n.costo$ es menor al costo de L por lo que no se necesita modificarlo. Como $\neg n.S \subseteq S$ (línea 4), hay que hacer un *split*. Dentro del *split*, $u = 2$ y la secuencia *mayores* contiene únicamente a n_3. Como $n.labels = L_1 \neq \emptyset$, se entra al primer caso del *split* (línea 3). Se crea el nodo nu con $nu.S = \{2\}$ y $nu.unique = 2$. nu recibe al conjunto L_1 de labels y n_3 pasa a ser hijo de nu. El costo de nu es el mínimo el mínimo de L_1 y el costo de n_3, que es 5. nu se agrega a los hijos de n y $n.S$ es $\{2\} \cap \{1, 4\} = \{\}$. Llamamos n_4 al nuevo nodo nu en el árbol.



(a) Split de un nodo con labels (Algoritmo 8, línea 3)



(b) Split de un nodo sin labels caso 1 (Algoritmo 8, línea 9)



(c) Split de un nodo sin labels caso 2 (Algoritmo 8, línea 14)

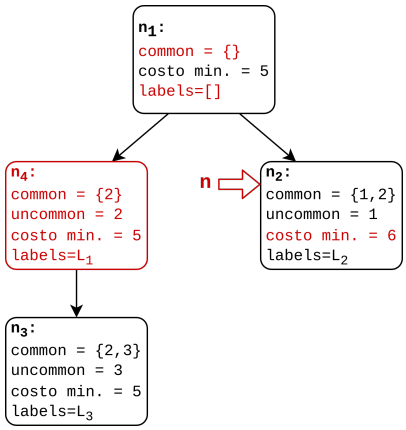
Fig. 3.6: Movimiento del cursor de un Label Tree

Algorithm 8: Split

Input:
T, un Label Tree,
n, un puntero a un nodo de T,
S, un conjunto de vértices,
Result: Modifica a T tal que $n.S \subseteq S$

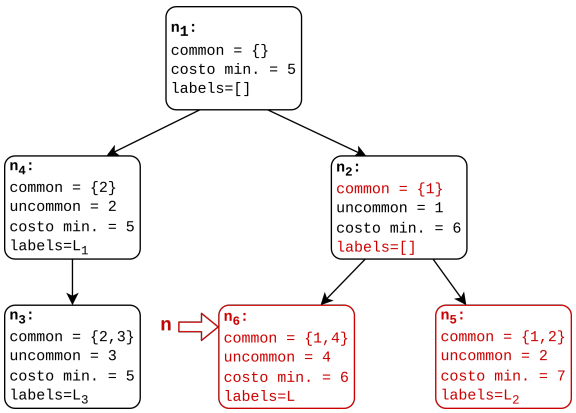
Sea u el primer vértice que no pertenece a S y si a $n.S$
Se extrae la secuencia *mayores* de hijos de n con $unique > u$
if $n.labels \neq \emptyset$ **then**
    Sea nu un nodo nuevo con $nu.S = n.S$ y $nu.unique = u$
    Se mueven los elementos de $n.labels$ a $nu.labels$
    Se inserta *mayores* a la secuencia de hijos de nu
else
    Sea S_{nu} la intersección del S de todos nodos en *mayores*
    if $\exists nu \in mayores \mid nu.S = S_{nu}$ **then**
        $nu.unique \leftarrow u$
        Se realiza un *merge* entre la secuencia *mayores* a la secuencia de hijos de nu
        Se recalculan los *unique* de los hijos de nu
    else
        Sea nu un nodo nuevo con $nu.S = n.S$ y $nu.unique = u$
        Se inserta *mayores* a la secuencia de hijos de nu
$nu.costo \leftarrow$ mínimo costo de su subárbol
Se agrega nu a la lista de hijos de n, en orden por costo
Se actualiza $n.S$ con la intersección entre sí mismo y S

Como $n.S \neq S$, no se entra al `if` de la línea 6. Luego, $u = 1$ y $next = n_2$ ya que su $unique = u$. Se elimina a n_2 de los hijos de n, se actualiza su costo con el costo de $L = 6$ y se vuelve a insertar. Esto se hace ya que el orden puede haber cambiado. Avanzamos n a n_2 para la próxima iteración.



