Trabajo Fin de Grado
Grado en Ingeniería de las Tecnologías Industriales

Una búsqueda tabú reactiva para la optimización del acarreo terrestre con vehículos heterogéneos

Autor: Jose Mariano Casallo Clapés

Tutor: Alejandro Escudero Santana

Dpto. Organización Industrial y Gestión de Empresas II
Escuela Técnica Superior de Ingeniería

Sevilla, 2020

Trabajo Fin de Grado
Grado en Ingenieria de las Tecnologías Industriales

Una búsqueda tabú reactiva para la optimización del acarreo terrestre con vehículos heterogéneos

Autor:
Jose Mariano Casallo Clapés

Tutor:
Alejandro Escudero Santana
Profesor Titular de Universidad

Dpto. de Organización Industrial y Gestión de Empresas II
Escuela Técnica Superior de Ingeniería
Universidad de Sevilla
Sevilla, 2020

Trabajo Fin de Grado: Una búsqueda tabú reactiva para la optimización del acarreo terrestre con vehículos heterogéneos

Autor: Jose Mariano Casallo Clapés

Tutor: Alejandro Escudero Santana

El tribunal nombrado para juzgar el Proyecto arriba indicado, compuesto por los siguientes miembros:

Presidente:

Secretario:

Acuerdan otorgarle la calificación de:

Sevilla, 2020

El Secretario del Tribunal

A Antonio

Agradecimientos

A la paciencia de mi familia,
Al optimismo de mis amigos.

José Mariano Casallo Clapés
Sevilla, 2020

Resumen

Este proyecto se basa en la aplicación de la búsqueda tabú reactiva para la resolución de un problema de acarreo diario con vehículos de capacidades distintas (MSDDP).

Se trata de la optimización del problema de ruteo de vehículos basado en operaciones de recogida y entrega, en una zona próxima a un puerto marítimo, en la cual se pueden llevar a cabo tanto tareas de importación como de exportación.

El proyecto tiene en cuenta, además, las ventanas temporales de cada uno de los clientes, y el tamaño de la unidad de carga, así como las mismas operaciones con contenedores vacíos.

Abstract

This project is based on the application of the reactive taboo search for the resolution of a daily drayage problem with vehicles of different capacities (MSDDP).

The aim is to optimize the vehicle routing problem based on pickup and delivery operations, in an area next to a seaport, in which both import and export tasks can be carried out.

The project also takes into account the time windows of each of the clients, and the size of the loading unit, as well as the same operations with empty containers.

Índice

ÍNDICE DE TABLAS

ÍNDICE DE FIGURAS

1 INTRODUCCIÓN

Actualmente, la distribución de mercancías se considera uno de los campos de la logística más interesantes en cuanto a optimización se refiere. Una parte importante de los costes de una empresa se debe a estos costes logísticos, y en un entorno cada vez más competitivo, la búsqueda de la mejora del servicio al cliente genera un claro aumento de dichos costes de transporte.

Además, debido al desequilibrio actual en el sistema de transporte de la UE, aparecen ciertos efectos externos de carácter negativo, a saber: costes de congestión, contaminación tanto atmosférica como acústica o accidentes.

Por todo ello surge este deseo de mejorar la eficiencia en el transporte, persiguiendo que sea más adecuado en costes, más seguro, y logrando una disminución de su impacto ambiental, así como una mayor aceptación social. Esta búsqueda del transporte sostenible ha llevado a un nuevo concepto de política de distribución de mercancías: el transporte intermodal.

Este concepto se trata de una integración del sistema de transporte, y se basa en recurrir en cada tramo de la cadena al medio de transporte más adecuado. Decide, para cada etapa, entre el medio marítimo, aéreo, el ferrocarril y el transporte por carretera, optimizando así tanto los diferentes tramos como la cadena global.

Ante el objetivo de mejorar la flexibilidad, y conseguir un servicio de entrega puerta a puerta, destaca sobre los demás un medio: el transporte por carretera. Teniendo en cuenta que la unidad de carga es el contenedor, tanto el tramo inicial de una tarea como el final, va a ser llevado a cabo por camiones, y a este movimiento se le denomina acarreo terrestre o drayage.

El acarreo terrestre supone una parte imprescindible en la cadena intermodal, ya que aporta la flexibilidad necesaria para cumplir con las exigencias relacionadas con el servicio al cliente que aparecen en la actualidad.

Esta parte de la cadena sigue suponiendo un porcentaje importante de los costes logísticos, y es en este proyecto donde se propondrá una optimización para reducir dichos costes, y por tanto conseguir una mejora en el rendimiento de la actividad.

El acarreo terrestre se define como un problema de optimización NP-Hard en cuanto a su complejidad. Todavía son muchas las empresas que resuelven dicho problema manualmente, y es por tanto muy interesante la búsqueda de su resolución mediante herramientas informáticas.

Técnicas heurísticas y meta-heurísticas pueden proporcionar una solución eficiente para el problema del drayage, y este documento mostrará algunas de las más interesantes.

Este proyecto ha optado por la Búsqueda Tabú o Tabu Search, la cual será analizada en profundidad. Mediante dicho método se buscará por tanto conseguir una solución que suponga una mejora en cuanto a eficiencia y ahorro de costes, además de conseguir una reducción en los problemas presentes del impacto ambiental y en el tiempo requerido para dicha actividad.

1.1. Objetivos

Este proyecto va a tratar de ofrecer la solución óptima al problema del acarreo terrestre, buscando minimizar los tiempos de transporte mediante la elección correcta de las rutas formadas por las diferentes localizaciones, y minimizar por tanto los costes logísticos que van ligados con dichas rutas. Este ahorro en costes será debido, entre otros, a la reducción de combustible necesario para llevar a cabo la tarea, por lo que también se perseguirá una minimización de la contaminación que el petróleo y la emisión de gases de combustión puede ocasionar durante el transcurso de la actividad.

Para la resolución de dicho problema, se presentará una descripción detallada del drayage, así como del método que va a ser empleado para afrontar dicho problema. Además, se incluirá el caso práctico en cuestión y las simplificaciones pertinentes para poder llevar a cabo el proyecto.

El objetivo es, por tanto, mostrar cómo, mediante herramientas informáticas, se implementa una heurística

para resolver de manera óptima el problema del acarreo terrestre, consiguiendo aumentar la eficiencia y flexibilidad del transporte por carretera reduciendo además los costes de la tarea.

1.2. Estructura del trabajo

Este documento se estructura en 8 capítulos:

- En el capítulo 1 se muestra una introducción, donde se expone la motivación del problema y los objetivos.
- En el capítulo 2 se analiza el transporte intermodal y sus características.
- En el capítulo 3 se pasa a profundizar en el acarreo terrestre: características, revisión bibliográfica, variantes del problema y su modelado.
- El capítulo 4 recoge los diferentes tipos de métodos de resolución, diferenciando entre exactos, heurísticas y metaheurísticas, así como describiendo algunos de los algoritmos más empleados de cada tipo.
- El capítulo 5 se centra en la búsqueda tabú, su definición, características, y una introducción a la búsqueda tabú reactiva, la cual se emplea en este proyecto.
- En el capítulo 6, se aplica el método de la Búsqueda Tabú al problema del Acarreo Terrestre. Se detallan las variables, parámetros necesarios, así como la estructura de resolución y las diferentes funciones empleadas en Python.
- El capítulo 7 se exponen los resultados obtenidos en las simulaciones.
- En el capítulo 8 se recogen las conclusiones que surgen del desarrollo del proyecto.

2 EL TRANSPORTE INTERMODAL

2.1. Concepto de logística y modos de transporte

Antes de exponer el significado de transporte intermodal, y de desarrollar dicho concepto, resulta de interés llevar a cabo una introducción sobre la logística para comprender mejor el entorno que va a ser analizado.

Según la CLM (Council of Logistic Management), la logística es la parte de la cadena de suministros que planifica, implementa y controla el almacenamiento y flujo eficiente y efectivo de bienes, servicios e información relacionada desde el punto de origen al punto de consumo con el objetivo de cubrir las necesidades de los clientes. (Wu, 2007)

Es en el flujo de bienes entre diferentes localizaciones, en el que este documento se centra. Según su modo, el transporte se divide en dos grandes conceptos: transporte unimodal y transporte intermodal.

Atendiendo a la definición de modo de transporte como el conjunto de técnicas que utilizan vías de transporte de la misma naturaleza, se entiende como unimodal, aquel que solamente emplea un modo e intermodal por tanto aquel en el que se recurre a varios.

Los diferentes modos de transporte a tener en cuenta son:

- Marítimo: Emplea como medio de transporte el mar, y es considerado el óptimo para la distribución de mercancías en grandes volúmenes, entre dos localizaciones alejadas geográficamente.
- Ferrocarril: Emplea la red ferroviaria del territorio en cuestión, para el traslado de la mercancía. Está por tanto muy condicionado por la infraestructura, lo cual conduce a una falta de flexibilidad que desemboca en una preferencia por el transporte marítimo para el trayecto principal.
- Aéreo: Es el más moderno y como ventajas aporta la rapidez, muy útil en envíos urgentes. Supone, por otra parte, muy restrictivo a la hora de contemplar características como el volumen o el peso.
- Carretera: La unidad de carga es transportada por camiones, desde el punto de recogida al punto de destino. Es el modo más flexible debido a la amplia red de infraestructura, y es por eso que se recurre a este modo tanto para el trayecto inicial de la cadena como para el final.

Los siguientes gráficos muestran el porcentaje de empleo de cada uno de los modos de transporte, tanto en la Unión Europea (izqda.) como en España (dcha.). Se observa claramente cómo el transporte por carretera es con diferencia el más empleado.

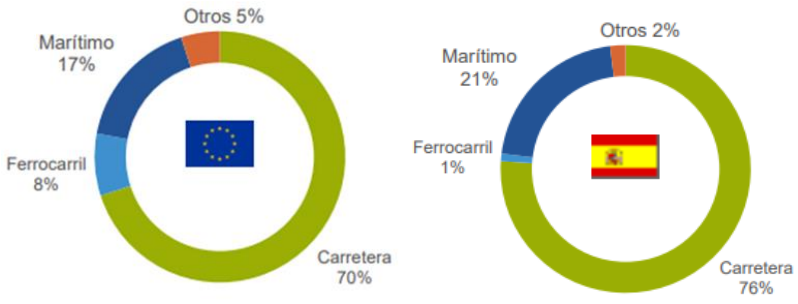


Ilustración 1. Proporción de los diferentes modos de transporte empleados en UE y España respectivamente (Fuente: https://contenidos.ceoe.es)

2.2. Definición

Una vez conocidos los diferentes modos, es posible comprender mejor la definición de transporte intermodal:

- "*Movimiento de bienes en una misma unidad de carga o vehículo, que usa sucesivamente varios modos de transporte sin manejo de los bienes en los cambios de modos".* (Van Duin & Van Ham, 1998)
- "*Movimiento de camiones o contenedores sobre vagones de tren, con transporte por carretera en los tramos finales"* (Nozick & Morlok, 1997)

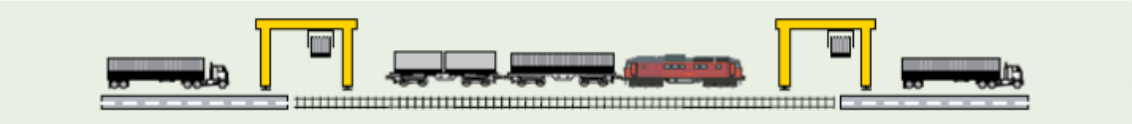


Ilustración 2.Esquema transporte intermodal: Ferrocarril - Carretera

Estas dos definiciones suponen la introducción perfecta para poder comenzar a profundizar en el transporte intermodal, así como para hablar de ciertos conceptos esenciales para la resolución del problema planteado.

Uno de estos conceptos fundamentales es la unidad de carga. En lo que atañe a este proyecto, se toma como dicha unidad el contenedor, el cual se analizará más adelante.

Otro concepto básico para comprender el problema será la terminal. La terminal será el punto donde se lleven a cabo los trasbordos de una unidad de carga de un modo de transporte al siguiente.

También es interesante introducir el concepto de depósito, o localización donde se encontrarán los contenedores, a la espera de ser recogidos por el modo de transporte pertinente. Los depósitos se suelen localizar en emplazamientos cercanos a las terminales intermodales, para facilitar el transporte entre ambos y minimizar las distancias recorridas por las empresas transportistas.

2.3. Ventajas e inconvenientes

El transporte intermodal presenta ventajas en varios aspectos. Algunos de los más importantes son:

- Reducción de costes: En cada una de las etapas, se decide entre el modo de transporte más económico, atendiendo a las características de cada operación.
- Aumento de la seguridad: Como se ha comentado anteriormente, la unidad de carga es el contenedor. Éste se cierra herméticamente, lo cual influye de manera positiva en la reducción de robos de mercancía. Al disminuir el riesgo de hurto, se reduce sustancialmente la necesidad de inspecciones y revisiones, lo cual resulta de gran interés. Este cierre hermético de la unidad de carga favorece además la conservación de la mercancía frente a situaciones climáticas desfavorables.
- Optimización del tiempo: Se realiza un análisis para decidir qué modo es más favorable teniendo en cuenta el tráfico y las condiciones que influyan en el tiempo del trayecto.
- Reducción de documentación: La documentación necesaria para llevar a cabo el transporte abarca toda la combinación de modos. Es decir, se reducen los trámites al no tener que documentar los diferentes trayectos por separado.
- Reducción de tiempos en cuanto a carga y descarga: La estandarización de la unidad de carga permite un ahorro de tiempo en el manejo del contenedor, al ser siempre las mismas dimensiones. Se reducen los tiempos de manipulación.

- Medio ambiente: El transporte por carretera es el más contaminante. Al combinarse con otros modos de transporte, se consigue una disminución de emisiones de CO2, lo cual supone también una ventaja para la opinión social.
- Reducción de plazos: Los tiempos de transporte también se analizan al decidir el medio, buscando aquel que suponga un ahorro de tiempo en cuanto a plazo de entrega.

Sin embargo, el transporte intermodal también cuenta con características desfavorables para la distribución de mercancías. Alguna de las más destacables son:

- Costes de manejo: Aunque el contenedor supone una ventaja debido a su estandarización, es cierto que requiere un equipo para su manipulación que supone un coste para la empresa.
- Aumento de riesgo: A mayor número de modos de transporte usado, mayor probabilidad de fallo en las transiciones de uno a otro. Estos fallos pueden consistir en daños durante la manipulación, imprevistos como colas, o ventanas temporales.
- Velocidad: Hay una gran diferencia en cuanto a velocidad si se analizan los diferentes medios de transporte. El aéreo o el ferrocarril son más rápidos que el marítimo y el transporte por carretera. Al combinarlos, a pesar de conseguir las ventajas antes nombradas, se percibe una reducción en la velocidad.

Esta selección de ventajas e inconvenientes permiten concluir que la rentabilidad del transporte intermodal se persigue mediante el análisis de cada caso particular. Este análisis consiste en observar los modos de transporte disponibles, atendiendo a la infraestructura necesaria y al tiempo de entrega previsto. Una vez obtenidos los datos, se pasa a tomar una decisión con la combinación de modos de transporte que mejor se adapte a las necesidades de la empresa.

3 EL ACARREO TERRESTRE

El acarreo terrestre o *drayage* se define como el conjunto de movimientos pertenecientes al transporte intermodal cuyo modo de transporte empleado es la carretera y comprende las operaciones necesarias para enlazar las terminales intermodales con los orígenes y destinos de las mercancías. (Escudero Santana, 2013)

Por tanto, el acarreo terrestre abarca un modo de transporte de corta distancia, enfocado en la recogida y entrega de mercancías entre dos puntos, que resultaran ser por un lado la terminal (puerto marítimo o punto fronterizo, entre otros), y por otro lado los diferentes clientes. Se trata además de una opción que por regla general se realiza durante el transcurso de un turno de trabajo.

Cabe añadir que, a pesar de la flexibilidad que esta parte de la cadena aporta al trayecto global, el *drayage* es muy interesante a la hora de analizar los costes, ya que se le atribuye un alto porcentaje de dichos costes asociados al transporte.

3.1. Operaciones y características del acarreo terrestre

Para comprender mejor las diferentes situaciones que pueden darse durante el acarreo, resulta de interés definir los diferentes estados en los que pueden encontrarse los camiones.

Teniendo en cuenta la notación anglosajona, se definen a continuación los tres estados fundamentales:

a) Load: El camión se encuentra cargado con un contenedor lleno.
b) Dead-Head: El camión se encuentra cargado con un contenedor vacío.
c) Bobtail: El camión se encuentra descargado.

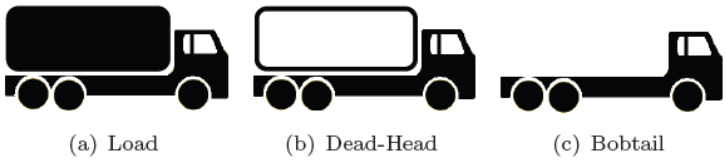


Ilustración 3. Estados diferentes del camión (Escudero 2013)

3.1.1 Tareas de acarreo

Una vez realizada la introducción de los tres estados en los que se puede encontrar un camión, se pasa a analizar las diferentes tareas que se realizan en el acarreo terrestre.

Las tareas de acarreo constan de dos operaciones fundamentales: Recogida de la unidad de carga en el cliente o en la terminal (Pick up) y entrega de dicha unidad de carga (Delivery).

Las diferentes operaciones que pueden llevarse a cabo a la hora de realizar una tarea se muestran en el diagrama siguiente.

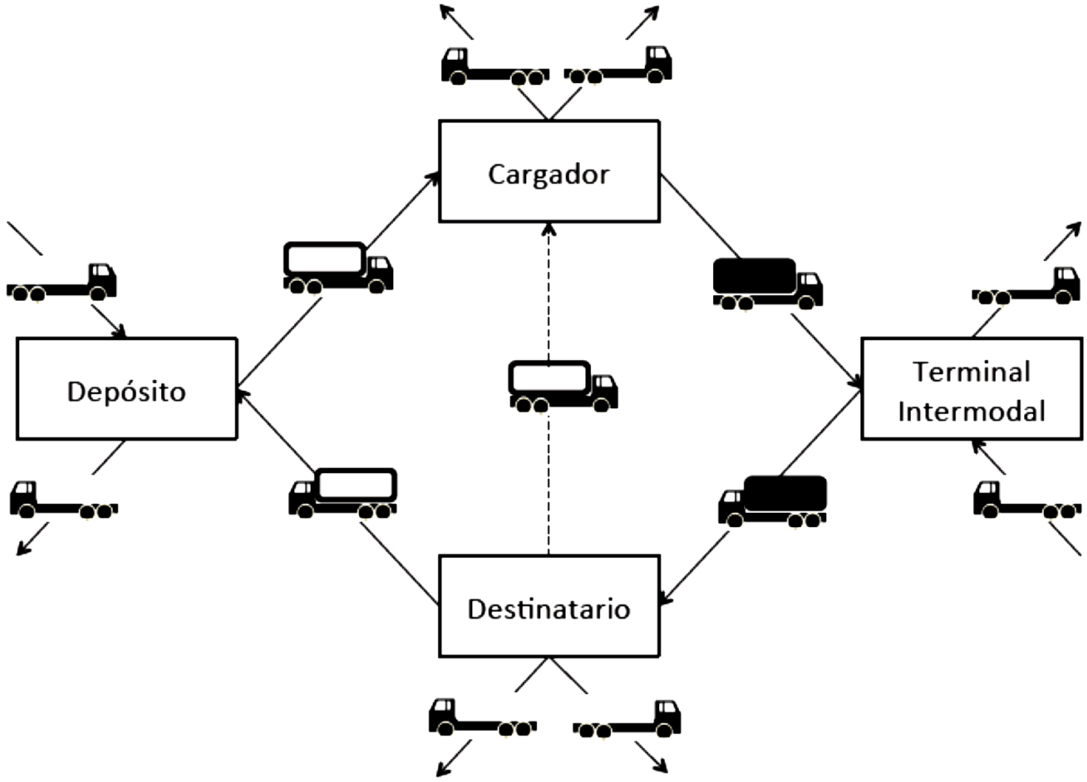


Ilustración 4. Diferentes movimientos de las operaciones de acarreo terrestre (Escudero Santana, 2013)

Como se observa en el diagrama anterior, una parte importante de las tareas de acarreo son llevadas a cabo con contenedores vacíos. En un principio, dichas operaciones deben ser reducidas al máximo, pues suponen desplazamientos que únicamente añaden coste a la tarea. Uno de las soluciones para conseguir dicha reducción es la que aparece en línea discontinua, y se denomina bypass.

La operación bypass permite reducir los kilómetros recorridos durante la tarea de los contenedores vacíos, y consiste en el envío de un destinatario que cuenta con un contenedor vacío, a un cargador que en ese momento lo requiere, en vez de tener que pasar dicha unidad de carga obligatoriamente por el depósito.

3.1.2. Tareas de importación y exportación

Las tareas del acarreo terrestre se dividen en dos tipos:

- Tarea de importación. Consiste en el desplazamiento de la unidad de carga desde la terminal portuaria al destinatario.
- Tarea de exportación. Desplazamiento de la unidad de carga del cliente o cargador a la terminal.

