PONTIFICIA UNIVERSIDAD CATÓLICA DE VALPARAÍSO
FACULTAD DE INGENIERÍA
ESCUELA DE INGENIERÍA INFORMÁTICA

"APLICACIÓN DE HEURÍSTICAS PARA EL INVENTORY ROUTING PROBLEMS"

VANESSA INÉS VIVANCO NAVARRO

INFORME FINAL DEL PROYECTO
PARA OPTAR AL TÍTULO PROFESIONAL DE
INGENIERO CIVIL EN INFORMÁTICA

DICIEMBRE, 2008

Pontificia Universidad Católica de Valparaíso
Facultad de Ingeniería
Escuela de Ingeniería Informática

"APLICACIÓN DE HEURÍSTICAS PARA EL INVENTORY ROUTING PROBLEMS"

VANESSA INÉS VIVANCO NAVARRO

Profesor Guía: **Guillermo Nicolás Cabrera Guerrero**

Profesor Co-referente: **Broderick Crawford Labrin**

Carrera: **Ingeniería Civil en Informática**

Diciembre, 2008

Dedicatoria

Dedicado a mis padres por todo su apoyo incondicional y constante.
A mi tía quien ha sido una segunda madre para mí y
a mi hermano que también ha estado ahí en
esta larga etapa que es la Universidad.

Vanessa Vivanco Navarro.

Agradecimientos

A mi profesor Guía quien mantuvo su confianza en mí a lo largo de todo este proceso. A mi familia quienes han sido parte fundamental en mi desarrollo profesional y a todas las personas que han estado apoyándome en este camino.

Resumen

El Vendor Managed Inventory es una práctica de la cadena de abastecimiento donde el inventario es monitoreado, planificado y gestionado por el proveedor a nombre de la organización que lo consume. El Inventory Routing Problems capta las características básicas de la situación al aplicarse el VMI, el control de inventario tanto del mismo proveedor como de los clientes que éste tiene, junto con el manejo de los vehículos y las rutas a utilizar le da complejidad a este problema.

Se presenta un esquema de dos etapas para la resolución del VMI las cuales son de planificación de la demanda, y la otra de enrutamiento de vehículos donde es en esta etapa donde se implementarán las heurísticas, y luego se integraran a lasa heurísticas del manejo del inventario, todo esto sustentado en una arquitectura propuesta para el VMI.

Abstract

Vendor Managed Inventory is a supply chain's practice where the inventory is monitored, planned and managed by the supplier representing the organization that consumes it. Inventory Routing Problems captures the basic characteristics of the situation because the VMI is applied to that, the inventory control of the same supplier and the customers that it has, with the vehicles driving and the routes to use gives complexity to this problem.

A Problem of two stages is showed to the VMI's solving, which are: demand's scheduling and vehicles routing. The last stage named is where the heuristics will be implemented and then they will be integrated to the inventory's heuristics. All of this are sustained on an architecture proposed for VMI.

Índice

Lista de Abreviaturas o Siglas

CPFR: Planeamiento Colaborativo, Proyectado y de Reabastecimiento.
CVRP: Capacitated Vehicle Routing Problem.
ECR: Respuesta eficiente al consumidor.
EDI: Intercambio electrónico de datos.
EOQ: Economic Order Quantity
IRP: Inventory Routing Problem.
LS: Búsqueda Local.
SA: Simulated Annealing.
SIRP: Stochastic Inventory Routing Problem
TS: Tabu Search.
VMI: Vendor Managed Inventory.

Índice de Figuras

Lista de Tablas

Lista de Ecuaciones

Capítulo 1

Introducción

1.1 Introducción

El rol de la gestión de la logística está cambiando. Muchas compañías se están dando cuenta que un valor para el cliente puede, en parte, ser creado a través de la disponibilidad del producto, la puntualidad y la consistencia de la entrega, así como también la facilidad de la realización de los pedidos. En consecuencia, desde la década de los 90 [1], la logística está empezando a ser reconocida como un esencial elemento de la satisfacción del cliente en un número creciente de mercados.

El Vendor Managed Inventory (VMI) es un ejemplo de está creación de valor en la logística, mientras que el Inventory Routing Problems (IRPs) es el nombre que se le da al problema de integrar el inventario y el enrutamiento de vehículos. VMI trata de una entidad central encargada de tomar decisiones referidas al inventario de sus clientes (ubicados en diferentes áreas geográficas) e integrarlas con el manejo de rutas de las flotas de vehículos que el proveedor posee para el manejo del stock de sus clientes.

El VMI difiere de la administración convencional del inventario por lo siguiente: de la manera convencional son los clientes los encargados de manejar el stock de sus propios inventarios, con esto se refiere a que el cliente es el encargado de indicar por medio de ordenes al proveedor cuando requiere que se realice una reposición, indicando la demanda que el cliente estime necesaria. Esto último puede conllevar varias desventajas, una de ellas es que las ordenes no lleguen de manera uniforme al proveedor, el cual sólo se encarga de recibirlas y preparar el producto para satisfacerlas, no se preocupa si el stock del cliente se mantiene en un nivel deseable o si el tiempo en que se ha demorado en llegar esta orden resulta ser mayor a lo que el cliente pueda desear, por lo que no es posible reconocer las ordenes urgentes de las que no lo son. En cambio en la implementación del VMI, al ser el proveedor quien controla el inventario de sus clientes, es el responsable por el mantenimiento de un nivel deseable en ellos y es quien decide a qué clientes deben reponer, con qué frecuencia, y la demanda de producto a entregar [2]. Para tomar estas decisiones el proveedor debe tener acceso a una gran cantidad de información relevante, por ejemplo, el nivel actual del inventario del cliente, los niveles deseables del inventario de estos, el comportamiento de la demanda, entre otras cosas. A su vez también debe tener conocimiento de la capacidad y disponibilidad de los vehículos y conductores en determinados tiempos.

El IRPs es un problema de optimización de combinatoria de tipo NP-Hard (NP-duro) [3]. Cuando se refiere a un problema de clase de complejidad NP, indicamos que existe un algoritmo que resuelve el problema en un tiempo polinómico pero solamente en una máquina no determinista, por lo cual la solución en una máquina normal el tiempo se hace exponencial y por lo tanto intratable [4]. Ahora si a esta clasificación la referimos como NP-hard se indica que el problema de clase NP es posible reducirlo a un tiempo polinomial de resolución a P. Una de las formas de permitir lo anteriormente dicho es por medio de aplicación de heurísticas, cual nos entrega un valor muy cercano al óptimo en tiempos razonables de ejecución.

En el actual proyecto se propone una arquitectura para el VMI, la cual está formada por cuatro módulos: Interfaz, Proyección, Información e IRP. Estos cuatros módulos engloban el funcionamiento de transferencia de información y toma de decisiones necesarias en la descripción del VMI. El módulo de Información es el encargado de la transferencia hacia y desde el cliente, como también la seguridad de ésta. El módulo de proyección es el módulo que ayudará al proveedor a proyectarse hacia el futuro, simulando demanda o hacer un mejor uso de los recursos disponibles. El módulo de Interfaz es el encargado de la interacción de los usuarios con el sistema, o de un módulo con otro. Y por último el módulo de IRP, el cual se encarga del problema de la integración de inventario con el manejo de vehículos y la toma de decisiones. En esta arquitectura se basará para realizar una comparación entre una situación utilizando VMI y sin utilizarlo. El módulo a implementar para lograr lo mencionado anteriormente es el módulo de "IRP". Cabe mencionar que la implementación de este módulo requiere simular la implementación de los módulos de Información y de Interfaz para obtener los datos de entrada y la entrega de resultados.

Para la implementación del módulo IRP se tomará la demanda de los clientes como ordenes de entrega para la situación sin el VMI, con esto nos referimos que el proveedor debe cumplir con entregar los productos exactamente como se lo piden, y para la situación con VMI se pretende mejorar el uso de los vehículos disponibles del proveedor, como una regla de inventario, enviar lo máximo posible en los vehículos, esto se logrará hacer gracias al conocimiento del proveedor del inventario del cliente y la aproximación de la futura demanda de estos, eligiendo a los clientes más convenientes para agregarlos en la ruta y a largo plazo reducir el tiempo de frecuencia de entrega de los productos a ellos. En ambos casos es necesario la implementación del un problema de CVRP en el cual se implementarán las heurísticas Tabu Search y Simulated Annealing (y luego comprobar cual de estas funciona mejor), para luego a este problema irle agregando la elección de la demanda transformando este CVRP en un IRP.

1.2 Objetivos

1.2.1 Objetivo General

Proponer una Arquitectura para la práctica del Vendor Managed Inventory (VMI) e implementar una solución para el Inventory Routing Problem (IRP) a través de heurísticas.

1.2.2 Objetivos Específicos

- Comprender la práctica del VMI para luego diseñar una arquitectura para ésta.
- Analizar el Inventory Routing Problems y obtener un modelo matemático para su implementación.
- Analizar Tabu Search y Simulated Annealing para solucionar el problema del IRP y diseñar una solución.
- Implementar las Heurísticas en instancias del VRP para escoger solamente la que de mejores resultados.
- Implementar el diseño de solución con la heurística escogida para el IRPs.
- Comparar costos al utilizar VMI y al no utilizarlo.

El presente informe tendrá la siguiente división de su contenido: El capítulo 2 se presenta el estado del arte del problema que se desea solucionar con el proyecto, se explicaran de que se trata el VMI y se introducirá más profundamente en IRP, el cual es el problema donde se le aplicaran las heurísticas. El capítulo 3 se presenta de manera detallada las heurísticas a utilizar para resolver el problema indicado en el capítulo anterior, estas heurísticas son Tabu Search y Simulated Annealing.

En el capítulo 4 se presenta una descripción de la arquitectura propuesta para el VMI y el modelo matemático utilizado para la implementación del módulo IRP de la arquitectura. Luego en el capítulo 5 se presenta el diseño de la solución, en el cual se indica la forma en que se utilizarán las heurísticas mencionadas anteriormente, y como se pretende obtener la lista de los clientes a visitar.

En el capítulo 6 se presenta las pruebas y resultados obtenidos, se muestran los resultados de las pruebas realizadas para comprobar el funcionamiento de las heurísticas con instancias conocidas y los resultados entregados de un caso de prueba para ver las diferencias entre el uso del VMI y la forma tradicional.

Finalmente en el capítulo 7 se presentan las conclusiones obtenidas e ideas para futuros trabajos de proyecto, tomando como base éste.

Capítulo 2

Estado del Arte

2.1 Vendor Managed Inventory

Como se ha presentado anteriormente en la introducción, la logística cada vez está tomando mayor importancia en los diferentes tipos de mercados, ya que se han reconocido que la rentabilidad y el crecimiento de los ingresos están directamente relacionados con la eficiencia de la cadena de suministro. El Vendor Managed Inventory (VMI) es un ejemplo de la creación de valor en la logística.

El VMI es una práctica de la cadena de abastecimiento donde el inventario es monitoreado, planificado y gestionado por el proveedor a nombre de la organización que lo consume, basándose en la demanda esperada y en los niveles de inventarios mínimos y máximos previamente pactados. Tradicionalmente, el éxito en la gestión de la cadena de abastecimiento se deriva del entendimiento y gestión del diferencial existente entre el costo de inventario y el nivel de servicio. Los proyectos de VMI pueden dar mejorías a lo largo de ambas dimensiones.

El VMI comenzó en el comercio al por menor y creció a partir de Respuesta Eficiente al Consumidor (ECR), donde la satisfacción del consumidor o en lugar de ello, la expectativa de disposición de inventario para el consumidor es una forma importante para tener un margen competitivo. A finales de los años 80 los grandes almacenes, como por ejemplo Wal-Mart, automatizaron el VMI, provocando un surgimiento de éste [5]. Ahora está progresando gradualmente hacia formas basadas en la asociación estratégica. Esta influencia la manera en que las compañías planifican su inventario, evolucionando desde el Planeamiento Colaborativo, Proyectado y de Reabastecimiento (CPFR).

2.1.1 Ventajas

EL VMI trae ventajas para los clientes, al traspasar la información de sus inventarios al proveedor le traspasan la responsabilidad de la planificación y las actividades de reposición, razón con la cual pueden reducir los costos de llevar el inventario ellos mismos [6]. También la reducción de los costos administrativos ya que el proveedor tiene la mayor parte de la responsabilidad de la reposición, y hay menos órdenes urgentes de demanda ya que los inventarios son vigilados con más frecuencia.

El cliente y el proveedor utilizan datos comunes por lo que hay un gran flujo de intercambio de información, es aquí que la reducción de los costos de la tecnología ha permitido que el VMI se extienda en su uso. También hay una menor posibilidad de stock out, ya que el proveedor con el VMI tiene ahora la plena visibilidad de la demanda real, junto con una mejor información de factores como el tiempo de preparación, los lanzamientos de productos, cambios, entre otras cosas que pueden ayudar a gestionar mejor. Con el VMI el proveedor tiene más posibilidades de responder e investigar los acontecimientos inusuales que afectan a la demanda del inventario.

El VMI también presenta una serie de ventajas para el proveedor, ahora ellos pueden responder más rápidamente a los cambios inesperados en la demanda de los consumidores, aumentar los niveles de servicio y controlar mejor el nivel de producción junto con los costos de los inventarios. Con la utilización del VMI el proveedor puede obtener por lo general un régimen más uniforme de la utilización de los recursos, también reduce la cantidad del inventario que el proveedor debe de mantener para alcanzar un nivel adecuado de servicio al cliente. Otra ventaja del VMI al proveedor es que permite la reducción de los costos de transporte más allá de la reducción alcanzada al utilizar este recurso de manera más uniforme, mediante la planificación proactiva sobre la base de la información adicional disponible en vez de una respuesta reactiva a la medida de que llegan las órdenes de los clientes, siendo posible un aumento de la frecuencia de los vehículos a bajo costo, o enviar vehículos con cargamento completo con una frecuencia menor, y el posible manejo más eficaz de las rutas de los vehículos a la hora de reposicionar los productos a sus clientes. Por último el VMI puede aumentar los niveles de servicio, medidos en términos de fiabilidad en la disponibilidad de los productos, que a su vez es también un importante beneficio para los clientes, ya que ahora con la información que el proveedor posee, puede gestionar que entregas pueden ser aplazadas para dar cabida a las entregas urgentes, con lo cual puede fortalecer sus relaciones con los clientes.

Las actividades claves de los procesos del VMI residen principalmente en el proveedor. Estas involucran los pasos descritos por [7] que se presentan a continuación:

- Recopilar información: los datos para los programas del VMI focalizados en los centros de distribución se obtienen, principalmente, de las salidas de almacén de los minoristas. Algunas empresas complementan esta información con los datos de los puntos de venta para que los niveles de inventarios de las tiendas tengan mayor visibilidad.
- Previsión de ventas: la previsión se realiza sobre las salidas de los almacenes de los centros de distribución. El detalle de las ventas a clientes se reconcilia con las ventas al mercado mediante procesos de agregación de las previsiones.
- Previsión de pedidos: la previsión de pedidos es controlada por el proveedor y, generalmente, se realiza sobre los niveles de inventario acordados y los costes de transporte. Esto le permite al proveedor planificar el inventario para los clientes específicos.
- Generación de pedidos: el proveedor controla la generación de las órdenes de compra. Éstas se derivan del reabastecimiento del inventario de las tiendas. Como es

el proveedor el que controla el proceso, el cliente normalmente recibe prioridad en el servicio cuando ocurra alguna escasez.

- Entrega de pedidos: el fabricante entrega el producto principalmente desde el inventario. Las promociones pueden obligar a los proveedores a producir pedidos promociónales.

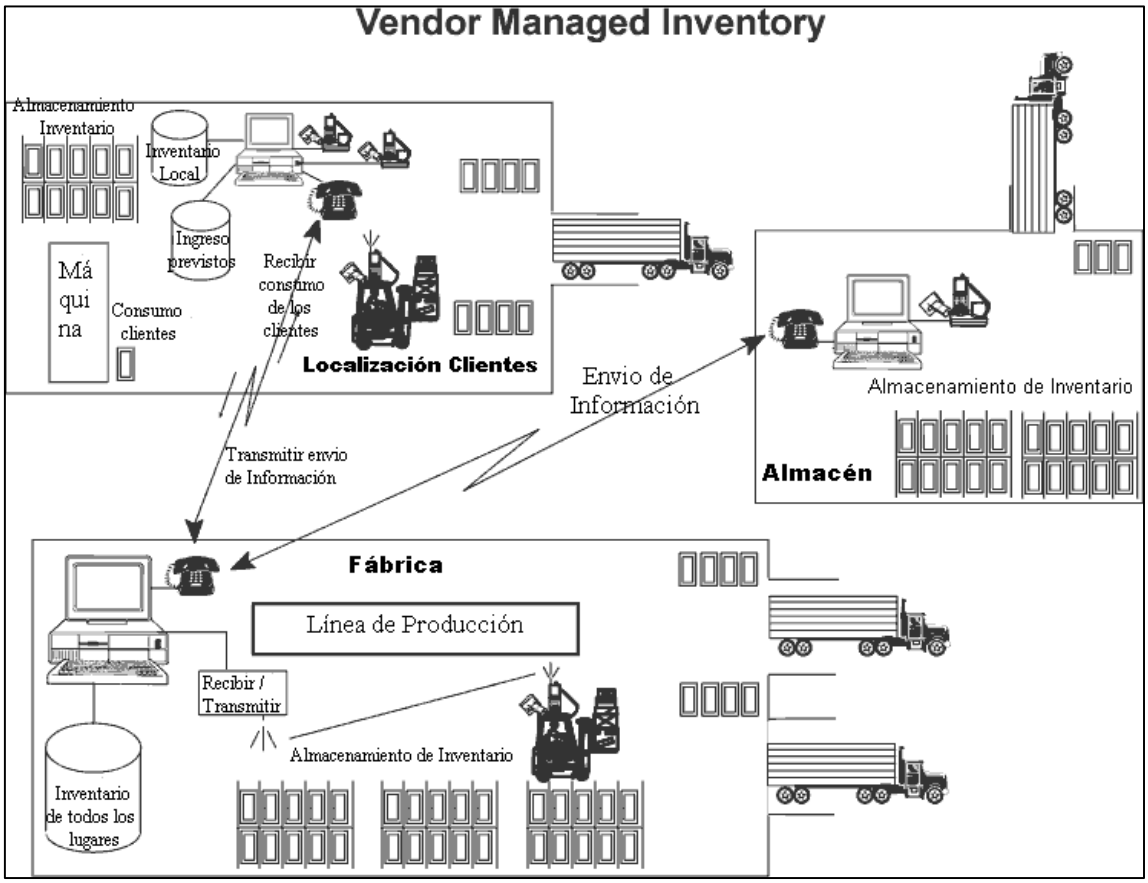

Figura 2.1: Vista del Proveedor con intercambio de información en el VMI.

En Figura 2.1, se representa un intercambio de información de los clientes con los proveedores. Esta información es ocupada por los proveedores para así poder gestionar de manera más eficiente la utilización de sus recursos.

Aplicaciones del VMI [5]:

- Industrias sensibles al error, como por ejemplo del sector farmacéutico
- Almacenes múltiples, bienes de consumo rápido.
- Productos merecedores.
- Componentes valiosos e imprevisibles.
- Fuerte Competencia (con márgenes pequeños), como por ejemplo la industria automovilística, como distribuidores de partes.

2.1.2 Limitantes

Limitantes del VMI indicadas en [5] son:

- El éxito del VMI depende de la relación entre los proveedores o vendedores con sus clientes.
- Aumento en el costo de cambio desde el punto de vista del cliente.
- Falta de confianza en intercambiar información puede dar lugar a malas prácticas que se expresan en la invisibilidad del inventario o en el desequilibrio de este.
- Costos de Tecnologías y de una organización cambiante.
- Se requiere de información extensa y pruebas de EDI (intercambio electrónico de datos).
- La pérdida del espacio necesario en los estantes del vendedor puede dar lugar a menor atención por parte de los compradores, comparados con los competidores que aún no utilizan el VMI.
- Las promociones o eventos especiales necesitan ser comunicadas de antemano para evitar errores de planeamiento del reabastecimiento (perdida de flexibilidad)
- Creciente vulnerabilidad ante los riesgos no previsibles tales como las huelgas de empleados, huracanes, terremotos, etc. Debido a niveles de inventario más bajos.
- La mayor parte de los beneficios son para el cliente final y para la parte vendedora, mientras que el proveedor logístico hace la mayor parte del trabajo.

El VMI es generalmente exitoso para las industrias y las organizaciones con las siguientes características [5]:

- Múltiples almacenes, porque esto aumenta los beneficios comparativamente con la gestión tradicional de inventarios.
- Consecuencias severas en caso de errores humanos (farmacéuticos)
- Industrias con volúmenes constantes y altos (al por menor, productos de consumo)
- Industrias con inventario de alto valor y de un alto nivel de imprevisión de la demanda (alta tecnología)
- Administración con un fuerte liderazgo para formar sociedades estratégicas de largo plazo (empresas automotrices)

2.2 Inventory Routing Problems

Como fue mencionado anteriormente en la introducción el IRPs es unos de los problemas que se presentan a la hora de aplicar la práctica llamada VMI, la cual integra dos componentes de la cadena de gestión de suministro: La gestión del inventario y la gestión del transporte. Comúnmente estos dos componentes son tratados por separados, pero su integración puede tener un gran impacto en el rendimiento general del sistema [8].

El IRPs es una variación del Vehicule Routing Problem (VRP). El VRP se produce cuando los clientes realizan las órdenes y la empresa, en un día determinado, asigna las

órdenes de ese día a las rutas de vehículos. En el IRPs es la empresa, y no el cliente, quien decide la cantidad a entregar a los clientes cada día. Por lo tanto no existe ninguna orden del cliente. En lugar de eso, la empresa opera con la restricción de que sus clientes no deben quedarse sin productos. Otra diferencia es el horizonte de planificación, los VRP suelen tratar con un solo día, con el único requisito de que todos los pedidos tienen que ser entregados al final del día, el IRPs hace frente a un horizonte más largo, cada día la empresa distribuidora toma la decisión respecto a qué clientes visitar y qué cantidad de producto entregar a cada ellos, tomando en cuenta que las decisiones tomadas en el presente impactarán en sus futuras decisiones [9].

De manera más formal, según [1], el IRPs se define de la siguiente manera:

El IRPs se refiere a la distribución repetida de un producto para un conjunto de N clientes durante un horizonte de planificación de longitud T, posiblemente infinito. El cliente i consume el producto a un determinado radio U_i (volumen por día) y tiene la capacidad de mantener un inventario local del producto hasta un máximo C_i. El inventario del Cliente i es I_i en el tiempo 0. Una flota de M vehículos homogéneos, con capacidad Q está disponible para la distribución del producto.

El objetivo es reducir al mínimo el promedio de distribución de los costos durante el periodo de planificación sin causar que el cliente se quede sin el producto requerido en su inventario.

Tres importantes decisiones se tienen que realizar:

- ¿Cuándo reponer el stock del inventario a un cliente?
- ¿Cuánto de producto a entregar a un cliente cuando éste se visita?
- ¿Qué rutas de entrega usar?

El IRP definido anteriormente es determinista y estático debido a que se está asumiendo que las tasas de consumo de los clientes son conocidas y constantes. Esto en la vida real no es así, el problema es estocástico y dinámico. Es por eso que una variante del IRP es el que utiliza demanda estocástica, y lo difiere del IRP normal en que en el futuro el uso de los productos por lo clientes es incierto. En el SIRP (Stochastic Inventory Routing Problem), se da una distribución de la demanda U_{it} del cliente i entre los puntos de decisión t y $t+1$ para $t=1,...,T-1$. Porque la demanda futura es incierta, puede que exista una probabilidad de que un cliente se quede sin existencias. El costo de que suceda esto puede ser modeladas de varias maneras, una de ellas es por medio de una función de penalización la cual consta con una componente fija y otra variable, por ejemplo $S_i + s_i d$ donde S_i es el costo del stock out fija y s_i es el costo del stock out variable (por unidad que falta) y d es lo faltante. El objetivo es elegir una política de entrega que minimiza el costo promedio por unidad de tiempo, o el costo total previsto, en el horizonte de planificación.

2.2.1 Problema con un cliente

Se considera como un análisis de un solo cliente cuando se tienen múltiples clientes pero en cada ruta el vehículo visita a un único cliente ya que existe una cantidad suficiente de vehículos para realizar esto [1].

Se considera el IRP de la siguiente forma: la tasa de uso del cliente es U, la capacidad del depósito del cliente es C, el nivel inicial del inventario es I, el costo de entrega al cliente es c, la capacidad del vehículo es Q y el horizonte de planeación es T.

Una política óptima es llenar los vehículos cuando estos son vaciados, en la fórmula (2.1) se presenta el costo v_t para un periodo de planificación de longitud T, el costo puede dar 0 si el cliente no es visitado.

$$v_t = \max\left(0, \left\lceil \frac{TU - I}{\min(C,Q)} \right\rceil\right) c \tag{2.1}$$

Cuando se considera el IRP como estocástico (SIRP) también se debe decidir los clientes que serán visitados y cuales no para formar las rutas de los vehículos. La demanda U entre puntos de decisiones consecutivos es una variable aleatoria con conocida probabilidad de distribución. Si se analiza la política "d-día", que es una política donde se entrega a los clientes cada "d" días una cantidad de productos suficiente como sea posible, a menos que haya un cliente haya quedado sin productos anteriormente. Cuando esto último sucede, el transporte es enviado de inmediato, lo que conlleva un costo S. Se supone que las entregas son instantáneas, por lo que las sanciones por ocurrir que el cliente se quede sin productos no son agregadas. Decimos que P_j es la probabilidad de que un cliente se quede sin productos ocurra en un día j ($1 \leq j \leq d\text{-}1$). Entonces $p = p_1 + p_2 + ... + p_{d-1}$ es la probabilidad de que ocurra un stock out y 1 – p es la probabilidad de que no ocurra un stock out en el periodo [1,…, d – 1]. La fórmula (2.2) nos indica el costo total esperado $v_y(d)$ con esta política en un periodo de tiempo de largo T cuando tenemos $d > T$, mientras que la fórmula (2.3) nos indica el costo cuando $d \leq T$.

$$v_T(d) = \sum_{1 \leq j \leq T} p_j \left(v_{T-j}(d) + S\right) \tag{2.2}$$

$$v_T(d) = \sum_{1 \leq j \leq T} p_j \left(v_{T-j}(d) + S\right) + (1-p)\left(v_{T-d}(d) + c\right) \tag{2.3}$$

Finalmente, el costo total de espera de los clientes para mantener el stock de sus inventarios en los días d es indicado en la fórmula (2.4), y es dada para un periodo de T días, con $T \geq d$, donde $\alpha(d)$ es una constante que depende de d, mientras que $f(T,d)$ es una función que de manera exponencial va a cero tan rápidamente como T se dirija al ∞,

y por último el valor $\beta(d)$ mostrado indicado en la fórmula (2.5) es el promedio del costo por día en el largo plazo.

$$v_T(d) = \alpha(d) + \beta(d)T + f(T,d) \tag{2.4}$$

$$\beta(d) = \frac{pS + (1-p)c}{\sum_{1 \le j \le d} jp_j} \tag{2.5}$$

Para encontrar la mejor política, se tiene que minimizar $v_t(d)$, por lo cual se debe encontrar un d que minimice $\beta(d)$.