Volviendo a la línea 4, nuevamente se cumple la condición ya que $n.S = \{1, 2\}$ y $S = \{1, 4\}$ y se realiza el *split*. El primer vértice u es 2 y como n_2 no tiene hijos, *mayores* es vacío. Como $n.labels = L_2 \neq \emptyset$, se entra, nuevamente, al primer caso del *split* (línea 3) y se crea un nuevo nodo nu. $nu.S = \{1, 2\}$, $nu.unique = 2$ y $nu.costo = 7$. nu se inserta como n_5 en los hijos de n y $n.S = \{1, 2\} \cap \{1, 4\} = \{1\}$.

No se entra al `if` de la línea 6 y $u = 4$. No existe ningún hijo de n con $S = u$. Por lo tanto $next$ es un nuevo nodo con $next.S = \{1, 4\}$, $next.unique = 4$ y $next.costo = 6$. Se agrega a $next$ como n_6 en los hijos de n_2 y se avanza n a $next$. En la siguiente iteración, $n.S = S$ por lo que se entra al `if` de la línea 6 y se asigna a L a los labels de n para finalizar.



4. EXPERIMENTACIÓN

Para poder evaluar la utilidad del Label Tree, se utilizará una serie experimentos computacionales.

Si bien el objetivo es acelerar el tiempo total que se tarda en resolver una instancia del TDTSPTW, comenzaremos extrayendo únicamente la parte que fue afectada directamente y evaluándola independientemente del resto del algoritmo. De esta manera, será más fácil determinar el rendimiento de la estructura, ya que podremos analizar más instancias en menor tiempo y cualquier diferencia resultante, no se verá atenuada por el tiempo demorado en otros pasos de la resolución, que no fueron modificados.

Recordemos que el algoritmo para resolver el TDTSPTW comienza con resolver la versión relajada del problema con el método NG y luego utiliza esos resultados como cotas que lo ayudan a resolver el problema original. La mejora implementada afecta únicamente la resolución del problema con la relajación NG. Dentro del algoritmo que resuelve esta versión relajada, comenzamos con vecindarios iniciales para cada nodo y se resuelve el problema con esos vecindarios. Luego, se amplían los vecindarios y se repite el proceso hasta que se encuentra una solución que también es solución del problema original o, más comúnmente, se alcanza el límite de tiempo asignado para esta etapa. Este es el proceso DNA. Normalmente en instancias suficientemente difíciles, la ejecución del DNA termina porque se alcanza el límite de tiempo, y luego, se utiliza la ultima instancia de NG resuelta como cota para resolver el problema exacto.

Por eso, primero vamos a evaluar únicamente iteraciones individuales del proceso DNA. Luego, evaluaremos el algoritmo total para analizar el impacto que tiene éste cambio en la resolución del TDTSPTW.

4.1 Implementación

El algoritmo fue implementado en el lenguaje de programación C++ y ejecutado en un único thread de una computadora con un procesador AMD Ryzen 7 5800h 3.2GHz y 24GB de memoria, corriendo el sistema operativo Ubuntu 22.04.

4.2 Experimento 1

En este primer experimento, vamos a comparar el tiempo de ejecución de iteraciones de DNA del algoritmo NG con el algoritmo original y con el algoritmo modificado para utilizar el Label Tree.

Recordemos que en el TDTSPTW-NG, el algoritmo DNA se ejecuta iterativamente, comenzando con un vecindario inicial para cada nodo y luego, en cada iteración, se amplían los vecindarios y se resuelve el problema con los nuevos vecindarios. Lo que significa que el problema comienza siendo muy fácil y se va complicando a medida que se avanza en las iteraciones. Por lo tanto, las iteraciones finales son las más complejas y las primeras son demasiado fáciles de resolver para poder comparar ambos algoritmos.