Así como las tareas con contenedores llenos están perfectamente definidas en su origen y destino, es decir, las operaciones de Pick up y Delivery cuentan con localizaciones conocidas, las tareas con contenedores vacíos van a permitir una holgura en cuanto a su comienzo o fin. No tiene por qué conocerse el lugar de procedencia del contenedor vacío si es un cliente quien lo requiere, y no tiene por qué conocerse el lugar de destino de aquel cargador que libera un contenedor. Por ello, mientras que las tareas con contenedores llenos serán denominadas tareas bien definidas, las tareas con contenedores vacíos pasarán a denominarse tareas flexibles.

Así pues, resulta de gran interés encontrar una manera de optimizar las rutas alternativas que estos contenedores vacíos pueden hacer antes de fijar por defecto su retorno a la terminal. Para ilustrarlo con un ejemplo:

Supóngase que un cliente desea realizar una tarea de exportación. Para ello necesitará contar con un contenedor vacío para poder cargar en él su mercancía, antes de realizar el envío. Dicho cliente tendería a solicitar al depósito un contenedor vacío. Sin embargo, se sabe que a su vez existe un segundo cliente que ha realizado una tarea de importación y ha descargado ya su contenedor: cuenta con un contenedor vacío que en un principio iba a liberar y devolver al depósito. Dichos clientes se encuentran mutuamente a una distancia menor que el depósito al que ambos desean recurrir. Es aquí donde la opción del bypass cobra utilidad: El cliente que cuenta con el contenedor vacío podrá enviárselo al que lo requiere para su tarea de exportación, ahorrando ambos costes y optimizándose de esta manera la ruta de dos tareas diferentes.

3.1.3. Operaciones complejas

En apartados anteriores se han definido tareas sencillas en las que un cargador demanda un contenedor para llevar a cabo su exportación, o bien un cliente importa un contenedor, llevándose a cabo su recogida en el depósito.

Sin embargo, el ejemplo anterior refleja una situación en la que es necesario que se lleve a cabo una combinación de operaciones. Este tipo de situaciones se definen como operaciones complejas. Intervienen varias órdenes que deben estar encadenadas, y para su enlace es necesario prestar atención al estado de los diferentes vehículos.

Ya se ha indicado en la introducción de este apartado 3.1 el esquema de los tres estados en los que un camión puede encontrarse: Load, Dead-Head y Bobtail. Así pues, para poder combinar varias operaciones será necesario que el estado al finalizar una operación sea el mismo que el estado previo a comenzar la operación siguiente.

Ante el problema de las operaciones complejas se presentan dos maneras diferentes de afrontar la solución:

- *Stay-with* (SW): Durante la operación de carga o descarga, el conductor permanece junto al contenedor.
- *Drop&pick* (D&P): El conductor abandona la localización tras entregar el contenedor al cliente, y continúa realizando la siguiente operación. Una vez el cliente ha finalizado la operación, dicho contenedor puede ser recogido por el mismo conductor o por otro diferente.

Queda así definido el problema de acarreo con operaciones complejas, entendiéndose como tal la combinación de acciones de recogida y entrega tanto de contenedores llenos como de contenedores vacíos en caso de que un cliente los requiera.

Sólo queda indicar la notación que se va a emplear para trabajar con el problema del acarreo terrestre, teniendo en cuenta los diferentes estados del vehículo, las órdenes que estos van a percibir y el modo de afrontar el problema.

- En mayúscula se indica el tipo de operación: **P** (Pick up) y **D** (Delivery)
- El superíndice señala el estado de la unidad de carga: **l** (Load) y **e** (Empty)
- El subíndice indica la resolución del problema: **SW** (Stay-with) y **D&P** (Drop&Pick)

Tabla 1. Conjunto de operaciones y estados en el acarreo terrestre. Fuente: Escudero Santana (2013)

Nombre operación	Descripción	Estado inicial	Estado final
$\boldsymbol{P^l}$	Recogida de contenedor lleno	Bobtail	Load
$\boldsymbol{D^l}$	Entrega de contenedor lleno	Load	Bobtail
$\boldsymbol{P^e}$	Recogida contenedor vacío	Bobtail	Dead-Head

D^e	Entrega contenedor vacío	Dead-Head	Bobtail
$D^e\&P^l_{D\&P}$	Entrega contenedor vacío y recogida de contenedor lleno	Dead-Head	Load
$D^l\&P^e_{D\&P}$	Entrega contenedor lleno y recogida contenedor vacío	Load	Dead-Head
$D^l\&P^l_{D\&P}$	Entrega contenedor lleno y recogida contenedor lleno	Load	Load
$D^e\&P^l_{SW}$	Entrega contenedor vacío y recogida contenedor lleno	Dead-Head	Load
$D^l\&P^e_{SW}$	Entrega contenedor lleno y recogida contenedor vacío	Load	Dead-Head
$D^l\&P^l_{SW}$	Entrega contenedor lleno y recogida contenedor lleno	Load	Load

3.1.4. Ventanas temporales

Como ya se ha comentado anteriormente, en el problema del acarreo terrestre participan dos entidades: por una parte, intervienen todas aquellas empresas que requieren una tarea de importación o exportación. Por la otra, la terminal.

Ambas introducen una variable fundamental a tener en cuenta: las ventanas temporales.

Se entiende por ventana temporal aquel intervalo de tiempo en el cual está permitida la actividad que se desea llevar a cabo. En este caso en concreto, estará referida al horario en el cual es posible realizar las operaciones de recogida y entrega de los contenedores.

Por tanto, es en este intervalo temporal, independiente en los diferentes puntos a analizar, donde las operaciones de Pick up y Delivery son posibles, y aparece de esta manera en cada uno de los clientes una restricción que habrá que tener en cuenta a la hora de recalcular las rutas. La manera de tener en cuenta dichas restricciones temporales no será otra que penalizar aquellas operaciones de acarreo que se realicen fuera de dichos intervalos de tiempo.

Así pues, en el caso de las tareas de exportación, existirá una hora programada para la salida del barco (o del tren), y no será de utilidad un programa en el que el camión llega a la terminal más tarde de dicho horario. Por otra parte, en las tareas de importación estará fijada la hora a la que el barco ha llegado a puerto y por tanto, la hora a la que está disponible la mercancía. No tendría sentido así, la llegada del camión antes de dicha hora.

3.1.5. Tamaños de contenedor.

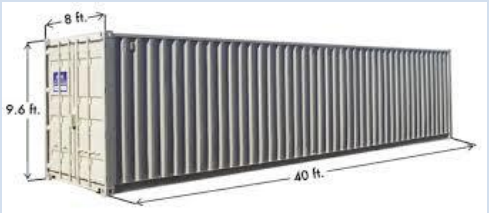
La unidad de carga con la que se trabaja en el problema del acarreo terrestre es, como se ha comentado en los puntos anteriores, el contenedor. Actualmente, los contenedores siguen unas medidas estandarizadas que permiten facilitar la planificación del espacio en los vehículos de los diferentes modos de transporte, y evitan imprevistos a la hora de llevar a cabo envíos entre países diferentes.

Los dos tamaños más empleados son de 20 pies (20'), conocido como TEU (Twenty Feet Equivalent Unit) y 40 pies (40'), conocido como FEU (Forty Feet Equivalent Unit).

El tamaño del contenedor, junto con las ventanas temporales, es un factor importante a tener en cuenta a la hora de plantear la resolución del problema propuesto en este trabajo, pues determinará la distribución y la planificación de las rutas de transporte, al variar la disponibilidad de los camiones.

Se detallan a continuación algunos de los diferentes tipos de contenedor que se emplean en el transporte intermodal.

Tabla 2. Tipos de contenedor según sus diferentes funciones

DRY VAN **(O cerrados- también conocidos como "Box")**	
HIGH CUBE **(Contenedores estándar comúnmente de 40 pies)**	
REEFER **(Frigorífico)**	
OPEN TOP **(Techo abierto)**	
FLAT RACK **(Plataforma)**	
ISO TANK **(Contenedor cisterna)**	

3.1.6. Revisión bibliográfica

Son muchos los autores que a lo largo de los últimos años se han dedicado a la investigación en cuanto al transporte intermodal y, en concreto, al acarreo terrestre.

El *drayage*, como ya se ha mencionado anteriormente, supone una pieza fundamental en el transporte intermodal, y se detalla a continuación una selección de algunos de los autores más relevantes.

Como introducción, resulta de gran interés el trabajo de investigación de Macharis y Bontekoning (2004), en el cuál se lleva a cabo un análisis de los estudios que se han realizado en los años anteriores, atendiendo a su horizonte de planificación, destacando dos enfoques: Estratégico-táctico y operativo.

Se muestra a continuación un análisis bibliográfico relacionado, en primer lugar, con el enfoque estratégico-táctico, y en segundo lugar con el enfoque operacional.

- **Enfoque estratégico-táctico.**

  El enfoque estratégico-táctico se centra en el análisis de costes y calidad de servicio de las diferentes medidas que pueden adoptarse en el transporte intermodal, y en concreto, indagan en el beneficio de dichas medidas que se genera en todo el conjunto.

  Desde el punto de vista cronológico, destaca en primer lugar Fowkes & Nash. (1991). En dicho estudio se enfoca el análisis de costes en la relación de los diferentes puntos logísticos con la terminal, considerando la localización entra ambos como punto estratégico clave para la optimización del problema.

  Más adelante, Spasovic y Morlok (1993) se centran, en cambio, en los diversos precios que pueden atribuirse a las distintas tareas de acarreo, así como un análisis previo relacionado con la planificación de recursos para optimizar el problema del *drayage* en cuanto a costes y calidad de servicio. Para ello desarrollan un modelo lineal entero que permite llevar a porcentajes el ahorro de costes de las diversas tareas (Morlok & Spasovic, 1994).

  Nierat (1997) analiza la relación entre la planificación de los recursos antes comentada y el aumento de la zona de influencia terrestre del puerto marítimo, *hinterland,* teniendo en cuenta el factor de ocupación de los vehículos o el número de operaciones de acarreo realizadas por el camión.

  Taylor (2002), propone dos alternativas diferentes en cuanto a la elección de la terminal de trasbordo, y expone las conclusiones en cuanto a la optimización de costes relacionados con las distancias recorridas.

  Cheung (2008) analizan las diferentes políticas relacionadas con el acarreo empleadas por el gobierno, diferenciando entre dos:

  - 4up-4down: conductor, cabeza tractora, chasis y contenedor forman una unidad, indivisible, encargada de llevar a cabo tanto el viaje de ida como el de vuelta.
  - 2up-2down: También llamada política de total libertad, en la que solamente el conductor y cabeza tractora forman parte de dicho conjunto.

- **Enfoque operativo**

  Este segundo enfoque se basa en buscar la resolución al problema del acarreo terrestre diario, *daily drayage problem* (DDP).

  Según Escudero-Santana (2013), su objetivo se define como minimizar los costes de todas las operaciones de acarreo que son llevadas a cabo por una empresa, o consorcio de empresas, en una región determinada a lo largo de una jornada laboral, teniendo en cuenta unos recursos limitados y una serie de restricciones a cumplir.

  Cuando una de las restricciones es la referida a las ventanas temporales, en la que el problema debe resolverse teniendo en cuenta los intervalos de tiempo en los que es posible realizar tanto la entrega como la recogida, se hablará de *daily drayage problem with time windows (DDPTW).*

  Otra restricción a estudiar será el tamaño del contenedor o unidad de carga. Es interesante afrontar el problema teniendo en cuenta este factor, pues tanto las empresas como los cargadores pueden recibir, o realizar encargos tanto de contenedores de 20 pies como de contenedores de 40.

  Este tipo de problema recibiría el nombre de Multi-size daily drayage problem (MSDDP), y es el estudiado en este trabajo.

  Los diferentes problemas son variantes específicas del problema de enrutamiento de vehículos VRP,

denominado en inglés *vehicle routing problem,* y en concreto del VRPTW (vehicle routing problem with time windows), un problema considerado binario al basarse en dos situaciones: O lleva contenedor, o no lo lleva.

Muchos autores han llevado a cabo proyectos de mejora para la optimización de diferentes variantes del problema de enrutamiento comentado VRPTW.

Algunos de los más importantes son:

- TSP: Traveling Salesman Problem With Time Windows
- PDPTW: Pickup and Delivery Problem with Time Windows

Para la resolución de estos problemas y de sus diferentes variantes, se recurre a tres métodos:

- Métodos exactos
- Heurísticas
- Metaheurísticas

En el siguiente capítulo se describirán en detalle los tres tipos de métodos de resolución.

Por tanto, el daily drayage problema with time windows puede afrontarse como una variante de los dos problemas de enrutamiento expuestos (TSP, PDPTW).

Jula et al. (2005) consideran dicho problema como un asymetric multi-traveling salesman problem with time windows (am-TSPTW), suponiendo que en vez de un sujeto realizando las diversas rutas, aparecen varios en el problema.

Gronalt et al. (2003) plantean en cambio el problema como un full-load pickup and delivery problem with time windows (FLPDPTW), desarrollando una heurística híbrida para su resolución.

Por otra parte, cabe destacar también la investigación relacionada con el problema de los contenedores vacíos. Smilowitz (2006), introduce la aplicación del Multi-Resource Routing Problem (MRRP) with flexible tasks, es decir, modela el problema de acarreo como un problema de enrutamiento que considera tanto contenedores llenos como vacíos, empleando tareas flexibles (no es obligatorio conocerse la localización de origen o de destino, empleadas para los contenedores vacíos, ya que puede ser irrelevante la procedencia de un contenedor vacío para aquel que lo requiere, y puede ser irrelevante su destino para aquel que quiera liberarlo tras finalizar su tarea concreta.

3.2. Problema del acarreo diario con varios tamaños de contenedores.

El acarreo terrestre se realiza fundamentalmente en un área local determinada. Como ya se ha comentado anteriormente, resulta de gran utilidad para llegar a localizaciones que con otros modos de transporte sería totalmente inviable. Aporta la flexibilidad necesaria para poder tener ese alcance que permite la entrega y recogida de mercancías en clientes locales.

Ya se ha introducido el concepto de acarreo diario o Daily Drayage Problem. En este apartado se profundizará en un caso particular de dicho problema, que tiene en cuenta como restricción el tamaño de la unidad de carga, sabiendo que éste podrá ser o bien de 20 pies o bien de 40.

Se analiza por tanto a continuación el problema de acarreo diario con varios tamaños de contenedores, denominado en inglés como Multi-Size Daily Drayage Problem (MSDDP).

La resolución de dicho problema consistirá en la búsqueda de la combinación de rutas óptima en cuanto a ahorro de costes se refiere. Los costes a tener en cuenta a la hora de buscar la solución son:

- Costes de penalización: Aquellos que se aplican al inclumplir alguna de las restricciones propuestas en el modelo generado. Serán costes de penalización aquellos por incumplir las ventanas temporales, por ejemplo.
- Costes fijos. Relacionados con la flota de vehículos. Dependerán del número de camiones empleados para llevar a cabo las tareas, e implican costes como el sueldo del empleado, el propio coste del camión, etc.

- Costes variables: Relacionados con la distancia en kilómetros recorrida, implicando este número el gasto de combustible, desgaste de neumático y mantenimiento de los vehículos.

3.2.1. Simplificación del problema

Se llevan las siguientes simplificaciones para la resolución del problema propuesto:

- Entorno determinista. Las diferentes variables que aparecen en el problema son conocidas, lo que permite afrontar el problema desde un punto de vista estático.
- Se consideran tareas de importación y tareas de exportación, y se añaden a dicho problema las tareas flexibles, en las que se lleva a cabo el bypass, o movimiento de contenedores vacíos entre clientes, en vez de recurrir al transporte de la terminal a las diferentes localizaciones.
- Se considerará la misma velocidad (constante) durante todo el trayecto, en todos los vehículos. Todos los vehículos son de las mismas características, y misma capacidad.
- Un vehículo será capaz de transportar, o bien, dos contenedores de 20', o bien, un contenedor de 40'.
- No se tendrá en cuenta el peso de los contenedores, distinguiéndose solamente entre lleno (load) y vacío (empty).
- Debido a la proximidad con la que se encuentran en la realidad el depósito y la terminal, se considera en la formulación del problema una misma ubicación para ambos. Dicha localización se conoce de antemano, y es fija.
- No se tendrán en cuenta tiempos de descanso de los conductores

3.2.2. Tareas reales y ficticias

Es importante tener en cuenta que tanto el punto de partida de los vehículos como el punto en el que deben acabar al fin de la jornada laboral, es el depósito. Se conoce, para cada tarea, el punto de inicio y el de destino, y por tanto, cada camión tendrá asociada dos tareas ficticias: Una de inicio de jornada, y otra de finalización.

Esto se lleva a cabo para asegurar, en el caso de tareas de exportación, que el vehículo se desplaza desde un punto determinado hasta el cliente, y en el caso de las tareas de importación, para modelar el regreso del camión al depósito una vez realizadas las operaciones pertinentes.

Así pues, quedan determinados dos tipos diferentes de tareas, que serán de gran utilidad a la hora de modelar el problema:

- Tareas reales (T_r). Conjunto de ordenes que engloba el acarreo para llevar a cabo a lo largo de la jornada laboral. Inluyen tanto los trayectos entre clientes, como aquellos movimientos del depósito a éstos.
- Tareas ficticias(T_f). Tienen características diferentes a las reales. Sirven para modelar las limitaciones temporales en el depósito, así como los movimientos de retorno de los camiones al depósito.

Por tanto, las tareas ficticias hacen posible el poder comenzar el problema con los vehículos en la terminal (o depósito). El punto de origen de éstas seran las coordenadas de tal ubicación, y las variables de distancia euclídea y tiempos de trayecto serán nulos para no influir en el análisis del problema a resolver.

Así pues, si se agrupan los diferentes tipos de tareas vistos hasta ahora, se obtienen las siguientes conclusiones:

- $T_r = \quad T^I \cup T^E \cup T^F$
- $T_f = T_f^{ini} \cup T_f^{end}$
- $T = T_f \cup T_r$
- $T_f \cap T_r = \emptyset$

3.2.3. Formulación

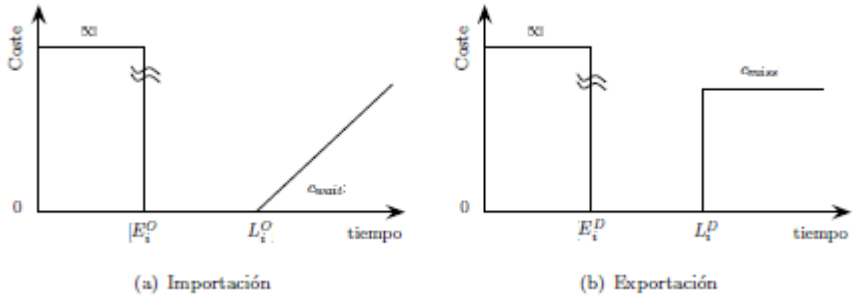
Se procede a continuación a representar el modelo matemático que refleja el problema propuesto, indicándose los diferentes parámetros empleados, las variables y la función objetivo.

Parámetros

En caso de no cumplirse estos intervalos de tiempo, se aplicarían penalizaciones. Estas penalizaciones van a modelarse como parámetros, y se van a traducir en los diferentes costes a tener en cuenta en el desarrollo del problema.

Dependiendo del retraso, la penalización será diferente, diferenciando entre:

- Retraso en la recogida del contenedor [c_{wait}]: Implica una penalización que simboliza el tiempo extra que dicho contenedor debe permanecer esperando en la terminal. Supondrá una función en la que el coste incrementará linealmente con el tiempo de retraso.
- Retraso en la entrega [c_{miss}]: Implica una penalización mayor, ya que puede significar que el siguiente medio de transporte de la cadena intermodal haya efectuado su salida, perdiéndose en ese caso el envío. Se representa como una función escalón, con coste c_{miss} si la entrega es posterior al momento L_i^D.



(a) Importación (b) Exportación

Ilustración 5. Ventanas temporales de las tareas. Fuente: Escudero-Santana (2013)

Otros costes que se tendrán en cuenta también serán modelados como parámetros:

- c_v: Coste fijo por vehículo usado. Penaliza el exceso de camiones empleados para la resolución del problema.
- c_{km}: Coste asociado a la distancia recorrida en km.
- c_{depot}: Coste debido a retrasos en el retorno al depósito tras finalizar la jornada laboral

Además, la jornada laboral también se modela mediante las ventanas temporales, siendo necesario que todos los vehículos acaben en el depósito dentro del intervalo de tiempo que se haya estipulado en el enunciado del problema.

Variables

Las variables del problema se representan en la siguiente tabla:

Tabla 3. Variables del problema MSDDP

d_i	Distancia euclídea entre origen y destino de la tarea
t_i	Tiempo de realización de la tarea i

s_i^O y s_i^D	Tiempos de servicio de origen y destino (Carga y descarga del contenedor
T^I y T^E	Tareas de importación y exportación
T^F	Tarea flexible. Tanto origen como destino será la ubicación de un cliente determinado.
$[E_i^O; L_i^O]$	Ventana temporal en la que se permite la carga del contenedor en origen.
$[E_i^D; L_i^D]$	Ventana temporal en la que se permite la descarga del contenedor en destino.