La política de los d-días tiene la ventaja de poder ser utilizada incluso cuando el inventario de los clientes no es posible de medirse y se informa solamente cuando ocurre el stock out. A pesar de esto también tiene una serie de desventajas, esta política no es la óptima en general si el inventario del cliente es posible de medir. Las políticas que usan el uso del conocimiento del inventario de los clientes son mejores que la política mencionada anteriormente. La otra desventaja es que la probabilidad de stock out P_j usada en el análisis de la política de los d-días es muy difícil de obtener, y muchos no son bien definidos, a menos que el nivel de los inventarios de los clientes estén en el mismo nivel luego de la reposición. La razón es que probabilidad P_j de un stock out sea exacto j días después de la anterior reposición del inventario, depende del nivel de inventario después de la reposición.

2.2.2 El problema con múltiples clientes

Cuando más de un cliente es visitado, el problema se coloca más difícil. No solo hay que decidir la próxima visita al cliente, sino también la forma de combinar en las rutas de los vehículos, y la cantidad de productos a entregar a cada cliente. Incluso si sólo hay dos clientes, estas decisiones no son fáciles.

Hay dos soluciones extremas en este tipo de problemas, visitar a cada cliente por sí solo cada vez, ó visitar tantos clientes juntos como sea posible [1]. La fórmula (2.6) es el costo asociado a estas soluciones, para dos clientes se asume que $I1 = I2 = 0$, donde es posible tener implementada la solución con un solo vehículo o con dos de ellos, y la fórmula (2.7) muestra el costo de la ruta óptima del problema donde c_{12} es el costo óptimo del vendedor viajero entre ambos clientes.

$$v_t = \max\left(0, \left\lceil \frac{Tu_1}{\min(C_1, Q)} \right\rceil\right) c_1 + \max\left(0, \left\lceil \frac{Tu_2}{\min(C_2, Q)} \right\rceil\right) c_2 \tag{2.6}$$

$$v_T = \left\lceil \frac{T}{\min\left(\frac{C_1}{u_1}, \frac{C_2}{u_2}, \frac{Q}{u_1 + u_2}\right)} \right\rceil c_{12} \tag{2.7}$$

Si ambos son visitados juntos, dada la cantidad entregada al primer cliente, es óptimo entregar lo más posible al segundo cliente (determinado por la capacidad restante del vehículo, y la capacidad restante para el segundo cliente).

2.2.3 Enfoques de solución

El IRPs es un problema de control dinámico y de planificación a largo plazo. Este término de largo plazo es lo que transforma a este problema en uno muy difícil, casi imposible de resolver. Es de esta forma que casi todos los enfoques que se han propuesto e investigados sólo le dan una solución de corto plazo [1]. A principios el plazo de planificación ha sido de sólo un día y luego se ha aumentado de a poco.

Dos temas claves deben ser resueltos con todos estos criterios: la forma de modelar el efecto a largo plazo a decisiones de corto plazo, y que clientes incluir en el periodo de planificación de corto plazo.

Una solución a corto plazo tiende a aplazar el mayor número de entregas para el próximo periodo de planificación, lo que puede conducir a una situación indeseable para el próximo periodo de planificación. Por lo que una adecuada proyección de un objetivo a largo plazo a corto plazo es esencial. El efecto a largo plazo de decisiones en el corto plazo para capturar las necesidades de los costos y beneficios de realizar una entrega a un cliente antes de lo necesario. Hacerlo antes de lo necesario conduce a un mayor costo de distribución en el futuro, pero reduce el riego de que un stock out suceda, por lo que puede reducir los costos de escasez en el futuro.

Es así que se puede distinguir dos enfoques en el corto plazo. La primera, parte del supuesto en que todos los clientes en el corto plazo del periodo de planificación deben ser visitados. El segundo, parte que todos los clientes en el periodo de corto plazo pueden ser visitados, pero la decisión de si lo serán o no todavía no es realizada.

Las decisiones de acerca a quien visitar y cuanto se debe entregar son usualmente guiados por los siguientes supuestos acerca de lo que constituyen unas buenas soluciones:

- Siempre tratar de maximizar la cantidad entregada por visita.
- Siempre tratar de enviar camiones con una carga completa.

Cuando el periodo de corto plazo de planificación consiste de un solo día, este problema puede ser considerado como una extensión al problema del VRP, por lo que pueden adaptarse técnicas de solución del VRP. El enfoque de un solo día generalmente se

basa en decisiones de la última lectura del inventario, y tal vez, en un uso previsto para ese día. Por lo tanto, evitar la dificultad de previsión de largo plazo de uso, hace que el IRPs puede verse de una manera más sencilla.

Cuando el periodo de corto plazo consiste en varios días, el problema se complica, pero el potencial de rendimiento son mucho mejor las soluciones. Típicamente la resolución de este problema a corto plazo son formulados con programas matemáticos y resueltos usando técnicas de descomposición.

Pero como se ha mencionado anteriormente, el IRP en la vida real es un problema dinámico y estocástico [9]. Con esto se refiere que para cualquier periodo de planificación para el plan de distribución que abarca más de un par de días nunca realmente se ejecuta de la manera en que es previsto. Los volúmenes suministrados en la realidad muchas veces difieren del volumen planeado para suministrar porque el radio de uso de los clientes a veces es diferente de lo previsto, el tiempo planeado de la conducción también puede ser diferente por la congestión del tráfico, entre otras cosas. Con esto se trata de decir que cualquier sistema de planificación tiene que ser flexible, y es necesario aprovechar los últimos datos para ver los cambios que se han efectuado. Es de esta manera que un enfoque con unos pocos días de planificación es muchas veces más conveniente que un enfoque con un periodo de planificación muy largo.

2.2.4 Discusión Bibliográfica

Como se ha podido apreciar el IRPs tiene diferentes maneras de abarcarse, por ejemplo para [10] se toma el problema de la siguiente manera, considera una distribución repetida de un producto para varios clientes durante un periodo de tiempo, el consumo de los clientes varia todos los días. Para minimizar el total de los gastos anuales de los gastos de envío se utiliza un marco de horizonte móvil, para lograr esto se utiliza la simulación de Monte Carlo.

En [11] el problema es abarcado como un sistema de distribución en el cual un almacén es responsable de la reposición de un producto para sus clientes, donde la demanda utilizada es constante, con una entrega eficiente de rutas. Propone una política de partición fija para este problema, donde la partición va por regiones de los clientes. Utilizando el poder de dos principios. Un límite inferior para la media del costo en el largo plazo de cualquier estrategia viable para el sistema de distribución considerado, y un algoritmo de Tabu Search diseñado para encontrar la óptima partición de las regiones para los clientes, fijado en el marco de política utilizado.

En [12] se analizan tres problemas de colas de control de un modelo de sistema de distribución estocástica, donde un solo vehículo capacitado sirve a un finito número de clientes en un make-to-stock fashion. El objetivo es reducir al mínimo el inventario promedio en un largo plazo y los costos de transporte. El primero problema presentado es donde un vehículo viaja a lo largo de preestablecidas rutas, y un controlador dinámicamente decide el número de unidades a entregar para cada cliente. El segundo problema es donde el vehículo ofrece toda la carga a un cliente, y el controlador decide la próxima visita al

cliente. Y por último el tercer problema que sería más dinámico permite la elección entre los dos problemas anteriores. Utilizando el teorema del límite del tráfico pesado existente, se realiza una escala de tiempo de descomposición de tiempo que nos permite una aproximación a estos problemas de control, que están resueltos para una ruta fija.

En [3] se considera IRP compuesto de un almacén central y un conjunto de empresas a las cuales produce diferentes tipos de productos. Los productos se mantienen en el almacén central y se enfrentan a constantes demandas deterministas. Una política de inventario con una cantidad de orden fija es aprobada y cuando hay necesidad de reposición a un cliente un vehículo del almacén central va hacia los clientes. Se presenta la construcción de una heurística u una mejora del algoritmo llamado un muy larga escala de barrio, very large scale neighborhood (VLSN) del algoritmo de búsqueda presente. La heurística secuencial genera una primera solución factible y luego el VLSN se aplica para mejorar esta solución.

Por último tanto en [8] como en [9] se presenta un enfoque de descomposición del problema del IRP en dos fases. La primera fase es crear un calendario de entrega el cual se utiliza la programación entera para realizar esto. Mientras que la segunda fase es la construcción de un conjunto de rutas para poder lograr efectuar el calendario de entrega calculado en la primera fase, es en esta segunda fase que aparece la utilización de las heurísticas para cumplir con este objetivo. Trata de crear una metodología adecuada para la solución a escala para casos de la vida real.

Es en este último enfoque en que se basará parte de la implementación del modulo de IRPs del proyecto presente. Ocupando la separación en dos fases del problema y aplicando diferentes heurísticas para la fase del esquema de rutas.

Capítulo 3

Heurísticas

Las Heurísticas se han utilizado desde siempre para hacer frente a los problemas de combinatoria. Desde los años 70 se ha puesto en manifiesto que la mayoría de estos problemas son del tipo NP-duro, lo que significa que hay pocas esperanzas de encontrar la solución exacta. Esto último hizo hincapié para la utilización de las heurísticas para la solución de problemas de combinatoria que se presenta en la vida real. A continuación se mostrarán los estados del arte de las heurísticas que se utilizarán en este proyecto.

3.1 Simulated Annealing

Simulated Annealing (SA) conocida también como recocido simulado o enfriamiento simulado es una técnica de búsqueda en forma aleatoria la cual explora una analogía entre el calentamiento y enfriamiento de una estructura cristalina y la búsqueda de un mínimo o una solución óptima para muchos sistemas generales [13].

La técnica del SA fue definida en el inicio de la década de los 80, en publicaciones independientes de Kirkpatrick (1983) sobre diseño de circuitos VLSI, y Cerny (1985) para el TSP. Surgió del campo de la termodinámica como consecuencia de la comparación de problemas formulados en este campo, con los del campo de la investigación operacional. El SA es una técnica de optimización de combinatoria que se usa para afrontar problemas de gran complejidad matemática, de modo que se obtengan soluciones cercanas a la óptima.

La idea que dio lugar al SA es llamada algoritmo de Metrópolis (1953) [14], se fundamenta en el proceso físico de calentamiento de un sólido, seguido por un enfriamiento hasta lograr un estado cristalino con una estructura perfecta. Durante este proceso, la energía libre del sólido es minimizada.

A una temperatura T, en el estado con nivel de energía E_i, se genera el estado siguiente *j* a través de de una pequeña perturbación. Si la diferencia de energía $E_i - E_j$ es menor o igual a cero, el estado *j* es aceptado. Si la diferencia de energía es mayor que cero, el estado *j* es aceptado con una probabilidad dada por la fórmula (3.1) donde k_B es la constante de Boltzmann. Si la disminución de la temperatura es hecha de manera paulatina, el sólido puede alcanzar el estado de equilibrio en cada nivel de temperatura.

$$e^{\frac{E_i - E_j}{k_B T}} \tag{3.1}$$

A partir del estado i con costo $f(i)$, se genera el estado j con costo $f(j)$. El criterio de aceptación determina si el nuevo estado es aceptado o no, para eso se calcula la probabilidad indicada en la fórmula (3.2).

$$\text{Prob.Acep.j} = \begin{cases} 1 & \text{si } f(j) \le f(i) \\ e^{\frac{f(i)-f(j)}{T}} & \text{si } f(j) > f(i) \end{cases} \tag{3.2}$$

Con esto se intenta evitar el quedar atrapado en mínimos locales. Al comienzo cuando T es alto, es permitido el uso de configuraciones que deterioren la función objetivo, pero a medida que T vaya disminuyendo la probabilidad de aceptar estas configuraciones disminuye también.

3.1.1 Algoritmo del Simulated Annealing

A continuación se presenta el seudo código (Figura 3.1) donde se implementa el Simulated Annealing, obtenido de [14], junto con un diagrama de su funcionamiento mostrado en Figura 3.2. En este pseudocódigo el óptimo es la minimización de la función objetivo $f(configuracion)$, los primeros pasos son inicializar las variables T_0 que es la variable de la temperatura inicial y N_0 que es el número de soluciones aceptadas por nivel de temperatura. Por otro lado tenemos R_i que es la configuración o solución inicial, de a partir de ella se van obteniendo las demás configuraciones R_j. Si la nueva configuración disminuye la función objetivo se selecciona como la mejor R_i, sino dependiendo de una probabilidad esta solución es aceptada o no. Cada vez que se calculan N_k soluciones por nivel, la Temperatura va decreciendo, como también las soluciones aceptadas. El algoritmo termina cuando se cumple un criterio de parada, por ejemplo que la temperatura no se modifique.

```
inicializar T0, N0
generar configuración inicial Ri
inicializar k
hacer
   para h desde 1 hasta Nk
      generar Rj a partir de Ri
      Si f(j) ≤ f(i) entonces
         Ri = Rj
      sino
         Si exp{[f(i) − f(j)] / Tk} > aleatorio[0,1] entonces
            Ri = Rj
      fin si
   fin para
   incrementar k
   calculo de Nk
   calculo de Tk
mientras (criterio de parada no se haya presentado)
```

Figura 3.1: Pseudocódigo Simulated Annealing

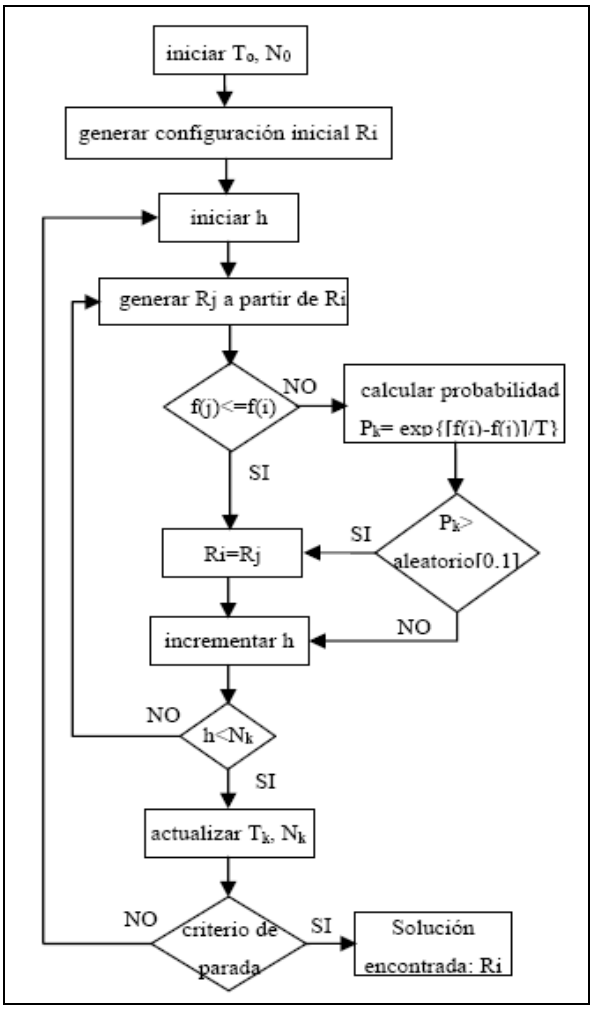

Figura 3.2: Diagrama de Flujo Simulated Annealing.

3.1.2 Temperatura

La determinación de la Temperatura inicial T_0 es un factor clave [13], debe permitir que se realice una adecuada exploración en las etapas del SA, y es la que resulta en un aumento promedio p_0 de unos 0.8 En otras palabras, hay un 80% de posibilidades de que un cambio que aumenta la función objetivo será aceptado. Existen diferentes formas para calcular la temperatura inicial, se puede estimar mediante la realización de una búsqueda inicial en el que todos los aumentos son aceptados y el cálculo de la media del aumento observado δf^+. La T_0 es entonces dada por:

$$T_0 = \frac{-\delta f^+}{\ln(p_0)} \tag{3.3}$$

También es necesario definir la tasa B de disminución de temperatura [14], para obtener un enfriamiento controlado. Tomándose $T_{k+1} = BT_k$.

Teóricamente la temperatura final debería reducirse a cero, pero en la práctica la búsqueda converge por lo general a su óptimo local final bastante antes de ese valor nulo. Por eso la temperatura final [13] de enfriamiento es aquella en la que ya es muy poco probable que se acepten soluciones que empeoren la situación, es por eso que en algunas soluciones simples del algoritmo SA es aquí donde termina el proceso. Alternativamente, la búsqueda puede ser detenida cuando se deja de avanzar. La falta de progreso puede ser definida de varias maneras, pero una utilización básica es:

- Ninguna mejora, es decir, sin ninguna nueva solución se encuentra en toda una cadena de Harkov en una temperatura, en combinación con
- La aceptación de una radio de fallo por debajo de un determinado valor p_f (pequeño).

La determinación de la longitud de la cadena de tentativas para cada nivel de temperatura T_k es otro factor clave, como regla general se maneja una longitud creciente a medida que la temperatura va diminuyendo, con el fin de permitir una exploración más intensa a temperaturas bajas. Por lo cual se toma $N_{k+1} = pN_k$, con p≥1.

Para definir la longitud inicial N_0 de la cadena, este se escoge proporcionalmente al tamaño del problema, por ejemplo k veces el numero de la variables: $N_0 = k \cdot \#\,\text{variables}$.

Se pueden usar varios criterios de parada, como por ejemplo:

- Se fija un número determinado de niveles de temperatura, usualmente entre 6 y 50.
- Si la función objetivo no mejora para varios niveles consecutivos, entonces se termina el proceso.
- Si no se cumple con un número mínimo de aceptaciones en el nivel de temperatura.

3.1.3 Generación de una nueva configuración

Se debe definir una estructura de vecindad, de la cual se toma la nueva configuración a partir de la actual. La estructura de vecindad está conformada por aquellas configuraciones (no necesariamente todas) que se pueden generar mediante pequeñas modificaciones, como sacar un elemento activo, agregar uno, o intercambiar un elemento que ésta presente en la configuración por otro que no lo está [14].

Para esto un componente importante es la generación de números aleatorios, que se utilizan tanto para generación aleatoria de los cambios en las variables de control y del incremento de la aceptación de la prueba, lo último depende también de la temperatura. La configuración de la vecindad es aleatoria pero también se puede incrementar la probabilidad de selección para aquellas configuraciones que tienen un índice de sensibilidad que las identifica como atractivas para conseguir un mejoramiento en la función objetivo.

3.1.4 Fuerzas y debilidades

El SA puede hacer frente a modelos altamente no lineales (complejos), a datos caóticos y ruidosos y con muchas limitaciones. El SA se trata de una técnica sólida y general [13]. Sus principales ventajas respecto a otros métodos de búsqueda locales son su flexibilidad y su capacidad de mantener un enfoque global óptimo. Este algoritmo es muy versátil, ya que no confía en ninguna de las propiedades restricciones del modelo.

Desde que el SA es una metaheurística, es requerido un montón de opciones necesarias para convertirlo en un verdadero algoritmo. Existe un claro compromiso entre la calidad de las soluciones y el tiempo de cálculo necesario. El trabajo necesario para tener en cuenta los diferentes tipos de limitaciones y la afinación de los parámetros del algoritmo puede ser bastante delicado. La precisión de los números utilizados en la aplicación del SA puede tener un efecto significativo sobre la calidad del resultado.

3.2 Tabu Search

Este método fue propuesto por Glover en el año 1986 en el mismo artículo que introdujo el término de metaheurística. El Tabu Search (TS) o Búsqueda Tabú original se trataba de un superposición a la Búsqueda Local (LS) encargándose de evitar que caiga en óptimos locales prohibiendo ciertos movimientos. Para prevenir la visita a los óptimos locales es impedido por el uso de memoria, llamada lista tabú, que registrará la historia reciente de la búsqueda de esta manera evitar que caigan en ciclos durante la búsqueda.

Es importante remarcar que Glover no vio el TS como una adecuada heurística, sino como una meta-heurística, es decir, guiar un procedimiento heurístico de búsqueda local en la búsqueda del óptimo global [15]. Su filosofía se basa en derivar y explotar una colección de estrategias inteligentes para la resolución de problemas, basadas en procedimientos implícitos y explícitos de aprendizaje [16].

Desde la perspectiva de la Inteligencia Artificial, el TS trata de simular el comportamiento de una persona [17]. Es bien sabido que el ser humano posee un mecanismo avanzado de intuición que nos permite operar aún con información mínima o nula, por lo que generalmente solemos introducir un elemento aleatorio en dichas decisiones, lo cual promueve un cierto nivel de "inconsistencia" en nuestro comportamiento. La tendencia resultante en estos casos suele desviarnos de una cierta trayectoria predefinida, lo cual algunas veces puede ser una fuente de errores, pero en otros casos puede llevarnos a una solución mejor. El TS trata de simular este mecanismo, pero sin la utilización de elementos aleatorios, sino más bien asumiendo que no hay razón para escoger un movimiento que nos lleve a una peor solución, a menos que estemos tratando de evitar una ruta que ya se examino previamente. Con esta excepción, la técnica buscará el mejor movimiento posible de acuerdo con la métrica utilizada por el problema en cuestión. Esto hace que en un principio esta técnica se dirija inicialmente a un óptimo local, pero la búsqueda no se detendrá ahí, sino que se reinicia manteniendo la capacidad inicial de detectar el mejor movimiento posible. Además, se mantiene información referente a los

movimientos más recientes en las listas tabú, reconociendo que está prohibición no es absoluta sino condicional.

Más particularmente el TS está basada en la premisa de que para clasificar un procedimiento de resolución como inteligente, es necesario que éste incorpore memoria adaptativa y exploración responsiva. La memoria adaptativa en TS permite la implementación de procedimientos capaces de realizar la búsqueda en el espacio de soluciones eficaz y eficientemente. Dado que las decisiones locales están por tanto guiadas por información obtenida a lo largo del proceso de búsqueda, el TS contrasta con diseños que por el contrario confían en procesos semialeatorios, que implementan una forma de muestreo [16].

El TS se encuentra fundamentado en tres cosas principales [15]:

- El uso de estructuras flexibles de memoria basado en atributos, diseñada para permitir una mejor exploración de los criterios de evaluación y la información histórica de la búsqueda.
- Un mecanismo asociado de control basado en la interacción entre las condiciones que limitan y hacen más flexible el proceso de búsqueda. Este mecanismo se encuentra inmerso en la técnica en la forma de restricciones y criterios de aspiración.
- Incorporación de memorias de diferente tiempo de duración, para implementar técnicas que intensifiquen y diversifiquen la búsqueda.

3.2.1 Algoritmo del Tabu Search

A continuación se presenta el pseudocódigo de una implementación del Tabu Search en Figura 3.3. La función objetivo de este código es minimizar el valor de la función $f(solución)$, el primer paso es inicializar las variables, aquí en este ejemplo son la lista Tabú que estará vacía, y el número máximo de iteraciones permitidas para mejorar una solución que es la variable N, mientras la variable n es un contador que indica el número de iteraciones. Luego se procede a generar una solución inicial, y se le asigna como la mejor solución, esta solución entra en un bucle donde se irá modificando para encontrar soluciones mejores, cuando se realiza un movimiento se debe ver si esta solución generada es permitida o no, esto se ve en la función del mejor movimiento, donde se ve si el movimiento realizado es un movimiento Tabú, y si lo es si cumple con los criterios de aspiración para determinar si se ira a ocupar o no. Si la solución no es aceptada se reinicia el proceso generando otra posible solución, también es posible que se reinicie si en las n iteraciones la solución no mejora, se deba cambiar. Si la solución no es Tabú y mejora la solución es considerada como mejor solución que la anterior. Hay varias maneras de detener este algoritmo, el que presenta el ejemplo es cuando se conoce la solución óptima, por lo que cuando el algoritmo llegue a esa solución se detendrá. Si la solución encontrada luego de todas las iteraciones no es el óptimo puede ser posible elegir si se realiza una diversificación o una intensificación, si la respuesta es una diversificación se genera otra solución inicial aleatoria, que a su vez cumpla algunos requisitos. En cambio si se ha

escogido la intensificación se realiza mejoras a la solución encontrada. Luego estas soluciones vuelven entrar al ciclo de procesos hasta que se haya encontrado la solución final.

La Figura 3.4 presenta el diagrama de flujo del algoritmo, la evaluación del mejor movimiento será explicado más detalladamente en el punto 4.2.6.

```
inicializar listaTabu, n, N
Generar configuración inicial s
Mejor solución encontrada s*=s
mientras(no se cumpla condición de parada)
    mientras(no se cumpla condición de reinicio)
        identificar mejor movimiento s' = BestMove(s, listaTabu)
        if (s' == NULL)
            se cumple condición reinicio
        if (n == N)
            se cumple condición reinicio
        AñadirMovTabu( )
        if (f(s') < f(s*))
            s* = s'
        s = s'
    fin mientras
    if (s* == s_buscada) entonces
        cumple condición de salida
    sino
        escoger entre diversificación o intensificación
        if (diversificacion es escogida) entonces
            Generar solución aleatoria s
        sino
            Mejorar solución encontrada s*
        fin si
        Reiniciar listaTabu
    fin si
fin mientras
```

Figura 3.3: Pseudocódigo algoritmo Tabu Search

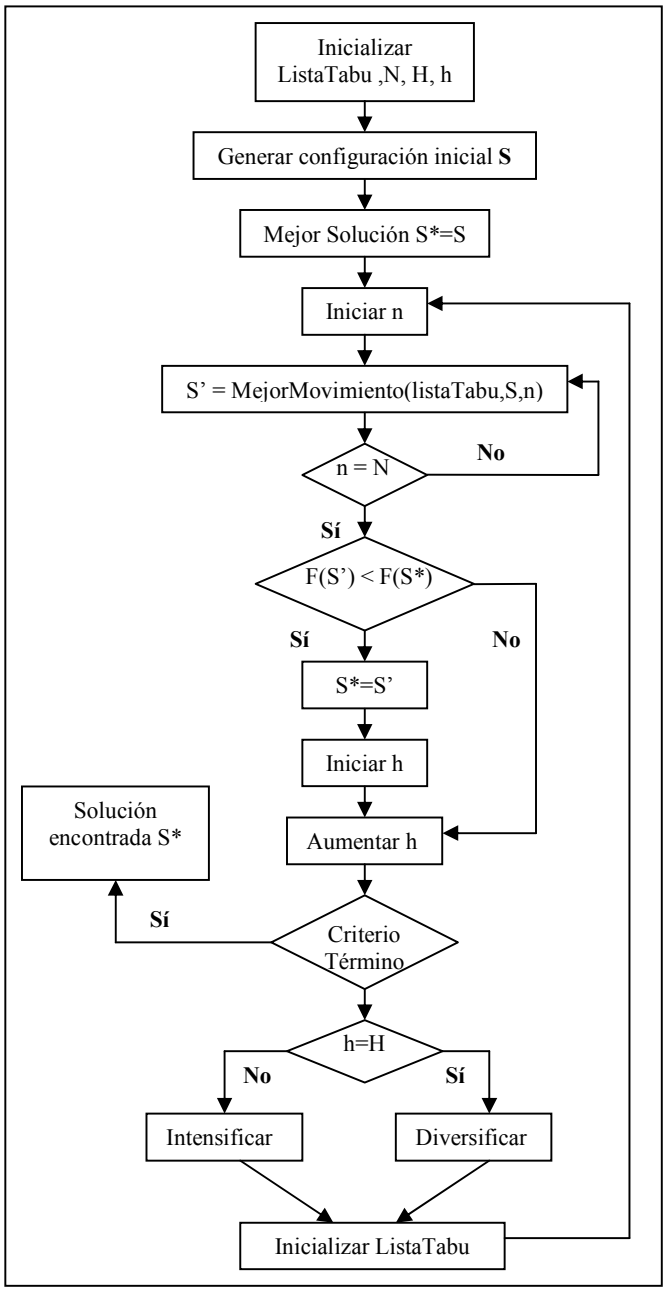

Figura 3.4: Diagrama de Flujo Tabu Search

3.2.2 Uso de Memoria

Las estructuras de memoria del TS funcionan mediante referencia a cuatro dimensiones principales, consistentes en la propiedad de ser reciente, en frecuencia, en calidad, y en influencia [16]. Las memorias basadas en lo reciente y en frecuencia se complementan para lograr el balance entre intensificación y diversificación. La dimensión de calidad hace referencia a la habilidad para diferenciar la bondad de las soluciones visitadas a lo largo del proceso. La calidad constituye un fundamento para el aprendizaje

basado en incentivos, donde se refuerzan las acciones que conducen a buenas soluciones y se penalizan aquellas que conducen a malas soluciones. La flexibilidad de las estructuras permite guiar la búsqueda en un entorno multi-objetivo, dado que se determina la bondad de una dirección de búsqueda particular mediante más de una función. La influencia considera el impacto de las decisiones tomadas durante la búsqueda, no solo referente a la calidad de soluciones, sino también en lo referente a la estructura de las mismas.