Para obtener instancias válidas y complejas del problema de labeling, se corrieron instancias del TDTSPTW-NG con el algoritmo original, con tiempo limite de 10 minutos. Para cada una, se extrajo la ultima iteración completada una vez alcanzado este tiempo

límite para ser utilizadas como instancias de labeling. Como estas instancias son de alta complejidad, en todos los casos se alcanzó el tiempo límite, obteniendo instancias de labeling correspondientemente complejas. Las instancias del TDTSPTW provinieron del conjunto propuesto por Vu et al. (2020).

Luego, se utilizaron estas instancias obtenidas para correr el algoritmo labeling original y con Label Tree, y se compararon los tiempos de ejecución de cada uno. Podemos observar los resultados en la Tabla 4.1, incluyendo el tiempo total de ejecución, el tiempo de dominación y el tiempo demorado fuera de la dominación ($total - dominación$). Hacemos esta distinción para poder observar el tiempo que toma insertar en el árbol (que no es parte del tiempo de dominación) ya que es relativamente complejo. Todos los resultados se muestran en segundos.

Como podemos ver, en todos los casos se observa una mejora importante en el tiempo de ejecución total. En el peor caso, el Label Tree es solamente 1,2 veces más rápido, pero en otras instancias, la mejora es de hasta 6 veces. En general, podemos ver que cuanto mayor es el tiempo de ejecución del original, mayor es la mejora del Label Tree. Esto sugiere que la función de tiempo de ejecución en tamaño del input, podría ser de orden menor usando el Label Tree.

Mirando la columna del tiempo demorado en el proceso de dominación, encontramos que toda la mejora en tiempo de ejecución proviene de esta etapa. Esto es esperable, ya que el Label Tree fue diseñado justamente para este proceso. En todos los casos, el Label Tree es más rápido que el algoritmo original. Podemos ver, por ejemplo, la instancia 100_98_A_180_1 donde la dominación demoraba originalmente 114.235 segundos y con Label Tree paso a solo 3.694 segundos.

Otra cosa que podemos observar en estos datos es que el tiempo de ejecución excluyendo el proceso de dominación es ligeramente mayor en el caso del Label Tree. Eso ocurre porque, si bien el Label Tree es más rápido en el proceso de dominación, es más lento en el proceso de inserción. Esto se debe a que el Label Tree tiene que reordenar sus nodos de acuerdo a cambios en los costos, cada vez que se inserta una nueva secuencia de labels. Sin embargo, como el proceso de dominación es mucho más costoso que el de inserción y el incremento en el tiempo de inserción no es tan grande, la diferencia en el tiempo total es positiva.

4.3 Experimento 2

Para el segundo experimento, buscamos poder observar el impacto del Label Tree en instancias completas del TDTSPTW. Como dijimos, el algoritmo que resuelve el problema tiene dos etapas, comienza resolviendo el TDTSPTW-NG y luego resuelve el problema original. La solución obtenida del TDTSPTW-NG, sirve para elegir en que orden se eligen las subsoluciones a extender en el algoritmo del TDTSPTW. La idea es que, cuanto mejor sea este orden, menos subsoluciones se necesitarán extender, por lo que el algoritmo será más rápido. La complejidad del TDTSPTW-NG está dada por el tamaño de los vecindarios, que se extienden en cada paso del DNA.

Para este experimento, la primera idea sería correr instancias enteras del problema dos veces, una con el algoritmo original y otra con el Label Tree, y comparar el tiempo demorado. Si bien es una idea intuitiva, esta opción tiene dos problemas: primero, en cada paso del DNA, ambas ejecuciones pueden agregar vértices distintos en los vecindarios. Esto implica que, al tomar distintas decisiones, la cantidad de iteraciones puede terminar