Las dos últimas filas de la tabla anterior muestran las ventanas temporales, las cuales caracterizan las diferentes tareas.

- **Variables de decisión**

Se emplean las siguientes variables binarias:

- $nv_i = \begin{cases} 1 & \text{Si el vehículo i es usado por primera vez} \\ 0 & \text{en otro caso} \end{cases}$
- $x_{ij} = \begin{cases} 1 & \text{la tarea i y la tarea j son servidas consecutivamente} \\ 0 & \text{en otro caso} \end{cases}$
- $l_i^O = \begin{cases} 1 & \text{existe retraso al servir en el origen la tarea i} \\ 0 & \text{en otro caso} \end{cases}$
- $l_j^D = \begin{cases} 1 & \text{existe retraso al servir en el destino la tarea j} \\ 0 & \text{en otro caso} \end{cases}$
- $l_j^{depot} = \begin{cases} 1 & \text{el vehículo que realiza la tarea j llega al depósito con retraso} \\ 0 & \text{en otro caso} \end{cases}$

- **Función objetivo**

Se expone a continuación la función objetivo, que minimiza el coste total teniendo en cuenta la realización de todas las tareas, así como las diferentes penalizaciones expuestas:

$$\min\left(c_v \sum_{i\epsilon T_f^{ini}} nv_i + \sum_{i\epsilon T_f^{ini}\cup T_r} \sum_{j\epsilon T_r\cup T_f^{end}} d_{ij}x_{ij} + c_{wait} \sum_{i\epsilon T_r^O} (st_i - L_i^O) l_i^O + c_{miss} \sum_{j\epsilon T_r^D} l_j^D + c_{depot} \sum_{i\epsilon T_f^{end}} (st_i - L_i^O) l_i^{depot} \right)$$

4 MÉTODOS DE RESOLUCIÓN

Son muchas las variantes que a lo largo de los años han surgido del famoso problema de ruteo VRP. Se han enumerado ya algunas de las más importantes, en las cuales, restricciones como las ventanas temporales o el tamaño de los contenedores juegan un papel fundamental.

Conforme pasan los años, aumenta por tanto la investigación en lo que respecta a la búsqueda de una solución óptima para dicho problema, y se recurre para ello a diferentes caminos, diferentes métodos de resolución dependiendo del autor.

En este capítulo se llevará a cabo un análisis de los diferentes métodos que existen para la búsqueda de la optimización de un problema, exponiendo sus principales características y estudiando los diferentes tipos de cada uno de ellos.

4.1. Métodos exactos

Son aquellos que ofrecen una solución óptima global al problema propuesto. Computacionalmente resultan ser más lentos que cualquier heurística, y por ello solo se emplean para aquellos problemas de dimensiones pequeñas, en los que no se manejen grandes cantidades de datos.

En el caso de un VRP, podrían emplearse siempre y cuando el problema supusiera una única localización para el depósito o terminal, resultando poco recomendable recurrir a un método exacto para la resolución de un problema de enrutamiento con varios emplazamientos de depósitos y terminales. Se muestran a continuación algunos de los métodos más relevantes:

4.1.1. Branch and Bound (B&B)

Según Mullin & Belotti (2016)el algoritmo Branch and Bound (Ramificación y Acotación) emplea la partición recursiva para resolver problemas de optimización no-convexos, y sirve para resolver problemas que van desde la optimización lineal de enteros mixtos (Mixed-Integer Linear Optimization), donde una función objetivo lineal se minimiza sujeta a restricciones lineales e íntegramente de un subconjunto de variables (Nemhauser & Wosley, 1988), a una optimización global, donde ambas funciones objetivo son forzadas a ser enteros (Horst & Tuy, 1996)

Se entiende por branch, la ramificación de uno de los nodos en dos nuevos a analizar, y dicha ramificación se lleva a cabo cuando alguna de las cotas (bound) de dichos nodos se ve modificada.

El algoritmo se resume en los siguientes pasos:

Paso 0. Inicialización

Se inicializa la lista de nodos para explorar con el nodo raíz.

Paso 1. Exploración

Se pasa al siguiente nodo y se calcula su relajación lineal.

Paso 2. Límite

Se compara el valor obtenido en el nodo actual con el anterior. Si es menor, se procede al siguiente paso. Si es mayor que el anterior, se deshecha dicha opción y se vuelve al paso 1.

Paso 3. Actualización

Si la relajación del nodo actual es una solución entera, entonces se ha mejorado dicha solución. Debe tenerse en cuenta que la solución empleando este método es siempre entera.

Ésta pasa a ser la nueva solución actualizada, y se vuelve a proceder desde el paso 1.

Paso 4. Ramificación

Se ramifica el nodo actual en dos. Se comprueba la solución de ambos. Si es válida, se continúa explorando dicho nodo y se procede al paso 1 de nuevo.

A continuación se expone un ejemplo gráfico de un problema resuelto mediante el método B&B

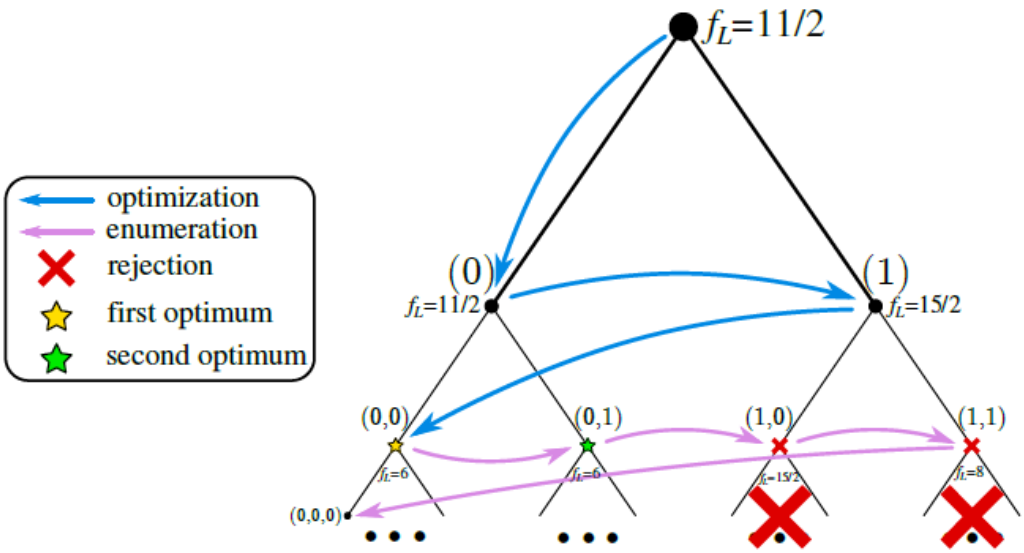


Ilustración 6. Esquema de resolución de un problema mediante algoritmo Branch & Bound (Briheche, Barbaresco, Bennis, & Chablat, 2020)

4.1.2. Branch and cut

El procedimiento Branch and cut resulta similar en cuanto a la búsqueda de la solución óptima mediante el desarrollo de nodos. Cada nodo representa un subproblema del tipo lineal (Linear Programming) o cuadrático (Quadratic Programming) que debe ser procesado; es decir, debe ser resuelto, comprobarse que es entero y posteriormente ser analizado. Se denominará a los nodos *activos* si todavía no han sido procesados.

Se denominará *cut* (corte) a toda aquella restricción que se añada al modelo. El propósito de añadir cortes al problema no es otro que el de reducir el tamaño de dicho problema representado en los nodos. Dicho de otra forma, añadiendo cortes se persigue prescindir de nuevas ramificaciones innecesarias.

Una vez aplicada la técnica de adición del plano de corte o restricción comentada, se procede a aplicar el algoritmo Branch and Bound expuesto anteriormente.

4.2. Heurísticas

Se califica de heurístico al procedimiento en el que se tiene un alto grado de confianza en que encuentre soluciones de alta calidad con un coste computacional razonable, aunque no se garantice su optimalidad o su factibilidad, e incluso, en algunos casos, no se llegue a establecer lo cerca que se está de dicha situación

Por tanto, una heurística es una técnica que busca soluciones siguiendo una serie de pasos previamente fijados y que se aproximan al óptimo, a un costo computacional razonable.

Mediante esta breve definición se obtiene un buen resumen del concepto de método heurístico, el cual, es empleado para la resolución de problemas de optimización de un tamaño considerable. Como se expone en la definición, no garantizan la optimalidad del problema; sin embargo, ofrecen una aproximación muy buena, teniendo en cuenta el notable incremento de velocidad computacional respecto a los métodos exactos.

Orientando de nuevo este tipo de método aproximado al problema de ruteo de vehículos, destacan tres

variantes atendiendo a la forma en que se desarrollan: Constructivas, de mejora, y aquellas que emplean técnicas de relajación.

- Constructivas: Siguiendo un criterio determinado, se va construyendo una solución admisible, progresivamente conforme avanza el algoritmo. Una de las más famosas es la heurística de ahorros. Empleada para problemas de ruteo, busca reducir la distancia global recorrida modificando el orden de los grafos, entendiéndose cada grafo como una localización.
  Otra heurística de este tipo sería el método del barrido, el cual fija alguna de las restricciones como por ejemplo la distancia límite a recorrer, y se "recorre" todas las opciones siguiendo un parámetro fijado de antemano al que se le denomina "ángulo".
- Heurísticas de mejora: A partir de una solución admisible, el algoritmo se dedica a proponer variaciones, ya sea en una de las rutas propuestas, modificándose el orden de las localizaciones visitadas, o bien, variando las rutas entre ellas en búsqueda de la mejora de la solución actual.
- Heurísticas de relajación: Como su propio nombre indica, tratan de relajar las restricciones, es decir, discernir entre aquellas restricciones de difícil cumplimiento y aquellas que no lo suponen, pasando a formar parte las primeras de la función objetivo, multiplicadas por una penalización.
  Destaca la Relajación Lagrangiana, la cual lleva a cabo el procedimiento comentado para conseguir un problema acotado y acelerar la resolución.

Una vez expuestos los tres tipos de heurísticas atendiendo al problema de ruteo de vehículos, resulta interesante indagar en alguno de los métodos heurísticos que existen en el campo de la optimización. Se exponen a continuación los más relevantes:

4.2.1. Método de Clark y Wright

También conocido como método de ahorros, es uno de los algoritmos más recurrentes a la hora de afrontar el problema de ruteo de vehículos. Su tiempo de computación resulta ser reducido, lo cual permite ofrecer una solución rápida, y próxima al óptimo.

En el caso del VRP, consiste en asignar la posición 0 al depósito (Se debe tener en cuenta que trabaja con un único depósito) y corresponderán a los diferentes clientes las posiciones desde 1 a n.

Cada cliente se representa mediante una variable x_{ij}, y se registra en una base de datos cada uno de los costes que hay entre el depósito y los clientes, así como todos los costes entre cada uno de los diferentes clientes. Se designan mediante c_{ij}.

El algoritmo partirá de la solución admisible que une cada uno de los clientes con el depósito, es decir, la solución inicial no es otra que tantas rutas como clientes, siendo dicha ruta:

$$depot - c_i\,;\ \ c_i - depot \ \square\ \text{i} = 1,\ldots,\text{n}$$

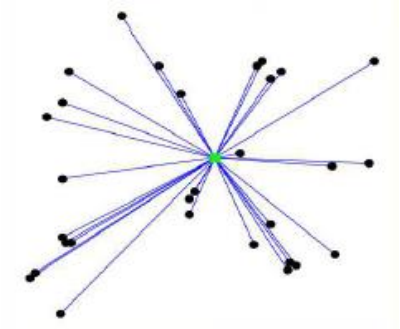


Ilustración 7. Solución inicial de Clark & Wright

Coste inicial:

$$2\sum_{j-1}^{n} c_{0j}$$

El algoritmo irá probando diferentes combinaciones de rutas, uniendo aquellas que terminan en **i** con las que comienzan por **j**, surgiendo así el arco (i, j), y calculándose su ahorro (s_{ij}) mediante el análisis de los costes, mediante la siguiente ecuación:

$$\text{Ahorro: } s_{ij} = 2(c_{0i} + c_{0j}) - (c_{0i} + c_{ij} + c_{0j}) = c_{0i} + c_{0j} - c_{ij}$$

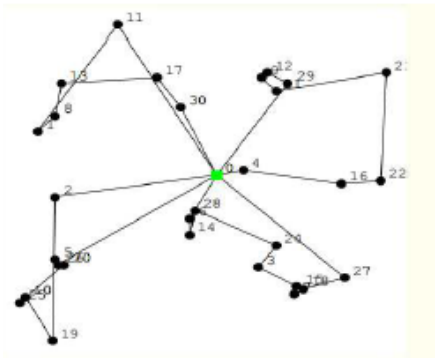


Ilustración 8. Solución óptima, Clark & Wright

4.2.2. Método del barrido

Esta heurística trabaja sobre un problema plano como puede ser el mapa de los diferentes clientes y el depósito. Consiste en trazar una semirrecta con origen dicho depósito, que gira respecto a éste abarcando un sector del mapa determinado en el cual las diferentes localizaciones de los clientes se van incluyendo. El ángulo de barrido dependerá de la restricción que quiera fijarse, siendo habitual que ésta sea la capacidad de los vehículos.

Una vez llevada a cabo esta selección de zonas barridas por la recta, se aplica a cada una de ellas un algoritmo de ruteo para decidir la ruta óptima en cada uno de dichos sectores. Esta segunda parte suele resolverse aplicándose un TSP (Traveling Salesman Problem), ya que cada sector barrido se asocia a un único vehículo.

Se muestra a continuación un ejemplo gráfico de un problema resuelto mediante dicha heurística:

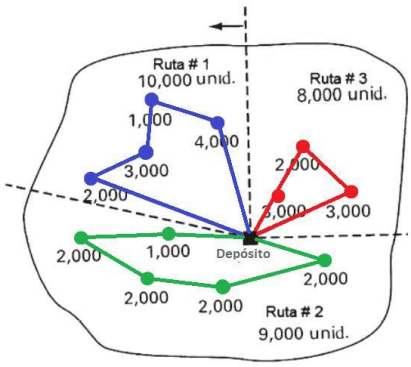


Ilustración 9. Ejemplo de resolución mediante método del barrido (Arango Serna, Gil Gómez, & Zapata Cortes, 2009)

4.3. Metaheurísticas

Según Melián et al. (2003), las metaheurísticas son estrategias inteligentes para diseñar o mejorar procedimientos heurísticos (definidos en el apartado anterior) muy generales con un alto rendimiento.

Así, una metaheurística puede emplearse para la resolución de problemas más complejos, con un volumen de datos mayor, y va a poder adaptarse a ciertas variantes que surjan de dichos problemas. Los algoritmos metaheurísticos son más robustos, y aportan una velocidad mayor.

Atendiendo al tipo de procedimiento empleado por el algoritmo, se distinguen varios tipos de metaheurísticas. Los más importantes son los métodos de relajación, constructivos, búsqueda de entornos y evolutivos.

- Metaheurísticas de relajación. Se proponen modificaciones del modelo para facilitar o simplificar la resolución del problema. El problema se simplifica mediante la modificación o incluso eliminación de alguna de las restricciones, y es posible a su vez la realización de ciertos cambios en la función objetivo. Un ejemplo de esta técnica seria la relajación de un problema de programación lineal entera, eliminando para simplificar el procedimiento, la restricción de que las variables deban ser enteras. Dicho ejemplo se usa frecuentemente en problemas que requieren del método Simplex para la obtención de la solución.
- Metaheurísticas constructivas. Partiendo de una solución vacía, se va generando una estructura mediante la incorporación iterativa de elementos. Mediante un procedimiento estratégico se van eligiendo aquellas componentes que generen una buena solución. Una de las más empleadas es la metaheurística GRASP.
- Metaheurística de búsqueda. Recorre el espacio de posibles soluciones transformando de forma iterativa un conjunto de soluciones de partida. Este conjunto de soluciones con las que trabaja el algoritmo se denomina habitualmente población. Destacan las búsquedas locales, las cuales, mediante la modificación de una parte determinada de la solución, analizan la función objetivo para comprobar si se mejora. Cuando la solución se modifica de esta manera, se habla de vecindades. Un ejemplo de ello sería el intercambio de dos elementos de un vector, siendo dicho vector la representación de la solución a analizar.
  Destacan los métodos de Recocido Simulado y la Búsqueda Tabú.
- Metaheurística evolutiva. Combina la información aportada por los elementos de la población para obtener nuevas soluciones, en búsqueda del óptimo.
  Inspirado en la naturaleza, el más conocido es el algoritmo genético.

  Otros algoritmos evolutivos son los meméticos, o los de re-encadenamiento de caminos (*Path Relinking*).

Resulta de gran interés analizar algunos de los algoritmos expuestos anteriormente, para comprender mejor en qué consisten las metaheurísticas, y su utilidad a la hora de afrontar los problemas de optimización.

4.3.1. GRASP

Un GRASP (Procedimiento de búsqueda miope aleatorizado y adaptativo) es un método multi-arranque diseñado para resolver problemas difíciles en optimización combinatoria. En su versión básica cada iteración consiste en dos fases: una fase constructiva cuyo producto es una solución factible buena, aunque no necesariamente un óptimo local, y una búsqueda local, durante la cual se examinan vecindades de la solución, al llegar a un óptimo local la iteración termina (Resende & Gonzalez-Velarde, 2003)

Se expone a continuación un ejemplo de GRASP básico para minimización, en pseudo-codigo:

```
procedure GRASP

Require: i_máx
  f* ← ∞;
  for i ≤ i_máx do
    x ← GreedyRandomized();
    x ← LocalSearch(x);
    if f(x) < f* then
      f* ← f(x);
      x* ← x;
    end if
  end for
  return x*;
```

Ilustración 10. Pseudo-código GRASP (Resende et al. 2003)

Por tanto, este algoritmo iterativo trabaja en dos fases:

1. Construir una solución aleatoria (Greedy)
2. Realizar búsqueda local (vecindad) para obtener mejoras en la función objetivo.

La solución será actualizada cada vez que se haya obtenido una mejora tras la búsqueda local.

Mientras que en los algoritmos de relajación comentados anteriormente se obtenía una primera solución con restricciones modificadas o incluso eliminadas en primera estancia, la metaheurística GRASP aporta una solución de partida de calidad (proceso constructivo), lo cual supone una gran ventaja frente a los primeros.

4.3.2. Algoritmo Genético

Los principios del Algoritmo Genético (AG) residen en el comportamiento evolutivo de la naturaleza.

El algoritmo genético trabaja con un conjunto de individuos al cual se le denomina población, representando cada uno de ellos una solución factible al problema propuesto.

Cada componente de dicha población tendrá asignado una puntuación, que depende de la bondad de dicha solución. Este concepto de bondad se asemeja a la capacidad que en la naturaleza tendría un individuo para sobrevivir o luchar por unos determinados recursos. En el caso matemático, la bondad estará referida al nivel de adaptación que la solución tiene al problema propuesto.

Cuanto mayor sea la bondad del individuo, mayor será la probabilidad de conseguir "reproducirse", es decir, tendrá asignada una probabilidad mayor para ser cruzado con otro individuo seleccionado bajo el mismo criterio. Este cruce no es más que un intercambio de material genético, es decir, un intercambio de las características de ambas soluciones para posteriormente valorar si dicha actualización resulta favorable en cuanto a los objetivos del problema, o por el contrario, ha empeorado la adaptación a las diferentes restricciones propuestas. Dicha actualización resultante del cruce equivaldría al "hijo" en el símil natural, y de hecho en muchos casos adapta ese nombre en el algoritmo. Por tanto, la adaptabilidad de un individuo a la solución se asemeja a la probabilidad de éste a cruzarse con otro individuo. Tras este proceso resulta una nueva población de posibles soluciones, la cual tendrá cada vez una proporción mayor de características favorables.

La UPV propone un ejemplo de éste método, el algoritmo genético simple, el cual se muestra a continuación en pseudo-código:

```
BEGIN /* Algoritmo Genetico Simple */
    Generar una poblacion inicial.
    Computar la funcion de evaluacion de cada individuo.
    WHILE NOT Terminado DO
    BEGIN /* Producir nueva generacion */
        FOR Tamaño poblacion/2 DO
        BEGIN /*Ciclo Reproductivo */
            Seleccionar dos individuos de la anterior generacion,
            para el cruce (probabilidad de seleccion proporcional
            a la funcion de evaluacion del individuo).
            Cruzar con cierta probabilidad los dos
            individuos obteniendo dos descendientes.
            Mutar los dos descendientes con cierta probabilidad.
            Computar la funcion de evaluacion de los dos
            descendientes mutados.
            Insertar los dos descendientes mutados en la nueva generacion.
        END
        IF  la poblacion ha convergido THEN
            Terminado := TRUE
    END
END
```

Ilustración 11. Pseudo-código Algoritmo Genético Simple (UPV)

El pseudo-código anterior sirve de explicación del proceso iterativo que se lleva a cabo. En primer lugar se genera de manera aleatoria una población inicial, formada por individuos que resultan ser, por regla general, vectores (cromosomas) de componentes binarios {1,0}. Dichos componentes reciben el nombre de genes, y son los que tras el cruce verán alterado su orden. Se computa la función de evaluación y dependiendo del resultado de atribuye a cada individuo una probabilidad de selección para el cruce. Aquellos con probabilidades similares son cruzados, y sus descendientes vuelven a pasar por la función de evaluación. En caso de ser una evolución positiva, estos descendientes son insertados en la nueva generación. Este proceso iterativo se lleva a cabo hasta cumplir el criterio de convergencia que se haya impuesto.