El uso de memoria en el TS es tanto explícita como implícita [16]. En lo primero, se almacenan en memoria soluciones completas, mientras que en lo segundo se almacena información de determinado atributos de las soluciones que cambian al pasar de una solución a otra.

3.2.3 Criterio de Aspiración

Si bien algo central en el TS es la lista Tabú, a veces está es demasiada poderosa ya que pueden prohibir movimientos atractivos, incluso cuando no hay peligro de que suceda un ciclo, o pueden conducir a un estancamiento general del proceso de búsqueda. Por lo tanto, es necesario utilizar dispositivos en el algoritmo que permitan a uno revocar Tabú, los cuales son llamados como criterios de Aspiración [15]. La más simple y utilizada consiste en dejar que avance, a pesar de que es Tabú, si se produce una solución objetivo con un valor mejor que la actual solución conocida.

Las aspiraciones se pueden dividir en dos clases:

- Aspiraciones de movimiento, que cuando se satisface se revoca la condición de tabú del movimiento.
- Aspiraciones de atributo, cuando se satisface revoca el estatus tabú del atributo, el movimiento puede o no cambiar su condición de Tabú, dependiendo si la condición tabú puede activarse por más de un atributo.

Se pueden tener los siguientes criterios:

- Aspiración por Default, si todos los movimientos posibles son clasificados como tabú, entonces se selecciona el movimiento menos tabú.
- Aspiración por Objetivo, una aspiración de movimiento se satisface, permitiendo que un movimiento x sea candidatos si la solución es mejor que la actual.
- Aspiración por dirección de búsqueda, un atributo de aspiración para la solución “s” se satisface si la dirección en “s” proporciona un mejoramiento y el actual movimiento es un movimiento de mejora.

3.2.4 Intensificación y Diversificación

La Intensificación y Diversificación constituyen a dos elementos altamente importantes en el proceso de TS. Las estrategias de Intensificación se basan en la modificación de reglas de selección para favorecer la elección de buenas combinaciones de

movimientos y características de soluciones encontradas [16], regresan a regiones ya exploradas para estudiarlas más a fondo, para ello se favorece la aparición de aquellos atributos asociados a buenas soluciones.

Por otro lado, las estrategias de diversificación tratan de conducir la búsqueda a zonas del espacio de soluciones no visitadas anteriormente y generar nuevas soluciones que difieran significativamente de las ya evaluadas. Para ello se modifican las reglas de elección para incorporar a las soluciones atributos que no han sido usados frecuentemente.

3.2.5 Memoria de corto plazo o basado en lo reciente

La parte medular del TS se localiza en el proceso de memoria a corto plazo, y muchas de estas consideraciones estratégicas en que se fundamenta este proceso reaparecen, amplificadas pero sin mayores modificaciones, en los procesos de memoria de largo plazo [17]. La memoria de corto plazo del TS constituye una forma de exploración agresiva que intenta realizar el mejor movimiento posible sujeto a las restricciones impuestas por el problema. Su funcionamiento básico se presenta más detalladamente en la Figura 3.5, perteneciente a [17].

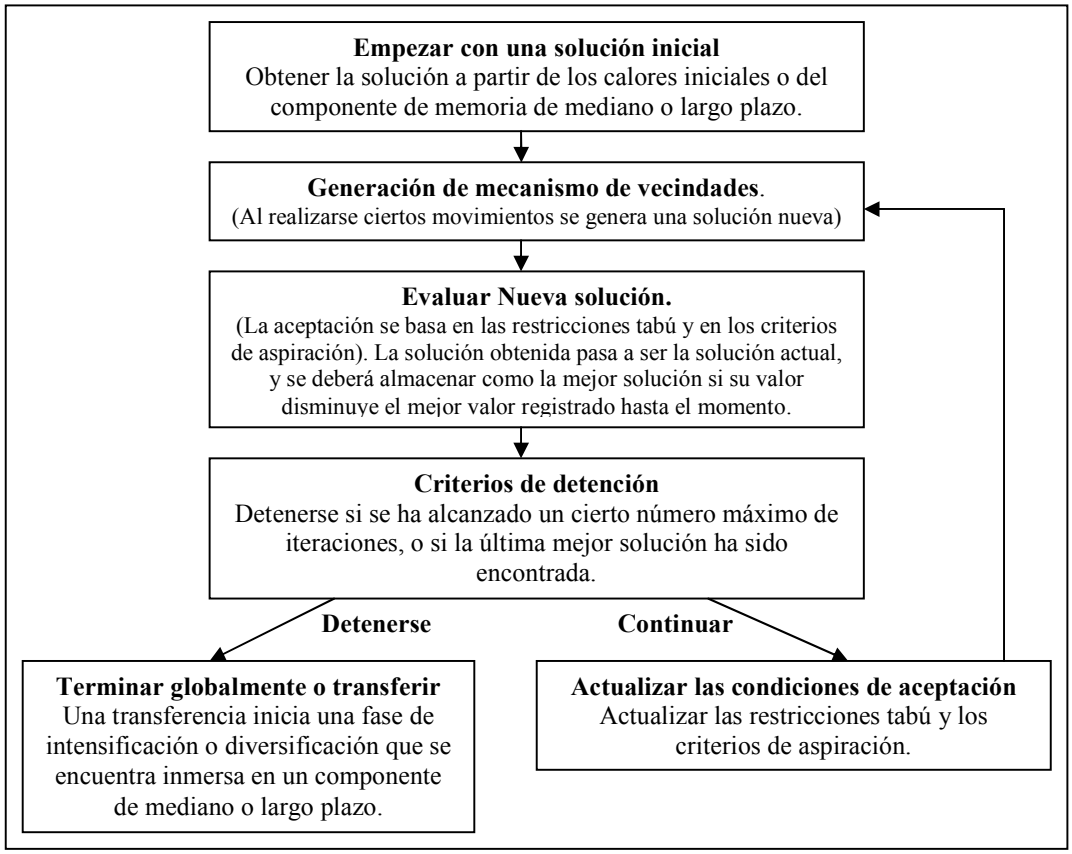

Figura 3.5: Componente de Memoria de Corto Plazo del TS.

Las restricciones impuestas por el problema se utilizan para evitar que se reviertan o repitan ciertos movimientos, que se convierten en prohibidos. Estas restricciones no actúan

de manera aislada, sino que se contrabalancean mediante la aplicación de ciertos criterios de aspiración derivados del planteamiento del problema.

3.2.6 Determinando el mejor movimiento

La selección del mejor movimiento es un paso crítico, en Figura 3.6, obtenida de [17], ilustra esquemáticamente este mecanismo.

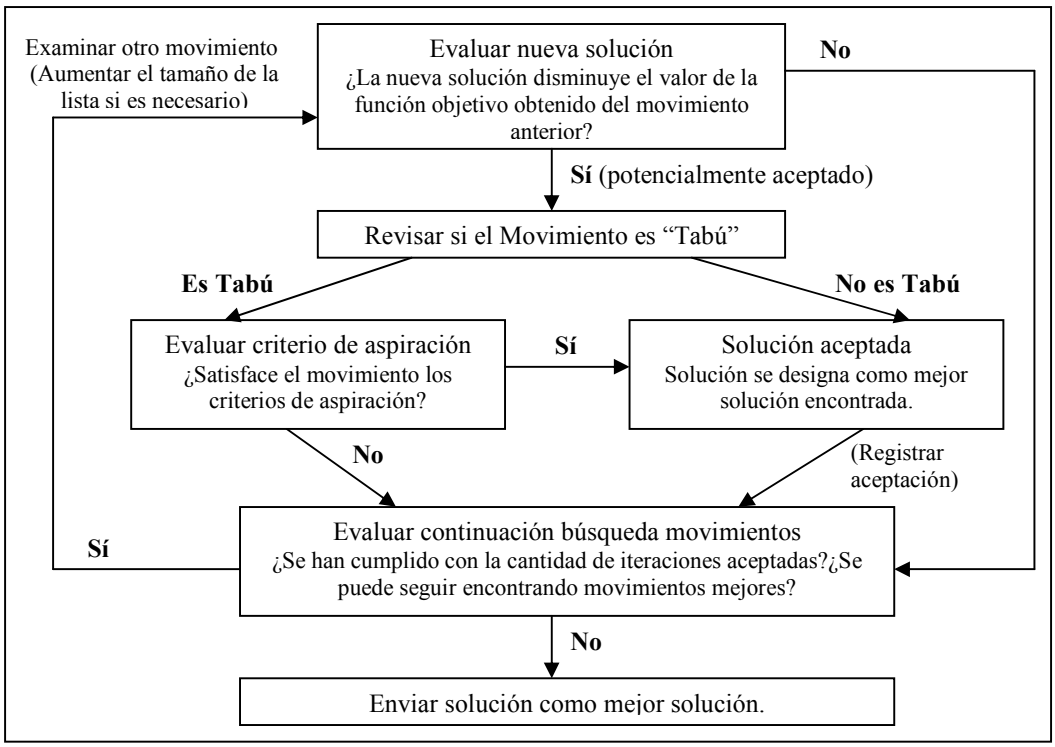

Figura 3.6: Selección del mejor movimiento posible.

El primer paso es evaluar cada uno de los movimientos de la lista de candidatos de turno. A veces esta evaluación se basa en la afectación en la función objetivo para obtener una solución más favorable. Otras veces, en donde la determinación del movimiento es más difícil de determinar o donde no ha todas las variables se les ha asignado un valor todavía, es más conveniente evaluar en base a soluciones aproximadas, o utilizar una medida local de estimación de la cercanía de una solución al óptimo. A medida que el algoritmo vaya avanzando, la evaluación utilizada se vuelve más adaptativa, incorporando referencia a los parámetros de intensificación y diversificación. Como la mayoría de los casos los movimientos tabú son menores al número de movimientos válidos; asumiendo que el costo de evaluación no es demasiado grande, normalmente se prefiere evaluar si el movimiento es mejor que sus predecesores aceptados, antes de ver si es un movimiento prohibido. Luego se revisa si el elemento es tabú, de no serlo es aceptado. De lo contrario, se utilizan los criterios de aspiración para dicho movimiento y evaluar si es conveniente cambiar su estado de tabú a que no lo sea. Si las restricciones tabú y los criterios de aspiración son suficientes limitantes, ninguno de los movimientos posibles calificarán como aceptados, en

estas situaciones se almacena un movimiento "menos aceptado", que se escogerá sólo si no hay ninguna alternativa de aceptación posible.

3.2.7 Memoria de largo plazo o basada en frecuencia

La memoria basada en la frecuencia proporciona un tipo de información que complementa la información proporcionada por la memoria basada en lo reciente, ampliando la base para seleccionar movimientos preferidos. En esta estructura de memoria se registra la frecuencia de ocurrencias de los movimientos, las soluciones o sus atributos y pueden ser:

- Frecuencia de transiciones, cantidad de veces que una solución es la mejor o cantidad de veces que un atributo pertenece a una solución generada.
- Frecuencia de residencia, cantidad de iteraciones durante la cual un atributo pertenece a la solución generada.

3.2.8 Criterios de Término

En teoría una búsqueda podría durar para siempre, a menos que el valor óptimo del problema en que se ocupa es conocido de antemano. En la práctica, la búsqueda debe ser detenida en algún momento. Los criterios más frecuentes en TS son [15]:

- Haber alcanzado cierto número máximo de iteraciones.
- Haber realizado cierta cantidad de iteraciones sin actualización de la mejor solución encontrada.
- Haber encontrado una solución cuyo costo sea mejor que un valor predefinido.
- La combinación de algunas de las condiciones anteriores.

En esquemas más complejos de Tabú, la búsqueda se detiene después de completar una secuencia de fases, la duración de cada fase está determinada por uno de los criterios anteriormente mencionados.

Capítulo 4

Arquitectura y Modelo Matemático

4.1 Arquitectura

A continuación se presenta una propuesta de una arquitectura para el Vendor Managed Inventory. Desde una vista general, mostrada en Figura 4.1, se puede apreciar 4 módulos: de Información, de Proyección, la interfaz y por último el del IRP. Es este último módulo (color naranjo) el que se va a implementar en lo que abarca este proyecto y el cual se adentrará más en su explicación.

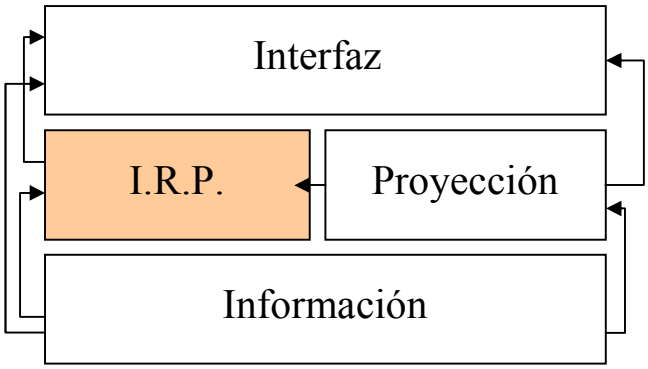

Figura 4.1: Arquitectura VMI

4.1.1 Módulo de información.

El manejo de la información es lo más esencial para el VMI, junto con la seguridad de la integridad de está tanto de los proveedores como de los clientes, es la fuente del funcionamiento correcto del VMI. Para realizar este objetivo el módulo se compondrá de lo siguiente: Componente de Comunicación, Seguridad y Administración. Todo esto es mostrado en la Figura 4.2 de a continuación.

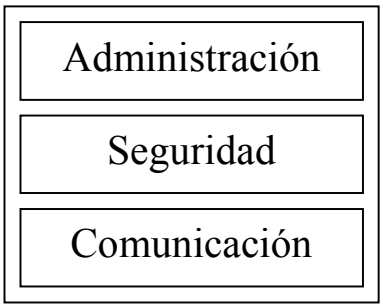

Figura 4.2: Arquitectura Módulo de Información

Ningún componente de éste módulo será implementado, solo se simularán los resultados, como por ejemplo la demanda de cada cliente, ubicación, entre otras cosas, utilizando archivos de texto necesarios como entrada para el módulo del IRP y de ayuda para la predicción de demanda (que también será simulada).

- **Componente de Comunicación.**

El proveedor posee múltiples clientes al cual atender, cada cliente debe enviar la información requerida por el VMI, y las formas en que esto sea posible es bastante variada. Es de esta manera que el Componente de Comunicación es el encargado de establecer ciertos estándares para el recibo de los Informes de Inventario que los clientes deben enviar con una periodicidad establecida para cada tipo de clientes. A su vez, deben entregar información de ubicación, de momentos excepcionales donde su oferta aumenta o disminuye (por ejemplo, Promociones, Cambio de Temporada), para que el proveedor pueda prever de mejor manera estas situaciones. Otra entrada de este módulo, aparte de la información de los clientes, es necesaria la información del mismo proveedor ya que gran parte de sus recursos participan activamente con el VMI (ejemplo: Transporte, Propio inventario) y pueden afectar casi directamente en la producción de sus productos y/o servicios.

Como se menciono en un principio, la transferencia de los datos es posible que sea bastante variada y este componente también debe encargarse de poder realizar un enlace para que esto sea posible y si es necesario poder transformar informes llegados a un estándar interno.

En resumen este componente debe:

- Establecer una vía de acceso continua de información tanto del cliente como del proveedor.
- Estandarizar la información recibida.
- E incluso por medio de esta vía de enlace realizar avisos importantes hacia el cliente (indicando que reportes deben enviar) o hacia el mismo proveedor.

- **Componente de Seguridad.**

Este componente es el encargado de mantener protegida la información entregada de los clientes como del proveedor, respaldando cada cierta cantidad de tiempo. Por otro lado se encarga del manejo de errores que pueda aparecer en el transcurso de algún proceso, como también se encarga de la autentificación necesaria para la protección del sistema.

En resumen este componente debe:

- Respaldar cada cierto periodo la información que se posee.
- Analizar la información en busca de errores y corregirlos.
- Autentificar información enviada (ver si es realmente del cliente o el proveedor).

- Autentificar a los usuarios del sistema.
- Proteger zonas del sistema.

- **Componente de Administración**

Con toda la información entregada anteriormente, es necesario analizarla y realizar resúmenes que serán enviados a los otros dos módulos del VMI. Estos resúmenes son tanto de los clientes como del proveedor. El resumen del proveedor debe mostrar cantidad de sus recursos que puedan ser utilizados, o estado de su inventario, entre otras cosas. Mientras que el resumen de los clientes deben mostrar partes como la ubicación y el estado de sus inventarios.

En resumen este componente debe:

- Recopilar la información entregada y analizarla.
- Realizar estadísticas de la información.
- Generar Reporte de inventario de los clientes.
- Generar Reporte de recursos de los proveedores.
- Enviar los reportes a los otros módulos.

4.1.2 Módulo del IRP

Primero que nada, se debe mencionar como punto importante que es éste módulo el cual se pretende implementar al final del proyecto. El cual es el encargado de responder a las tres preguntas que definen el VMI, las cuales son: ¿Cuánto de producto se debe entregar al cliente cuando es visitado?, ¿Cuándo se debe visitar al Cliente para realizar la reposición de su inventario? Y por último ¿Qué rutas se deben usar para realizar las entregas de los productos?

Para cumplir con esto, este módulo se compone con lo siguiente: Componente de Enrutamiento de Vehículos, Asignación de Demanda y de Cluster. Todo esto se puede ver en Figura 4.3 de a continuación.

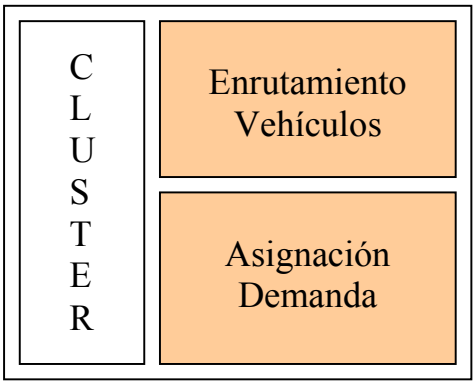

Figura 4.3: Arquitectura Módulo de IRP

- **Componente de Asignación de Demanda.**

Este componente responde a las dos primeras preguntas mencionadas anteriormente, para realizar esto se obtiene la información de los estados actuales de los inventarios de los clientes (Inventario Inicial junto con el límite de entrega o Inventario final que el cliente pude poseer en sus bodegas), como también se requiere de la tasa de uso de cada cliente para así definir si es realmente urgente visitar al cliente, si necesita una gran cantidad de productos para no caer en stockout, o sí puede esperar y en otro día ser visitado.

Como resultado, nos entrega si el cliente debe o no ser visitado en la ruta en un plazo de tiempo en específico y que tan urgente es realizar esto, la cantidad de demanda que se debe entregar. Esta información pasa al siguiente componente.

Este componente será implementado completamente para poder ejecutar el caso de prueba del proyecto.

- **Componente de Enrutamiento.**

Este componente es el encargado de responder a la pregunta faltante, la cual es la siguiente: ¿Qué rutas se deben usar para realizar las entregas de los productos? Para esto es necesario contar con la demanda de cada cliente y con la información de los costos en la visita a cada cliente o de pasar de un cliente "i" a un cliente "j", junto con la información de los recursos del proveedor, como por ejemplo la capacidad de los vehículos.

Como resultado nos entrega las rutas a utilizar, junto con el orden en que deben ser visitados cada cliente en cada ruta y el costo que realizarla supone al proveedor.

Es este componente conjunto con el anterior el que se implementará de manera completa para realizar el caso de prueba del proyecto.

- **Componente de Cluster.**

Es posible también que la cantidad de clientes del proveedor sea bastante amplia y procesar la información más complicada. Agrupar a los clientes para que sean estos los grupos que pueden ser visitados en determinadas rutas puede ayudar a simplificar el problema (por lo menos en tiempos de duración). Para realizar Cluster mejores se debe tener en consideración que tan importantes son algunos clientes al proveedor o que tan seguido requieren de la visita para reponer los productos.

Para el caso de prueba a realizar el componente de Cluster no será implementado.

4.1.3 Módulo de Proyección

Con la gran información que se posee de los clientes, es posible para el proveedor poder realizar proyecciones a largo plazo (junto con los riesgos que significa realizar esto) para de esta manera prever problemas o las maneras de responder a los sucesos de mejor

manera y así obtener una mejor satisfacción en los clientes y una mejora en el manejo de los recursos utilizados interiormente en la empresa para así obtener más beneficios.

Para lograr esto este módulo se separa en los siguientes componentes: Predicción de Demanda y en el de Predicción de Recursos, como se indica en Figura 4.4.

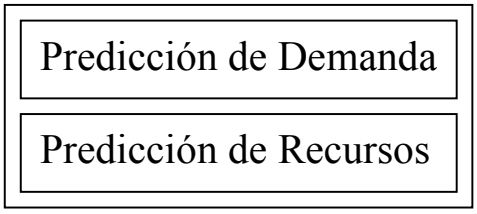

Figura 4.4: Arquitectura Módulo de Proyección

- **Componente de Predicción de Demanda**

Este componente es el encargado de prever la tasa de demanda de cada cliente dependiendo de la información de los inventario entregados por ellos. Al estar atentos y en proyectar esta tasa se pretende poder estar preparados para poder enfrentarse a momentos en que esta tasa pueda alterarse de gran manera, y de esta manera poder estar apegados lo más cerca de la realidad de sus clientes para así no obtener un stockout o un gasto de los recursos del proveedor innecesario al no poder enfrentarse al cambio. Este componente es el que entrega la tasa de consumo de los clientes necesaria en el otro módulo.

Para el caso de prueba a realizar el componente de predicción de demanda será en cierta manera simulado, como datos de entrada se utilizarán demandas esperadas para cada cliente en los próximos periodos, esto es algo que éste componente entrega como resultado, la simulación constara de un calculo de demanda por medio de un archivo Excel.

- **Componente de Predicción de Recursos.**

Aprovechando los datos de los clientes, el proveedor puede también poder realizar un análisis de la forma en que están ocupando sus propios recursos o inventarios. Con este análisis pueden cambiar algunas formas de la realización de sus procesos y de ser posible realizar una utilización de sus recursos de una manera mucho más óptima, comparada con la anterior utilizada.

Para el caso de prueba a realizar el componente de predicción de Recursos no será implementado.

4.1.4 Módulo de Interfaz

Por último tenemos es módulo de la Interfaz, este modulo nos permite tener una interacción entre el sistema y el usuario de este. Permitiendo poder obtener los resultados de los componentes anteriores de manera intercomunicándose con ellos, como también permite mostrar los reportes que se vayan generando

Este módulo interactúa con todos los demás módulos de la arquitectura del VMI en su globalidad como también con los componentes de cada módulo.

En el caso de prueba que se va a realizar este módulo se va a simular, es necesario mostrar los resultados obtenidos del módulo de IRP, estos resultados serán entregados de manera sencilla por medio de un documento de texto. Para obtener los valores necesario para realizar el caso también se ocupara como entrada un documento de texto. Por lo tanto este módulo se simulará básicamente por medio de archivos de texto.

4.2 Modelo Matemático

4.2.1 Inventory Routing Problems

Para enfrentarnos a la implementación del módulo de IRP se siguió como base lo propuesto en [7], el cual presenta una propuesta para descomponer este problema en dos fases, una fase de planeación donde ve la demanda de los clientes y si en una ruta este cliente debe ser visitado o no. Mientras que en la otra fase se presenta el enrutamiento de los vehículos para realizar lo indicado en la fase anterior. Para la primera fase no se tomo un modelo matemático en particular, ya que solo estaremos comprobando si la capacidad de los vehículos está ocupada, si no es así ver si se pueden ingresar más clientes, los cuales estén cerca del punto de reposicionamiento.

Para la fase 2 se ocupará el problema como un CVRP, en otras palabras a un problema de Capacitated Vehicle Routing Problem que nos indica que los vehículos tienen una capacidad conocida por lo cual las rutas tendrán la restricción de no superar esta capacidad.