Instancia	Dominación		Total		Total - Dominación	
	Original	Label Tree	Original	Label Tree	Original	Label Tree
60_70_A_180_3	9.45	1.915	14.327	7.897	4.877	5.982
60_80_A_180_3	11.083	3.407	18.898	12.801	7.815	9.394
60_80_B_180_0	12.555	2.545	19.111	10.516	6.556	7.971
60_90_A_180_4	9.874	1.148	14.441	6.724	4.567	5.576
60_90_B_180_3	14.852	4.639	21.593	12.829	6.741	8.19
60_98_A_180_4	20.136	2.299	28.866	12.647	8.73	10.348
60_98_B_180_4	20.331	2.614	27.213	10.384	6.882	7.77
80_70_A_180_1	12.531	3.659	21.913	15.077	9.382	11.418
80_70_A_180_3	53.218	7.924	67.283	22.826	14.065	14.902
80_70_B_180_1	9.849	1.503	16.673	9.624	6.824	8.121
80_70_B_180_3	68.063	4.035	81.769	18.642	13.706	14.607
80_80_A_180_0	49.214	6.186	60.558	18.668	11.344	12.482
80_80_A_180_2	75.418	6.318	92.472	24.913	17.054	18.595
80_80_A_180_4	48.487	9.311	63.558	26.273	15.071	16.962
80_80_B_180_1	36.923	1.97	44.668	10.468	7.745	8.498
80_80_B_180_3	78.067	9.485	89.459	21.582	11.392	12.097
80_90_A_180_0	99.021	5.153	114.633	21.759	15.612	16.606
80_90_A_180_2	16.801	5.171	27.987	18.691	11.186	13.52
80_90_B_180_0	18.175	8.34	27.865	19.931	9.69	11.591
80_90_B_180_2	44.413	10.107	57.882	25.065	13.469	14.958
80_90_B_180_4	82.895	10.34	101.341	30.478	18.446	20.138
80_98_A_180_1	101.5	4.089	117.124	20.597	15.624	16.508
80_98_A_180_3	11.22	1.912	19.2	11.714	7.98	9.802
80_98_B_180_0	12.719	3.51	22.144	15.027	9.425	11.517
80_98_B_180_2	41.974	9.02	58.154	27.763	16.18	18.743
80_98_B_180_4	53.718	13.586	69.055	30.61	15.337	17.024
100_70_A_180_1	31.419	7.141	42.564	20.332	11.145	13.191
100_70_A_180_1	24.248	6.641	36.225	20.646	11.977	14.005
100_70_A_180_3	29.179	8.483	38.006	17.922	8.827	9.439
100_70_B_180_0	56.832	9.363	79.903	33.688	23.071	24.325
100_70_B_180_2	34.982	17.829	47.848	33.28	12.866	15.451
100_70_B_180_4	24.121	7.051	40.995	26.269	16.874	19.218
100_80_A_180_1	31.441	13.64	39.535	23.272	8.094	9.632
100_80_A_180_3	87.473	14.929	110.678	41.358	23.205	26.429
100_80_B_180_0	58.504	7.634	75.655	27.067	17.151	19.433
100_80_B_180_2	26.605	9.498	36.082	20.438	9.477	10.94
100_80_B_180_4	68.413	9.903	84.723	29.299	16.31	19.396
100_90_A_180_1	69.441	14.717	86.06	34.891	16.619	20.174
100_90_A_180_3	34.036	9.103	45.609	23.135	11.573	14.032
100_90_B_180_0	24.314	8.903	36.786	23.782	12.472	14.879
100_90_B_180_2	91.153	6.015	107.331	24.122	16.178	18.107
100_90_B_180_4	89.38	33.029	103.818	50.33	14.438	17.301
100_98_A_180_1	114.235	3.694	128.476	19.107	14.241	15.413
100_98_A_180_3	11.755	3.486	20.737	15.335	8.982	11.849
100_98_B_180_0	21.899	14.646	34.652	31.151	12.753	16.505
100_98_B_180_2	72.078	10.337	87.553	28.374	15.475	18.037
100_98_B_180_4	71.67	9.367	86.98	27.593	15.31	18.226

Tab. 4.1: Resultados Experimento 1

Instance	DNA Label Tree	DNA Original
100_70_A_180_0	452.72	1088.5
100_70_A_180_1	474.71	1157.17
100_70_A_180_2	429	1146.02
100_70_A_180_3	811.33	1121.22
100_70_A_180_4	523.53	1056.98
100_70_B_180_0	481.25	1064.14
100_70_B_180_1	578.71	1197.85
100_70_B_180_2	906	1198.61
100_70_B_180_3	738.88	1180.2
100_70_B_180_4	769.3	1138.45
100_80_A_180_0	411.86	1014.27
100_80_A_180_1	474.29	1155.04
100_80_A_180_2	406.55	1180.65
100_80_A_180_3	343.72	814.17
100_80_A_180_4	448.11	1166.57
100_80_B_180_0	570.23	1187.88
100_80_B_180_1	606.81	1168.06
100_80_B_180_2	691.63	1163.18
100_80_B_180_3	558.2	1149.1
100_80_B_180_4	471.76	1031.87