4.3.3. Recocido Simulado

En la industria metalúrgica, uno de los procesos habituales consiste en calentar un determinado material para después enfriarlo gradualmente, de manera controlada. Con ello se consigue un aumento de los cristales de los que se compone, así como una reducción de ciertos defectos.

El algoritmo de Recocido Simulado (en inglés, Simulated- Annealing) se basa en este proceso, con los siguientes símiles:

- La solución a analizar equivale a la estructura cristalina que se persigue modificar en el proceso
- La variación de la puntuación equivale a la energía en el recocido
- La distancia que se modifica en el algoritmo equivale a la temperatura de recocido

Tricas García (2015) propone los siguientes pasos para el algoritmo:

1. Inicialización. Se genera una solución aleatoria, equivalente a una “Temperatura elevada”.
2. Movimiento. Se perturba la solución mediante alguna alteración en su orden.
3. Evaluación. Cálculo de la variación en la puntuación (Energía)
4. Decisión. Dependiendo del resultado de la evaluación, se decide entre aceptar o rechazar. La probabilidad de aceptación depende de la temperatura:
    a. Si el vecino es mejor, se elige.
    b. Si es peor, se le asigna una probabilidad según la Distribución de Boltzmann

$$e^{-\frac{|E(s)-E(s')|}{K_B \cdot Temp}}$$

Donde:

s es el estado actual, y *s'* es el vecino

E es la función objetivo

Temp disminuye conforme avanza el algoritmo

K_B es constante

5. Actualización. Se reduce el valor de la temperatura y retorno al paso 2 hasta alcanzar el 'punto de enfriamiento' o condición de finalización de algoritmo.

```
SIMULATED-ANNEALING()
 1  Crear solución inicial s
 2  Inicializar la temperatura Temp
 3  repeat
 4      for i = 1 to numIteraciones
 5
 6          Generar un s' vecino aleatorio de s
 7          if (E(s) ≥ E(s'))
 8              s = s'
 9          else
10              if (e^((E(s)−E(s'))/(K_B·Temp)) > random[0,1))
11                  s = s'
12          Reducir temperatura Temp
13  until (no hay cambios en E(s))
14  Resultado s
```

Ilustración 12. Pseudo-código Recocido Simulado (Tricas, 2015)

5 BÚSQUEDA TABÚ

Para la resolución del problema MSDDP, analizado en este proyecto, se ha empleado el algoritmo de búsqueda tabú, uno de los más populares métodos de búsqueda local.

En este capítulo se analizará dicho método, profundizando en la definición y sus características, el uso de memoria, y la variante a la que se ha recurrido en este problema: la búsqueda tabú reactiva.

5.1. Definición

Según Glover (1986), la búsqueda tabú es una estrategia para resolver problemas de optimización combinatoria, con la capacidad de hacer uso de muchos otros métodos, tales como algoritmos de programación lineal y heurística especializada, y a los cuales dirige para superar las limitaciones de la optimización local.

Dicho método está estrechamente relacionado con la inteligencia artificial. Según Martí (2003), TS toma de la IA (Inteligencia Artificial) el concepto de memoria y lo implementa mediante estructuras simples con el objetivo de dirigir la búsqueda teniendo en cuenta la historia de ésta. Es decir, el procedimiento trata de extraer información de lo sucedido y actuar en consecuencia. En este sentido puede decirse que hay un cierto aprendizaje y que la búsqueda es inteligente.

El procedimiento de búsqueda se desarrolla mediante vecindades, es decir, se van realizando intercambios entre dos componentes del vector solución, y tras cada intercambio se evalúa la función objetivo para valorar la mejora, o empeoramiento de la solución actual.

Así pues, la búsqueda tabú utiliza la memoria para almacenar información perteneciente a las últimas soluciones analizadas, con el propósito de guiar la búsqueda del óptimo y evitar recaer en óptimos locales, o soluciones visitadas recientemente.

Mediante ese uso de memoria "prohíbe" durante cierto número de iteraciones, ciertas soluciones candidatas, y las almacena en una lista, denominada lista tabú. De esa prohibición temporal es de donde se obtiene el nombre de dicho método.

Se exponen a continuación las principales características del algoritmo.

5.2. Características

Resulta interesante profundizar en este método mediante el análisis de las principales características, empezando por cómo emplea la memoria, cómo lleva a cabo la búsqueda del óptimo, conocer los criterios de aspiración, y analizar por último las estrategias de búsqueda, a saber: intensificación y diversificación.

5.2.1. Lista tabú

Durante el proceso de búsqueda, en cada iteración, para que el algoritmo no se quede atrapado en un ciclo, se almacena la información relativa a las soluciones recientemente visitadas en una lista llamada lista tabú (Sait & Youssef, 1999).

Resultaría altamente costoso computacionalmente, almacenar en la lista tabú soluciones completas. Es por eso que dicha lista se limita a guardar movimientos, entendiéndose por tales, aquellas operaciones por las que se ha llevado a cabo la vecindad oportuna.

Estos movimientos quedarán prohibidos durante un número de ciclos, siendo posible volver a realizarlos antes de lo debido en caso de cumplir un criterio de aspiración. Este último concepto se analiza en profundidad en el

apartado 5.2.3.

Ejemplo. Sea la solución con la que se trabaja un vector de 5 posiciones:

x_i =1,..,5 ∀i

1	4	3	5	2

Ilustración 13. Solucion inicial

Se genera una lista tabú, inicialmente vacía, con los posibles movimientos:

	1	2	3	4	5
1					
2					
3					
4					
5					

Ilustración 14. Lista Tabú de 5 elementos

Si se llevara a cabo en la siguiente iteración en la figura 10, la vecindad 1-5 (intercambio de dichos elementos), y asumiendo que la prohibición de dicho movimiento es durante 3 ciclos en este ejemplo, la lista quedaría actualizada como en la ilustración 11:

5	4	3	1	2

Ilustración 15. Solución resultado de aplicar vecindad 1-5

	1	2	3	4	5
1					3
2					
3					
4					
5					

Ilustración 16. Lista Tabú actualizada

En el siguiente ciclo, si ahora se aplica un intercambio de los elementos 5 y 3, el vector solución actualizado y la nueva lista actualizada quedarán de la siguiente manera:

3	4	5	1	2

Ilustración 17. Vector solucion actualizado

	1	2	3	4	5
1			3		2
2					
3					
4					
5					

Ilustración 18. Lista Tabú actualizada. Segundo ciclo

Con esto, se consigue ahorrar tiempo computacional, ya que al estar prohibidos temporalmente los movimientos registrados en la lista, se reduce el número de acciones que deben ejecutarse en cada uno de los ciclos.

5.2.2. Vecindad

En el apartado anterior ya se introduce el concepto de vecindad mediante el ejemplo propuesto.

El algoritmo tabú emplea para la búsqueda del óptimo un método iterativo por el cual va aplicando en cada ciclo intercambios entre los diferentes elementos que componen la solución.

Cada intercambio o vecindad supone una variación de la función objetivo, y será entonces necesario valorar si dicha modificación resulta una función mejor o peor que la anterior.

Así pues, se entiende por vecindad ese intercambio entre dos elementos de la solución, siendo tres los conceptos de vecindad más populares:

- Adjacent Swap: Una solución S de n elementos tiene n-1 vecinos.
  Se realizan intercambios exclusivamente entre aquellos elementos adyacentes.

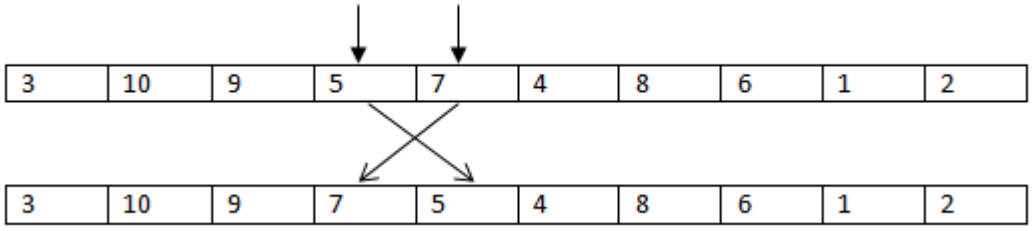


Ilustración 19. Adjacent Swap

- General Swap: Una solución S de n elementos tiene $\frac{n(n-1)}{2}$ vecinos.
  Un elemento puede intercambiarse por cualquiera de los demás.

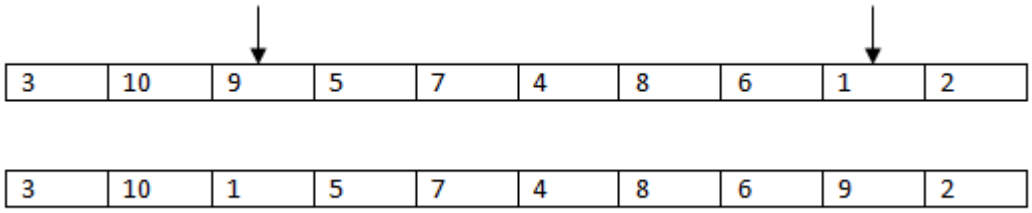


Ilustración 20. General Swap

- Insertion: Una solución S de n elementos tiene $(n-1)^2$ vecinos.
  Se extrae un elemento y se "inserta" en una determinada posición.

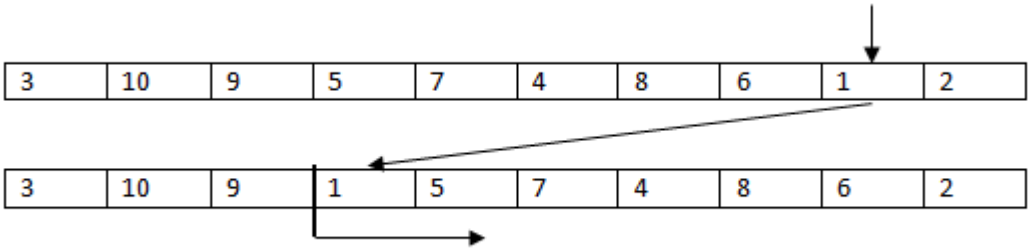


Ilustración 21. Insertion

5.2.3. Niveles de aspiración

Hay ocasiones en las que interesa otorgar al algoritmo cierta flexibilidad. El criterio de aspiración permite determinar ciertas condiciones en las cuales está permitido incluir alguno de los movimientos restringidos que se encuentran en la lista tabú.

Los niveles de aspiración se clasifican en dos:

- Aspiraciones de movimiento: Se invalida la restricción en la lista tabú de un cierto movimiento, en caso de que se cumpla el criterio.
- Aspiraciones de atributo: Se invalida la restricción en la lista de un determinado atributo.

Según Guerrero & Escudero Santana (2014) , los criterios de aspiración surgen por tres motivos:

- Si todos los posibles movimientos se consideran tabú, entonces se deberá escoger el movimiento "menos tabú" de todos ellos. Esto es que si se tienen varios movimientos restringidos, se escogerá el movimiento que le resten menos iteraciones para volver a ser permitido.
- En el caso de que el valor de la función objetivo de una solución, sea mejor que el mejor valor obtenido hasta el momento. Por ejemplo, un movimiento m es aceptado sí, en un problema de minimización, el valor de la solución x resultante es menor que el menor coste obtenido hasta entonces.
- Para los casos en los que un atributo de aspiración para alcanzar una solución x se satisface si la dirección para en x otorga una mejora y el actual movimiento también es de mejora. En ese caso, se toma a x como posible candidato.

5.2.4. Estrategias de búsqueda

El algoritmo de la búsqueda tabú lleva asociada la interacción entre la memoria a corto plazo y a largo plazo.

En cuanto a la memoria a corto plazo, ha sido analizado el método por el cual, mediante la lista tabú, se

almacenan aquellos movimientos recientes para no recaer en óptimos locales. Esta es la estrategia fundamental que dicha memoria sigue.

Mientras que la memoria a corto plazo se orienta a la búsqueda de un óptimo local, la memoria a largo plazo tendrá como fin encontrar un óptimo global. Así, en este apartado se analizan las dos estrategias más importantes en cuanto al uso de memoria a largo plazo.

En búsqueda tabú, se entiende por memoria a largo plazo el registro y almacenaje de la frecuencia con que un determinado atributo aparece en una región de soluciones determinada. La identificación de diferentes regiones es la que rige este tipo de uso de memoria.

Martí (2003) define estas dos estrategias de la siguiente manera:

- Diversificación: consiste en visitar nuevas áreas no exploradas del espacio de soluciones. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente.
- Intensificación: consiste en regresar a regiones ya exploradas para estudiarlas más a fondo. Para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones encontradas

5.3. Búsqueda tabú reactiva

5.3.1. Definición

Según Battiti y Tecchiolli (1994), la búsqueda tabú reactiva (RTS) aumenta la robustez del algoritmo, y se basa en un simple mecanismo de adaptación del tamaño de la lista tabú acorde con las propiedades del problema de optimización.

Así, las configuraciones visitadas durante la búsqueda y el correspondiente número de iteraciones se guardan en la memoria para que, tras la elección del último movimiento, se pueda verificar la repetición de las configuraciones y calcular el intervalo entre dos visitas.

El mecanismo básico de "reacción" rápida aumenta el tamaño de la lista cuando ciertas configuraciones se repiten. Esto va acompañado de un mecanismo de reducción más lento para disminuir aquellas regiones del espacio de búsqueda que no necesiten tamaños tan grandes.

En resumen, se trata de una variación del algoritmo en la que los tamaños de la lista tabú pasan a variar dependiendo de las necesidades en cada momento durante la ejecución del mismo.

5.3.2. Características

La búsqueda tabú reactiva constituye una variante de gran interés en lo que respecta a la búsqueda del óptimo, siendo de gran utilidad a la hora de mantener el control sobre posibles movimientos cíclicos.

Como se ha comentado anteriormente, tras cada iteración se almacena en memoria la configuración de las soluciones obtenidas. Tras esto, se valora si el procedimiento ha quedado "estancado" en un ciclo, situación que ocurre al dar con un óptimo local (atractor). Cuando se sospecha de la presencia de un atractor fuerte, el método RTS añade un mecanismo de diversificación, descrito en el apartado anterior, que permite abandonar la solución actual, y seguir profundizando en el análisis de zonas que no han sido tocadas por el método hasta ese momento.

Pueden analizarse tres procedimientos distintos a la hora de ejecutar el método RTS:

1. Mecanismo de reacción básico. Consiste en aumentar la lista tabú cuando las configuraciones de la solución actual llevan repitiéndose durante un número determinado de iteraciones. Este mecanismo también puede encargarse de reducir dicha lista en caso de que no aparezca una necesidad de aumento de la misma.
2. Mecanismo de diversificación. A lo largo de la búsqueda del óptimo, es frecuente que el método de búsqueda caiga en un óptimo local o *atractor (*Battiti, 1994). El atractor puede desencadenar en una

búsqueda cíclica, en la cual, al cabo de un número de iteraciones N, la solución analizada siga siendo la misma. Para evitar este comportamiento cíclico, se aplica un mecanismo de diversificación por el cual, como se ha descrito en el apartado anterior, se pasan a analizar nuevos conjuntos de soluciones abandonando así el óptimo local.

3. Restricción del área de búsqueda. Un procedimiento interesante para huir de los óptimos locales es el de restringir la trayectoria a una determinada área del espacio de búsqueda, e ir variando dicha área de manera aleatoria. Battiti et al. (1994) introducen el concepto de *caos* para describir dicho procedimiento, en el cual, hablan de *atractores caóticos,* caracterizados por la "contracción de áreas", por la cual, «diferentes trayectorias comienzan con condiciones iniciales diferentes, siendo comprimidas en un área limitada del espacio de configuraciones fijado, con dependencia de las condiciones iniciales, por lo que las diferentes trayectorias acaban divergiendo».

5.3.3. Caos y exponente de Lyapunov

Como se expone en el apartado anterior, RTS recurre a dos conceptos relacionados con la rama de los sistemas dinámicos: Caos y el exponente de Lyapunov.

El concepto de Caos es empleado para muchas aplicaciones en ingeniería informática, en lo que respecta a sistemas dinámicos. Se muestra a continuación una definición matemática del concepto expuesta en la Universidad Politécnica de Madrid (Escribano Iglesias & Giraldo Carbajo):

Se dice que un sistema dinámico (X, f) es caótico si

- es sensible respecto a las condiciones iniciales
- es topológicamente transitivo,
- sus puntos periódicos son densos en X.

Si (X, f) es sensible respecto a las condiciones iniciales, pequeños errores en la estimación de valores de la función se pueden ampliar considerablemente al iterarla.

Si (X, f) es topológicamente transitivo no puede descomponerse en dos subconjuntos disjuntos invariantes con interior no vacío. (Si f posee una órbita densa entonces (X,f) es topológicamente transitivo).

Por tanto, si un sistema dinámico es caótico, tiene una componente de impredicibilidad, una componente de irreducibilidad pero aun así tiene una tercera componente de regularidad.

Según Battiti (1994), sea una función g que mapea un punto en el paso n a un punto en el paso n + 1, g^k(x) se define como el mapeo obtenido al interactuar g, k veces. Si se empieza con configuraciones cercanas x_0 y $x_0 + \varepsilon$, el exponente de Lyapunov λ se define por la relación:

$$\epsilon e^{n\lambda} \approx || \, g^n(x_0 + \epsilon) - g^n(x_o) \, ||$$

Si λ > 0, los puntos iniciales divergen exponencialmente, y si las trayectorias se mantienen dentro de una región en el espacio, uno obtiene lo que se llama caos determinístico.

Estos dos conceptos permiten al método de búsqueda una acción aleatoria, a pesar de ser en realidad determinística, y sirve de gran utilidad para escapar de aquellas iteraciones en las que se

ha entrado en un ciclo, es decir, en un análisis del mismo óptimo local durante un número límite de iteraciones.

5.3.4. Procedimiento y método de escape

Así pues, el algoritmo propuesto que emplea estos conceptos se expone de la siguiente manera:

1. Evaluación de los posibles movimientos elementales a partir de la configuración actual
2. Búsqueda de la última configuración analizada y decisión sobre la ejecución del mecanismo de escape
3. En caso de no ejecutarse el escape, búsqueda del mejor movimiento
4. En caso afirmativo, ejecución de una serie de movimientos aleatorios

Cuando se lleva a cabo una repetición en el movimiento, la lista tabú aumenta, consiguiéndose así a largo plazo evitar en próximas iteraciones la recaída en un movimiento cíclico.

La estrategia de escape propuesta por Battiti (1994) consiste en realizar un determinado número de movimientos aleatorios, proporcionales al número promedio de movimientos entre ciclos.

```
procedure escape
begin
Clean the hashing memory structure.
Generate a random number of steps in the given range. rand returns a random no. in [0,1)
steps := 1 + (1+rand)×moving_average/2
for i=1 to steps do
    begin
    set (r_chosen,s_chosen) := random exchange of two units
    make_tabu(r_chosen,s_chosen)
    update_current_and_best
    Find the decrease in function value for all possible elementary moves.
    end
end
```

Ilustración 22. Función escape del algoritmo RTS en Pascal. (Battiti et al, 1994)

El pseudo-código de la función *escape* expuesto en la Figura (22), se resume en lo siguiente:

- Reinicio del desglose en la estructura de memoria
- Generar un número aleatorio en el rango (0,1)
- Durante un número determinado (*steps),* generar vecindades nuevas en la solución
- Actualizar la lista tabú.

En resumen, la búsqueda tabú reactiva supone una aplicación de los sistemas dinámicos al método clásico de la búsqueda tabú, aportando a ésta la independencia para generar cambios en la lista de movimientos tabú en función de la situación. Esto, permite una mayor versatilidad en cuanto a la búsqueda del óptimo, así como un incremento de la eficiencia.

6 BÚSQUEDA TABÚ REACTIVA APLICADA AL ACARREO TERRESTRE

6.1. Planteamiento del problema

El fin de este proyecto consiste en aplicar la búsqueda tabú reactiva, comentada y analizada en el capítulo anterior, al problema del acarreo diario con vehículos heterogéneos.

Ya se ha hablado de la dificultad de dicho problema, ya que es necesario trabajar con números elevados de clientes, surgiendo de cada uno de ellos un cierto número de tareas y operaciones de acarreo que deben ser ejecutadas cumpliendo ciertas restricciones, entre las cuales las más importantes son las ventanas temporales y el tamaño de la unidad de carga.

Es por ello que un método dinámico como el RTS ofrece, en un tiempo razonable como se apreciará más adelante, esa solución buscada, la cual sería complicada de obtener siguiendo otras técnicas de optimización.