El modelo matemático fue formulado tomando en base el modelo explicado en [18] y es el siguiente:

$$min \sum_{(i,j)\in R} c_{ij} y_{ij} \tag{4.1}$$

Sujeto a:

$$\sum_{1\le j\le n} y_{0j} = k \tag{4.2}$$

$$\sum_{1\le i\le n} y_{i0} = k \tag{4.3}$$

$$\sum_{1\le k\le K} v_{ij}^{k} = y_{ij} \qquad \forall i, j \tag{4.4}$$

$$\sum_{1\le j\le n} y_{ij} = 1 \qquad \forall i \tag{4.5}$$

$$\sum_{1\le i\le n} y_{ij} = 1 \qquad \forall j \tag{4.6}$$

$$\sum_{1\le i\le n} \sum_{1\le j\le n} d_i v_{ij}^{k} \le Q \qquad \forall k \tag{4.7}$$

$$k \le K \tag{4.8}$$

$$y_{ij}\ , v_{ij}^{k}\ \in\ \{0,1\}\ \ \forall (i,j) \in R, \forall k \tag{4.9}$$

$$\text{El conjunto R se define como: } \mathrm{R} = \{(i,j) : y_{ij} = 1\} \tag{4.10}$$

La ecuación (4.1) es la función Objetivo, que las rutas tengan el menor costo posible, c_{ij} es el costo que tiene visitar al cliente i y después al cliente j. Mientras que y_{ij} indica si se ha realizado un recorrido desde i hasta j. Las restricciones (4.2) y (4.3) indican que *k* es la cantidad de vehículos utilizados en la solución y que todos los que parten desde el proveedor deben volver a él. La restricción (4.4) se encarga de hacer obligatorio la asignación de un vehículo a la ruta (i,j) si esta es recorrida, la variable v_{ij}^{k} indica sí si $(v_{ij}^{k} = 1)$ o no $(v_{ij}^{k} = 0)$ se utiliza el vehículo k en el arco (i,j). La variable y_{ij} de las restricciones (4.5) y (4. 6) indica la activación del arco (i,j) y asegura que todo cliente es un nodo intermedio de alguna ruta. La restricción (4.7) observa que cada vehículo no sobrepase su capacidad, siendo Q la capacidad del vehículo y d_i la demanda del cliente *i*. La restricción (4.8) limita el número máximo de vehículos a utilizar lo cual el máximo es la cantidad de vehículos que el proveedor posee. Por último la restricción (4.9) indica que tanto la variable v_{ij}^{k} como la variable y_{ij} son binarias.

4.2.2Vendor Managed Inventory

Para esta parte del proyecto se va a basar en [19] y [20] pero se ira generalizando a una situación de más clientes para que el costo de enrutamiento sea más notorio que en la situación de un solo cliente. En [19] y [20] presenta un modelo de trabajo en el cual se considera la existencia de un proveedor junto con la de un sólo Comprador (Cliente), es en esta formulación del costo del sistema total en la que se va a basar el proyecto pero con las siguientes modificaciones:

- La cantidad de clientes es mucho mayor que uno solo, para el caso práctico de la simulación se tomarán la cantidad de 20 clientes.
- Al haber una cantidad mayor de clientes, el costo de transporte (enrutamiento) se convierte en algo importante, por lo que no se puede asumir como innecesario.

El modelo propuesto para la situación sin VMI en [19] (ver Figura 4.5) nos muestra que no hay integración entre el comprador y el proveedor, se tienen los costos por separado cada uno donde H y h es la carga de llevar el inventario por unidad del proveedor y del comprador respectivamente, mientras que C y c son los costos de colocar una orden en el proveedor y en el comprador respectivamente, por último Q y q son la cantidad de orden a entregar. Generalizando a más cliente (N es la cantidad de clientes) el modelo queda como se ve en la Figura 4.6, donde se agrega lo mismo anteriormente dicho más CV que es el costo de enrutamiento de los vehículos a la hora de realizar la entrega de los productos.

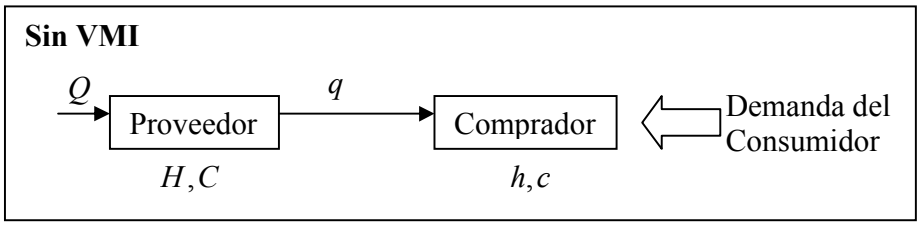

Figura 4.5: Modelo de trabajo sin VMI según [18].

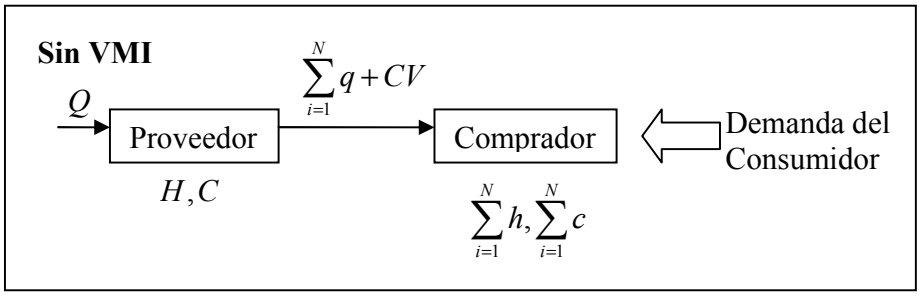

Figura 4.6: Modelo de trabajo sin VMI generalizado a más clientes.

Por otro lado en la situación en que se utiliza VMI en [19] (ver Figura 4.7) se muestra que existe una integración entre estos dos componentes de la cadena de suministro y el que toma las decisiones es el proveedor, por otro lado aún existe una c'aunque el valor es mucho menor que c ($c' < c$), generalizando a más clientes queda como Figura 4.8.

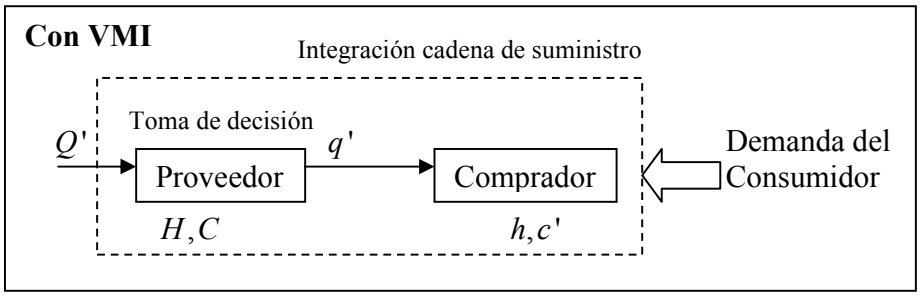

Figura 4.7: Modelo de trabajo con VMI según [18].

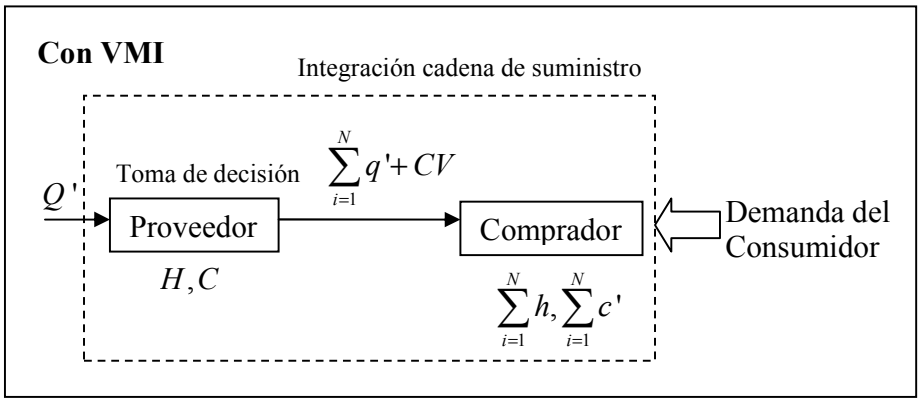

Figura 4.8: Modelo de trabajo con VMI generalizado a más clientes.

Análisis de Costo de Inventario

Para la situación sin VMI se utiliza el modelo EOQ para encontrar el tamaño óptimo de orden q, el EOQ y el costo total del inventario son calculados según [19] de las maneras mostradas en las ecuaciones (4.11) para el comprador y (4.12) para el proveedor, donde r y R es la demanda por unidad anual, y tc y TC es el costo total (las demás variables fueron explicadas anteriormente en el modelo de trabajo), por último * denota el valor óptimo. Uniendo las ecuaciones (4.11) y (4.12) para sacar el costo total del modelo, se tiene la ecuación (4.13) donde se da el supuesto que el proveedor vende todos sus productos, la demanda anual de los clientes es igual a lo producido por el proveedor $(R = r)$.

$$q^* = \sqrt{\frac{2cr}{h}} \; ; \; tc^* = hq^* = \sqrt{2crh} \tag{4.11}$$

$$Q^* = \sqrt{\frac{2CR}{H}} \; ; \; TC^* = HQ^* = \sqrt{2CRH} \tag{4.12}$$

$$TC^*_{\sin VMI} = \sqrt{2CRH} \cdot \sqrt{2cRh} \tag{4.13}$$

Ahora para pasar de esta situación a la situación donde se utiliza el VMI se tienen los siguientes cambios, ahora el proveedor tiene conocimiento de la demanda del cliente y es responsable de manejar ambas partes. En un periodo de tiempo, el pedido o la producción del proveedor dependerán de la frecuencia de reposición de productos al comprador $(Q = k_{VMI} q)$. El nivel del inventario del proveedor está determinado por Q, q y k_{VMI}. Cada vez que se envía q unidades al comprador el inventario del proveedor se va disminuyendo $(Q - q)$, luego en el próximo envío el nivel del inventario es $(Q - 2q)$ y así sucesivamente. Para simplificar el problema se va a suponer que cuando el $Q = 0$ de manera automática recibe Q productos nuevos. De esta manera el promedio de inventario queda de la siguiente manera:

$$I_p = Q - \frac{k_{VMI} - 1}{2} q \tag{4.14}$$

Y el costo total queda así:

$$TC_{VMI} = \frac{CR}{Q} + HI_P + \frac{c'R}{q} + \frac{hq}{2} \tag{4.15}$$

El primer término de la ecuación (4.15) es el costo total de ordenamiento del proveedor (Ordering Cost), el segundo término es el nivel de inventario promedio del proveedor (Holding Cost), el tercer término es el total costo del orden del comprador y el último es el inventario promedio del comprador.

Las formulas anteriormente mencionadas están tomando el caso de un proveedor con un solo comprador (el [19] y [20]) ahora generalizando ambas situaciones a una cantidad mayor de clientes, las formulas para el costo total de inventario se modificarían. Para la situación del no uso del VMI podemos simplificar que el proveedor sigue vendiendo todos sus productos anualmente a todos sus compradores (clientes) con lo cual $R = \sum_{i=1}^{N} r_i$. De esta manera el total de inventario en la situación sin VMI queda como la ecuación (6.16), donde N es la cantidad de clientes:

$$TC^*_{\sin VMI} = \sqrt{2CRH} \cdot R \sum_{i=1}^{N} \sqrt{2c_i h_i} \tag{4.16}$$

Para la situación del uso del VMI el promedio del inventario del proveedor queda de la siguiente forma:

$$I_p = Q - \sum_{i=1}^{N} \frac{k_{VMIi} - 1}{2} q_i \tag{4.17}$$

Y el costo total es:

$$TC_{VMI} = \frac{CR}{Q} + HI_P + R \sum_{i=1}^{N} \frac{c'_i}{q_i} + \sum_{i=1}^{N} \frac{h_i q_i}{2} \tag{4.18}$$

A estos costos se le deben considerar el costo de enrutamiento de los vehículos, este valor al cual se le va a llamar $CV_{\sin VMI}$ y CV_{VMI} será calculado por el programa que se esta haciendo en este proyecto. La diferencia entre ambos estará en la parte en que el $CV_{\sin VMI}$ sólo llegara a la etapa donde las heurísticas calcularán el valor de costo total, mientras que el CV_{VMI} se le agregará además una etapa de reajuste para utilizar los vehículos al 100%. Por último las formulas del Costo total del inventario están dadas por las ecuaciones (4.19) para situación sin VMI y (4.20) para la situación con el VMI. Indicar que I_{Ci} es el

inventario promedio de los clientes y que su valor es $\frac{q_i}{2}$ entregado por la política de inventario EOQ.

$$TC_{\sin VMI} = \sqrt{2CRH} \cdot R\sum_{i=1}^{N}\sqrt{2c_i h_i} + CV_{\sin VMI} \quad (4.19)$$

$$TC_{VMI} = \frac{CR}{Q} + HI_P + R\sum_{i=1}^{N}\frac{c'_i}{q_i} + \sum_{i=1}^{N} h_i I_{Ci} + CV_{VMI} \quad (4.20)$$

Como se menciono el $CV_{\sin VMI}$ y CV_{VMI} es el costo entregado por el programa de IRP, sin y con ajuste respectivamente, por lo que la función de costo del vehículo es como la ecuación (4.21). Donde la tercera función representa el valor de lo anteriormente mencionado, mientras que las dos primeras funciones indica que el costo esta basado en la distancia de los clientes y el valor a trasladar un producto.

$$CV = f(d_{ij}) + f(p) + f(\#\,\mathrm{Prod} \cdot \mathrm{V}) \quad (4.21)$$

Capítulo 5

Diseño de la Solución

A continuación se presentará un diseño para solucionar el módulo del IRP de la arquitectura propuesta anteriormente para el VMI. La Figura 5.1 presenta un esquema general del diseño, con ambas heurísticas, al final sólo se escogerá una (Tabu Search), esto será explicado en el capítulo de comprobación de heurísticas.

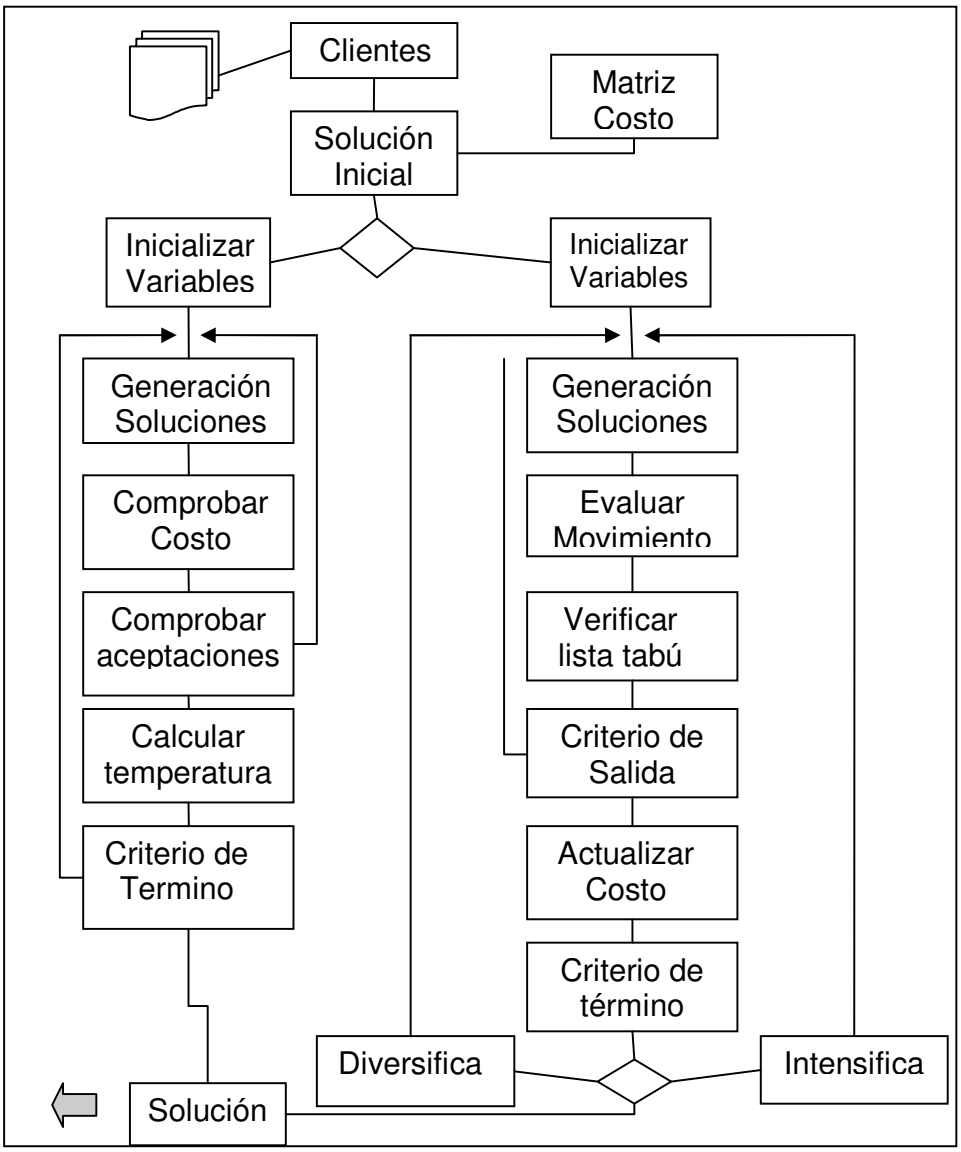

Figura 5.1: Esquema de diseño de algoritmo para el IRP.

5.1 Clientes

Como se presenta en este esquema, en un módulo aparte se realiza la determinación de los clientes que se van a visitar, este módulo es el que realiza la diferencia entre usar VMI o usar la manera tradicional, como ingreso de éste modulo son los informes de demanda de los clientes en el periodo que se quiere revisar. En estos informes también se indica la capacidad de los vehículos, la capacidad de las bodegas de los inventarios y el punto de reposicionamiento. Se trabaja con una demanda estocástica modelada en una función normal.

Existen dos formas de seleccionar los clientes, la primera nos lleva a realizar una situación sin VMI, ya que se toman las demandas de los clientes como ordenes, y el proveedor tiene que satisfacer estas ordenes sin importar que suceda en el próximo periodo, la segunda forma nos lleva a realizar una situación con VMI, ya que luego de ingresar los clientes en la lista de clientes (estructura mostrada en Figura 5.2) , la cual nos informa sobre el identificador del cliente y la demanda que requiere, se comprueba la cantidad de vehículos a utilizar, este valor se calcula viendo la demanda total de los clientes "obligatorios" dividido por la capacidad de los vehículos, como se muestra en la Figura 5.3 (que también nos presenta la lista de clientes y como los nuevos clientes se agregan al final de la lista). Para utilizar VMI se ve si queda espacio en los vehículos, y si es así se procede a tratar de ingresar otros clientes que estén cercanos al punto de reposicionamiento, para así completar la capacidad desperdiciada del vehículo, esto último repercute en los próximos periodos y es posible realizar ya que es el proveedor el encargado de manejar en inventario de sus clientes.

```
struct Demanda{
      struct Demanda *sig;
      int cliente;
      double demandaEntregada;
}*demandaCliente
```

Figura 5.2: Estructura Lista de Demanda

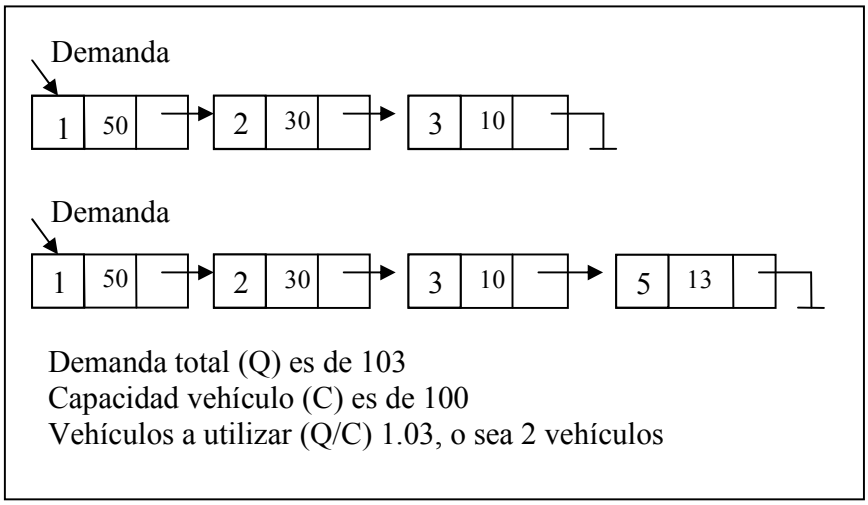

Figura 5.3 : Generación de lista de demanda.

5.2 Matriz de Costos

La matriz de costos va a ser de tamaño N (cantidad de clientes), sin importar si ese cliente fue visitado o no, la creación de ella depende de la distancia que existe entre cada uno de los clientes con los otros y también con el proveedor. Es éste criterio el que se utilizará para calcular el costo de enviar un vehículo a un cliente. A continuación se presenta como ejemplo una tabla de 9 clientes (ver

Tabla 5.1), la empresa que posee el "0" es el proveedor. Esta tabla posee todos los clientes, sin importar si es visitado o no, y debería ser generada una sola vez, al menos que se agreguen más clientes.

Tabla 5.1: Ejemplo de la matriz de costo con 9 clientes

	0	1	2	3	4	5	6	7	8	9
0	0	100	120	50	254	321	154	241	154	122
1		0	21	48	36	211	100	142	120	254
2			0	111	52	61	22	21	145	125
3				0	25	154	21	322	90	354
4					0	112	154	123	88	120
5						0	54	102	21	252
6							0	111	101	263
7								0	45	221
8									0	151
9										0

5.3 Generación de una Solución Inicial

Simulated Annealing y Tabu Search poseen algunas similitudes, una de ellas es que trabajan sobre una solución inicial para poder llegar a una solución óptima. El cálculo de una solución inicial por lo tanto es realmente importante, para realizar esto primero que nada se necesitará la lista de la demanda de los clientes, el cual posee el dato de identificación del cliente, junto con la demanda que este requiere. Con esta lista se empezarán a tomar a clientes de forma aleatoria y se irán agregando a una ruta, donde cada ruta representa a un vehículo y no se viola las restricciones que cada ruta debe tener. Si es factible encontrar una solución se debe escoger la heurística que se va a utilizar, por el contrario si no es posible realizar una solución factible se avisa en el programa para que el proveedor vea la cantidad de clientes se va a requerir o si agrega más recursos para cumplir. Sobre las rutas, estas serán representadas por una lista donde cada nodo representará a un cliente. La estructura a utilizar es la mostrada en Figura 5.4, como se ve se utilizan dos puntero que nos mostrará el cliente anterior y el siguiente, esto es para facilitar luego los movimientos que se puedan realizar sobre ella. A cada cliente se le especifica la demanda que se le entrego, y el costo que se efectuó al visitar al cliente. Este costo es calculado dependiendo de cliente anterior y se saca de la matriz de costo que fue antes calculada. Para la visita de cada ruta generada es por medio de un arreglo de puntero de tamaño K, donde K es la cantidad de vehículos que el proveedor posee, con esto se indica que cada vehículo sólo tiene una ruta.

```
struct Ruta{
      struct Ruta *ant;
      struct Ruta *sig;
      int cliente;
      int demandaEntregada;
      double costocliente;
}*Rutas[K]
```

Figura 5.4: Estructura de las Rutas.

La Figura 5.5 representa el estado inicial de lo descrito sobre la generación de la solución inicial, donde se escoge al azar el primer cliente de la lista de demanda y las rutas no tienen aún ningún cliente en ellas. La capacidad del vehículo es de 70 y son 3 vehículos los que posee el proveedor en este ejemplo.

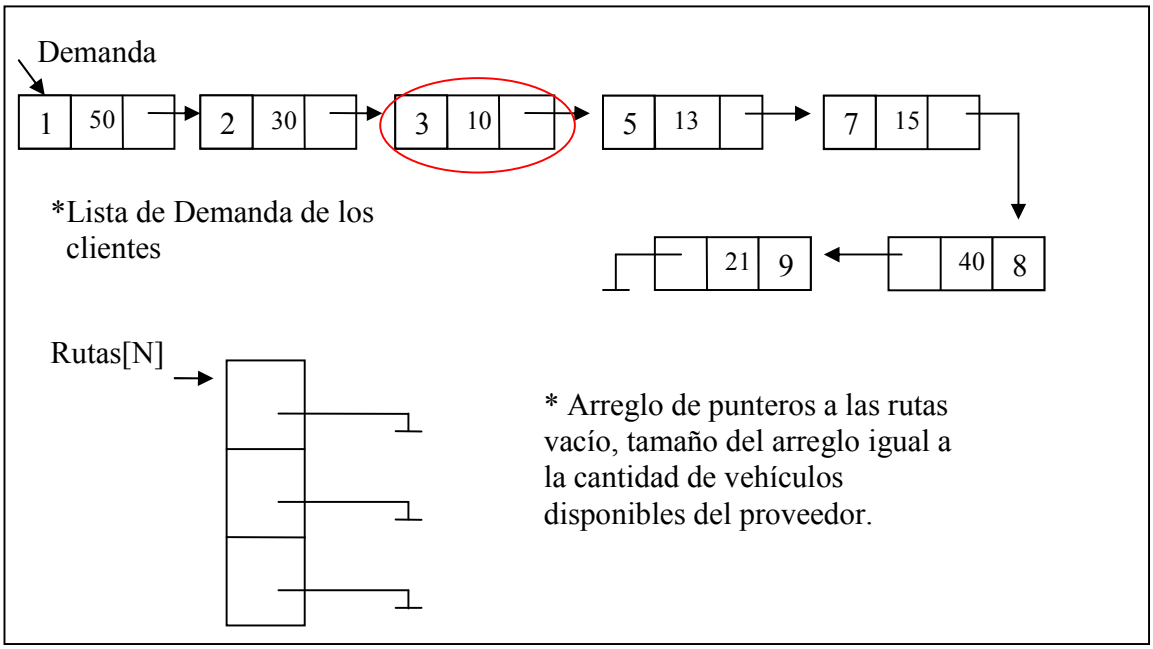

Figura 5.5: Estado inicial de la generación de solución inicial.

La Figura 5.6 representa el ingreso del cliente a una ruta, como no supera la capacidad del vehículo se ingresa, la capacidad total del vehículo disminuye (se le resta la demanda) y como es el primero en ser ingresado a una ruta el costo de esta operación es el costo del vehículo i desde el proveedor hasta el cliente 3, el costo de los demás ingresos de clientes dependerán del cliente anterior y se sacará el costo de la matriz de costo. Se escoge el segundo cliente a ingresar en la ruta al azar.