Tab. 4.2: Resultados Experimento 2

siendo distinta y, si bien deberíamos poder observar el impacto promedio del uso del Label Tree con suficientes corridas, esto introduce azar en los tiempos de ejecución. Como cada una es muy costosa de ejecutar, queremos eliminar este efecto para poder aislar el efecto del Label Tree y analizarlo con menos ejecuciones. Segundo, como en las instancias difíciles esta etapa continua hasta alcanzar el tiempo límite, ambas versiones del algoritmo tardarían lo mismo, pero el Label Tree terminaría haciendo más iteraciones. Nuevamente, con suficientes ejecuciones, deberíamos poder detectar una diferencia en el tiempo de ejecución de resolver el problema original, habiendo realizado mas iteraciones de DNA gracias al Label Tree. Sin embargo, esto también hace que sea difícil de medir el efecto de la mejora.

Por estos motivos, para este experimento, correremos primero el algoritmo original, guardando los vecindarios en cada paso del DNA, y luego correremos el algoritmo con Label Tree, utilizando los vecindarios generados por la ejecución anterior. Esto nos permite eliminar el componente de azar en la dificultad y aislar el impacto de los cambios implementados.

En la tabla 4.2 vemos el tiempo de ejecución del proceso de DNA para un conjunto de instancias complejas del set de datos provisto por Vu et al. (2020). Podemos observar que el tiempo de ejecución se reduce en casi un 50% gracias al uso del Label Tree. Esta reducción no es tan marcada como en el experimento anterior. Esto sucede ya que en ese experimento se corrieron las ultimas instancias de labelling extraídas de sus respectivos DNA y estas son las que más se benefician por el uso del Label Tree. Al analizar el DNA completo, se reduce el efecto del Label Tree ya que incluimos también las iteraciones más fáciles. Decidimos no incluir el tiempo de ejecución de la etapa que resuelve el problema original, luego del DNA, ya que no veríamos diferencias en su tiempo de ejecución y no es relevante para este análisis.

5. CONCLUSIONES Y TRABAJO A FUTURO

En esta tesis diseñamos una estructura de datos nueva para representar una colección de conjuntos de elementos. Su propósito es poder buscar eficientemente aquellos elementos que sean subconjunto de un conjunto de búsqueda, sin tener que recorrer uno por uno. Esta estructura de datos, llamada Label Tree, es utilizada para resolver el problema de *labeling* en el contexto del Problema del Viajante de comercio con dependencia temporal y ventanas de tiempo (TDTSPTW).

Se implementó el Label Tree como una mejora en el tiempo de ejecución de un algoritmo moderno que resuelve el TDTSPTW y se realizaron experimentos para evaluar su impacto. Los resultados muestran que el Label Tree es una mejora efectiva para el proceso de labeling, reduciendo el tiempo de ejecución significativamente en todos los casos evaluados. Además, se encontró que el Label Tree es más efectivo en instancias con mayor complejidad. Un segundo experimento demostró que esta mejora se refleja en el tiempo de ejecución del algoritmo total para el TDTSPTW. Sin embargo, su impacto es limitado ya que solo pude ser aplicado a una parte del algoritmo total.

Por este motivo, se propone como trabajo a futuro la implementación del Label Tree para otros problemas de esta familia donde pueda ser utilizado durante el proceso de búsqueda de la solución exacta. Por ejemplo, el problema de ruteo de vehículos con ventanas de tiempo (VRPTW) o el problema de ruteo de vehículos con ventanas de tiempo y dependencia temporal (TDVRPTW), entre otras versiones del VRP, que es una generalización del TSP en el que se cuenta con más de un vehículo para visitar todos los destinos. En estos problemas, el proceso de búsqueda de la solución exacta puede tomar ventaja del Label Tree.