Para comenzar con el planteamiento del problema, se muestran los parámetros que acotan el tamaño del mismo:

- Número de vehículos: El parámetro **num_veh** indica el tamaño de la flota de vehículos con la que cuenta el depósito. Dicha dimensión de la flota influirá en la capacidad de variación de la solución. El problema contará con más flexibilidad cuanto mayor sea en número de vehículos, pues habrá más opciones en cuanto a combinación de tareas. A modo de ejemplo, si la flota cuenta con tan solo un camión, el problema se reduciría a un problema similar al TSP, en el que el mismo vehículo debe pasar por todos los puntos. Por otro lado, una flota amplia permitirá una combinación de muchas rutas más pequeñas.
- Número de tareas: El parámetro **num_tasks** dimensiona el problema en cuanto al tamaño y dificultad. La solución variará dependiendo del número de tareas u operaciones a realizar, y de ello dependerá a su vez la elección de una flota de un tamaño diferente.
  Este proyecto cuenta con dos tipos de tareas: exportación e importación, cada una de ellas, dividida a su vez en dos tipos de operaciones: Recogida (Pick-up) y Entrega (Delivery), de las cuales en algunos casos, serán ficticias, como ya se ha comentado en el capítulo del acarreo terrestre.

Además de los parámetros que se enfocan en la dimensión del problema, y que conciernen a la teoría del acarreo, existen en este proyecto un segundo conjunto de parámetros, los cuales resultan de interés tanto a nivel computacional, como a la hora de permitir al usuario una mayor flexibilidad. Dichos parámetros son las penalizaciones.

En muchos métodos de optimización se recurre a penalizaciones, las cuales posteriormente se emplean en la función de evaluación para aportar más importancia a ciertos aspectos de la solución que en determinados casos pueden cobrar más o menos relevancia.

Es posible que en futuras aplicaciones, el programa en cuestión sea empleado para realizar la gestión de la flota, y sea interesante dar más importancia al cumplimiento de los horarios (ventanas temporales) que al precio del conjunto resultante de las rutas.

Las penalizaciones permiten asignar costes en función de los intereses del usuario, y además permiten al método descartar soluciones de una manera más rápida, para encontrar el óptimo de una manera más eficiente.

Así pues, se exponen las penalizaciones empleadas como parámetros:

- Penalización por número de camiones (penal_trucks): Se asigna un determinado coste por cada vehículo que se añade a la solución. Dicho coste se acumula con el número de vehículos empleados, y más adelante se añade a la evaluación de la solución. Con ello, el programa puede discernir entre la opción de utilizar más o menos vehículos para el acarreo.
- Penalización ventanas temporales (penal_retraso): Durante la búsqueda del óptimo, muchas de las soluciones incumplirán las restricciones relacionadas con las ventanas temporales, ya sea en cuanto a retrasos, o bien debido a llegadas más tarde de lo que el intervalo indicado expone. Es importante penalizar dicho incumplimiento, y aplicar un coste a la solución si esto ocurre. Dependiendo de la decisión del usuario, podrá asignarse una penalización más o menos alta para que el programa decida sobre la importancia del retraso en la búsqueda del óptimo.
- Penalización del orden: El parámetro **penal_orden** se emplea exclusivamente para dirigir la solución a la región de admisibilidad. Como se expuso anteriormente, es necesario realizar en primer lugar una recogida antes del reparto. Cualquier cliente debe tener antes la orden *pick-up* que la orden *delivery*. El inverso es inadmisible, y una manera de evitar dicha situación es penalizándola con un número de orden superior al rango de soluciones, para conseguir que el algoritmo huya de esas soluciones.

Como ya se ha comentado, las diferentes penalizaciones expuestas anteriormente se emplean a modo de coste que se incluye posteriormente en la función de evaluación.

En segundo lugar, cabe destacar el papel de los costes en el problema. Como ya se ha comentado, el objetivo es reducir los costes en el acarreo de los vehículos mediante la reducción de los kilómetros recorridos, así como de los vehículos empleados para las tareas.

Por ellos, los diferentes costes implicados son:

- Coste por kilómetro recorrido.

- Coste por vehículo

- Costes asociados a las penalizaciones (Ventanas temporales, tareas sin realizar)

Los costes se emplean en la función evaluación, la cual se explica en el siguiente apartado, y juegan el papel principal en el resultado (función objetivo).

6.2. Codificación general del algoritmo en Python

Se expone a continuación el esquema empleado para la resolución del problema:

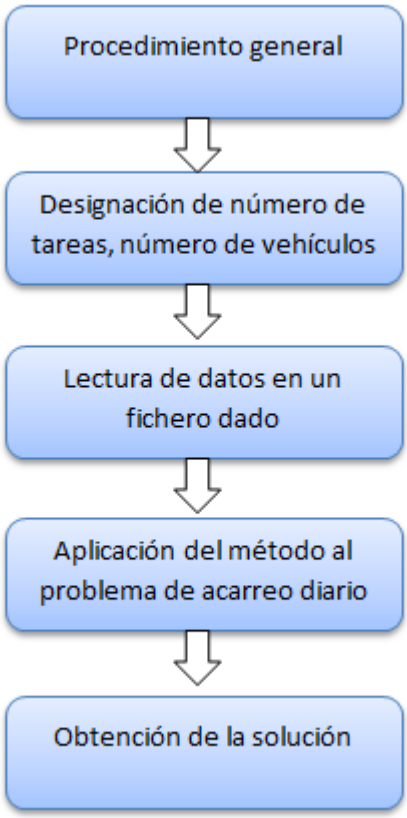


Ilustración 23. Esquema general resolución del problema (Elaboración propia)

En primer lugar, se designa valor a los diferentes parámetros indicados en el apartado anterior. Una vez indicados los diferentes pesos de las penalizaciones, y el número de vehículos y tareas, el programa pasa a leer los diferentes datos, obteniendo la información en este caso de una hoja Excel. En ella se halla toda la información pertinente. Mediante la librería **Pandas**, Python permite leer de forma sencilla datos de una hoja Excel:

Se expone a continuación la función Python encargada de leer el problema:

```
86      def read_problem(self, file_name, sheet_name):
87          self.problem_name = file_name
88          self.issue = sheet_name
89          self.T = pd.read_excel(file_name, sheet_name, header=0, usecols='A:N')
90          wb = openpyxl.load_workbook(filename=file_name, read_only=True)
91          ws = wb[sheet_name]
92          self.depot_xy = [ws['Q3'].value, ws['R3'].value]
93          self.depot_TW = [ws['Q4'].value, ws['R4'].value]
94
95          if self.version == 'tasks':
96              self.read_tasks()
97          elif self.version == 'operations':
98              self.read_tasks()
99              self.read_operations()
```

Ilustración 24. Funcion Read_problem codificada en Python

Una vez realizada la tarea de lectura de datos, el siguiente paso será adaptar dichos datos al problema que atañe a este proyecto.

Para ello, el programa Python cuenta, en primer lugar, con dos funciones llamadas **read_tasks** y **read_operations**, encargadas de registrar en una estructura de datos la información obtenida de la hoja de

cálculo. Los datos pertenecientes a las diferentes tareas quedan guardados y ordenados en una estructura

```
{'id': 'p3', 'id_n': 3, 'task_id': 3, 'type': 'pf', 'position': 'terminal', 'xy': [35.0, 35.0], 'x': 35.0, 'y': 35.0, 'TWe': 614.0, 'TWl': 704.0, 'st': 15.0, 'container_size': 1.0, 'charge': 0, 'paired_id': 'p3', 'paired_empty': None}
```

diccionario, en la cual, la información referente se ordena como se muestra en el siguiente ejemplo:

```
{'id': 'p%d' % (index + 1),
                    'id_n': (index + 1),
                    'task_id': (index + 1),
                    'type': 'pf',
                    'position': 'terminal',
                    'xy': [task_i['Ox'], task_i['Oy']],
                    'x': task_i['Ox'],
                    'y': task_i['Oy'],
                    'TWe': task_i['TWO1'],
                    'TWl': task_i['TWO2'],
                    'st': task_i['Load'],
                    'container_size': task_i['CS'],
                    'charge': 0,
                    'paired_id': 'p%d' % (index + 1),
                    'paired_empty': None}
```

Ilustración 25. Ejemplo de estructura diccionario para almacenar las operaciones

Se explicará más adelante cada una de las componentes que aparecen en la ilustración anterior, ya que es necesario comprenderlo para entender el funcionamiento de la función de evaluación.

En cada componente del diccionario queda registrada toda la información obtenida de la hoja de cálculo.

Una vez leído el problema, y ya adaptado al código Python, el siguiente paso del esquema ya es la búsqueda de la solución óptima.

Para ello, se exponen a continuación las funciones empleadas para el procedimiento:

- Generar solución (*Create_initial_solution*): Dicha función genera una solución inicial que, a lo largo del programa, irá variando dependiendo de la función objetivo que resulte.

Dicha solución tiene la siguiente forma:

['p2', 'p3', 'p4', 'p6', 'd3', 'd2', 'd4', 'd6', 'oe13', 'oe17', 'ief20', 'ie20', 'oef13', 'oef17']

Cada uno de los componentes es un identificador de operación, sabiendo que hay seis tipos diferentes:

a) Pick-up : p
b) Delivery: d
c) Out-empty : oe
d) In-empty: ie
e) Out-empty ficticio: oef
f) In-empty ficticio: ief

Dichos identificadores se encuentran en el diccionario comentado en el apartado anterior, y permiten acceder

al resto de informacion correspondiente a la operación en cuestión. Más adelante, en la función evaluación, se explica con detalle cómo trabaja el problema con dichos identificadores y el diccionario.

- Generar vecindad (*SWAP_neighbourhood*): Dicha función se encarga, como su propio nombre indica, de generar vecinos en la solución actual. Ya se comentó en profundidad en el capítulo de la búsqueda tabú en qué consiste la vecindad, y se resume en el intercambio de dos elementos para conseguir un nuevo vector candidato a contar con una función objetivo mejor.
  En el caso de este proyecto, la función *SWAP_neighbourhood* realiza intercambios de cada uno de los elementos con aquellos que se encuentran en posiciones posteriores en el vector. A continuación se muestra un esquema del funcionamiento de la vecindad empleada en este trabajo:

Ejemplo:

P2	**P4**	**P7**	**P9**	**Oef13**	**Oe15**	**Ie10**	**Ief10**	**D4**	**D7**	**D2**	**Oe13**	**D9**	**Oef15**
P2	P4	P7	P9	Oef13	D9	Ie10	Ief10	D4	D7	D2	Oe13	Oe15	Oef15

Ilustración 26. Vecindad aplicada a una solucion aleatoria del problema

- Función de evaluación (*eval_function*): La función de evaluación recibe la solución actual y devuelve el coste de dicha opción de ruteo de los vehículos. Dicho valor es el que posteriormente se compara, y el que permite decidir si dicha solución supone una mejora en el proceso de búsqueda.
  La codificación de esta función se basa en la creación de dos vectores :
  - **trucks**: formado por tantas componentes como número de vehículos, cada una de dichas componentes representa la ruta de un camión. Las operaciones se van asignando al camión que ofrece un menor valor de la función objetivo. El vector (lista) para un problema con 5 camiones es de la siguiente forma:
    - Trucks = [[ ], [ ], [ ], [ ], [ ]]
  - **Truck_list:** Al igual que la lista anterior, este vector tiene tantas componentes como número de vehículos impuestos por el usuario. Cada una de estas componentes es una lista de dos elementos, los cuales representan la capacidad del camión.
    Si el problema cuenta con una flota de cinco camiones:
    - Truck_list = [[0, 0], [0, 0], [0, 0], [0, 0], [0, 0]]

Estos dos vectores se han creado para decidir qué camiones se encuentran disponibles en cada momento. El procedimiento de asignación de vehículo, dependiendo del tipo de operación, es el siguiente:

Operaciones de pick-up

Mediante un bucle “for”, el programa busca un espacio vacío en alguno de los camiones, es decir, un “0” en alguna de las componentes de **truck_list.**

Al encontrar alguna componente nula, asigna a dicho elemento el número identificativo de dicha tarea, siempre y cuando se cumplan las restricciones (ventanas temporales y tamaño del contenedor).

Operaciones de delivery

En este caso, el programa busca en **truck_list** el número que identifica dicha tarea de delivery, y vuelve a anular la lista, lo que simula la liberación del contenedor. En la teoría, el camión pasaría a tener ese espacio vacío de nuevo.

Este procedimiento se aplica de la misma forma para tareas con contenedores vacíos (out-empty, in-empty), así como con las ficticias, pues simulan todas ellas recogidas y entregas de la unidad de carga.

En cuanto a la restricción relacionada con el tamaño de los contenedores, el programa distingue entre unidades de 20' y 40'.

Se supone que cada vehículo cuenta con capacidad para dos contenedores de 20', o bien un contenedor de 40'. Por ello, en caso de estar ante un contenedor de 40', el programa buscará aquella componente del vector **truck_list** en el que ambos elementos sean nulos.

Tras asignar cada una de las tareas a los diferentes camiones, el programa calcula los kilómetros que supone dicha operación para el vehículo.

Se explica en el siguiente apartado el procedimiento que el algoritmo sigue para calcular la función objetivo.

6.2.1. Cálculo de la función objetivo

La técnica que el programa Python emplea para el cálculo de la función objetivo se basa en una estructura diccionario, construida durante la lectura del problema.

Como se ha comentado anteriormente, todos los datos necesarios para la resolución del problema de acarreo diario se encuentran en una hoja de cálculo que el programa lee al comienzo de su ejecución.

Todos estos datos son guardados en un diccionario, y podrán ser utilizados por el algoritmo mediante el concepto de objeto.

Un doble bucle "for" recorre, por un lado, la solución actual; por el otro, recorre el diccionario generado al comienzo hasta encontrar la similitud entre identificadores. El código resulta de la siguiente forma:

```
for i in range(0, len(current_solution)):
    for j in range(0, len(self.O)):
        if current_solution[i] == self.O[j]['id']:
```

Ilustración 27. Bucle for para la búsqueda de operaciones en el diccionario

Una vez localizado el identificador deseado, se tiene acceso a toda la información necesaria para realizar los cálculos. Dicha información se detalla a continuación:

Tipo	Ox	Oy	Dx	Dy	TWO1	TWO2	TWD1	TWD2	Load	Unload	CS	id	id_E
2	41	49	35	35	0	720	273	513	15	15	1	1	11

Ilustración 28. Información disponible para cada tarea de acarreo

En dicha fila aparecen los siguientes datos:

- Tipo: La tarea puede ser de importación (Tipo 1) o exportación (Tipo 2)
- (Ox, Oy): Coordenadas del origen del cliente
- (Dx, Dy): Coordenadas de destino del cliente
- (TWO1, TWO2): Intervalo horario en el que es posible recoger el contenedor
- (TWD1, TWD2): Intervalo horario en el que es posible entregar el contenedor
- Load: Tiempo de carga del contenedor
- Unload: Tiempo de descarga
- CS: Tamaño del contenedor
- Id. Código de identificación de la tarea

Por tanto, una vez localizados los datos en el diccionario, del identificador analizado, puede calcularse la distancia entre el punto en que se encuentra el camión actualmente y el punto de destino. El resultado se actualiza en la lista **km_list.**

```
trucks[n].append(current_solution[i])

km_list[n] = km_list[n] + self.NORMA(posicion_actual[n], self.O[j]['xy'])

t_f[n] = t_f[n] + self.O[j]['st'] + 60*(self.NORMA(posicion_actual[n],
self.O[j]['xy'])/velocidad_truck)
```

Ilustración 29. Actualizacion en Python de km_list, t_f y posicion_actual

Además, se comprueba si dicho camión cumple con los tiempos para realizar la entrega en el horario permitido, ya que se conocen los intervalos de tiempo y se conoce el instante actual de cada camión.

Por último, como se comentó en apartados anteriores, se calculan las diferentes penalizaciones en base a los datos obtenidos tras el análisis de cada tarea asignada.

Una vez calculada la distancia, y realizada la suma de las diferentes penalizaciones, puede obtenerse el valor de la función objetivo.

6.3. Método de resolución: Búsqueda Tabú Reactiva

En este apartado se explica cómo se aplica este método de optimización al problema del acarreo diario.

Se ha expuesto en el apartado anterior la estructura del problema, así como las diferentes funciones que sirven para modelarlo, y posteriormente poder aplicarlo al algoritmo tabú reactivo.

Este algoritmo se basa en la generación de vecindades en la solución que se está analizando, para comprobar en cada iteración, si se consigue una reducción en la función objetivo, o por el contrario, se aleja del posible óptimo.

Para ello, el algoritmo tabú cuenta con una lista, en la cual se almacenan los movimientos más recientes. Así, se evita caer en óptimos locales, y se explora todo el área de soluciones posibles.

Además, como ya se ha comentado, este algoritmo en particular (RTS), discierne en el tamaño de dicha lista tabú si se da el caso de iteraciones cíclicas durante un tiempo previamente estipulado. La búsqueda tabú reactiva permite así, aportar independencia al algoritmo, para ahorrar tiempo de computación, y optimizar la función de la lista tabú en función de la situación del análisis.

Se definen dos parámetros relacionados con la búsqueda tabú:

- Tamaño de la lista tabú (SIZE_TABU). La característica que define a la búsqueda tabú reactiva es el tamaño variable de su lista de movimientos no prohibidos (lista tabú). En dicho algoritmo, la función que se encarga de actualizar la lista, se encargará también de decidir el tamaño óptimo de la misma, en función del historial de soluciones obtenidas.
  En este proyecto, dicha lista varía de tamaño atendiendo a la frecuencia de repetición de una solución determinada.

  La función implementada en Python se expone más adelante.
- Numero de iteraciones (ITERATIONS). Como mínimo, la búsqueda tabú debe emplear entre 500 y 1000 iteraciones. Dicho valor se deja como parámetro, y determina el número de veces que va a ejecutarse el algoritmo.

Se expone a continuación el esquema del procedimiento de búsqueda tabú reactiva aplicado al problema del acarreo:

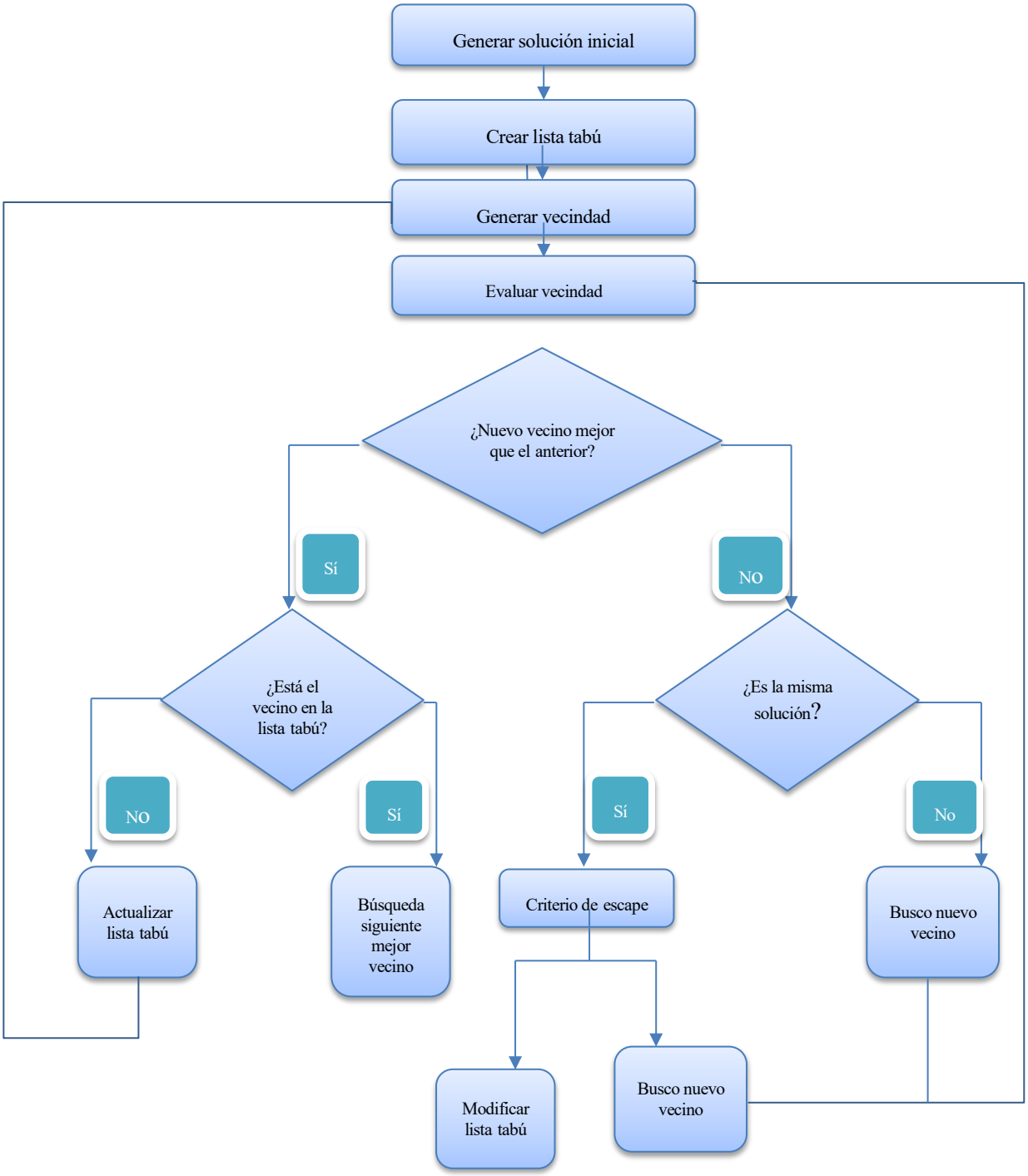


Ilustración 30. Diagrama RTS aplicado al problema de acarreo

6.3.1. Actualización de la lista tabú

En este proyecto, la función de actualización de la lista tabú cobra una función esencial: como se ha comentado con anterioridad, dicho algoritmo no solo se dedica a almacenar en memoria movimientos prohibidos, sino que decide el número de movimientos censurados que debe haber, atendiendo al número de repeticiones de cada una de las soluciones.