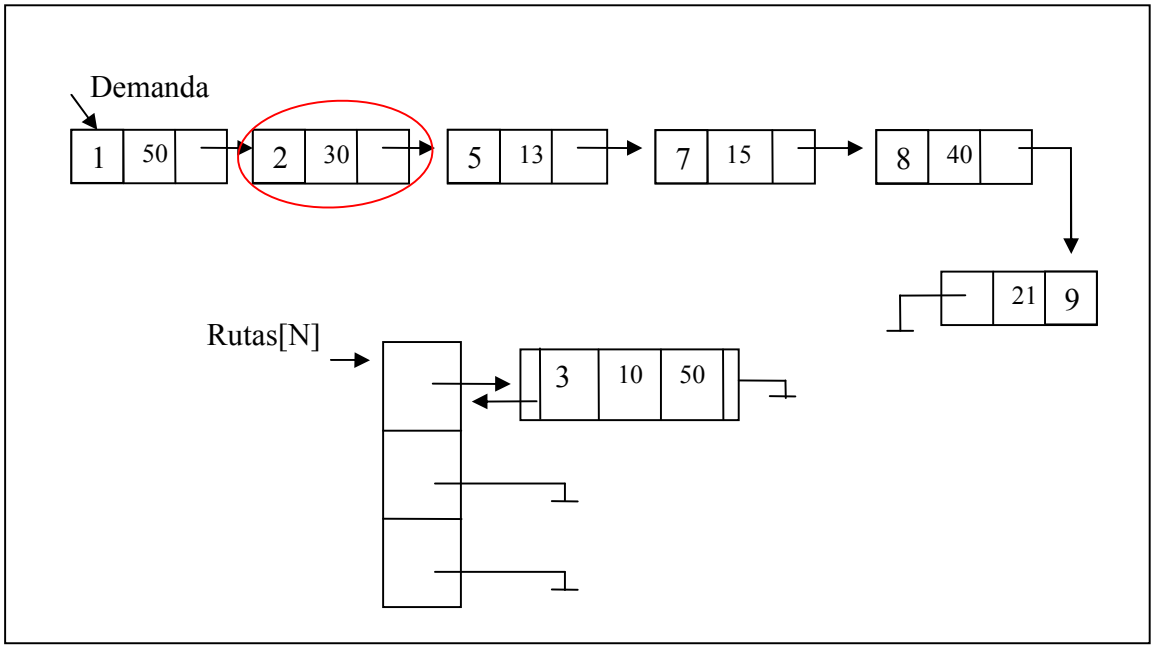

Figura 5.6: Cliente ingresado a una ruta y asignado un costo.

En Figura 5.7 se representa la solución inicial de rutas ya lista con los clientes escogidos, cada ruta cumple con la restricción de no superar la capacidad del vehículo (ya que cada vehículo representa una ruta). Y se logró colocar a todos los clientes en una ruta, quedando la lista de demanda vacía. Al final de cada ruta se debe de regresar al proveedor el cual se representa como cliente 0 con demanda 0.

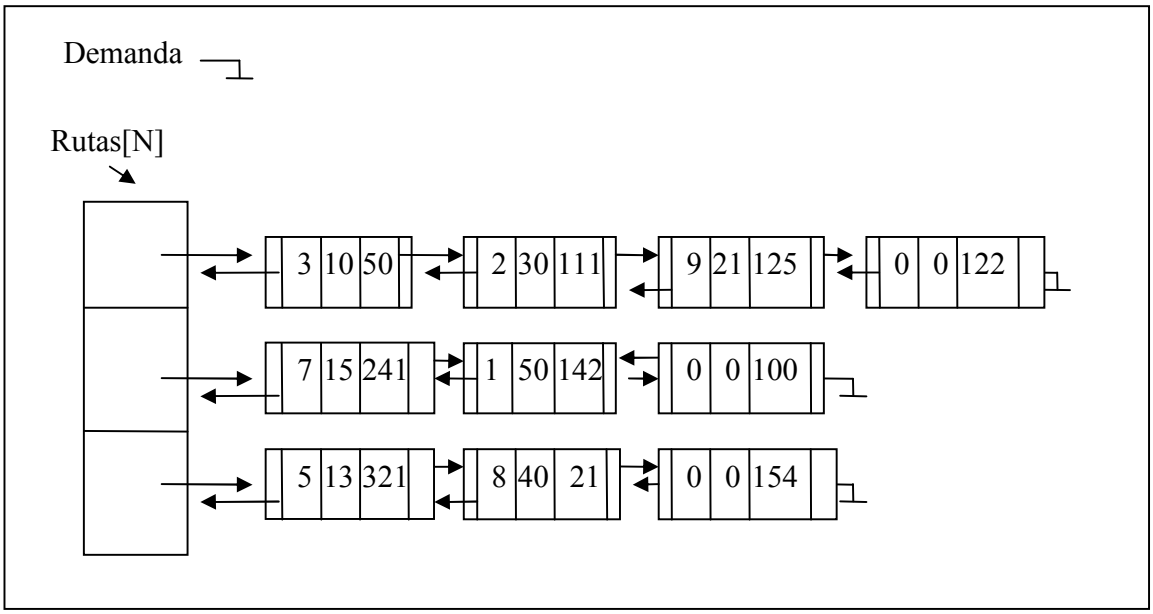

Figura 5.7: Solución inicial

5.4 Mecanismo de Generación de Vecindades

Otra cosa en común en Simulated Annealing y Tabu Search va a ser la forma de mecanismo de generación de vecindades. Para la realización de nuevas vecindades (generación de una nueva solución) a partir de una solución anterior, se presentan tres formas: Intercambiar clientes de una ruta por otro de otra ruta (forma 1), colocar un cliente de una ruta en otra ruta (forma 2), y por último escoger un cliente de una ruta y cambiarlo de posición en ésta (forma 3).La forma de escoger que forma se va a utilizar es de manera aleatoria (por probabilidad), aunque la forma 2 tendrá una mayor porcentaje de aparición por ser la que más cambios en el costo puede provocar (al realizar pruebas en el programa se logró llegar a esta conclusión). Se escogerán las rutas al azar (ver Figura 5.8 para el caso donde se escogen dos rutas) y el cliente será escogido de la misma manera (ver Figura 5.9 para el caso donde se tienen dos rutas) pero nunca será escogido el proveedor, el cual siempre debe estar al final de una ruta. Para no perder el orden de la solución anterior se duplica la solución en una solución auxiliar con la cual se va a trabajar y modificar.

Los clientes escogidos en la forma 1 son intercambiados solamente si al pasar a la otra ruta siguen respetando las restricciones de la capacidad del vehículo, como se representará desde Figura 5.10, los costos se cambian por los costos nuevos al cliente que se intercambio, como también del cliente siguiente ya que su costo depende del cliente anterior.

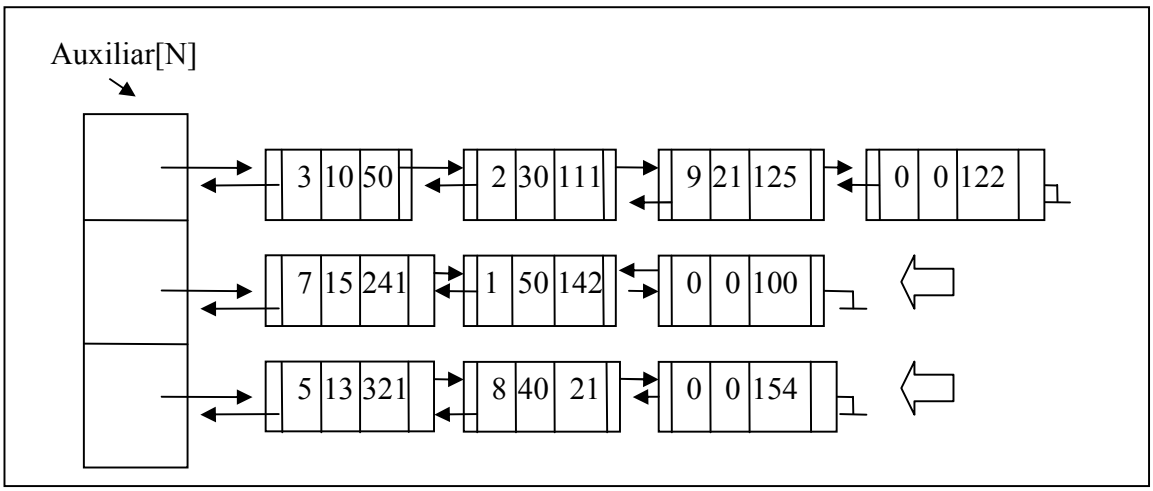

Figura 5.8: Se escogen dos rutas al azar

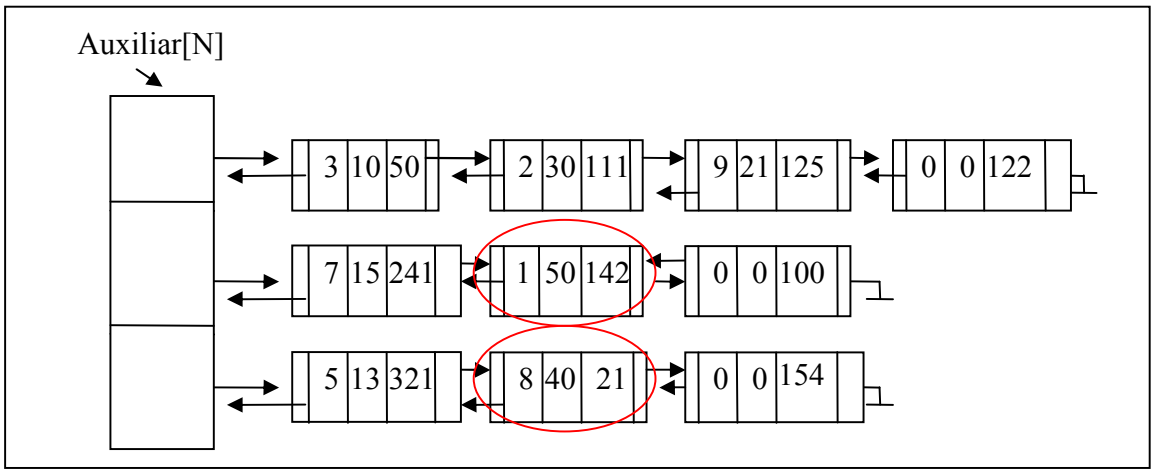

Figura 5.9: Se eligen un cliente al azar de las rutas escogidas.

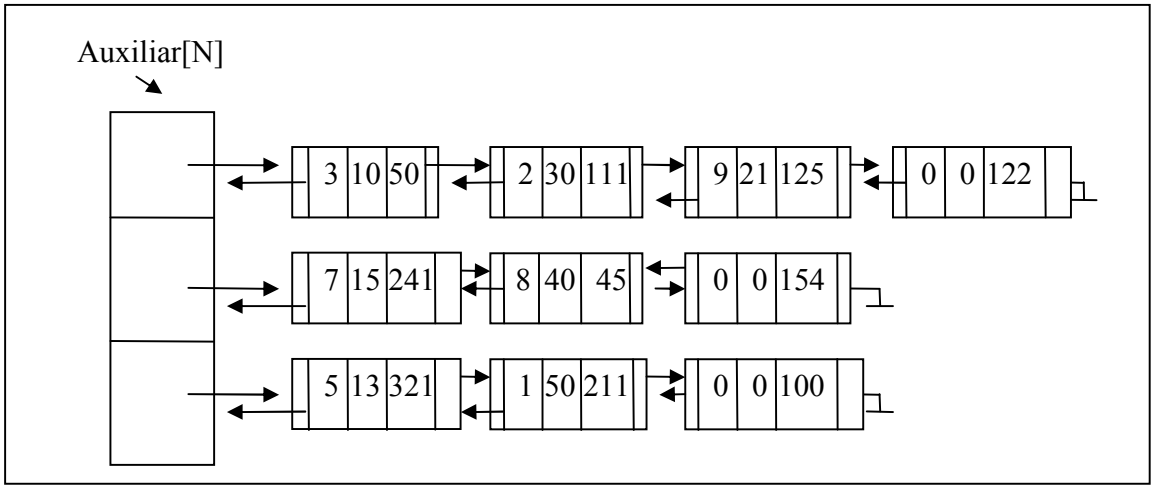

Figura 5.10: Intercambio de clientes entre las rutas

De la forma 2, el cliente escogido es colocado en una posición al azar en otra ruta solamente si al hacer esto se sigue respetando la restricción de capacidad, ver Figura 5.11 e Figura 5.12, cambiando los costos también de los clientes siguientes (tanto de la ruta en que se quito al cliente como la que se le agregó) y del cliente escogido.

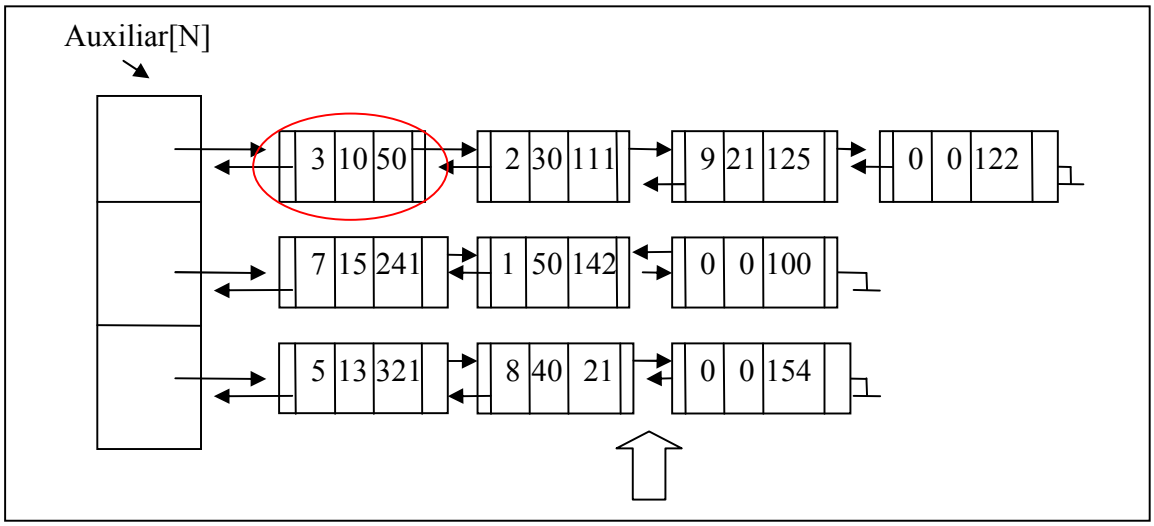

Figura 5.11: Escoge un cliente de una ruta y la posición a donde colocarlo.

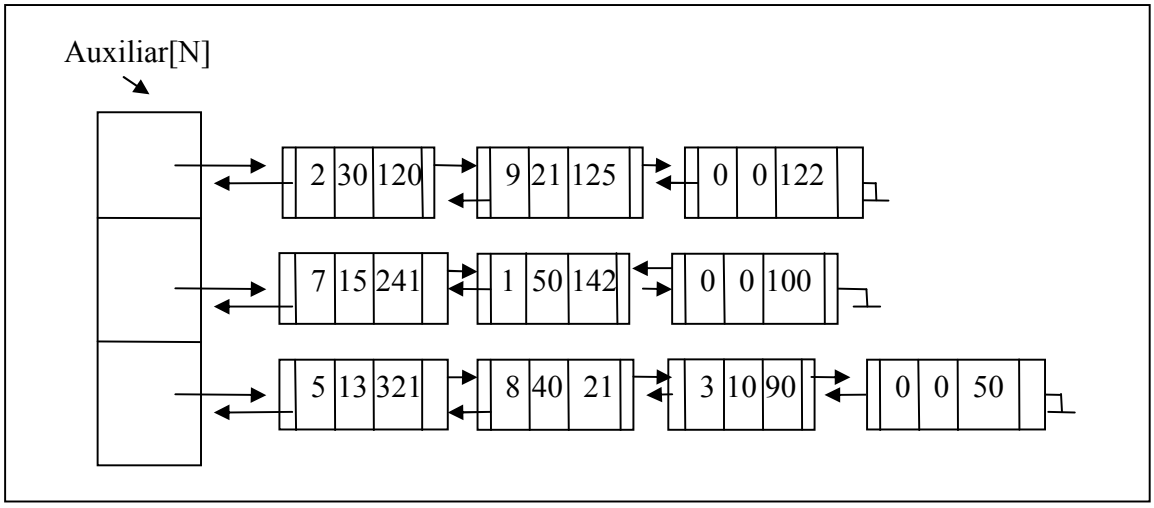

Figura 5.12: Quitar un cliente de una ruta y colocarlo en otra ruta

Por último de la forma 3, el cliente escogido es colocado en una posición al azar en la misma ruta donde pertenece el cliente, la posición escogida no debe ser la posición anterior o después del cliente escogido. Para que una ruta pueda ser escogida para realizar esta forma debe contar por lo menos con 3 clientes en ella, ver Figura 5.13 y Figura 5.14. Se cambian los costos del cliente anterior y del siguiente donde el cliente cambia de posición, y los costos del cliente que estaba después del cliente escogido antes de que éste cambie de posición.

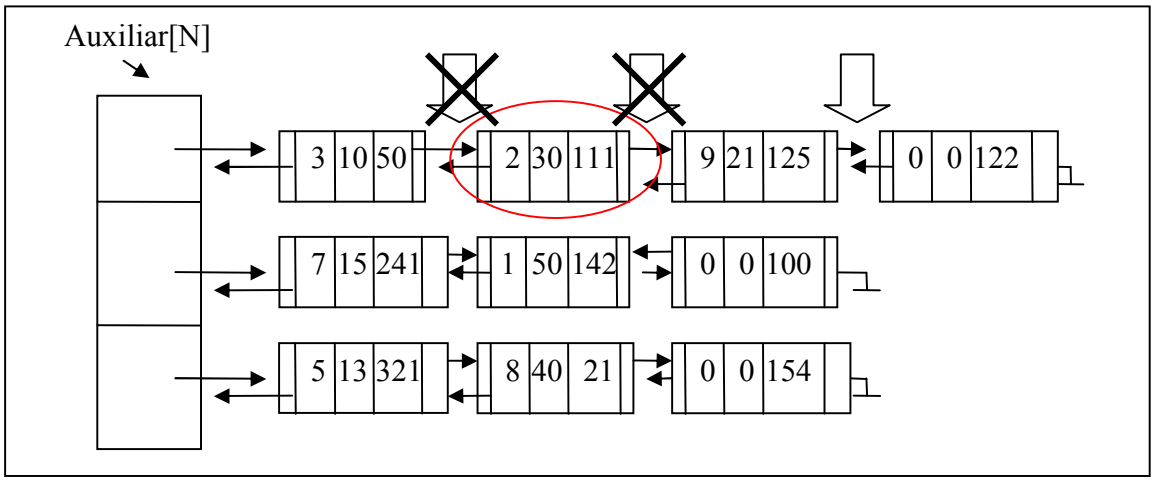

Figura 5.13: Escoge un cliente de una ruta y la posición a donde colocarlo.

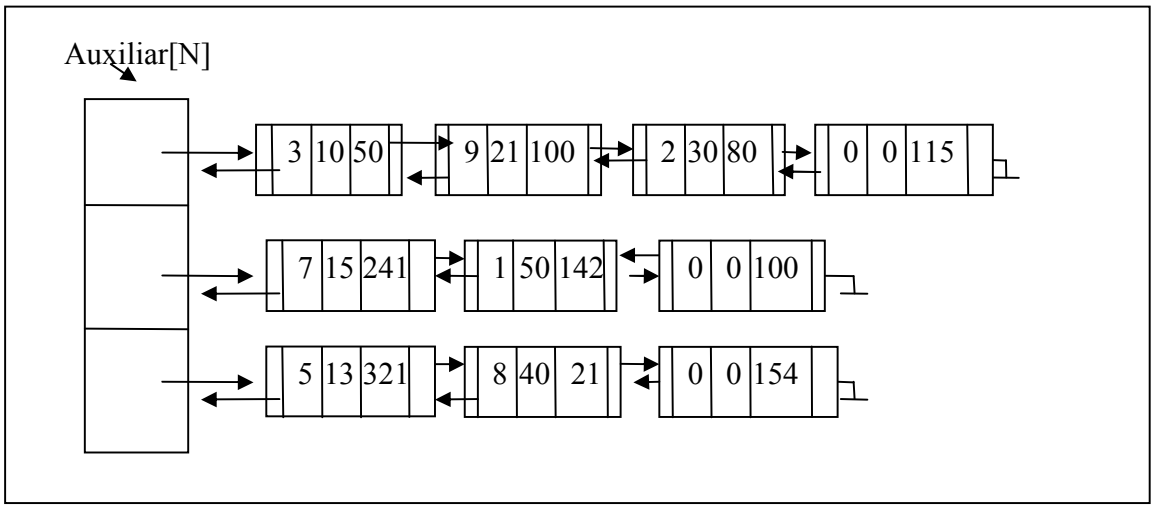

Figura 5.14: cambiar de posición el cliente en la ruta.

5.5 Simulated Annealing

Un esquema de cómo se realizará el funcionamiento de esta heurística sobre la solución inicial generada se presenta en Figura 5.15. De este esquema ya se explico como se Generaba la solución Inicial y los Mecanismos de Generación de vecindades anteriormente, ya que ambas cosas son algo común en ambas heurísticas.

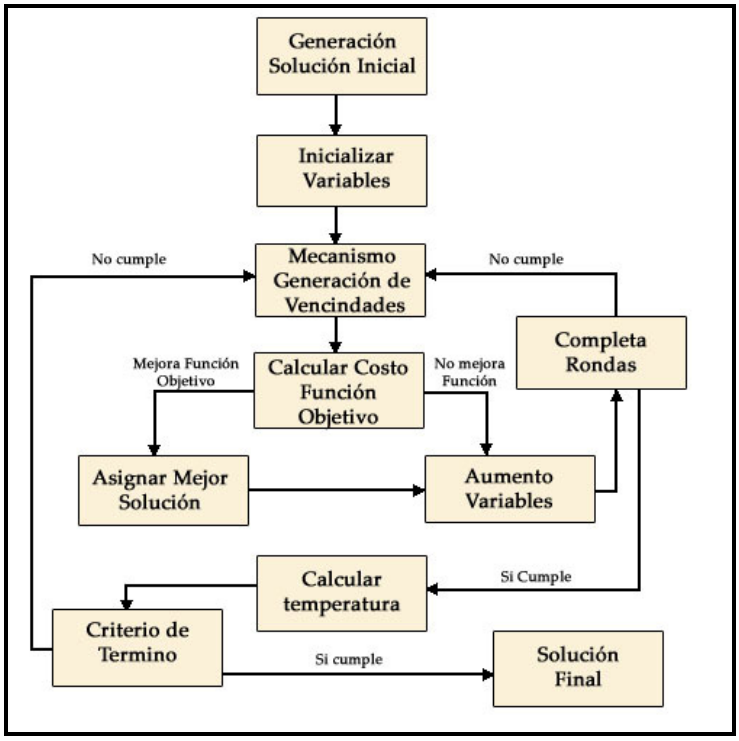

Figura 5.15: Esquema del Algoritmo Simulated Annealing utilizado.

- **Inicializar Variables.**

Las variables que se requieren inicializar son la temperatura inicial, y el número de aceptaciones por nivel de temperatura para mantener un equilibrio, T_0 y N_0 respectivamente. También se procede a realizar una duplicación de la solución inicial, para que en esta se realicen los cambios necesarios.

La temperatura inicial será calculada como se menciona en la página [12], la probabilidad de aceptación en la temperatura a utilizar será de 0.8, con esto nos referimos que hay un 80%, el promedio de los costos de la matriz de costo (fórmula en sección de Simulated Anealing). Cabe mencionar que los valores de las temperaturas serán guardados en un arreglo de tamaño del número de cambio de temperatura permitido.

La Solución inicial que se genera es guardada como solución anterior y también como mejor solución y es con esta con la cual se va a comparar las soluciones generadas por el mecanismo de vecindades en un principio hasta obtener (si es posible) una solución que sea con menos costo.

- **Calcular Costo Función Objetivo**

Cada vez que se realiza una generación de vecindad se esta modificando la solución anterior (Sol_anterior[k]) para ver si es posible obtener una solución mejor (Sol_mejor[k]), ambas conservan la misma estructura que las rutas. Para ver el costo de una solución

simplemente se procede a realizar una sumatoria de los costos de cada cliente en las rutas, como se ve en Figura 5.16.

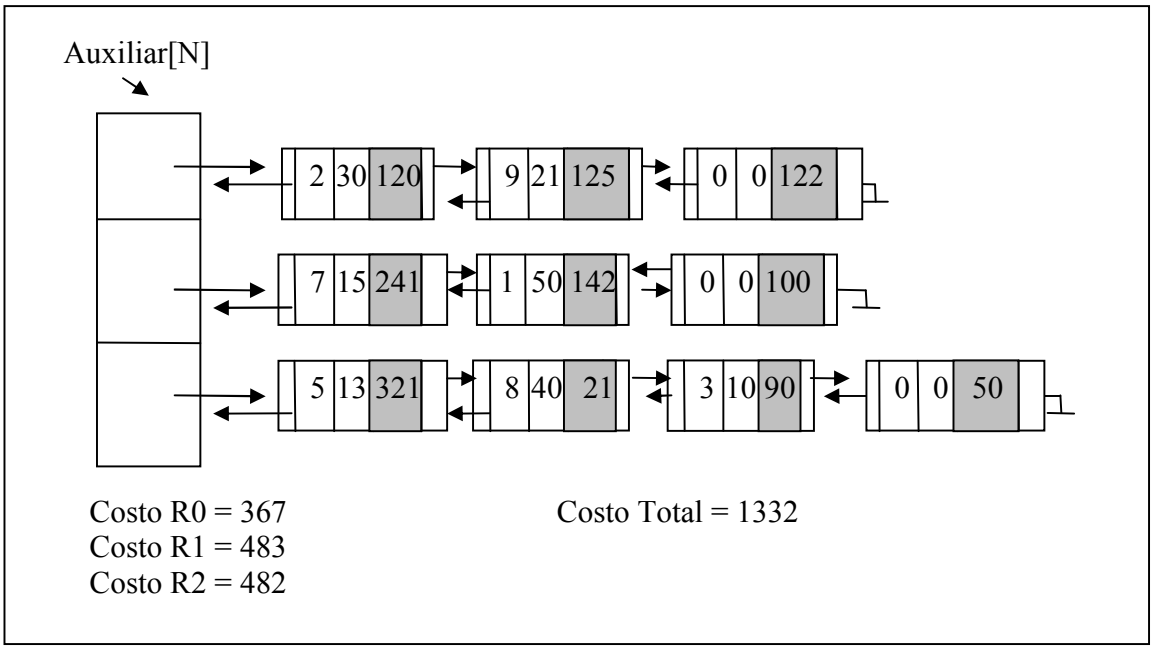

Figura 5.16: Calculo del costo de una Solución.

Cuando se realiza el cálculo de la solución generada (Sol_nueva[k]) y el de la mejor solución (Sol_mejor[k]) se puede presentar lo siguiente:

1. Solución que se genero (Sol_nueva[k]) entrega un costo menor que la mejor solución (Sol_mejor[k]).
2. Solución que se generó (Sol_nueva[k]) entrega un costo mayor que la mejor solución (Sol_mejor[k]).

Si sucede lo primero esta solución generada pasa a convertirse en la mejor solución (Sol_mejor[k]) y en la solución anterior (Sol_anterior[k]), con esto decimos que será con esta solución con que se comparará luego para ver si hay una solución más óptima, y de esta solución se va a efectuar modificaciones para obtener otra solución nueva.

Por el contrario, si el camino a tomar es el siguiente se debe calcular la función de probabilidad de aceptación con la diferencia de los costos de las dos soluciones junto con la temperatura del nivel en que se está el algoritmo. Este resultado lo comparamos con un número aleatorio que va de 0 a 1 que nos indica una probabilidad. Si el resultado es aceptado, la solución pasa a reemplazar la solución anterior (Sol_anterior[k]), lo que significa que en la siguiente vuelta es a esta nueva solución con la que se va a trabajar para conseguir una nueva. Por el contrario si esta solución no es aceptada, la solución anterior queda siendo la misma.