BIBLIOGRAPHY

Norbert Ascheuer, Matteo Fischetti, y Martin Grötschel. A polyhedral study of the asymmetric traveling salesman problem with time windows. *Networks*, 36(2):69–79, 2000. doi:10.1002/1097-0037(200009)36:2<69::AID-NET1>3.0.CO;2-Q.

Norbert Ascheuer, Matteo Fischetti, y Martin Grötschel. Solving the asymmetric travelling salesman problem with time windows by branch-and-cut. *Mathematical Programming*, 90:475–506, 01 2001. doi:10.1007/PL00011432.

Roberto Baldacci, Aristide Mingozzi, y Roberto Roberti. New route relaxation and pricing strategies for the vehicle routing problem. *Operations Research*, 59(5):1269–1283, 2011. doi:10.1287/opre.1110.0975.

Roberto Baldacci, Aristide Mingozzi, y Roberto Roberti. New state-space relaxations for solving the traveling salesman problem with time windows. *INFORMS J. Comput.*, 24: 356–371, 2012. doi:10.1287/ijoc.1110.0456.

Nicos Christofides, A. Mingozzi, y P. Toth. State-space relaxation procedures for the computation of bounds to routing problems. *Networks*, 11(2):145–164, 1981. doi:10.1002/net.3230110207.

Romain Fontaine, Jilles Dibangoye, y Christine Solnon. Exact and anytime approach for solving the time dependent traveling salesman problem with time windows. *European Journal of Operational Research*, 311(3):833–844, 2023. doi:10.1016/j.ejor.2023.06.001.

Martin Grötschel, Michael Jünger, y Gerhard Reinelt. Optimal control of plotting and drilling machines: A case study. *Zeitschrift für Operations Research*, 35:61–84, 1991. doi:10.1007/BF01415960.

Arthur V. Hill y William C. Benton. Modelling intra-city time-dependent travel speeds for vehicle scheduling problems. *Journal of the Operational Research Society*, 43:343–351, 1992. doi:10.1057/jors.1992.49.

Soumia Ichoua, Michel Gendreau, y Jean-Yves Potvin. Vehicle dispatching with time-dependent travel times. *European Journal of Operational Research*, 144(2):379–396, 2003. doi:10.1016/S0377-2217(02)00147-9.

M Joerss, J Schröder, F Neuhaus, C Klink, y F Mann. Parcel delivery: The future of last mile. Technical report, McKinsey & Company, 2016.

Gonzalo Lera-Romero y Juan José Miranda-Bront. A branch and cut algorithm for the time-dependent profitable tour problem with resource constraints. *European Journal of Operational Research*, 289(3):879–896, 2021. doi:10.1016/j.ejor.2019.07.014.

Gonzalo Lera-Romero, Juan José Miranda Bront, y Francisco J. Soulignac. Dynamic programming for the time-dependent traveling salesman problem with time windows. *INFORMS Journal on Computing*, 34(6):3292–3308, 2022. doi:10.1287/ijoc.2022.1236.

Chryssi Malandraki y Mark S. Daskin. Time dependent vehicle routing problems: Formulations, properties and heuristic algorithms. *Transportation Science*, 26(3):185–200, August 1992. doi:10.1287/trsc.26.3.185.

Aristide Mingozzi, Lucio Bianco, y Salvatore Ricciardelli. Dynamic programming strategies for the traveling salesman problem with time window and precedence constraints. *Operations Research*, 45(3):365–377, 1997. doi:10.1287/opre.45.3.365.

Christian Tilk y Stefan Irnich. Dynamic programming for the minimum tour duration problem. *Transportation Science*, 51(2):549–565, 2017. doi:10.1287/trsc.2015.0626.

Duc Minh Vu, Mike Hewitt, Natashia Boland, y Martin Savelsbergh. Dynamic discretization discovery for solving the time-dependent traveling salesman problem with time windows. *Transportation Science*, 54(3):703–720, 2020. doi:10.1287/trsc.2019.0911.