Se ha implementado en Python la función *update_tabu_list*, y se explica a continuación su funcionamiento.

Función *update_tabu_list*

La función *update_tabu_list* recoge como parámetros la solución actual, y un vector creado para almacenar el histórico de soluciones analizadas hasta el momento. Con ello, en cada iteración, busca en dicho vector el número de veces que la solución actual se ha repetido a durante la ejecución del algoritmo.

Según el número de repeticiones hasta el momento, la función update_tabu_list activa una serie de instrucciones que ponen en marcha la actualización del tamaño de la lista, dependiente de una variable denominada *chaos.*

La variable *Chaos* se encarga de fijar el límite de veces que es necesario repetirse una solución para que sea lícito cambiar el tamaño de la lista.

El funcionamiento es sencillo: Cuando el algoritmo caiga en un óptimo local, y una misma solución caiga en un número excesivo de repeticiones, esta función aumentará el tamaño de la lista. En caso contrario, el tamaño de la lista tabú disminuirá.

El aumento y disminución del tamaño están determinados por los parámetros *Increase* y *Decrease,* respectivamente. Dichos parámetros han sido fijados a 1, y se suman o restan al tamaño actual según proceda.

Se emplean para la implementación las siguientes variables:

Size_tabu: Tamaño inicial de la lista tabú
- Chaotic. Contador de frecuencia de repetición de soluciones
- MovAvg. Media móvil de repeticiones detectadas.
- GapRept. Espacio entre repeticiones consecutivas
- LastChange. Nº de iteraciones desde el último cambio
- LasTimeRept. Nº iteración de la última repetición.
- CurTimeRept. Nº iteración de la repetición actual

Los parámetros empleados en este mecanismo de búsqueda reactiva son:

- REP. Límite máximo de repeticiones
- Chaos. Límite máximo de soluciones que se repiten.
- Increase. Incremento de la lista tabú
- Decrease. Disminucion de la lista tabú
- GapMax. Espacio máximo entre repeticiones

Se expone a continuación el pseudo-código empleado para su implementación:

```
RTS Mechanism

(1) Búsqueda de repetición de S,
        If repeticion:
                GapRept = CurTimeRept- LasTimeRept
                Go to (2);
        Else go to (4).

(2) If repeticion de S > REP:
        set Chaotic = Chaotic+1;
        Else go to (3).
        If Chaotic > Chaos:
                Chaotic = 0
                Reconstruir vector current_solution (aleatorio)
                Evaluar nueva solucion
        Else go to (3).

(3) If GapRept < GapMax:
                SIZE_TABU = SIZE_TABU + Increase
                LastChange = 0
                MovAvg = 0.1×GapRept+ 0.9×MovAvg;
                Go to (5);
        Else go to (4).

(4) If LastChange > MovAvg:
                SIZE_TABU = SIZE_TABU - Decrease
                LastChange = 0;
        Else set LastChange = LastChange+1
        Almacena solucion S
        Go to (5).

(5) Stop the RTS mechanism.
```

Ilustración 31. Pseudo-código del mecanismo RTS

7 RESULTADOS

Una vez se han desarrollado los aspectos relacionados tanto con el problema como con el método de resolución empleado, se procede a exponer los diferentes resultados obtenidos mediante el programa implementado en Python.

El programa en cuestión obtiene los datos relevantes para la resolución de una hoja Excel llamada 'B2_R1_010_F.xlsx'.

Dicho documento consta de 7 enunciados, en los cuales aparecen variaciones relacionadas con las ventanas temporales de los diversos clientes, así como diferentes intervalos de las mismas.

Otra de las variaciones de los enunciados es el número de tareas que requieren operaciones con contenedores vacíos.

Resulta interesante especificar las características del ordenador con el que se pone a prueba el algoritmo, ya que influye en el tiempo de computación. Dichas características se exponen a continuación:

Ordenador: Packard Bell EasyNote TS44HR

Procesador: Intel® Core™ i5-2450M CPU @2.50GHz

Memoria RAM: 4,00GB

Sistema Operativo: Windows 10 Home 64bits.

A continuacion se mostrarán siete casos diferentes, los cuales han sido resueltos en primer lugar mediante la búsqueda tabú (TS) y posteriormente se ha aplicado la búsqueda tabú reactiva (RTS).

Se considera una flota máxima de ocho vehículos

El uso de cada camión adicional se penaliza con un coste de 10 u.m.

El km recorrido supone un coste de 1 u.m.

Las tablas mostrarán, para cada hoja Excel:

- Función objetivo
- Número de bypass realizados
- Número de camiones
- Tamaño final de la lista tabú
- Intervalo de las ventanas temporales

Se adjuntará además, para cada caso, una gráfica obtenida en Python con la evolución de la función objetivo y el número de Bypass considerados en cada iteración.

7.1. Resultados para 500 iteraciones y flota máxima de 8 camiones

- Caso 1.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	530.94	0	6	187	30/60
TS	636.28	1	6	10	30/60

Tabla 4. Resultados para el caso 1

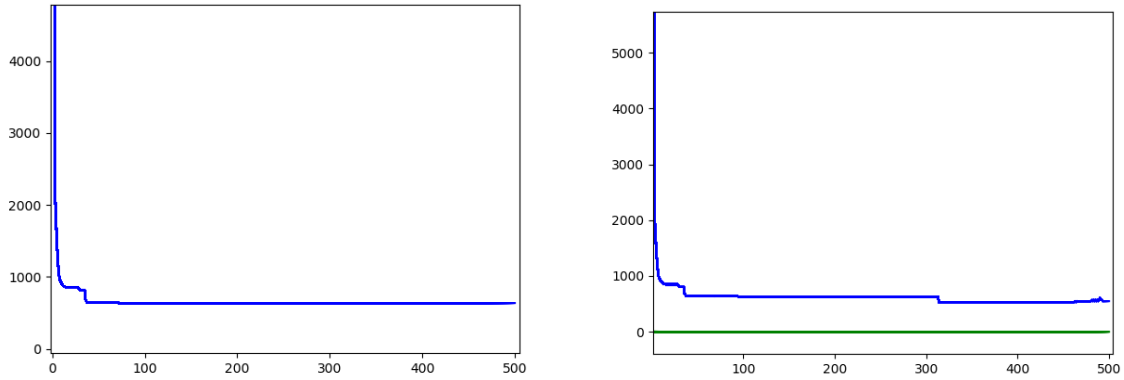


Ilustración 32. Izqda: Evolución de la f.o. mediante TS. Dcha: Mediante RTS

El algoritmo RTS (derecha) no haya una mejora en la solución hasta pasadas las 300 iteraciones, como se aprecia en la imagen ampliada. No se contemplan acciones de bypass, por lo que no se muestra la gráfica más ampliada.

Las ventanas temporales (30 ud. temporales en el caso de tareas con contenedores llenos, 60 para los contenedores vacíos), no permiten flexibilidad en la solución, ya que son intervalos de tiempo muy estrechos.

Diferencia de 100 uds. Monetarias entre el coste obtenido mediante la búsqueda tabú y el obtenido mediante la búsqueda tabú reactiiva

- Caso 2.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	517.86	0	7	128	60
TS	543.809	0	7	10	60

Tabla 5. Resultados para el caso 2

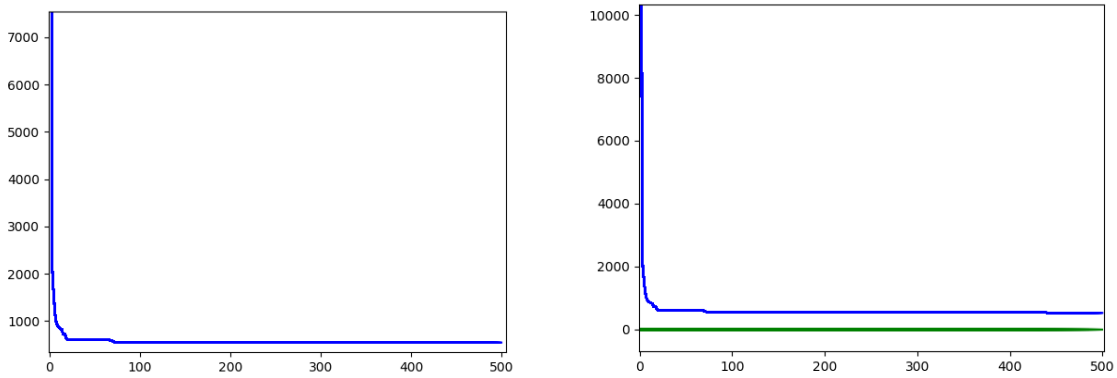


Ilustración 33. Izqda: Evolución de la f.o. con TS. Dcha: Gráfica RTS

- Caso 3.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	489.45	1	6	199	90
TS	492.38	1	6	10	90

Tabla 6. Resultados para el caso 3.

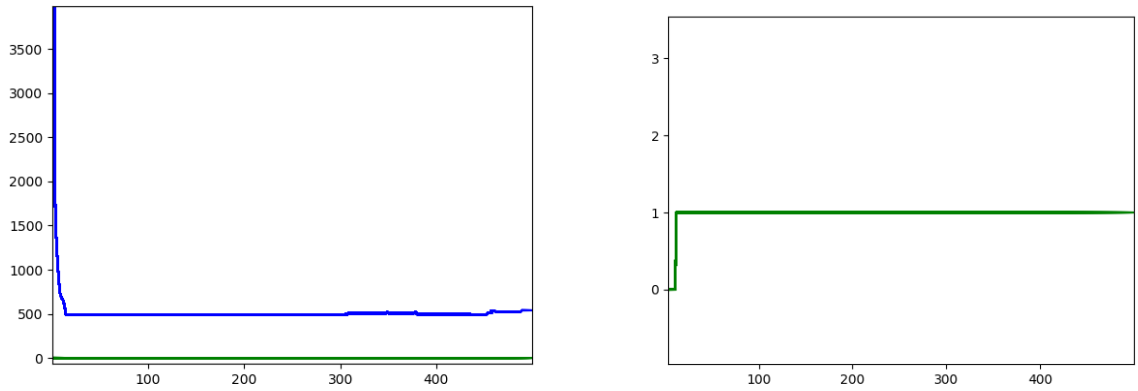


Ilustración 34. Izqda. Evolución (RTS) de f.o. para el caso 3. Dcha: Evolución del bypass

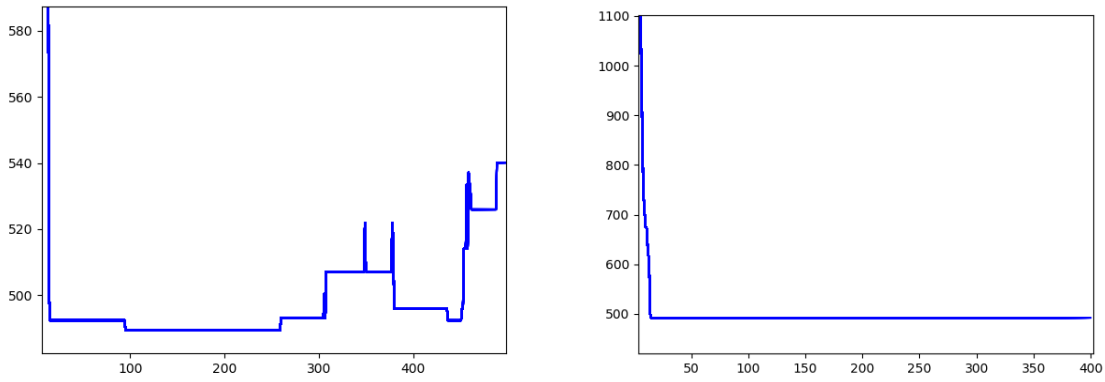


Ilustración 35. Comparativa de evolución de la f.o entre RTS (izqda.) y TS (dcha.)

- Caso 4.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	500.39	0	6	202	120
TS	534.37	0	7	10	120

Tabla 7. Resultados para el caso 4.

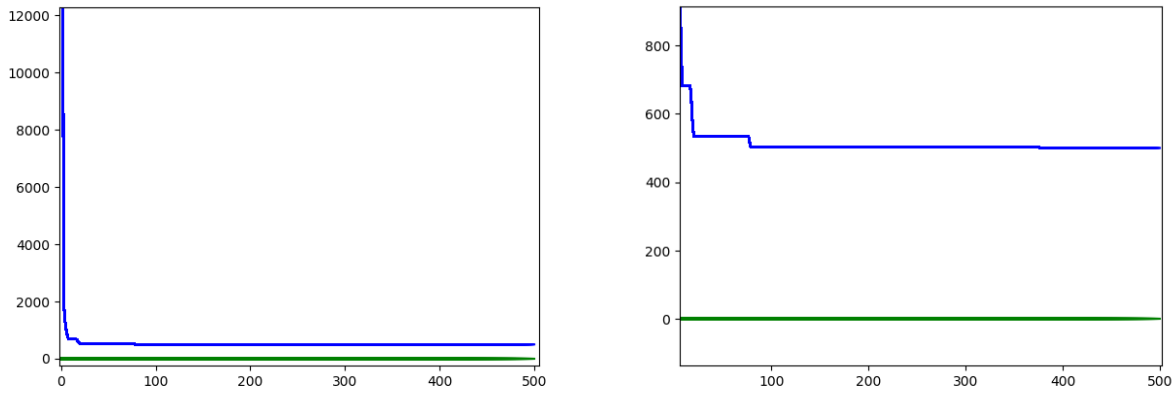


Ilustración 36. Izqda: Evolucion f.o. para el caso 4. Dcha: Gráfica ampliada

- Caso 5.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	363.71	4	5	168	240
TS	468.15	4	4	10	240

Tabla 8. Resultados para el caso 5.

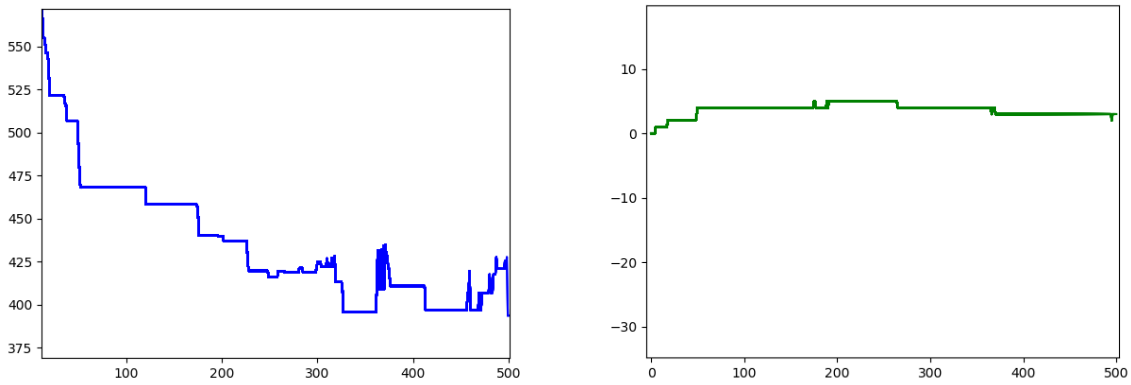


Ilustración 37. Izqda: Evolución f.o. en el caso 5. Dcha. Evolución Bypass (RTS)

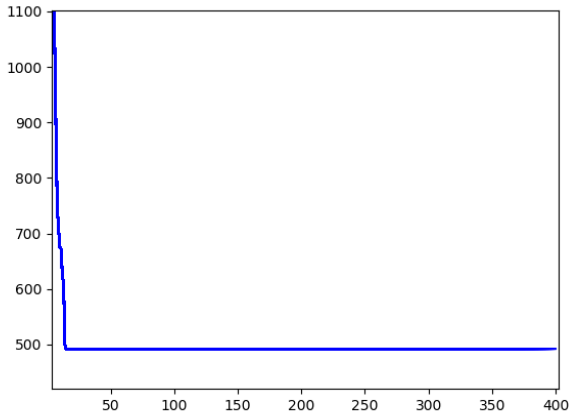


Ilustración 38. Evolución de f.o. en para el caso 5 (TS)

En este caso 5, se muestran las gráficas resultado por separado, y tras ser ampliadas. Así, se consigue apreciar el resultado, el cual indica una clara disminución de los costes, y una clara diferencia con respecto al resto.

En este supuesto práctico, el método tabú reactivo ha recurrido al abandono del óptimo local en sucesivas ocasiones.

Por otra parte, el bypass cobra protagonismo, llegándose a realizar 5 acciones alrededor de la iteración 200.

De este quinto caso obtenemos el mejor resultado, y el que mejor refleja el comportamiento del algoritmo.

En la ilustración 35 se aprecia el desarrollo de la búsqueda tabú básica. Mediante este método se consigue una función objetivo de 468 u.m

- Caso 6.

	F.O	BYPASS	FLOTA	LISTA TABU	TW
RTS	507.54	0	6	226	120
TS	654.34	0	7	10	120

Tabla 9. Resultados para el caso 6.

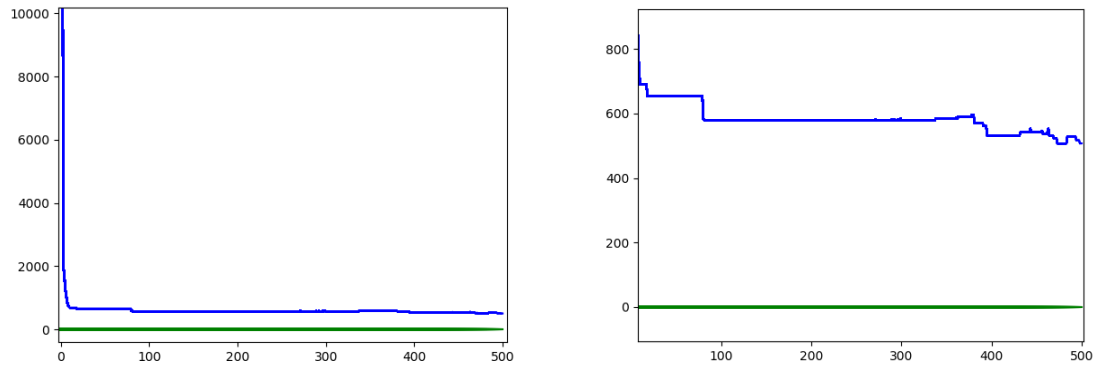


Ilustración 39. Izqda: Evolucion f.o. para el caso 6. Dcha: Gráfica ampliada (RTS)

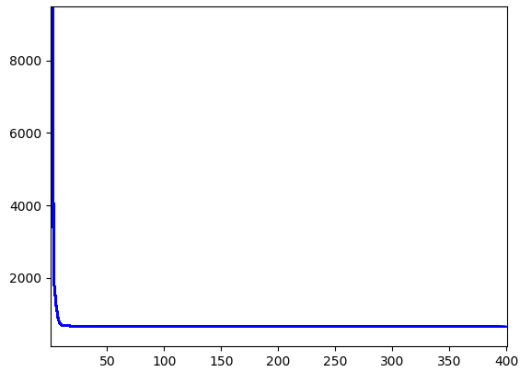


Ilustración 40. Evolución de la f.o. mediante TS

- Caso 7.

F.O	BYPASS	FLOTA	LISTA TABU	TW
425.71	2	5	194	240

Tabla 10. Resultados para el caso 7.

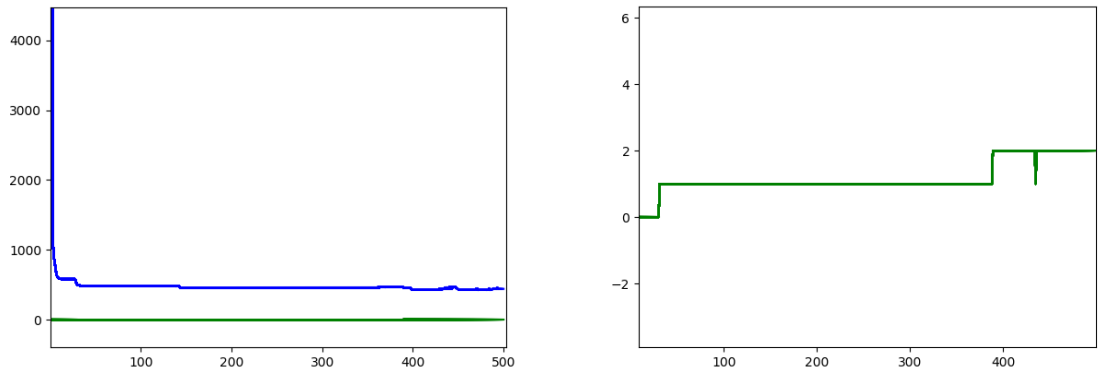


Ilustración 41 Izqda: Evolución f.o. en el caso 5. Dcha. Evolución Bypass

En este caso 7 se aprecia de nuevo cómo el aumento de las ventanas temporales influye en la función objetivo.