- **Aumento de variables**

Al entrar en este módulo se realiza a aumentar el contador de las aceptaciones por nivel de temperatura el cual luego es un factor para determinar si se debe salir de este nivel, y si se da el caso también aumenta la variable de que no ha habido cambio del valor de la mejor solución.

- **Completa Rondas**

Aquí se verifica si se ha llegado a la cantidad de soluciones aceptadas por el nivel de temperatura, si se alcanzo para a aumentar la temperatura, si no lo alcanza aún reinicia el proceso de modificar la solución anterior para generar una solución nueva que cumpla con la disminución de los costos. En este módulo se ve también si la función objetivo ha disminuido con cada aceptación o no.

- **Calcular Temperatura**

La nueva temperatura se calcula utilizando el valor de la temperatura anterior ocupando la fórmula $T_{k+1} = BT_k$, donde como valor para B se utilizara 0.30 (por el momento hasta encontrar un mejor valor por medio de análisis de sensibilidad). Este valor será asignado al arreglo en la posición que indique el contador de niveles de temperatura.

- **Criterio de término**

En este módulo se comprueba si se han cumplido con la cantidad de niveles de temperatura que se ha entregado, o si la función objetivo no se ha modificado en una cierta cantidad de niveles de temperatura. Si esto sucede se acepta la mejor solución encontrada como solución final y es entregada al usuario junto con su costo. Otro motivo de que el código termine su ejecución, pero casi imposible que suceda antes de lo mencionado anteriormente es si la temperatura llega a 0. Si el algoritmo no cumple ninguno de estos criterios sigue con el procesamiento desde el inicio pero con la temperatura cambiada. También se modifica el numero de aceptaciones por nivel por temperatura, multiplicando el número por el valor de número en que es visitado este módulo, para así tener una mejor búsqueda cada vez que la temperatura disminuye (una búsqueda más intensa).

Cuando se acepta el término del algoritmo, se entrega la solución final la cual es la Sol_mejor[k] que quedo al final de todo, y nos indica la formación de las rutas y el costo final. Para ver rendimiento del algoritmo también se presentara tiempo en que se demoró en encontrar la solución y cantidad de iteraciones que se tuvo que realizar.

5.6 Tabu Search

Un esquema de cómo se realizará el funcionamiento de esta heurística sobre la solución inicial generada se presenta en Figura 5.17.

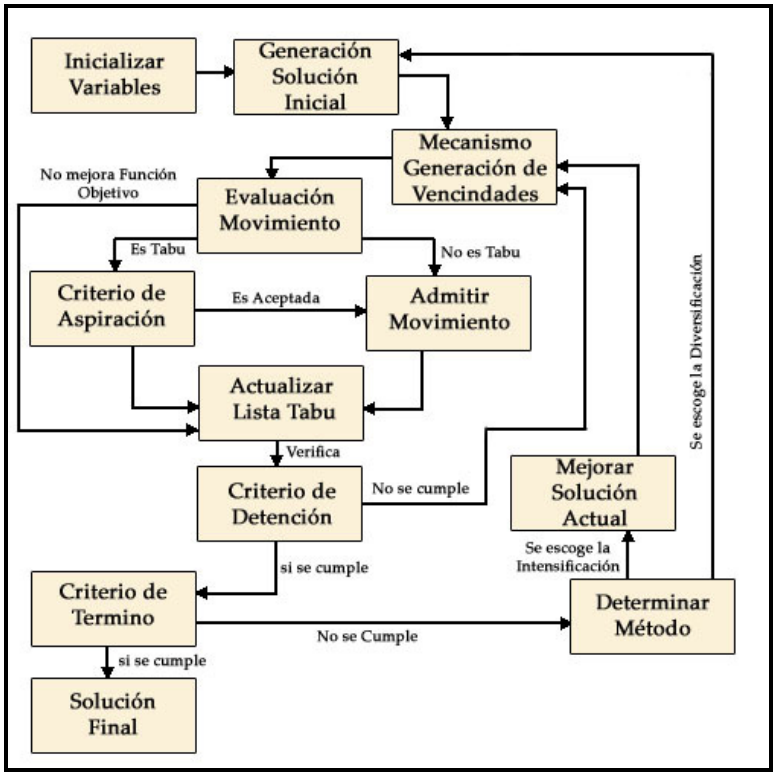

Figura 5.17: Esquema del algoritmo Tabu Search utilizado.

- **Inicializar Variables**

En este módulo se inicializa la lista tabú, esta lista va a tener un largo variable ya que en sí no tendrá un tamaño prefinido sobre la lista, aunque cada restricción tabú tendrá un número límite de permanencia (tiempo), con lo que en cierta manera se puede decir que se esta verificando que la lista tabú no aumente de manera considerable. La estructura de la lista tabú será la mostrada en Figura 5.18., donde se guarda el identificador del cliente, el número de la ruta donde es movido y el contador de tiempo de cuanto lleva este movimiento tabú en la lista para evaluar si se revoca o sigue en el lugar.

```
struct ListaTabu{
        struct ListaTabu *sig, *ant;
        int cliente;
        int numeroruta;
        int posicion;
        int permanencia;
}*Tabu
```

Figura 5.18: Estructura de la lista Tabú

Otras variables que se inicializan es indicar el número máximo que un elemento puede permanecer en la lista tabú, aquí este número será el mismo que la cantidad de clientes que hay, el número de iteraciones necesarias en el criterio de término cuando la

función objetivo no varia su valor (se pretende dejar el número igual al número de rutas que se formaron en la solución inicial).

- **Evaluación Movimiento**

En este módulo la solución generada por el mecanismo de generación de vecindades es contrastada con la lista tabú para ver si el movimiento recién realizado esta en ella, la búsqueda por la lista tabú será de la siguiente manera: se busca al cliente en la lista, si se encuentra se verifica a qué ruta indica que es tabú ser transferido y si es la misma ruta en el cual se ha movido la última vez es movimiento tabú, se realiza la búsqueda en toda la lista ya que es posible que el cliente este repetido en la lista. Si el movimiento que se realizó no es tabú se pasa a verificar si minimiza el costo de la función objetivo, si lo hace se traslada a admisión de movimiento, y si no a actualizar lista.

Por ejemplo la Figura 5.19 nos indica que se quiere realizar el intercambio de los clientes de la ruta, el cliente 1 va a ir a la ruta 2 (el índice del arreglo de punteros indica el número de ruta, tomando que el índice comienza en 0, cuando se presente al usuario el número de ruta el número a indicar será el índice más 1), mientras que el cliente 8 va a ir a la ruta 1. Se efectúa el movimiento y se busca en la lista tabú si esos movimientos los son, en Figura 5.20 mostramos que no es movimiento tabú, ya que la lista tabú tiene dos movimientos los cuales son de que el cliente 3 no puede ir a la ruta 1 y el cliente 5 no puede regresar a la ruta 1, mientras que el tercer número nos indica hace cuantas iteraciones este movimiento fue ingresado en la lista. El movimiento de cliente 1 a ruta 2 o cliente 8 a ruta 1 no es tabú.

Si esto sucede se ve si el costo de esta nueva solución es menor que el de la mejor solución (Sol_mejor[k]), o no. Si el costo es menor pasa a la etapa de Admisión de movimiento. Por el contrario si el movimiento no mejora la solución pasa a la etapa de actualizar lista tabu.

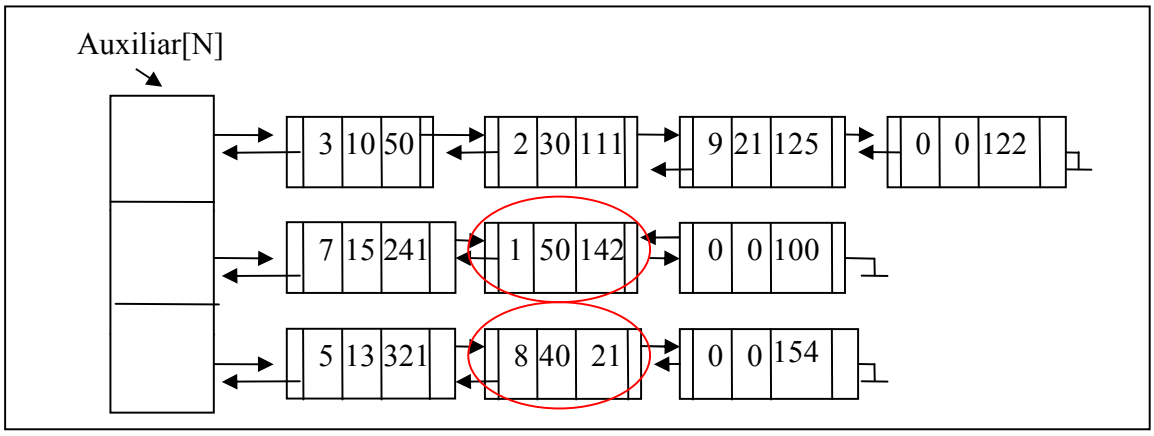

Figura 5.19: Se eligen un cliente al azar de las rutas escogidas para intercambiarse.

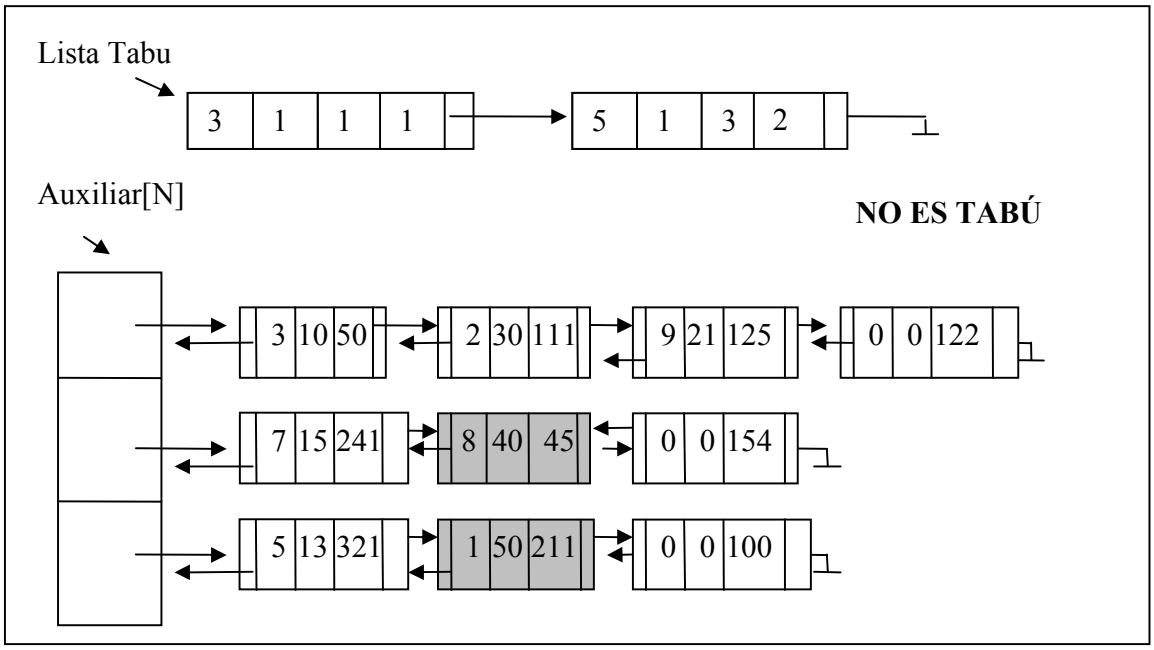

Figura 5.20: Búsqueda si el movimiento es tabú, el movimiento no resulta serlo.

La Figura 5.21 nos muestra el caso contrario, un movimiento realizado resulta ser un movimiento tabú, se comprueba si el valor de la solución mejora a la mejor solución encontrada por el momento (Sol_mejor[k]), si esto sucede por lo que pasa a la fase de comprobar los criterios de aspiración

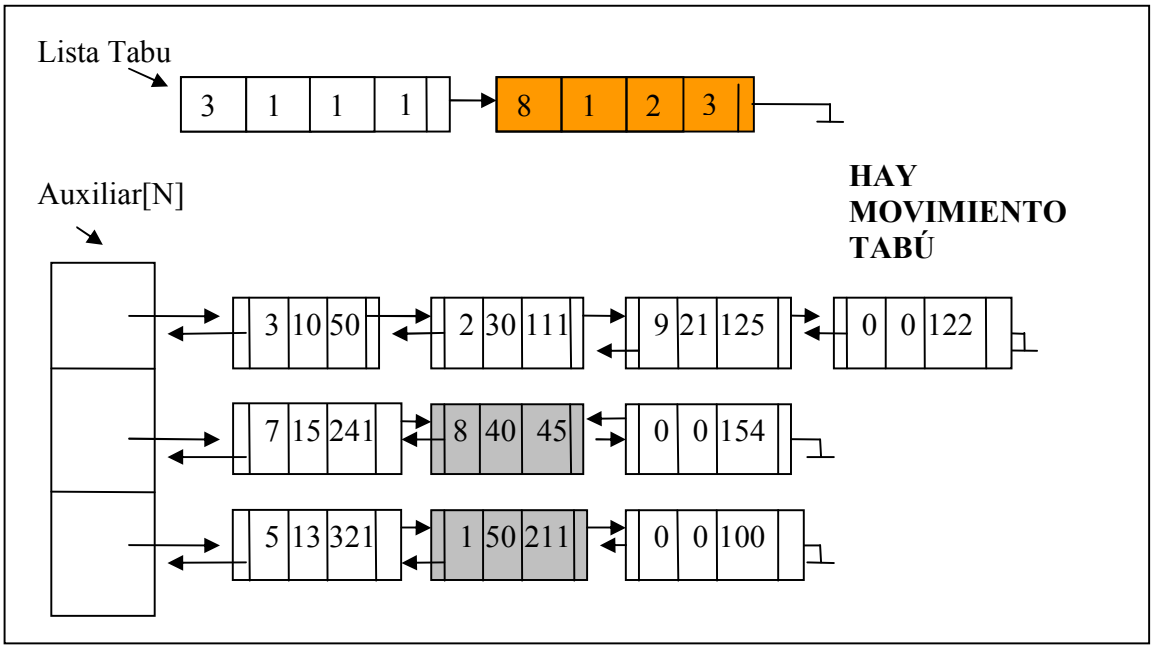

Figura 5.21: Se registra que un movimiento anterior es movimiento Tabú.

- **Admitir movimiento**

Cuando una solución pasa por este proceso significa que minimiza la función objetivo, es por esta razón que la solución ingresada pasa a ser la mejor solución (Sol_mejor[k]).

- **Criterio de Aspiración**

Uno de los principales criterios de aspiración es si el movimiento Tabú genera que el costo de la función objetivo se minimice (Auxiliar[k] < Sol_mejor[k]), si esto sucede el movimiento Tabú saldrá de la lista Tabú y la solución será aceptada. El otro criterio que se va a utilizar es que cuando un movimiento entra a formar parte de la lista tabú se le asigne un contador que va aumentando cada vez que se haga una operación de ingreso en la lista, si este contador llega a un número dado (puede ser el doble de la cantidad de rutas que se ocupan) el movimiento sale de la lista, ya que se puede considerar que a esa altura la solución presentada y analizada no tiene parecido a la solución cuando el movimiento paso a ser tabú.

- **Actualizar lista Tabú**

En este módulo se ingresa el movimiento tabú con el contador igual a 0 ya que es reciente, esto solamente si se llega a este proceso después de determinar si cumple con el criterio de aspiración. A todos los elementos de la lista se van aumentando su contador, y si en alguno de ellos alcanza el número límite se saca de la lista Tabú ya que cumple con un criterio de aspiración. Tomando como ejemplo el caso de que el movimiento no es Tabú (ver Figura 5.20), minimice o no la función objetivo, se debe agregar el movimiento que se realizo a la lista Tabú (ver Figura 5.22) y aumentar el contador de cada movimiento de la lista para ver el tiempo de posición en la lista.

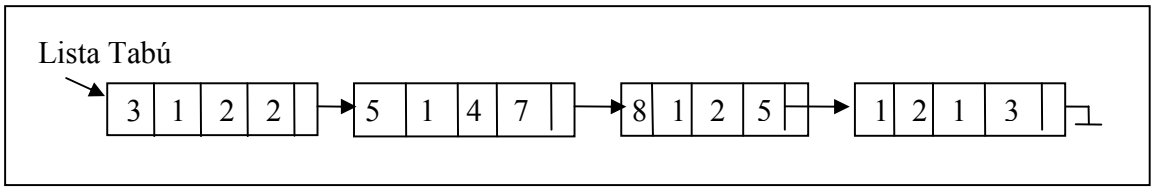

Figura 5.22: Actualización de la lista de tabú, ingresando movimientos Tabú

Cuando un movimiento de la lista tabú alcanza el límite de permanencia en la lista, que será igual al número de clientes, este movimiento dejará de ser un movimiento tabú y se quitara en la lista, como se puede apreciar en Figura 5.23.

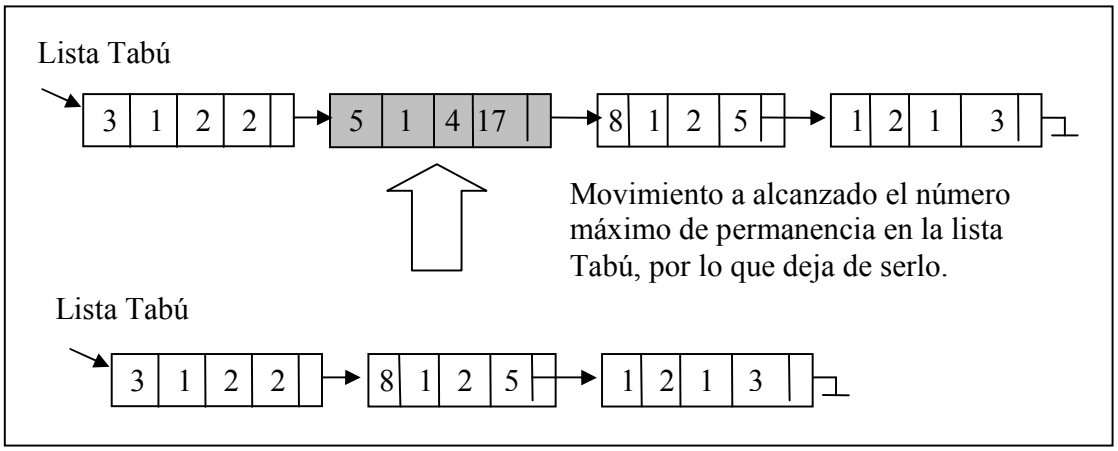

Figura 5.23: Eliminando movimiento tabú que alcanza su límite

- **Criterio de detención**

Se detiene si se a alcanzado un número máximo de iteraciones ya antes predefinidos, avanza hacia el proceso de criterio de termino, si no es así vuelve a la generación de una nueva solución.

- **Criterio de término**

El algoritmo puede terminar de la siguiente manera: si la mejor solución encontrada no varía de la anterior en una determinada cantidad de vueltas (se dejara con el número de la cantidad de rutas que tiene la solución). Otra manera es si se conoce el costo mínimo, que es posible para los casos de pruebas y detenerlo por si alcanza ese valor o un valor menor, de esta manera analizar mejor cuantas iteraciones son requeridas (proporcionalmente con el tamaño del problema) para llegar al valor y así determinar la cantidad de iteraciones necesarias de una mejor manera.

- **Determinar Método**

Cuando el criterio de término no se cumple, se puede seguir dos caminos: Diversificar o Intensificar. Diversificar se trata de probar con otra vecindad de soluciones, para eso se vuelve a llamar el módulo de Generar una solución inicial, ya que de manera aleatoria va a generar otra solución y la probabilidad de que sea la misma que la anterior es bastante difícil.

Por otro lado si se escoge Intensificar se va a tratar de mejorar la solución actual (si esto es posible), para esto se pretende reorganizar a los clientes en la rutas donde se encuentran, ya que al generarse las soluciones de manera aleatoria y los cambios que se realizan en ella de la misma manera puede haber la posibilidad de que los cliente no hayan quedado en el mejor orden, la forma de realizar esto último es empezar desde el primer nodo de una ruta, el cual por ser el primero se conoce que el empieza desde el proveedor, el cual es identificado como “0” y se busca a otro cliente que disminuya el costo si sale desde

el proveedor él en vez del otro cliente, el valor de los costos de los movimientos se encuentran en la matriz de costo antes realizada (como se ve en Figura 5.24).

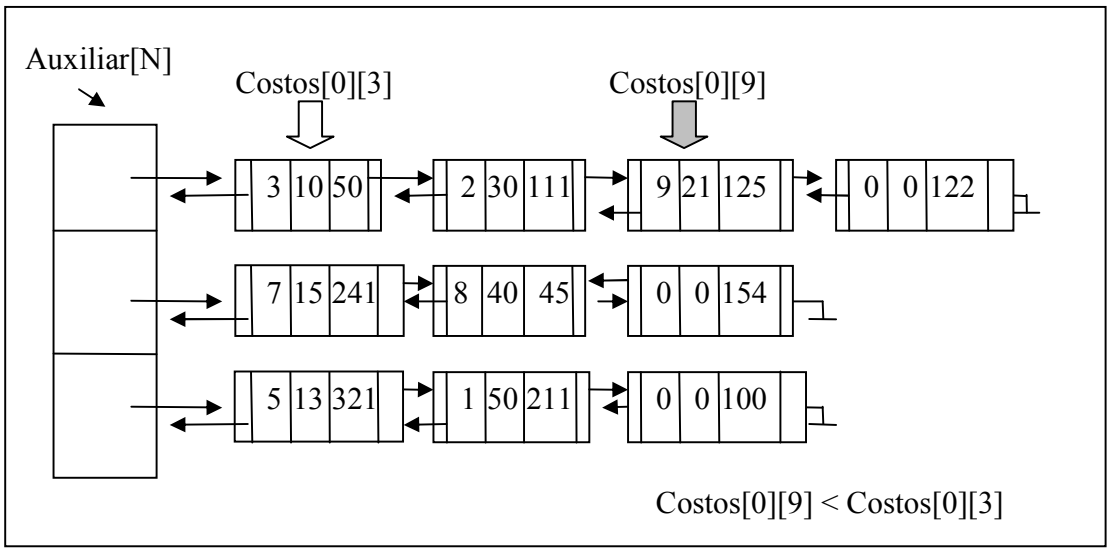

Figura 5.24: Búsqueda de un cliente que disminuya el costo si esta en esa posición

Cuando se encuentra un cliente que disminuya el costo en esa posición se procede a realizar el intercambio entre los clientes (ver Figura 5.25), al realizarse el cambio se ve que el valor de los costos se modifica en los clientes que siguen luego de los clientes intercambiados ya que el costo depende del valor del cliente anteriormente visitado. A continuación, se procede a tomar el segundo cliente y compararlos con los demás clientes utilizando el costo considerando al cliente anterior (ver Figura 5.26) y así sucesivamente hasta haber revisado todas las rutas de la solución encontrada. Al realizar esto hay una cierta posibilidad de que el valor de la solución no mejore, a pesar de realizarse los cambios, es por eso que por el momento esta solución no se considera como mejor, sino luego de pasar por el ciclo de nuevo se va a comparar si mejora la solución o no y si es así se procederá a identificarlo como mejor solución.

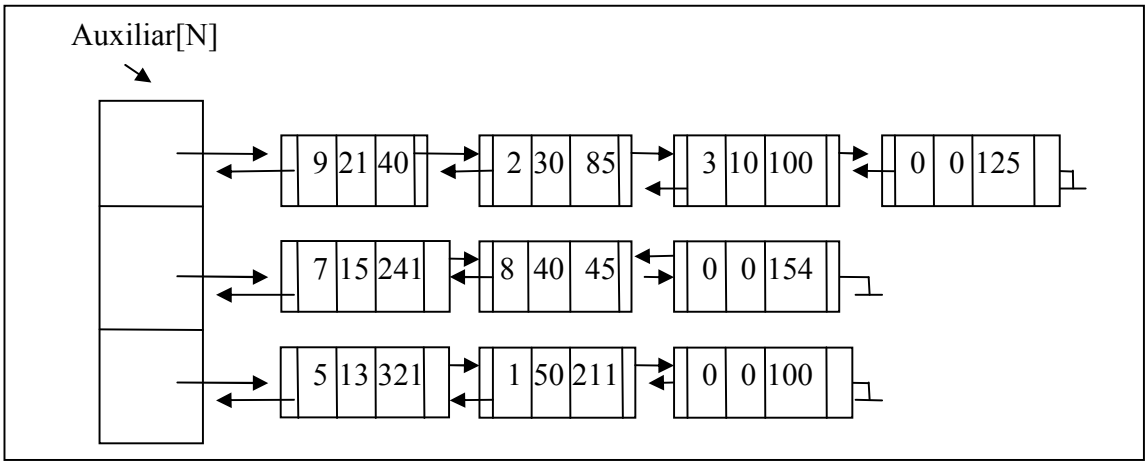

Figura 5.25: Se procede a intercambiar los clientes y actualizar costos.

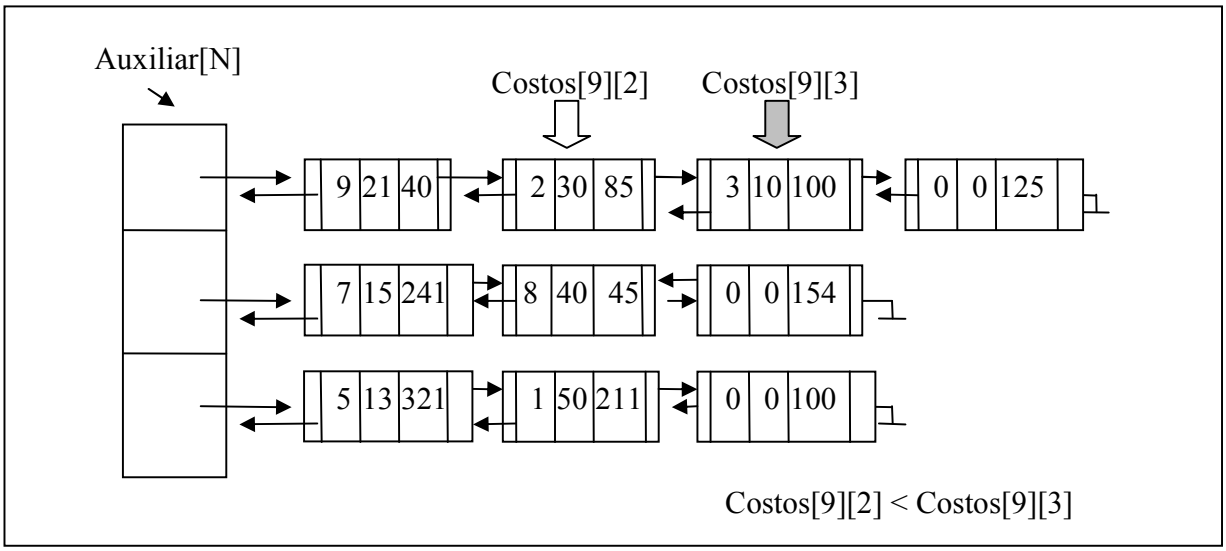

Figura 5.26: Comienzo a verificar nuevo cliente.

5.7 Diseño de reajuste de las soluciones

A continuación se procede a mostrar el diseño de la etapa de reajuste de la solución la cual solamente se va a realizar en las pruebas de la utilización del VMI. La razón es esto es por el conocimiento y el manejo de los inventarios de los clientes, a su vez conocer o predecir la demanda de todos los clientes en un momento dado. Cuando se realiza el envió de los productos en el caso donde no se ocupa el VMI, el proveedor solamente tiene como obligación enviar la cantidad de productos a los clientes que han solicitado, esto puede provocar el no uso eficiente de los vehículos disponibles por el proveedor, es a esto último donde se quiere centrar esta parte del proyecto.

Las rutas escogidas con las heurísticas se realizan con los clientes que están en su punto de re-orden o demasiado cerca de éste, tratando de simular que se han enviado órdenes desde los clientes como si no existiera el VMI. Con esto es posible que la capacidad de los vehículos no este ocupada al 100%, se pretende lograr que así sea cuando esto se pueda realizar, para esto se buscarán a los clientes que están cerca de el punto de re-orden pero que no alcanzaron a ser visitados en la ruta original y agregarlos en está con la cantidad de producto que sean posibles (llenar al vehículo de la ruta). Esto se hará de la siguiente manera.

El primer paso será ver comprobar la cantidad de productos que están llevando cada vehículo de la solución final encontrada, esto se puede apreciar en Figura 5.27, donde también se indica cual es el último cliente visitado (el cual no es el proveedor) dato que nos servirá luego en la elección de clientes que se van a visitar. $Qtotal$ es la capacidad máxima que un vehículo puede llevar, CF indica el último cliente visitado y los $Q_1 ... Q_n$ nos indica la demanda entregada por cada vehículo.

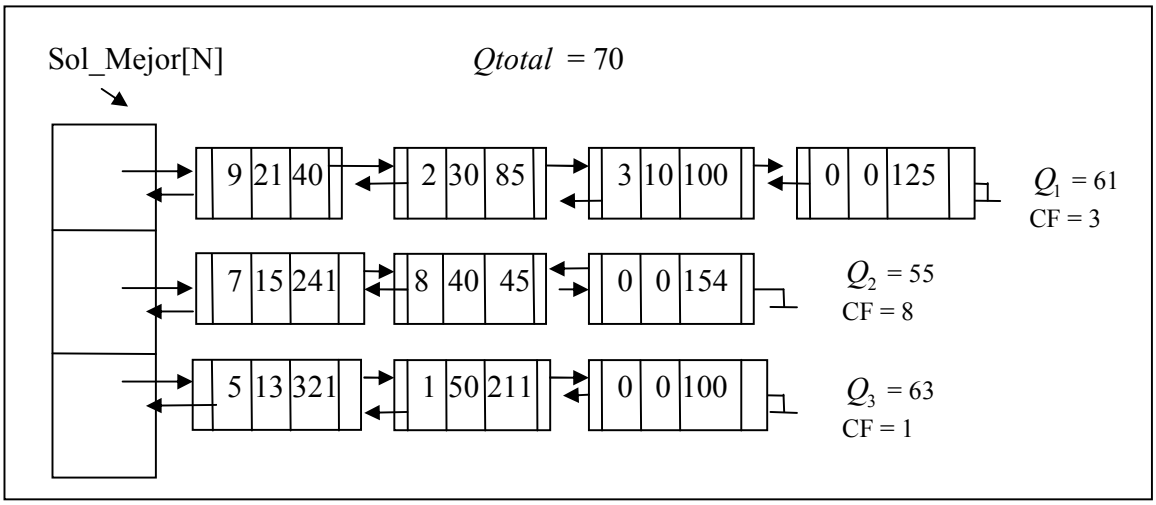

Figura 5.27: Conteo de productos llevados por cada vehículo

Estos datos se guardaran en una matriz de tamaño $nrutas \times 2$ donde $nrutas$ es la cantidad de rutas que tiene la solución (cantidad de vehículos utilizados). Ahora del archivo de clientes se selecciona los clientes más cercanos del punto de re-orden y que no fueron escogidos anteriormente, se escoge una cantidad igual a la cantidad de vehículos que se ocupa, se genera una lista que guarda el identificador del cliente junto con la cantidad posible demandada como se muestra en Figura 5.28.

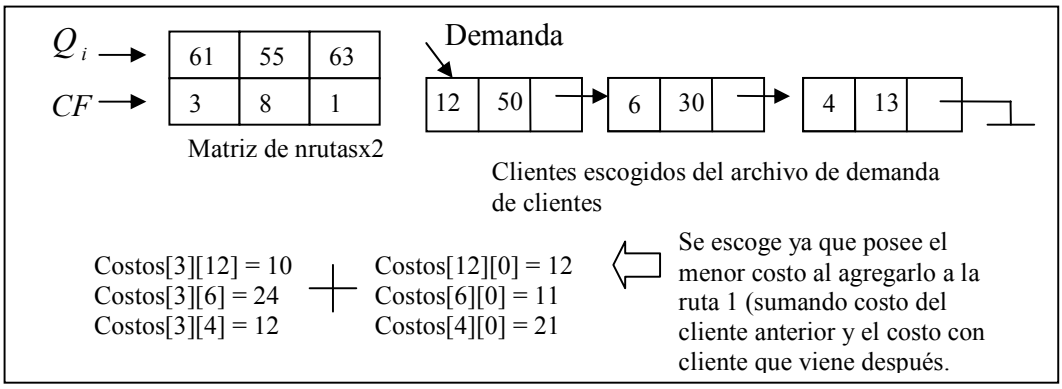

Figura 5.28: Clientes a ingresar en las rutas

Volviendo a la primera ruta, primer elemento de la matriz, se comprueba si queda espacio en el vehículo y el último cliente y se va a escoger de la ruta de los clientes en espera de ser incorporados en la ruta al que presente un menor costo de ubicación en la ruta ($\min(costos[CF][CN])$), donde CN es el identificador del cliente de la lista de clientes), se comprueba si la demanda del cliente puede ser satisfecha por completo, si no es así solamente se le entregará la cantidad con la cual el vehículo se ocupe al 100% de su capacidad, este cliente sigue permaneciendo en la lista de clientes pero a la demanda se le resta lo entregado por el vehículo si es que la demanda no se satisfacía por completo, por el contrario si la demanda se satisfaga el cliente es sacado de la lista de los clientes, ver Figura

5.29. Esto se realiza con todas los vehículos de la solución, y si sobran clientes en la lista se descartan de esta.

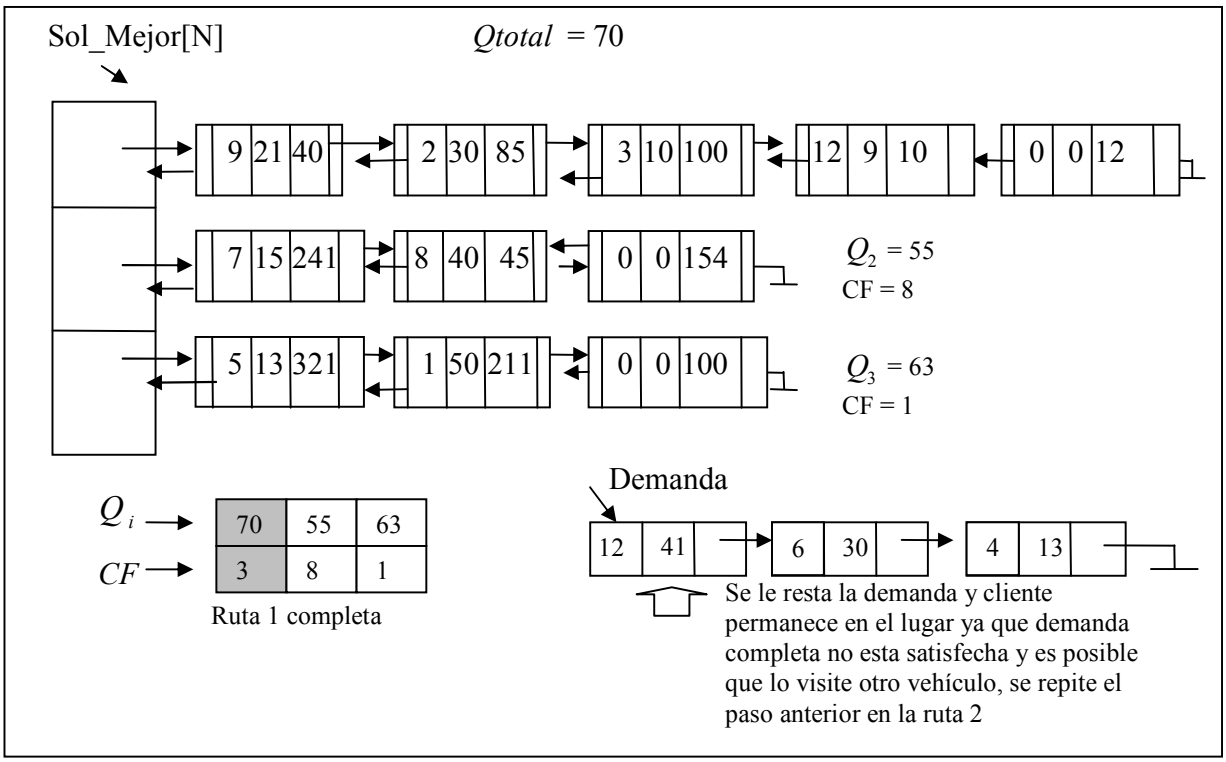

Figura 5.29: Ingreso de cliente, vehículo completo

Se hace una actualización al archivo de demanda de los clientes, ya que la demanda de los clientes visitados (de esta parte de reajuste) se verá modificada. Con la solución ya reajustada se procede a comprobar si es posible realizar un reordenamiento de los clientes en esta que permita que el costo sea menor, si es así se procede a actualizar la mejor solución sino se mantiene el orden con los clientes ingresados al final.

Capítulo 6

Pruebas y Resultados

6.1 Comprobación de las Heurísticas

Para comprobar el funcionamiento de las heurísticas se procedió a utilizar instancias encontradas en [21], estas instancias son de CVRP y se entrega la ubicación de cada cliente y del proveedor (coordenadas X e Y), la cantidad de clientes y la demanda a entregar para cada uno de ellos, como también la capacidad del vehículo. Cada instancia nos presenta el valor óptimo y es a este valor que trataremos de alcanzar con la implementación de las heurísticas.

En la Tabla 6.1 se entrega los resultados de la heurística Simulated Annealing (S.A.), y en la Tabla 6.2 con la heurística Tabu Search (TS), se comparan los resultados obtenidos en el informe de avance de proyecto 2, donde recién se comprobaban las heurísticas ocupando las instancias mencionadas (se diferencian estos valores en las tablas indicando entre paréntesis una P de pasado), junto con el valor obtenido al finalizar el proyecto, donde los resultados son más cercanos al óptimo entregado por la literatura (se diferencian estos valores en las tablas indicando entre paréntesis una A de actual)., en ambas tablas se indica el valor encontrado por cada heurística, el tiempo que ocupo el programa para llegar a estos resultados (tomados en segundos y para el valor actual) y el porcentaje de error comparándolo con el óptimo de las instancias indicado por la literatura (el porcentaje logrado anteriormente en el avance del proyecto (P) y al finalizar el proyecto (A)). El nombre del archivo de las instancias es de la siguiente manera "A-n32-k5.vrp" donde el número que acompaña a la "n" nos indica la cantidad de clientes al restarle 1, ya que incluye al proveedor en la cantidad, mientras que el número que acompaña a la "k" es la cantidad de vehículos a utilizar para llegar al óptimo indicado. El porcentaje de error está calculado de la siguiente manera:

$$\frac{(Sol\_encontrada - Sol\_óptima)}{Sol\_óptima} \cdot 100 \qquad (6.1)$$

Tabla 6.1: Resultados heurística Simulated Annealing en las diferentes instancias

Instancia	Óptimo	S.A. (P)	S.A. (A)	Tiempo	% (P)	% (A)	Semilla
B-n31-k5	672	698.868648	682.391357	1.156	3.99	1.54	1228311612
A-n32-k5	784	855.866962	832.964496	1.078	9.16	6.24	1228331372
A-n37-k5	669	777.790796	685.137416	1.031	16.26	2.41	1228331918
B-n38-k6	805	851.211038	819.105350	0.75	5.74	1.75	1228332257
B-n43-k6	742	825.264125	770.869596	1.062	11.22	3.89	1228332958
A-n48-k7	1073	1251.60239	1158.172906	1.921	16.64	7.93	1228333594

Tabla 6.2: Resultados heurística Tabu Search en las diferentes instancias

Instancia	Óptimo	T.S. (P)	T.S (A)	Tiempo	% (P)	% (A)	Semilla
B-n31-k5	672	681.167704	680.955682	3.578	1.36	1.33	1228254370
A-n32-k5	784	830.814125	787.081889	3.437	5.97	0.39	1228246654
A-n37-k5	669	741.8877	672.740749	10.125	10.89	0.55	1228256065
B-n38-k6	805	818.064055	808.702610	4.531	1.62	0.45	1228259490
B-n43-k6	742	790.605471	751.748867	12.453	6.55	1.31	1228259855
A-n48-k7	1073	1144.4825	1107.446922	21.438	6.66	3.21	1228260600

Con las tablas anteriores se puede apreciar lo siguiente: primero que nada que la heurística que nos entrega un valor más cercano al óptimo es Tabu Search, no solamente comparando los resultados de las tablas sino también en promedio los números entregados son menores, la otra conclusión que se puede obtener es que Tabu Search es la heurística que más se demora pero este tiempo no es tanto para llegar a preferir Simulated Annealing. Son por estas razones que para realizar la simulación de las situaciones con y sin VMI se utilizó solamente Tabu Search.

También podemos concluir que ambas heurísticas son aptas para realizar y solucionar problemas de tipo VRP, incluso ambas ya han sido utilizadas anteriormente con problemas de esta clase en [18][22][23], otros autores pueden ser encontrados en la página web indicada en [21], los valores son cercanos al óptimo para instancias con clientes de un tamaño regular, por eso podemos concluir que para nuestra prueba los resultados entregados o son el óptimo o muy cercanos a éste, ya que la cantidad de clientes es bastante menor y también que sin importar la semilla con que se utilice el programa los resultados entregados son los mismos.

6.2 Caso de Prueba

Como se ha mencionado desde el principio de la memoria, se quiere ver y comparar el uso del VMI y el no uso de éste, como esto afecta al costo de transporte y al costo de planificación, formula de costo de esto último mencionada en el capítulo de modelo matemático del VMI. Para poder efectuar esto se procedió a darnos un caso de prueba, este

caso de prueba consiste en lo siguiente: tenemos un total de 20 clientes cada uno de ellos con diferentes demandas, esta demanda es estocástica y se genero utilizando una función normal, teniendo la varianza y la media. La capacidad de las bodegas de los clientes es la misma, la cual es de 175 productos, y el punto de reorden es de 25 productos. Se probaron dos capacidades de vehículos, de 200 y de 250. Con capacidad de vehículo de 200 se realizaron la simulación de 238 periodos, mientras que con la capacidad de 250 se realizaron la simulación de 216 periodos. De estas simulaciones se obtuvieron distintos indicadores (ver Tabla 6.3), que a continuación se detallan y se entregan sus resultados.

Tabla 6.3: Indicadores encontrados tras realizar la simulación.

	C. Vehículo 200, Periodos 238		C. Vehículo 250, Periodos 216	
	Sin VMI	**Con VMI**	**Sin VMI**	**Con VMI**
Cantidad de periodos donde debe enviar productos.	185	117	166	82
Cantidad de periodos donde no se envía productos	53	121	50	166
Demanda promedio por periodo.	135.563944	194.80714	134.425102	224.977074
Costo de transporte	25079.3297	22792.4354	22314.567	18448.12

El primer indicador es la cantidad de periodos donde el proveedor tiene la necesidad de enviar sus vehículos a algunos de sus clientes para satisfacer su demanda, y el segundo indicador nos muestra los periodos en que ocurre todo lo contrario, se obtuvieron estos resultados ya que el VMI entrega productos a clientes no obligatorios, con eso estos clientes van a requerir que les envíe productos en periodos más alejados.

El tercer indicador es la demanda promedio, esta demanda nos indica qué tan ocupado ha estado el vehículo a la hora de entregar los productos, como se ve sin VMI no se aprovecha de ocupar la mayoría de la capacidad de los vehículos, incluso en algunos periodos sólo se visitaba a un cliente, en cambio VMI presenta un mejor uso de la capacidad de los vehículos, lo que afecta luego en la cantidad de periodos donde se visita al cliente.

Por último se muestra el costo de transporte, VMI es el que menos costo posee a pesar de que muchas veces cuando se visitan a los clientes, al agregar más clientes de la lista de negocio, el costo de ese periodo sea mayor al mismo periodo pero sin VMI, la razón va relacionado con la cantidad de visitas al proveedor.

A continuación se presenta unos gráficos donde se ve como ha variado los costos de transporte en un vehículo con capacidad de 200, en un comienzo se ve que al utilizar VMI causa que es costo de transporte sea mayor pero esto va comenzando a cambiar a medida que los periodos vayan pasando (Figura 6.1), como se puede apreciar la razón es que al ocupar VMI hay más periodos sin visitar a los clientes, es por eso que si se tira una línea de tendencia (ver Figura 6.2) se ve que la línea de costos de transporte con VMI es menor.

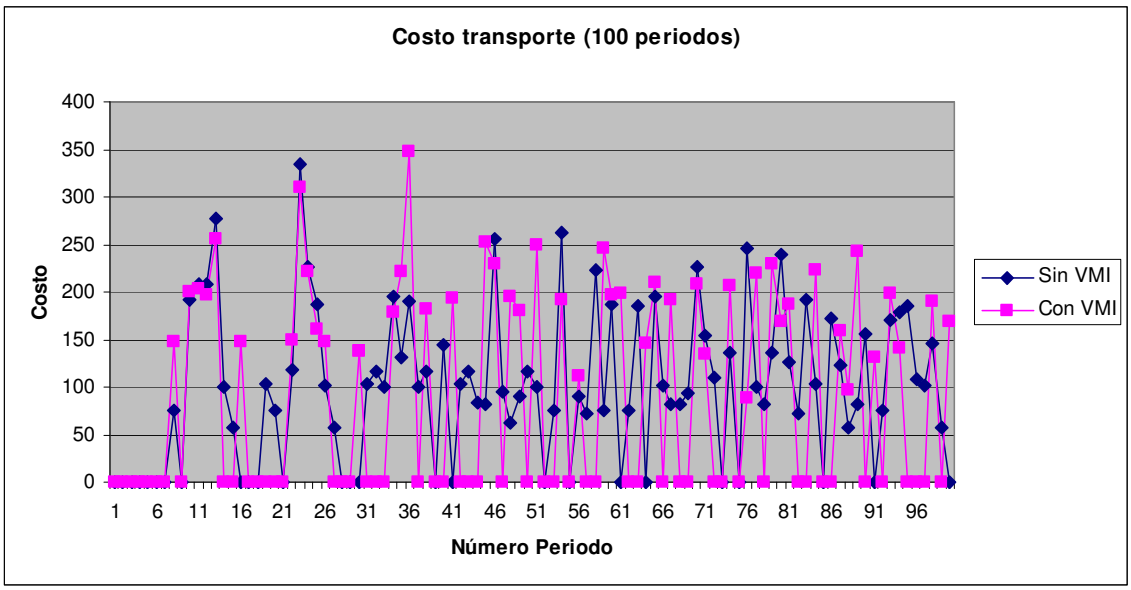

Figura 6.1: Gráfico de 100 periodos, vehículo de 200.

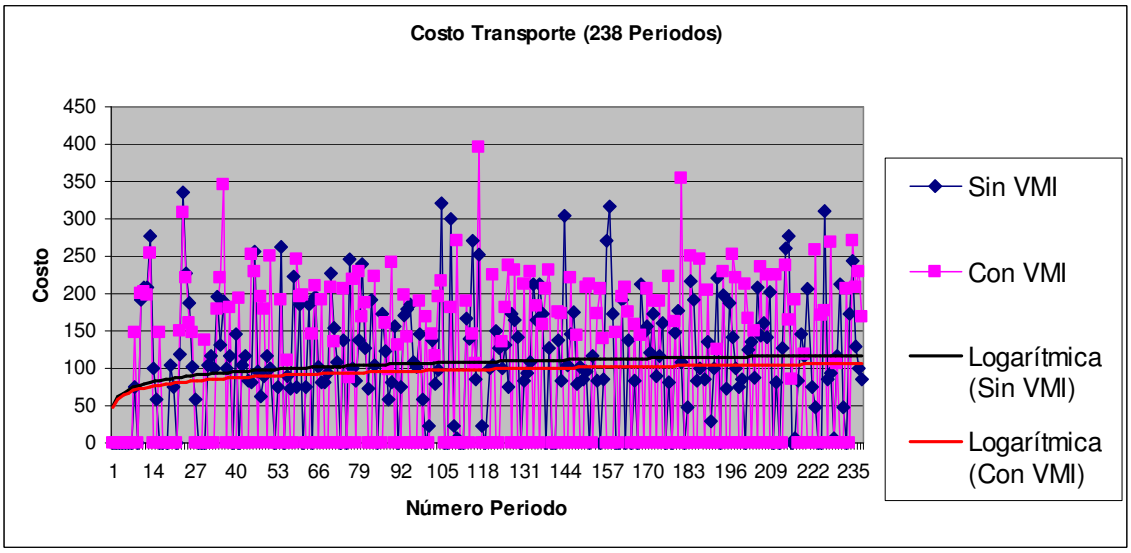

Figura 6.2: Gráfico de 238 periodos, vehículo de 200.

Lo mismo sucede cuando la capacidad de vehículos es de 250, como se ve en Figura 6.3 la línea de tendencia también es más baja que la que no utiliza VMI.

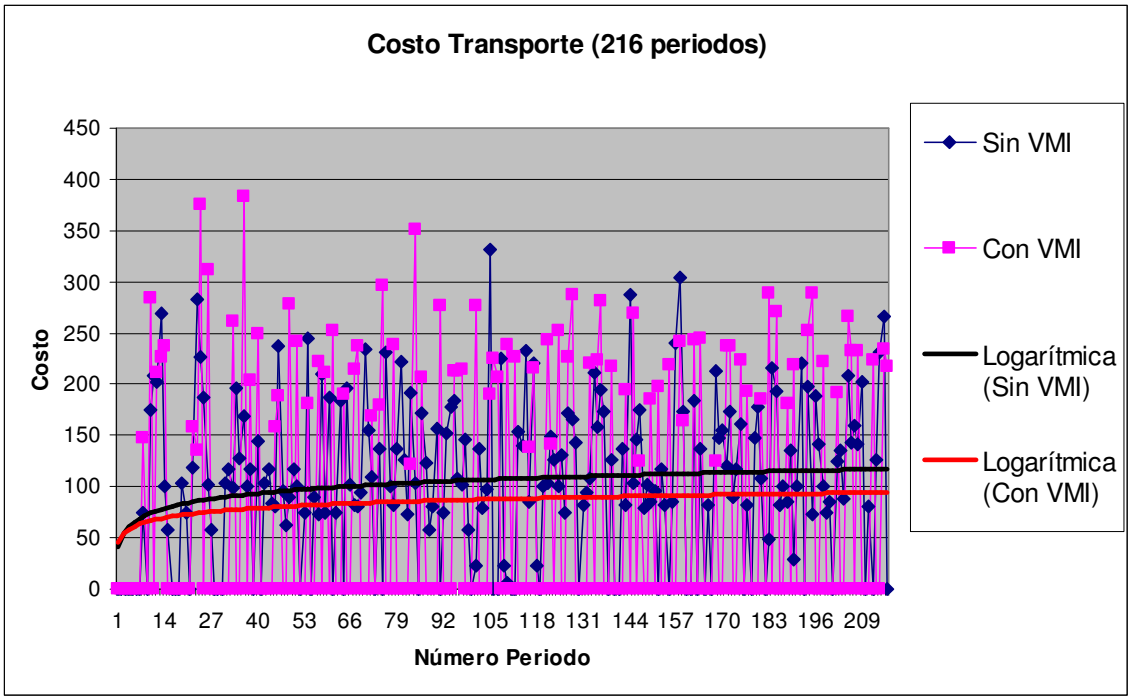

Figura 6.3: Gráfico de 216 periodos, vehículo de 250.

La evaluación de la función de costos totales mostrada en la ecuación (4.20) del capítulo de modelo matemático con los valores del VMI y sin VMI, reemplazando las variables R, Q, qi, kvmi, entre otros (estos datos serán entregados en la plantilla de resultados que vendrá en el CD junto con la memoria, la razón es por que dependiendo del cliente, en específico su demanda, los datos varían), y dándonos valores para las variables C, c y H, h se obtiene los siguientes resultados:

Simulación con capacidad de vehículos de 250:

- Costo total ocupando VMI = 20237167.4
- Costo total no ocupando VMI = 88300293.3
- Beneficio de un 77% aproximadamente.

Simulación con capacidad de vehículos de 200:

- Costo total ocupando VMI = 23161355.5
- Costo total no ocupando VMI = 97396986.3
- Beneficio de un 76% aproximadamente.

Para finalizar se presenta la Tabla 6.4, donde se presentan los porcentajes de beneficios entre VMI y sin VMI de los diferentes factores que afectan el resultado anterior de costo total.

Tabla 6.4: Porcentajes de Beneficios entre VMI y sin VMI

	C. Vehículo 200, Periodos 238	C. Vehículo 250, Periodos 216
Demanda entregada	1.09% aproximadamente	0.41% aproximadamente
Periodos sin visitar	128.30% aproximadamente	168% aproximadamente
Capacidad del vehículo promedio	43% aproximadamente	67% aproximadamente
Costo transporte	9.11% aproximadamente	17% aproximadamente

El beneficio referente a la demanda entregada no es muy alto, lo que se le entrega a los clientes es solo un poco más (adelantando demanda a clientes próximos a su punto de reorden) de la demanda que se esta pidiendo, cuidando de no sobrepasar la capacidades de almacenamiento. Donde se produce un beneficio más notable, y por consecuencia es uno de los factores importantes en el costo total, es en los periodos donde no se visitan los clientes, aquí se ven porcentajes que superan el 100% y los recursos que no se ocupan en ese día (camiones por ejemplo) pueden ser utilizados para otras áreas. La capacidad del vehiculo también es otro factor importante que afecta en el costo total, los porcentajes de beneficio alcanzan el 50%. Otro factor que puede afectar el costo, pero no de gran medida es costo de transporte, donde el beneficio entregado por VMI va desde los 10 a 20 %.