Además, se aprecia de nuevo cómo el algoritmo escapa de los óptimos locales, y propone nuevas áreas para encontrar el nuevo óptimo. El programa discierne entre el número de tareas ficticias solucionadas mediante bypass, concluyéndose en 2 al final de la ejecución.

8 CONCLUSIONES

Por último, queda comentar las conclusions obtenidas durante el desarrollo de este proyecto.

Para ello, se resume en primer lugar cada uno de los puntos en los que se ha profundizado.

En primer lugar, se ha indagado en el transporte intermodal, y en concreto en el acarreo terrestre, para conocer el ámbito en el que este trabajo puede aportar mejoras en la logística, y se ha comentado la importancia del transporte por carretera a la hora de conseguir flexibilidad y un mayor alcance en las tareas de distribución.

Seguidamente se ha expuesto un capítulo sobre el acarreo terrestre, analizando sus principales características, a saber: Conceptos de exportación e importación, el camión como vehículo empleado, consideración de los contenedores de 20' y 40' como unidad de carga, ventanas temporales, tareas ficticias y concepto de bypass.

Se ha realizado más adelante un análisis de los diversos métodos de optimización, distinguiendo entre métodos exactos, heurísticas y metaheurísticas, y en concreto se ha analizado el método de la búsqueda tabú.

Por último, se ha estudiado la búsqueda tabú reactiva, la cual ha sido la herramienta para la resolución de este problema.

Tras ver los resultados obtenidos mediante un código Python, en el capítulo anterior, se obtienen las siguientes conclusiones:

- Al variar la lista tabú mediante el mecanismo RTS, el programa escapa de óptimos locales, analizando nuevas áreas de soluciones. Se obtiene así una mejora en la función objetivo, lo cual se traduce en una disminución de los costes.
- La técnica del bypass cobra importancia a medida que aumentan las tareas con contenedores vacíos, ya que los vehículos tienen la opción de trasladar dicha unidad de carga de un cliente a otro sin necesidad de pasar por el depósito.
- A pesar de contar con una flota ficticia de 8 vehículos, se aprecia que en el caso más costoso, el programa requiere solamente 7, llegando a necesitar 5 vehículos en los últimos ejemplos.
- Cuanto mayor es el intervalo de las ventanas temporales, menor es el coste de la actividad. Una mayor franja horaria de los clientes permite una mayor versatilidad a la hora de mejorar las rutas de los vehículos.

8.1. Lineas de futuro

Este método de optimización supone una introducción a una técnica que cada vez se desarrolla con mayor velocidad: la inteligencia artificial.

La búsqueda tabú reactiva se basa en el uso de memoria del algoritmo tabú, para discernir en la toma de decisión de, en este caso, el tamaño de la lista en la que almacena los movimientos prohibidos.

Este proyecto supone una pequeña introducción al campo de la inteligencia artificial, y el desarrollo teórico del mismo resulta un campo de gran interés para aplicar dicha técnica, y conseguir mejores resultados requiriendo a su vez tiempos de computación mucho menores.

REFERENCIAS

Arango Serna, M. D., Gil Gómez, H., & Zapata Cortes, J. A. (2009). Lean logistics applied to transport in the miner sector. *Boletín de Ciencias de la Tierra*(25), 121-136.

Autor. (2012). Este es el ejemplo de una cita. *Tesis Doctoral, 2*(13).

Autor, O. (2001). Otra cita distinta. *revista*, pág. 12.

Battiti, R., & Tecchiolli, G. (1994). The Reactive Tabu Search. *ORSA Journal on computing*(6(2)), 126-140.

Briheche, Y., Barbaresco, F., Bennis, F., & Chablat, D. (2020). *Branch-and-Bound Method for Just-in-Time Optimization of Radar Search Patterns.*

Cheung. (2008). An attribute- decision model for cross-border drayage problem. *Transportation Research Part E- Logistics and Transportation Review*(44 (2)), 217-234.

Escribano Iglesias, M., & Giraldo Carbajo, A. (s.f.). *Departamento de Matemática Aplicada para las Tecnologías de la Información y las Comunicaciones*. Obtenido de http://www.dma.fi.upm.es/docencia/plan96/sistemas_dinamicos/

Escudero Santana, A. (2013). *Mejoras en el transporte intermodal: Optimización en tiempo real del acarreo terrestre, Capítulo 5. El problema del Acarreo Terrestre.* Tesis Doctoral, ETS de Ingeniería, Sevilla.

Fowkes, A. S., & Nash, C. (1991). *Analysing demand for rail travel.* Avebury: Aldershot, Hants; Brookfield, Vt.

Glover, F. (1986). Future Paths for Integer Programming and Links to Artificial Intelligence. *Computers & Operations Research*(13), 533-549.

Gronalt, M., Hartl, R. F., & Reinmann, M. (2003). New savings based algorithms for time constrained pickup and delivery of full truckloads. *European Journal of Operational Research, 151*(3), 520-535.

Guerrero Herruzo, J., & Escudero Santana, A. (2014). *Resolución del problema del Acarreo Terrestre mediante la implementación de un algoritmo basado en la metaheurística de Búsqueda Tabú.* Trabajo Fin de Grado, ETSI. Universidad de Sevilla, Departamento de Organización y Gestión de Empresas II, Sevilla.

Horst, R., & Tuy, H. (1996). *Global optimization: deterministic approaches.* Springer.

Jula, H., Dessouky, M., Ioannou, P., & Chassiakos, A. (2005). Container movement by trucks in metropolitan networks: modeling and optimization. *Transportation Research Part E- Logistics and Transportation Review*(41(3)), 235-259.

Macharis, C., & Bontekoning, Y. M. (2004). Opportunities for OR in intermodal freight transport research: A review. *European Journal of operational research, 153*(2), 400-416.

Martí, R. (2003). Procedimientos metaheurísticos en optimización combinatoria. En *Matemátiques* (Vol. 1, págs. 3-62). Valencia: Universidad de Valencia.

Melián, B., Pérez, J. A., & Moreno Vega, J. M. (2003). Metaheurísticas: Una visión global. Inteligencia Artificial. *Revista Iberoamericana de Inteligencia Artificial, 7*(19), 0.

Morlok, E. K., & Spasovic, L. N. (1994). Redesigning rail-truck intermodal drayage operations for enhanced service and cost performance. *Journal of the Transportation Research Forum*.

Mullin, T., & Belotti, P. (2016). Using branch-and-bound algorithms to optimize selection of a fixed-size breeding population under a relatedness constraint. *Tree Genetics & Genomes*(12(4)).

Nemhauser, G. L., & Wosley, L. A. (1988). *Integer and combinatorial optimization.* New York: Wiley-Interscience.

Nierat, P. (1997). *Market area of rail-truck terminals: Pertinence of the spatial theory. Transportation*

Research Part A: Policy and Practice (Vol. 31).

Nozick, L. K., & Morlok, E. K. (1997). Model for medium-term operations planning in an intermodal rail-truck service. *Transportation Research Part A-Policy and Practice*(31(2)), 91-107.

Resende, M., & Gonzalez-Velarde, J. (2003). GRASP: Procedimientos de búsqueda miopes aleatorizados y adaptativos. *Inteligencia Artificial: Revista Iberoamericana de Inteligencia Artificial*(19), 67-76.

Sait, S. M., & Youssef, H. (1999). *Iterative computer algorithms: and their applications in engineering.*

Smilowtiz. (2006). Multi-resource routing with exible tasks: an application in drayage operations. *IIE Transactions*(38(7)), 577-590.

Spasovic, L., & Morlok, E. K. (1993). Using marginal costs to evaluate drayage rates in rail-truck intermodal service. *Transportation Research Record*(1383).

Taylor. (2002). An analysis of intermodal ramp selection methods. *Transportation research Part E- Logistics and Transportation Review*(38(2)), 117-134.

Tricas García, F. (2015). *www.webdiis.unizar.es.* Obtenido de http://webdiis.unizar.es/asignaturas/APD/wp/wp-content/uploads/2013/09/151201simulatedAnnealing2.pdf (9/20, 13/20)

Van Duin, R., & Van Ham, H. (1998). Three-stage modeling approach for the design and organization of intermodal services. *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics*, *4*, págs. 4051-4056. San Diego.

Wu, Y. -C. (2007). "Contemporary logistics education: an international perspective". . *International Journal of Physical Distribution & Logistics Management., 37*, 504- 528.

ANEXOS

Se expone a continuación el código Python realizado para la obtención de resultados.

```
#-*- coding: utf-8 -*-

"""
Multi-size Daily Drayage Problem

Created: 25 april 2019
@author: Alejandro Escudero-Santana
Universidad de Sevilla

"""

import openpyxl
import random
import pandas as pd
import scipy.spatial.distance as distance

#from gurobipy import *

from itertools import permutations
import matplotlib.pyplot as plt
import numpy as np

# import copy
# from openpyxl.utils.dataframe import dataframe_to_rows
# import math



class TABU:
    SIZE_TABU = 2              # Parameters
    ITERATIONS = 500
    SOLUTIONS = []              # Esta lista alberga las soluciones que van
siendo analizadas, para poder detectar los casos de repeticion
    tabu_list = []
    iteration = 0

    #   Inicializo los parámetros relacionados con la búsqueda tabú reactiva



    def eval_function(self, current_solution):
        pass

    #   Defino funcion que calcula la norma para conocer el modulo de los
diferentes trayectos

    def NORMA(self, a, b):
        v = [(float(i - j)) for i, j in zip(a,b)]
        return np.linalg.norm(v)
```

```
    def SWAP_neighbourhood(self, current_solution):
        neighbourhood = []
        swap = []
        neighbourhood_value = []
        for i in range(len(current_solution)):
            for j in range(i + 1, len(current_solution)):
                ti = min(current_solution[i], current_solution[j])
                tj = max(current_solution[i], current_solution[j])
                if [ti,tj] not in self.tabu_list:        #
                    neighbour = current_solution[0:i] + [current_solution[j]]
+ current_solution[i + 1:j] + [
                        current_solution[i]] + current_solution[j + 1:]
                    neighbourhood.append(neighbour)
                    neighbourhood_value.append(self.eval_function(neighbour))
                    swap.append([current_solution[i], current_solution[j]])

        return zip(neighbourhood_value, swap, neighbourhood)

    def update_tabu_list(self, swap, current_solution, SOLUTIONS, Chaotic,
LastChange, MovAvg):

        self.tabu_list.append(swap)

        i = 0
        repeticion = 0
        REP = 3
        #Chaotic = 0
        Chaos = 6
        GapRept = 0
        LasTimeRept = 0
        CurTimeRept = 0
        #MovAvg = 0
        #LastChange= 0
        Increase = 1
        Decrease = 1
        GapMax = 25

        while i < len(SOLUTIONS)-2:
            if SOLUTIONS[i] == SOLUTIONS[len(SOLUTIONS)-1]:
                repeticion += 1
                LasTimeRept = i
                CurTimeRept = len(SOLUTIONS)
            i += 1
        if repeticion > 0:
            GapRept = CurTimeRept - LasTimeRept

            if repeticion > REP:
                Chaotic += 1
            else:
                if GapRept < GapMax:
                    self.SIZE_TABU = self.SIZE_TABU + Increase
                    LastChange = 0
                    MovAvg = 0.1 * GapRept + 0.9 * MovAvg
                    stop_RTS_mechanism = True
                else:

                    if LastChange > MovAvg:
                        self.SIZE_TABU = self.SIZE_TABU - Decrease
                        LastChange=0
                    else:
                        LastChange += 1
                        stop_RTS_mechanism = True
```

```
                if Chaotic > Chaos and not stop_RTS_mechanism:
                    Chaotic = 0
                    random.shuffle(current_solution)
                    print(current_solution)
                    stop_RTS_mechanism = True
                else:
                    if GapRept < GapMax:
                        self.SIZE_TABU = self.SIZE_TABU + Increase
                        LastChange = 0
                        MovAvg = 0.1*GapRept + 0.9*MovAvg
                        stop_RTS_mechanism = True
                    else:
                        if LastChange > MovAvg:
                            self.SIZE_TABU = self.SIZE_TABU - Decrease
                            LastChange = 0
                        else:
                            LastChange +=1
                            stop_RTS_mechanism = True

        while len(self.tabu_list) > self.SIZE_TABU:
            self.tabu_list.pop(0)

    def check_end_condition(self):
        self.iteration += 1
        print(self.iteration)
        end_condition = False
        if self.iteration > self.ITERATIONS:
            end_condition = True

        return end_condition

class TABU_MSDDP(TABU):
    def __init__(self, id=None, num_tasks=None, num_veh=None, version=None,
penal_orden=None, penal_trucks= None, penal_retraso=None, coste_km= None):
        self.id = id
        self.nv = num_veh
        self.nt = num_tasks
        self.version = version
        self.penal_orden = penal_orden
        self.penal_trucks = penal_trucks
        self.penal_retraso = penal_retraso
        self.coste_km = coste_km

    def read_problem(self, file_name, sheet_name):
        self.problem_name = file_name
        self.issue = sheet name
        self.T = pd.read_excel(file_name, sheet_name, header=0,
usecols='A:N')
        wb = openpyxl.load_workbook(filename=file_name, read_only=True)
        ws = wb[sheet_name]
        self.depot_xy = [ws['Q3'].value, ws['R3'].value]
        self.depot_TW = [ws['Q4'].value, ws['R4'].value]

        if self.version == 'tasks':
            self.read_tasks()
        elif self.version == 'operations':
            self.read_tasks()
            self.read_operations()

    def read_tasks(self):
        self.T_I = self.T[self.T.Tipo == 1]
```

```
        self.T_E = self.T[self.T.Tipo == 2]
        self.T_OE = self.T[self.T.Tipo == 3]
        self.T_IE = self.T[self.T.Tipo == 4]

        self.num_tasks = len(self.T_I) + len(self.T_E) + len(self.T_OE) +
len(self.T_IE)
        self.num_tasksWD = len(self.T_I) + len(self.T_E) + len(self.T_OE) +
len(self.T_IE)

    def read_operations(self):
        # id = 1
        self.O_P = []
        self.O_D = []
        self.O_P_V = []
        self.O_D_V = []
        self.O_P_F = []
        self.O_D_F = []
        self.O = []

        for index, task_i in self.T_I.iterrows():
            operation_pickup = {'id': 'p%d' % (index + 1),
                                'id_n': (index + 1),
                                'task_id': (index + 1),
                                'type': 'pf',
                                'position': 'terminal',
                                'xy': [task_i['Ox'], task_i['Oy']],
                                'x': task_i['Ox'],
                                'y': task_i['Oy'],
                                'TWe': task_i['TWO1'],
                                'TWl': task_i['TWO2'],
                                'st': task_i['Load'],
                                'container_size': task_i['CS'],
                                'charge': 0,
                                'paired_id': 'p%d' % (index + 1),
                                'paired_empty': None}

            # id += 1

            pair = 'oe%d' % task_i['id_E'] if not np.isnan(task_i['id_E'])
else None
            operation_delivery = {'id': 'd%d' % (index + 1),
                                  'id_n': (index + self.num_tasks + 1),
                                  'task_id': (index + 1),
                                  'type': 'df',
                                  'position': 'customer',
                                  'xy': [task_i['Dx'], task_i['Dy']],
                                  'x': task_i['Dx'],
                                  'y': task_i['Dy'],
                                  'TWe': task_i['TWD1'],
                                  'TWl': task_i['TWD2'],
                                  'st': task_i['Unload'],
                                  'container_size': -task_i['CS'],
                                  'charge': 0,
                                  'paired_id': 'p%d' % (index + 1),
                                  'paired_empty': pair}
            # id += 1

            self.O_P.append(operation_pickup)
            self.O_D.append(operation_delivery)

        for index, task_i in self.T_E.iterrows():
            task_i = task_i.to_dict()
```

```
            pair = 'ie%d' % task_i['id_E'] if not np.isnan(task_i['id_E'])
else None
            operation_pickup = {'id': 'p%d' % (index + 1),
                                'id_n': (index + 1),
                                'task_id': (index + 1),
                                'type': 'pf',
                                'position': 'customer',
                                'xy': [task_i['Ox'], task_i['Oy']],
                                'x': task_i['Ox'],
                                'y': task_i['Oy'],
                                'TWe': task_i['TWO1'],
                                'TWl': task_i['TWO2'],
                                'st': task_i['Load'],
                                'container_size': task_i['CS'],
                                'charge': 0,
                                'paired_id': 'd%d' % (index + 1),
                                'paired_empty': pair}
            # id += 1

            operation_delivery = {'id': 'd%d' % (index + 1),
                                  'id_n': index + self.num_tasks + 1,
                                  'task_id': index + 1,
                                  'type': 'df',
                                  'position': 'terminal',
                                  'xy': [task_i['Dx'], task_i['Dy']],
                                  'x': task_i['Dx'],
                                  'y': task_i['Dy'],
                                  'TWe': task_i['TWD1'],
                                  'TWl': task_i['TWD2'],
                                  'st': task_i['Unload'],
                                  'container_size': -task_i['CS'],
                                  'charge': 0,
                                  'paired_id': 'p%d' % (index + 1),
                                  'paired_empty': None}

            # id += 1

            self.O_P.append(operation_pickup)
            self.O_D.append(operation_delivery)

        for index, task_i in self.T_OE.iterrows():
            task_i = task_i.to_dict()

            operation_pickup = {'id': 'oe%d' % (index + 1),
                                'id_n': (index + 1),
                                'task_id': (index + 1),
                                'type': 'oe',
                                'position': 'customer',
                                'xy': [task_i['Ox'], task_i['Oy']],
                                'x': task_i['Ox'],
                                'y': task_i['Oy'],
                                'TWe': task_i['TWO1'],
                                'TWl': task_i['TWO2'],
                                'st': task_i['Load'],
                                'container_size': task_i['CS'],
                                'charge': 0,
                                'paired_id': 'oef%d' % (index + 1),
                                'paired_empty': None}
            # id += 1

            operation_delivery = {'id': 'oef%d' % (index + 1),
```

```
                                'id_n': index + self.num_tasks + 1,
                                'task_id': index + 1,
                                'type': 'oef',
                                'position': 'depot',
                                'xy': [task_i['Dx'], task_i['Dy']],
                                'x': task_i['Dx'],
                                'y': task_i['Dy'],
                                'TWe': task_i['TWD1'],
                                'TWl': task_i['TWD2'],
                                'st': task_i['Unload'],
                                'container_size': -task_i['CS'],
                                'charge': 0,
                                'paired_id': 'oe%d' % (index + 1),
                                'paired_empty': None}

            # id += 1

            #             self.O_OE.append(operation_pickup)
            #             self.O_OE_F.append(operation_delivery)
            self.O_P_V.append(operation_pickup)
            self.O_D_F.append(operation_delivery)

        for index, task_i in self.T_IE.iterrows():
            task_i = task_i.to_dict()

            operation_pickup = {'id': 'ief%d' % (index + 1),
                                'id_n': (index + 1),
                                'task_id': (index + 1),
                                'type': 'ief',
                                'position': 'depot',
                                'xy': [task_i['Ox'], task_i['Oy']],
                                'x': task_i['Ox'],
                                'y': task_i['Oy'],
                                'TWe': task_i['TWO1'],
                                'TWl': task_i['TWO2'],
                                'st': task_i['Load'],
                                'container_size': task_i['CS'],
                                'charge': 0,
                                'paired_id': 'ie%d' % (index + 1),
                                'paired_empty': None}
            # id += 1

            operation_delivery = {'id': 'ie%d' % (index + 1),
                                  'id_n': index + self.num_tasks + 1,
                                  'task_id': index + 1,
                                  'type': 'ie',
                                  'position': 'customer',
                                  'xy': [task_i['Dx'], task_i['Dy']],
                                  'x': task_i['Dx'],
                                  'y': task_i['Dy'],
                                  'TWe': task_i['TWD1'],
                                  'TWl': task_i['TWD2'],
                                  'st': task_i['Unload'],
                                  'container_size': -task_i['CS'],
                                  'charge': 0,
                                  'paired_id': 'ief%d' % (index + 1),
                                  'paired_empty': None}

            # id += 1
            self.O_P_F.append(operation_pickup)
            self.O_D_V.append(operation_delivery)
```

```
        depot = {'id': 'depot',
                 'id_n': 0,
                 'task_id': 0,
                 'type': 'depot',
                 'position': 'depot',
                 'xy': self.depot_xy,
                 'x': self.depot_xy[0],
                 'y': self.depot_xy[1],
                 'TWe': self.depot_TW[0],
                 'TWl': self.depot_TW[1],
                 'st': 0,  # <-------------------------------------
                 'container_size': 0,
                 'charge': 0,
                 'paired_id': None,
                 'paired_empty': None}

        self.O = self.O_P + self.O_D + self.O_P_V + self.O_D_V + self.O_P_F +
self.O_D_F + [depot]

        self.df_O = pd.DataFrame(self.O)

    def create_initial_solution(self):
        current_solution = []
        for order in self.O:
            current_solution.append(order['id'])
        return current_solution

    #   Escribo self donde habria puesto tabusearch

    #   Para tener en cuenta las ventanas temporales vamos a fijar una
velocidad de 50 km/h simulando los limites de velocidad en una ciudad


    def eval_function(self, current_solution):
        penal=0
        pickup_done = []
        delivery_done = []
        for h in range(len(self.O)):
            pickup_done.append(0)
            delivery_done.append(0)

        for i in range(0, len(current_solution)):
            for j in range(0, len(self.O)):
                if current_solution[i] == self.O[j]['id']:
                    if self.O[j]['id'][0] == 'p':
                        if pickup_done[int(self.O[j]['id'][1:3])] == 0:
                            pickup_done[int(self.O[j]['id'][1:3])] = 1

                    elif self.O[j]['id'][0] == 'd' and self.O[j]['id'][1] !=
'e':
                        if pickup_done[int(self.O[j]['id'][1:3])] == 0:
                            penal += self.penal_orden
                            #solution[i] = 'p%d' % int(self.O[j]['id'][1:3])
# Realizo el cambio en current solution
                            #pickup done[int(self.O[j]['id'][1:3])] = 1
                        else:
                            delivery_done[int(self.O[j]['id'][1:3])] = 1

            #   Hacemos lo mismo con las operaciones de contenedor vacio TIPO
3

                    if self.O[j]['id'][0:2] == 'oe' and self.O[j]['id'][2] !=
```

```
'f':
                        #if delivery_done[int(self.O[j]['id'][2:4])-10] == 1:
                            if pickup_done[int(self.O[j]['id'][2:4])] == 0:
                                pickup_done[int(self.O[j]['id'][2:4])] = 1

                    elif self.O[j]['id'][0:3] == 'oef':
                        if pickup_done[int(self.O[j]['id'][3:5])] == 0:
                            penal += self.penal_orden

            #   Hacemos lo mismo con las operaciones de contenedor vacio TIPO
4

                    if self.O[j]['id'][0:3] == 'ief':
                            if pickup_done[int(self.O[j]['id'][3:5])] == 0:
                                pickup_done[int(self.O[j]['id'][3:5])] = 1

                    elif self.O[j]['id'][0:2] == 'ie' and self.O[j]['id'][2]
!= 'f':
                        if pickup_done[int(self.O[j]['id'][2:4])] == 0:
                            penal += self.penal_orden

#   Vamos a definir una lista llamada truck_list en la que se ve el estado
ACTUAL de los camiones

        bypass = 0
        truck_list = []
        trucks = []
        km_list = []
        t_f = []
        posicion_actual = []

        for i in range(0, self.nv):
            truck_list.append([0, 0])
            trucks.append([])
            km_list.append(0)
            t_f.append(0)
            posicion_actual.append([35, 35])

        velocidad_truck = 50
        penaliza_retraso = 0

        for i in range(0, len(current_solution)):
            for j in range(0, len(self.O)):
                asignado = False                                    #  Cada 'p'
se descarta una vez ha sido asignada en un camion
                enviado = False                                     #  Cada 'd'
se descarta una vez enviado en un camion
                if current_solution[i] == self.O[j]['id']:
                    if self.O[j]['id'][0] == 'p':
                        for n in range(0, len(truck_list)):
                            if self.O[j]['container_size'] == 1:
                                for l in range(0, 2):
                                    if truck_list[n][l] == 0 and not
asignado:

                                        #   LA SIGUIENTE CONDICION COMPRUEBA
SI EL CAMION LLEGA AL CLIENTE DENTRO DE LA VENTANA TEMPORAL

                                        if (((t_f[n] + 60 *
self.NORMA(posicion_actual[n], self.O[j]['xy'])/ velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60 * self.NORMA(posicion_actual[n],
self.O[j]['xy'])/ velocidad_truck) >= self.O[j]['TWe'])):
```

```
                                        truck_list[n][l] =
int(current_solution[i][1:3])


trucks[n].append(current_solution[i])

                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n] + self.O[j]['st']
+ 60*(self.NORMA(posicion_actual[n], self.O[j]['xy'])/velocidad_truck)
                                        posicion_actual[n] =
self.O[j]['xy']             #   Ese punto tendrá que estar dentro del
intervalo de la ventana temporal
                                        asignado = True

                                    elif (((t_f[n] + 60 *
self.NORMA(posicion_actual[n], self.O[j]['xy'])/ velocidad_truck) <
self.O[j]['TWe'])):
                                        truck_list[n][l] =
int(current_solution[i][1:3])


trucks[n].append(current_solution[i])

                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                        t_f[n] = self.O[j]['TWe'] +
self.O[j]['st']
                                        posicion_actual[n] =
self.O[j]['xy']  # Ese punto tendrá que estar dentro del intervalo de la
ventana temporal
                                        asignado = True
                                    else:

                                        #   SI EL CAMION LLEGA FUERA DEL
INTERVALO DE TIEMPO, APLICAMOS UNA PENALIZACION

                                        penaliza_retraso +=
self.penal_retraso
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        posicion_actual[n] =
self.O[j]['xy']
                                        truck_list[n][l] =
int(current_solution[i][1:3])


                                        asignado = True

trucks[n].append(current_solution[i])

                            if self.O[j]['container_size'] == 2:
                                if truck_list[n][0] == 0 and truck_list[n][1]
== 0 and not asignado:
                                    if (((t_f[n] +
60*self.NORMA(posicion_actual[n], self.O[j]['xy'])/velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60*self.NORMA(posicion_actual[n],
self.O[j]['xy'])/ velocidad_truck) >= self.O[j]['TWe'])):
```

```
                                        truck_list[n][0] =
int(current_solution[i][1:3])
                                        truck_list[n][1] =
int(current_solution[i][1:3])

                                        trucks[n].append(current_solution[i])
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)
                                        posicion_actual[n] = self.O[j]['xy']
                                        asignado = True
                                        print('Contenedor de 40 recogido')

                                    elif (((t_f[n] + 60 *
self.NORMA(posicion_actual[n], self.O[j]['xy'])/ velocidad_truck) <
self.O[j]['TWe'])):

                                        truck_list[n][0] =
int(current_solution[i][1:3])
                                        truck_list[n][1] =
int(current_solution[i][1:3])

                                        trucks[n].append(current_solution[i])

                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                        t_f[n] = self.O[j]['TWe']+
self.O[j]['st']
                                        posicion_actual[n] = self.O[j]['xy']
                                        asignado = True
                                    else:
                                        penaliza_retraso +=
self.penal_retraso
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][0] =
int(current_solution[i][1:3])
                                        truck_list[n][1] =
int(current_solution[i][1:3])

                                        asignado = True
                                        trucks[n].append(current_solution[i])
                    if self.O[j]['id'][0] == 'd' and self.O[j]['id'][1] !=
'e':
                        for n in range(0, len(truck_list)):
                            for l in range(0, 2):
                                if truck_list[n][l] ==
int(current_solution[i][1:3]) and not enviado:
                                    if (((t_f[n] +
60*(self.NORMA(posicion_actual[n], self.O[j]['xy']))/velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60*(self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >= self.O[j]['TWe'])):
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] +
60*(self.NORMA(posicion_actual[n], self.O[j]['xy'])/velocidad_truck)
```

```
                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0

                                        enviado = True
                                        trucks[n].append(current_solution[i])

                                    elif (((t_f[n] +
60*(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck) <
self.O[j]['TWe'])):
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                        t_f[n] = self.O[j]['TWe']+
self.O[j]['st']

                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0

                                        enviado = True
                                        trucks[n].append(current_solution[i])

                                    else:
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        penaliza_retraso +=
self.penal_retraso
                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0

                                        enviado = True
                                        trucks[n].append(current_solution[i])

                    if self.O[j]['id'][0:2] == 'oe' and self.O[j]['id'][2] !=
'f':
                        for n in range(0, len(truck_list)):
                            if self.O[j]['container_size'] == 1:
                                for l in range(0, 2):
                                    if truck_list[n][l] == 0 and not
asignado:
                                        if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck))
<=self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >=self.O[j]['TWe'])):


                                            truck_list[n][l] =
int(current_solution[i][2:4])

trucks[n].append(current_solution[i])
                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                            t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)
                                            posicion_actual[n] =
self.O[j]['xy']
                                            asignado = True

                                        elif (((t_f[n] + 60 *
```

```
self.NORMA(posicion_actual[n], self.O[j]['xy'])/ velocidad_truck) <
self.O[j]['TWe'])):
                                                truck_list[n][l] =
int(current_solution[i][2:4])


trucks[n].append(current_solution[i])

                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                                t_f[n] = self.O[j]['TWe']+
self.O[j]['st']
                                                posicion_actual[n] =
self.O[j]['xy']  # Ese punto tendrá que estar dentro del intervalo de la
ventana temporal
                                                asignado = True
                                            else:

                                                #   SI EL CAMION LLEGA FUERA DEL
INTERVALO DE TIEMPO, APLICAMOS UNA PENALIZACION

                                                penaliza_retraso +=
self.penal_retraso
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                                t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                                posicion_actual[n] =
self.O[j]['xy']
                                                truck_list[n][l] =
int(current_solution[i][2:4])

                                                asignado = True

trucks[n].append(current_solution[i])

                            elif self.O[j]['container_size'] == 2:
                                if truck_list[n][0] == 0 and truck_list[n][1]
== 0 and not asignado:
                                    if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >= self.O[j]['TWe'])):
                                        truck_list[n][0] =
int(current_solution[i][2:4])
                                        truck_list[n][1] =
int(current_solution[i][2:4])

                                        trucks[n].append(current_solution[i])
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)
                                        posicion_actual[n] = self.O[j]['xy']
                                        asignado = True
                                        print('Contenedor de 40 recogido')
                                    elif (((t_f[n] + 60 *
self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck)
<self.O[j]['TWe'])):
```

```
                                        truck_list[n][0] =
int(current_solution[i][2:4])
                                        truck_list[n][1] =
int(current_solution[i][2:4])

                                        trucks[n].append(current_solution[i])

                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                        t_f[n] = self.O[j]['TWe']+
self.O[j]['st']
                                        posicion_actual[n] = self.O[j]['xy']
                                        asignado = True

                                    else:
                                        penaliza_retraso +=
self.penal_retraso
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][0] =
int(current_solution[i][1:3])
                                        truck_list[n][1] =
int(current_solution[i][1:3])

                                        asignado = True
                                        trucks[n].append(current_solution[i])
                    if self.O[j]['id'][0:3] == 'oef':
                        if i < len(current_solution)-1:
                            if current_solution[i+1][0:3] == 'ief':
                                    for n in range(0, len(truck_list)):
                                        for l in range(0,2):
                                            if truck_list[n][l] ==
int(current_solution[i][3:5]):
                                                truck_list[n][l] =
int(current_solution[i+1][3:5])
                                                bypass += 1
                            else:
                                for n in range(0, len(truck_list)):
                                    for l in range(0, 2):
                                        if truck_list[n][l] ==
int(current_solution[i][3:5]) and not enviado:
                                            if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck))
<=self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >= self.O[j]['TWe'])):

                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n],self.O[j]['xy'])
                                                t_f[n] = t_f[n]+
self.O[j]['st'] + 60 * (self.NORMA(posicion_actual[n],self.O[j]['xy']) /
velocidad_truck)

                                                posicion_actual[n] =
self.O[j]['xy']
                                                truck_list[n][l] = 0

                                                enviado = True
```

```
trucks[n].append(current_solution[i])
                                            elif (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck)
<self.O[j]['TWe'])):
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                                t_f[n] = self.O[j]['TWe']+
self.O[j]['st']

                                                posicion_actual[n] =
self.O[j]['xy']
                                                truck_list[n][l] = 0

                                                enviado = True

trucks[n].append(current_solution[i])

                                            else:
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                                t_f[n] = t_f[n]+
self.O[j]['st'] + 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) /
velocidad_truck)

                                                penaliza_retraso +=
self.penal_retraso
                                                posicion_actual[n] =
self.O[j]['xy']
                                                truck_list[n][l] = 0

                                                enviado = True

trucks[n].append(current_solution[i])


                        else:
                            for n in range(0, len(truck_list)):
                                for l in range(0, 2):
                                    if truck_list[n][l] ==
int(current_solution[i][3:5]) and not enviado:

                                        if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >= self.O[j]['TWe'])):

                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                            t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                            posicion_actual[n] =
self.O[j]['xy']
                                            truck_list[n][l] = 0
                                            enviado = True

trucks[n].append(current_solution[i])
                                        elif (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck) <
self.O[j]['TWe'])):
```

```
                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                            t_f[n] = self.O[j]['TWe']+
self.O[j]['st']

                                            posicion_actual[n] =
self.O[j]['xy']
                                            truck_list[n][l] = 0

                                            enviado = True

trucks[n].append(current_solution[i])

                                        else:
                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                            t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck)

                                            penaliza_retraso +=
self.penal_retraso
                                            posicion_actual[n] =
self.O[j]['xy']
                                            truck_list[n][l] = 0

                                            enviado = True

trucks[n].append(current_solution[i])

                    if self.O[j]['id'][0:3] == 'ief':
                        if current_solution[i-1][0:3] == 'oef':
                            pass
                        else:
                            for n in range(0, len(truck_list)):
                                if self.O[j]['container_size'] == 1:
                                    for l in range(0, 2):
                                        if truck_list[n][l] == 0 and not
asignado:
                                            if (((t_f[n] + 60 *
self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck))
<=self.O[j]['TWl']) and (((t_f[n] + 60 *
self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck)
>=self.O[j]['TWe'])):

                                                truck_list[n][l] =
int(current_solution[i][3:5])

trucks[n].append(current_solution[i])
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                                t_f[n] = t_f[n]+
self.O[j]['st'] + 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) /
velocidad_truck)
                                                posicion_actual[n] =
self.O[j]['xy']
                                                asignado = True
                                            elif (((t_f[n] +
60*(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck) <
self.O[j]['TWe'])):
                                                truck_list[n][l] =
int(current_solution[i][3:5])
```

```
trucks[n].append(current_solution[i])
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n],self.O[j]['xy'])
                                                t_f[n] = t_f[n]+
self.O[j]['st'] + 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) /
velocidad_truck)
                                                posicion_actual[n] =
self.O[j]['xy']
                                                asignado = True
                                            else:
                                                #   SI EL CAMION LLEGA FUERA
DEL INTERVALO DE TIEMPO, APLICAMOS UNA PENALIZACION

                                                penaliza_retraso +=
self.penal_retraso
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n],self.O[j]['xy'])
                                                t_f[n] = t_f[n]+
self.O[j]['st'] + 60 * (self.NORMA(posicion_actual[n],self.O[j]['xy']) /
velocidad_truck)

                                                posicion_actual[n] =
self.O[j]['xy']
                                                truck_list[n][l] =
int(current_solution[i][3:5])

                                                asignado = True

trucks[n].append(current_solution[i])

                                    elif self.O[j]['container_size'] == 2:
                                        if truck_list[n][0] == 0 and
truck_list[n][1] == 0 and not asignado:
                                            if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck))
<=self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >=self.O[j]['TWe'])):

                                                truck_list[n][0] =
int(current_solution[i][3:5])
                                                truck_list[n][1] =
int(current_solution[i][3:5])


trucks[n].append(current_solution[i])
                                                km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                                t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)
                                                posicion_actual[n] =
self.O[j]['xy']
                                                asignado = True
                                                print('Contenedor de 40
recogido')

                                            elif (((t_f[n] + 60 *
self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck)
<self.O[j]['TWe'])):

                                                truck_list[n][0] =
int(current_solution[i][3:5])
```

```
                                            truck_list[n][1] =
int(current_solution[i][3:5])


trucks[n].append(current_solution[i])

                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                            t_f[n] = self.O[j]['TWe']+
self.O[j]['st']
                                            posicion_actual[n] =
self.O[j]['xy']
                                            asignado = True
                                        else:
                                            km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                            t_f[n] = t_f[n]+ self.O[j]['st']
+ 60 * (self.NORMA(posicion_actual[n],self.O[j]['xy']) / velocidad_truck)

                                            penaliza_retraso +=
self.penal_retraso
                                            posicion_actual[n] =
self.O[j]['xy']
                                            truck_list[n][0] =
int(current_solution[i][3:5])
                                            truck_list[n][1] =
int(current_solution[i][3:5])

                                            enviado = True

trucks[n].append(current_solution[i])

                    if self.O[j]['id'][0:2] == 'ie' and self.O[j]['id'][2] !=
'f':

                        for n in range(0, len(truck_list)):
                            for l in range(0, 2):
                                if truck_list[n][l] ==
int(current_solution[i][2:4]) and not enviado:

                                    if (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck)) <=
self.O[j]['TWl']) and (((t_f[n] + 60 * (self.NORMA(posicion_actual[n],
self.O[j]['xy'])) / velocidad_truck) >= self.O[j]['TWe'])):

                                        km list[n] = km list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0
                                        enviado = True
                                        trucks[n].append(current_solution[i])
                                    elif (((t_f[n] + 60 *
(self.NORMA(posicion_actual[n], self.O[j]['xy'])) / velocidad_truck) <
self.O[j]['TWe'])):
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])

                                        t_f[n] = self.O[j]['TWe']+
```

```
self.O[j]['st']

                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0

                                        enviado = True
                                        trucks[n].append(current_solution[i])

                                    else:
                                        km_list[n] = km_list[n] +
self.NORMA(posicion_actual[n], self.O[j]['xy'])
                                        t_f[n] = t_f[n]+ self.O[j]['st'] + 60
* (self.NORMA(posicion_actual[n], self.O[j]['xy']) / velocidad_truck)

                                        penaliza_retraso +=
self.penal_retraso
                                        posicion_actual[n] = self.O[j]['xy']
                                        truck_list[n][l] = 0

                                        enviado = True
                                        trucks[n].append(current_solution[i])

        fo = 0

        penal_camiones = 0
        for i in range(0, len(km_list)):
            fo = fo + self.coste_km*km_list[i]

        for i in range(0, len(trucks)):
            if trucks[i]:
                penal_camiones += self.penal_trucks
        fo = fo + penal + penal_camiones + penaliza_retraso

        return fo, km_list, trucks, bypass


def main():

    global Chaotic, LastChange, MovAvg
    Chaotic = 0
    LastChange = 0
    MovAvg=0


    SOLUTIONS = [[]]
    Bypass = []
    #tabu_list_evolution = []
    HISTORICO_CURRENT_SOLUTION = []
    HISTORICO = []
    tabusearch = TABU_MSDDP(version='operations',num_veh=8,
penal_orden=10000, penal_trucks=10, penal_retraso=100, coste_km=1)
    tabusearch.read_problem('B2_R1_010_F.xlsx','07')
    best_solution_eval = float("inf")
    best_solution = []
    current_solution = tabusearch.create_initial_solution()


    end_condition = False
    while not end_condition:

        current_solution_eval = tabusearch.eval_function(current_solution)[0]
```

```
        #print(current_solution)

        #print(current_solution_eval)

        HISTORICO_CURRENT_SOLUTION.append(current_solution_eval)

        SOLUTIONS.append(current_solution)
        print('Current solution guardada en el vector SOLUTIONS')


        neighbourhood = tabusearch.SWAP_neighbourhood(current_solution)

        current_solution_value, swap, current_solution = min(neighbourhood)

        if current_solution_eval < best_solution_eval:
            best_solution_eval = current_solution_eval
            best_solution = current_solution[:]



        tabusearch.update_tabu_list(swap, current_solution, SOLUTIONS,
Chaotic, LastChange, MovAvg)


        end_condition = tabusearch.check_end_condition()
        HISTORICO.append(best_solution_eval)
        Bypass.append(tabusearch.eval_function(current_solution)[3])


        plt.plot(HISTORICO_CURRENT_SOLUTION, color='b', label="HISTORICO DE
SOLUCION ACTUAL")
        plt.plot(Bypass, color='g', label="Bypass realizados")

        plt.legend(('f(x)',), bbox_to_anchor=(1.6, 0.5))

        print('Lista tabu actualizada:',tabusearch.tabu_list)
        print('Vector de kms actualizado:',
tabusearch.eval_function(current_solution)[1])
        print('Vector de ordenes actualizado: ',
tabusearch.eval_function(current_solution)[2])
        print('Numero de BYPASS realizados:',
tabusearch.eval_function(current_solution)[3])


        print(best_solution)
        print('Valor de la funcion objetivo: ')
        print(best solution eval)
        print('Tamaño de la lista tabu: ', tabusearch.SIZE_TABU)
    plt.show()
    plt.show()


if __name__ == "__main__":
            main()
```