Capítulo 7

Conclusiones

En este proyecto se propuso una arquitectura para el VMI compuesta por cuatro módulos: Interfaz, IRP, Proyección e Información. Para realizar la comparación entre el uso del VMI y la forma tradicional se implemento casi por completo un módulo de la arquitectura, IRP, faltando solamente el componente de clusterización. Este módulo fue escogido ya que representa lo que se quiere realizar al implementar el VMI, que es integrar a la cadena de suministro los componentes proveedor y clientes, permitiendo la integración del inventario y el enrutamiento.

Para hacer frente al IRP se realizó una descomposición donde se separó el enrutamiento y la generación de la demanda o lista de clientes, para la parte del enrutamiento se ha implementado el problema del CVRP con las heurísticas de Simulated Annealing y Tabu Search, luego se opto por la que mejores resultados entregaba al realizar una prueba de las heurísticas con casos de pruebas ya resueltos anteriormente, Tabu Search fue la escogida. Al implementar ambas heurísticas se logro ver las diferencias que hay entre ambas en el momento de implementar, y en las entrega de resultados, por ejemplo Simulated Annealing es una heurística que no ocupa mucho tiempo de ejecución, por el contrario es la más rápida entre las dos, pero comparándola con Tabu Search solamente en el tipo de resultados que entrega, esta última gana a pesar de ser más lenta aunque el tiempo es bastante parecido y no supone un factor determinante a la hora de escoger en éste caso.

La integración de heurísticas para este tipo de problemas es algo fundamental para obtener buenos resultados, que no necesariamente deban ser el óptimo pero son mejores y ocuparon menos tiempo en obtenerse si se hubiera realizado de otra manera el problema. Las heurísticas (por lo menos las utilizadas en este proyecto) dependen mucho de la semillas dadas en un principio, esto afecta a los resultados, pero dependerá también del tamaño del problemas, un problema con muchas variables pueden tener resultados diferentes para cada semilla, y se trata de encontrar cual es la semilla que de el mejor resultado. Pero con problemas con no muchas variables de entrada (o no con tamaños muy grande en las variables), puede suceder que el resultado entregado sea el mismo sin importar la semilla utilizada. Esto último fue lo sucedido en el caso de prueba que se realizo para comprobar el uso y el no uso de VMI, en ambas partes se utilizaron las heurísticas para obtener el resultado.

Luego de las pruebas de las heurísticas se agregaron las partes faltantes para transformar el CVRP en IRP, al agregar la parte de selecciòn de clientes. Al revisar la capacidad de los vehículos que se utilizan y tratando de utilizar la mayor posible se comprobó que ayuda bastante en el mejoramiento de la situación utilizando VMI al largo plazo, esta regla es posible gracias al conocimiento del proveedor del inventario de sus clientes, y la posible demanda que se puede efectuar en los días próximos, ayuda a disminuir el costo de transporte, la cantidad de periodos de enviar vehículos, y la capacidad de utilización de estos aumentan.

Como resultados obtenidos se llegó a la conclusión de que el ocupar VMI trae beneficios para el proveedor en término de disminución de costos comparando con el uso tradicional del esquema de cliente-proveedor para el trabajo en el inventario y enrutamiento.

VMI trae las ventajas para el proveedor de poder comprobar lo que realmente el cliente necesita en sus inventarios y en lograr adelantarse para los próximos periodos, teniendo siempre presente que pueden ocurrir situaciones no pronosticadas por lo que es necesario también tener algún respaldo (por ejemplo una cierta cantidad de stock de productos adicionales) para enfrentarse a ellas. Estas ventajas solamente pueden presentarse si la comunicación entre el proveedor y el cliente es fluida, constante y actualizada, ya que es en esta información donde se sustenta el VMI y si no es correcta las ventajas de esta práctica no se presentarán. El factor de confianza que tiene que haber entre el proveedor y el cliente es lo más importante.

Para este proyecto se trabajó usando listas y punteros, dado que las soluciones y operaciones a usar son dinámicas respecto a los resultados que deben entregar. En un principio no hubo mayores problemas al implementar las heurísticas usando listas, y no se veía grandes diferencias con respecto al uso de estructuras estáticas como por ejemplo matrices. Pero esto cambio a la hora de dejar el programa corriendo por periodos más largos de tiempo, comparando con otros compañeros que pueden dejar sus programas corriendo toda una noche, al usar estas estructuras no se logró realizar esto, la razón principalmente era el pedido y la liberación de memoria. Al ejecutar el programa en una cierta cantidad de iteraciones se detiene indicando que no hay suficiente memoria virtual, la cantidad de iteraciones dependen de la capacidad del computador (en memoria RAM). La conclusión obtenida de esto es que para programas para uso comercial es posible utilizar este tipo de estructuras al no ser realmente necesario realizar una cantidad grande de iteraciones (proyecciones para futuros periodos) ya que muchas cosas pueden variar de una cierta cantidad de periodo a otro, pero para estudios donde es importante obtener los mejores resultados para alcanzar el óptimo y para lograrlo se debe dejar funcionando el programa realizado por periodos de tiempo largos (horas) no es aconsejable usar estructuras dinámicas.

Para trabajos futuros se puede trabajar en la implementación de los otros módulos de la arquitectura propuesta en forma más detallada, como por ejemplo, un sistema de información que permita comunicar al proveedor con sus clientes y obtener los datos necesarios para el buen funcionamiento del VMI.

Por otro lado para la implementación del módulo del IRP se realizo tomando ciertas restricciones, las cuales pueden modificarse o agregarse otras, un cambio en las restricciones puede enfocarse en modificar o agregar más detalles a la parte de selección de clientes para visitar, clusterizar clientes, entre otras cosas. Por la parte de el enrutamiento también es posible ir agregando más restricciones, como puede ser lead time, o ventanas de tiempo, o capacidades de vehículos no fijas con diferentes costos para la utilización de cada vehículo. Lo mencionado anteriormente solo siguiendo el tipo de desarrollo del IRP que es separado por dos etapas, una de selección de clientes y otra de enrutamiento de vehículos.

Algo importante de mencionar es que en este proyecto se trabajo solamente con dos niveles de la cadena de suministro, clientes y proveedor, se puede ir agregando más complejidad al problema por esta área e ir aumentando los niveles de la cadena, por ejemplo la aparición de puntos de suministro, cada vez llevando este problema lo más cercano a la vida real, donde es bastante complejo.

Por último, se puede cambiar las heurísticas o formas de solución que se van a implementar para resolver el problema.

Capítulo 8

Referencias

[1] CAMPBELL A., CLARKE L., KELYWEGT A.,SAVELSBERGH M., The Inventory Routing Problem, En: CRAINIC T., LAPORTE G. Fleet Management and Logistics, Atlanta, Springer, 1998, pp. 95-112.

[2] KLEYWEGT A., NORI V., SAVELSBERGH M., The Stochastic Inventory Routing with Direct Deliveries, *Transportation Science*, 2002, Vol. 36, Nº 1, pp. 94-118.

[3] SINDHUCHAO S., A Very Large Scale Neighborhood (Vlsn) Search Algorithm for an Inventory-Routing Problem, *Thammasat International Journal of Science and Technology*, 2006, Vol. 11, Nº 4, pp. 20-26.

[4] REINGOLD E., NIEVERGELT J., DEO N., Combinatorial Algorithms: Theory and Practice, *Prentice-Hall Inc.*, Englewood Cliffs, New Jersey 07632, 1977.

[5] http://www.12manage.com/methods_vendor_managed_inventory_es.html, página sobre la práctica del Inventario Manejado por el vendedor, indicando de que se trata, un poco de historia de sus orígenes y pasos a seguir, la página principal trata sobre los diferentes modelos y métodos de administración, visitada el 02 de Julio del 2008.

[6] http://www.datalliance.com/whatisvmi.html, página donde se explican los elementos básicos del Vendor Managed Inventory, la página principal es de una empresa dedicada a la implementación de esta práctica, visitada el 02 de Julio del 2008.

[7] POLER R., MULA J, ORTIZ A., LARIO F., Un modelo de Empresa Virtual para la Gestión del Proceso de Previsión Colaborativa en Cadenas de Suministro, En: CONGRESO de Ingeniería de Organización (X, 2006, España, Valencia), Centro de Investigación Gestión e Ingeniería de Producción. Universidad Politécnica de Valencia, 7 y 8 de septiembre de 2006.

[8] CAMPBELL A., SAVELSBERGH M., A Decomposition Approach for the Inventory-Routing Problem, *Transportation Science*, 2004, Vol. 38, Nº 4, pp. 488-502.

[9] CAMPBELL A., CLARKE L., SAVELSBERGH M., Inventory Routing in Practice, En: TOTH P., VIGO D. The Vehicle Routing Problem, Society for Industrial and Applied Mathematics, Paperback, 2001. pp. 309-330.

[10] JAILLET P., BARD J. F., HUANG L., DROR M., Delivery Cost Approximations for Inventory Routing Problems in a Rolling Horizon Framework, *Transportation Science*, 2002, Vol. 36, Nº 3, pp. 292-300.

[11] QUI-HONG Z., SHOU-YANG W., LAI K. K., A partition approach to the inventory/routing problem, European Journal of Operational Research, 2007, Vol. 177, Nº 2, pp. 786-802.

[12] REIMAN M. I., RUBIO R., WEIN L. M., Heavy Traffic Analysis of the Dynamic Stochastic Inventory-Routing Problem, Transportation Science, 1999, Vol. 33, Nº 4, pp. 361-380.

[13] http://www.geocities.com/francorbusetti/anneal.htm, página con artículos de heurísticas e inteligencia artificial, persona encargada de la página BUSETTI F. quien también tiene publicado artículos escrito por él en ella, visitada 18 de Junio del 2008.

[14] FRANCO J., TABARES P., Aplicación del Simulated Annealing al problema de las N reinas, *Scientia et Técnica*, 2005, Año XI, Nº 29.

[15] GENDREAU M., An Introduction to Tabu Search, En: Handbook of Metaheuristics, Vol. 57, New York, Springer, 2003, pp. 37-54.

[16] MELIÁN B., GLOVER F., Introducción a la Búsqueda Tabú, *Procedimientos Metaheurísticos en Economía y Empresa*, 2007, Vol. 3.

[17] COELLO C., Búsqueda Tabú: Evitando lo Prohibido, *Soluciones Avanzadas. Tecnologías de Información y Estrategias de Negocios*, 1997, Año 5, Nº 49, pp. 72-80.

[18] TAVAKKOLI-MOGHADDAM R., SAFARI N., KAH M.M.O., RABBANI M., A New Capacitated Vehicle Routing Problem with Split Service for Minimizing Fleet Cost by Simulated Annealing, *ScienceDirect Journal of the Franklin Institute*, 2007, Vol. 344, Nº 5, pp. 406-425.

[19] YAO Y., EVERS P., DRESNER M., Supply chain integration in vendor-managed inventory, Decision Support Systems, 2007, Vol. 43, pp. 663-674.

[20] VAN DER VLIST P., KUIK R., VERHEIJEN B., Note on supply chain integration in vendor-managed inventory, Decision Support Systems, 2007, Vol. 43, pp. 360-365.

[21] http://neo.lcc.uma.es/radi-aeb/WebVRP/, página con material relacionado con las distinats variantes del VRP entre formulación matemática hasta instancias de pruebas, página creada gracias a la colaboración de Auren y el "Languages and Computation Sciences department of the University of Málaga", persona encargada de la página DORRONSORO B., última actualización Marzo 2007.

[22] BRÄYSY O., GENDREAU M., Tabu Search Heuristics for the Vehicle Routing Problem with Time Windows, *Interna Report STF42 A01022, SINTEF Applied Mathematics, Department of Optimisation, Oslo, Norway,* 2007.

[23] THANGIAH S., OSMAN I., SUN T., Hybrid Genetic Algorithm Simulated Annealing and Tabu Search Methods for Vehicle Routing Problem with Time Windows, *Computer Science Department, Slippery Rock University,* 1994, Technical Report 27.

Anexo 1: Código Fuente

main.c

```
/*****************************************************************************************************************/
Archivo con el menú principal que se encarga de la incorporación de los datos a utilizarse por
ambas heurísticas (Simulated Annealing y Tabu Search) y la selección de clientes visitados
ocupando VMI.
/*****************************************************************************************************************/

        #include "cabecera.h"

        char caracter[20],□uxilia[10];
        int i, j, n, distancia[50][2],nperiodo, visitado[20];
        □uxil porcentaje_vehiculo, capacidad, demanda[50], Icap, entregado[21];
        struct ClientesEspera *Otros;

        //funciones locales del codigo fuente presente
        void comienzo_operacion();
        void ObtenerMatrizCosto();
        void clientes_obligatorios();
        int inicializar_clientes();
        void otrosCliente();
        void lecturadedatos(int j);
        void lecturadatosposicion(int Clientes);
        void EscogerOtrosClientes();
        void cantidadCliente();
        void actualizar_inventario();
        void eliminarclientes2();

        int main(int argc, char *argv[]){
            int operacion;
            for(i=0;i<20;i++)
              visitado[i] = 0;
            for(i=0;i<21;i++)
              entregado[i] = 0.0;
            VMI = 1;
          lecturadatosposicion(20);
          nperiodo = 1;
          do{
            lecturadedatos(nperiodo);
            printf("\nEl numero de □uxilia □u de %d",nperiodo);
            operacion = inicializar_clientes();
            if(carácter == 0)
              comienzo_operacion();
            nperiodo++;
          }while(nperiodo!=1001);
          system("pause");
          return 0; }
        void comienzo_operacion(){
          variable = 0;
         // Simulated_Annealing();
          Tabu_Search();
```

```
    actualizar_inventario();
}

void actualizar_inventario(){
    FILE *archivo_costo;
    char nombre[25] = "nuevosin200.txt";
    archivo_costo = fopen(nombre,"a");
    struct Ruta *Prueba;
    int obtenercliente;
    uxil demandaperiodo = 0.0;
    fprintf(archivo_costo,"\nPeriodo %d\nCliente\tDemanda\n",nperiodo);
    for(i=0;i<nrutas;i++){
      Prueba = Sol_nueva[i];
      do{
        obtenercliente = Prueba->cliente;
        fprintf(archivo_costo,"%d\t%f\n",Prueba->cliente,Prueba->demandaEntregada);
        demandaperiodo = demandaperiodo + Prueba->demandaEntregada;
        linicial[obtenercliente-1] = linicial[obtenercliente-1] + Prueba->demandaEntregada;
        visitado[obtenercliente-1] = visitado[obtenercliente-1] + 1;
        entregado[obtenercliente]=entregado[obtenercliente]+Prueba->demandaEntregada;
        Prueba = Prueba->sig;
      }while(Prueba!=NULL);
    }
    for(i=0;i<20;i++)
      fprintf(uxilia_costo,"%d\t%f\t",I,linicial[i]);
    fprintf(uxilia_costo,"Costo transporte es de %f\n",Costo_Fobjetivo(3));
    fprintf(archivo_costo,"Demanda periodo es de %f\n",demandaperiodo);
    elimina_Ruta(1);
    elimina_Ruta(2);
    elimina_Ruta(3);
    elimina_lista_tabu();
    eliminarclientes2();
    if(nperiodo == 238){
      for(i=0;i<20;i++)
        fprintf(archivo_costo,"Visitado %d\t%d\n",i+1,visitado[i]);
      for(i=1;i<21;i++){
        fprintf(archivo_costo,"entregado %d\t%f\n",i,entregado[i]);
        entregado[0] = entregado[0] + entregado[i];
      }
      fprintf(archivo_costo,"Demanda proveedor\t%f\n",entregado[0]);
    }
    fclose(uxilia_costo);
}

/*****************************************************************************************************************/
/*Datos de inventario: Función encargada de obtener los datos de la demanda de cada cliente   */
/*en un periodo dado ocupando una función normal                                              */
/*****************************************************************************************************************/

void lecturadedatos(int j){
  FILE *archivo_demanda;
  char nombre[14] = "demanda.txt";  //nombre del archivo
  archivo_demanda = fopen(nombre, "r" );  //abierto solamente en lectura
  if( archivo_demanda ){
    primero = 0;
    while (feof(archivo_demanda) == 0){
      fscanf(archivo_demanda,"%s",carácter);
      if(j == 1){
```

```
            if(strcmp(carácter,"Vehiculo") == 0){
               fscanf(archivo_demanda,"%s",caracter);
                //se obtiene la capacidad de los vehículos
               Q = atoi(caracter) ;
            }
            if(strcmp(caracter, »PR ») == 0){
                fscanf(archivo_demanda,"%s",caracter);
                 //se obtiene el punto de reorden
                Preorden = atoi(caracter);
            }
            if(strcmp(caracter,"Cliente") == 0){
                fscanf(archivo_demanda,"%s",caracter);
                 //se obtiene □u cantidad de clientes
                N = atoi(caracter);
            }
            if(strcmp(carácter,"Icap") == 0){
                 //se obtiene la capacidad del inventario por el momento es igual a       todos los
              clientes,
                //puede ser diferente (si hay tiempo modificar)
                fscanf(archivo_demanda,"%s",caracter);
                Icap = atof(caracter);
                for(i=0;i<N;i++) //como estado □ux inventario inicial
                    linicial[i] = Icap;
            }
           }
           if(strcmp(□uxiliary,itoa(j,periodo,10)) == 0)
              for(i=0;i<20;i++){
                 //Obtenemos la demanda y guardamos el estado del inventario
                 fscanf(archivo_demanda,"%s",caracter);
                 linicial[i] = linicial[i] – atof(caracter);
                 fscanf(archivo_demanda,"%s",caracter);
               }
        }
        fclose(archivo_demanda);
      }else {
        printf( "\nERROR: No se ha logrado abrir el archivo de demanda\n" );
        system("pause");
      }
      for(i=0;i<N;i++){ //Obtenemos la demanda de cada cliente
        demanda[i] = Icap – linicial[i];
      }
    }


/*****************************************************************************************************************/
/* Datos de distancia: Función encargada de obtener los datos de las coordenadas de los          */
/*clientes desde un archivo de configuración.                                                                    */
/*****************************************************************************************************************/

    void lecturadatosposicion(int Clientes){
      FILE *archivo_matriz;
      char nombre[20] = "□uxiliary□ión.txt";
      archivo_matriz = fopen(nombre, "r" );  //abierto solamente en lectura
      □u( archivo_matriz ){
        while (feof(archivo_matriz) == 0){
          fscanf(archivo_matriz,"%s",caracter);
          if(strcmp(carácter,"Y") == 0){
            fscanf(archivo_matriz,"%s",carácter);
```

```
        //Obtenemos las coordenadas X e Y de los clientes y del proveedor, se ingresan en
    una matriz
        for(i=0;i<=Clientes+1;i++){
          fscanf(archivo_matriz,"%s",caracter);
          distancia[i][0] = atoi(caracter); //coordenada X
          fscanf(archivo_matriz,"%s",caracter);
          distancia[i][1] = atoi(caracter); //coordenada Y
          fscanf(archivo_matriz,"%s",caracter);
        }
       }
      }
    }else{
        printf( "\nERROR: No se ha logrado abrir el archivo de demanda\n" );
        system("pause");
    }
    fclose(archivo_matriz);
    ObtenerMatrizCosto();
}

/*****************************************************************************************************************/
/*Matriz de Costo: Función encargada de obtener la matriz de costos Cij dependiendo de las*/
/*coordenadas X e Y de los clientes entregadas anteriormente.                              */
/*****************************************************************************************************************/

    void ObtenerMatrizCosto(){
      int disX, disY;
      for(i=0;i<21;i++) //inicializamos la matriz de costos agregando 0 a todos sus componentes
        for(j=0;j<21;j++)
          costos[i][j] = 0;
      for(i=0;i<21;i++) //obtenemos el costo dependiendo de la dist6ancia a recorrer
        for(j=0;j<21;j++)
          □u(i==j)
            costos[i][j] = 0;
          else{
            disX = distancia[j][0] – distancia[i][0];
            disY = distancia[j][1] – distancia[i][1];
            disX = disX * disX;
            disY = disY * disY;
            disX = disX + disY;
            costos[i][j] = sqrt(disX);       } }

/*****************************************************************************************************************/
/*Generación de una lista de clientes: Función encargada de la creación de una lista con los      */
/*clientes entregados  cada nodo posee el identificador del cliente y la demanda a entregar.      */
/*****************************************************************************************************************/

    int inicializar_clientes(){
      struct ClientesEspera *nuevo, *anterior;
      int contador_cliente=0, salida=0;
      Clientes = (struct ClientesEspera *)malloc(sizeof(struct ClientesEspera));
      if(¡(Clientes))
        printf("\nError: No hay memoria disponible para generar lista de Clientes");
      else{
        Clientes->sig = NULL;
        Clientes->ant = NULL;
        do{  // Ver si el cliente cumple con el punto de reorden
          //tratamos de obtener al primer cliente que cumpla con el punto de reorden
          if(demanda[contador_cliente] >= (Icap – Preorden)){
```

```
            Clientes->cliente = contador_cliente+1;
            Clientes->demanda = demanda[contador_cliente];
            contador_cliente++;
            salida = 1;
          }else{
            contador_cliente++;
            if(contador_cliente == N)
              salida = 1;
          }
        }while(salida == 0);
        if(contador_cliente == N){
          //no hay clientes que cumplan con estar listos para visitar
          printf("\n No es necesario visitar a ningun cliente en el periodo %d\n",nperiodo);
          free(Clientes);
          return 1;
        }else{
          anterior = Clientes;
          //empieza a ingresar a los clientes que obligadamente hay que visitar
          for(i=contador_cliente;i<N;i++)
            if(demanda[i] >= (Icap – Preorden)){
              //empezamos a buscar más clientes para agregarlos en la lista
              nuevo = (struct ClientesEspera *)malloc(sizeof(struct ClientesEspera));
              if(¡(nuevo))
                printf("\nERROR: No hay memoria disponible para agregar cliente");
              else{
                anterior->sig = nuevo;
                nuevo->ant = anterior;
                nuevo->cliente = i+1;
                nuevo->demanda = demanda[i];
                nuevo->sig = NULL;
                anterior = nuevo;
                nuevo = NULL;
                free(nuevo);
              }
            }
          anterior = NULL;
          free(anterior);
          EscogerOtrosClientes();
          return 0;
        }
      }
    }

/*****************************************************************************************************************/
/* Cantidad de vehículos a utilizar: Función encargada de comprobar gracias a la demanda total */
/*del periodo,  la  cantidad de vehículos que se van a ocupar.                                                   */
/*****************************************************************************************************************/

    void EscogerOtrosClientes(){
      struct ClientesEspera *ver;
      uxil deman, demandatotal=0.0;
      int demanint;
      ver = Clientes;
      do{
        demandatotal = demandatotal + ver->demanda;
          printf("\n PRUEBA Cliente %d con demanda %f",ver->cliente, ver->demanda);*/
        ver = ver->sig;
```

```
    }while(ver!=NULL);
    deman = demandatotal/Q ;
    demanint = (int)deman ;
    if((float)demanint < deman) //verificamos la cantidad de vehiculos que se van a ocupar
      F = demanint + 1;
    else
      F = demanint;
    capacidad = F * Q;  //Verificamos cuanto de espacio aun le queda al vehiculo
    capacidad = capacidad – demandatotal;
    // Se procede a efectuar la elección de otros clientes
    //empezamos a escoger a los otros clientes cercanos al punto de reorden
    otrosCliente();
  }

/*****************************************************************************************************************/
/* Clientes a visitar cercanos a Punto de Reorden: Función encargada de encontrar los clientes */
/*cercanos al punto de reorden que se van a visitar para llenar los vehículos utilizados                    */
/*****************************************************************************************************************/

  void otrosCliente(){
    struct ClientesEspera *nuevo, *otro, *antes, *ver, *auxiliar, *auxiliar2;
    int salir=0;
    otro = NULL;
    //se ingresan en una lista todos los clientes que no son obligadamente visitados en este
  periodo
    for(i=0;i<N;i++){
      if(demanda[i] < (Icap – Preorden)){
        nuevo = (struct ClientesEspera *)malloc(sizeof(struct ClientesEspera));
        nuevo->cliente = i+1;
        nuevo->demanda = demanda[i];
        nuevo->sig = NULL;
        if(otro == NULL){
          nuevo->ant = NULL;
          otro = nuevo;
        }else{
          nuevo->ant = antes;
          antes->sig = nuevo;
        }
        antes = nuevo;
        nuevo = NULL;
        free(nuevo);
      }
    }
    antes = NULL;
    free(antes);
    do{  //ahora se procede a seleccionar los más cercanos al punto de reorden
      auxiliar = otro;
      auxiliar2 = otro->sig;
      do{
        if((auxiliar2->demanda) > (auxiliar->demanda))
            auxiliar = auxiliar2;
        auxiliar2 = auxiliar2->sig;
      }while(auxiliar2!=NULL);
      capacidad = capacidad – (auxiliar->demanda);
      if(capacidad > 0){
        if(auxiliar->ant == NULL){
          otro = otro->sig;
          otro->ant = NULL;
```

```
            uxiliary->sig = NULL;
          }else if(uxiliary->sig == NULL){
            auxiliar2 = auxiliar->ant ;
            auxiliar2->sig = NULL ;
            auxiliar->ant = NULL ;
          }else{
            auxiliar2 = auxiliar->sig ;
            ver = auxiliar->ant ;
            ver->sig = auxiliar2 ;
            auxiliar2->ant = ver ;
            auxiliar->ant = NULL ;
            auxiliar->sig = NULL ;
          }
          ver = Clientes ;
          do{
             auxiliar2 = ver ;
             ver = ver->sig ;
          }while(ver !=NULL) ;
          auxiliar2->sig = auxiliar ;
          auxiliar->ant = auxiliar2 ;
          auxiliar->demanda = auxiliar->demanda ;
        }else
          salir = 1 ;
    }while(salir!=1);
    auxiliar = NULL;
    free(uxiliary);
    auxiliar2 = NULL;
    free(auxiliar2);
    ver = otro; //luego de escoger los clientes se procede a borrar la lista de otros clientes
generada
    do{
      otro = ver->sig;
      ver->sig = NULL;
      ver->ant = NULL;
      free(ver);
      ver = otro;
    }while(otro!=NULL);
    free(otro);
}

void eliminarclientes2(){
     struct ClientesEspera *ver;
    ver = clientes_utiliza;
    do{
      clientes_utiliza = ver->sig;
      ver->sig = NULL;
      ver->ant = NULL;
      free(ver);
      ver = clientes_utiliza;
    }while(clientes_utiliza!=NULL);
    free(clientes_utiliza);
}
```

Solucion_inicial.c

```
/*****************************************************************************************************************/
Archivo con las operaciones encargadas para generar una solución inicial a utilizarse por ambas
heurísticas
/*****************************************************************************************************************/

        #include "cabecera.h"

        struct ClientesEspera *Clientes, *clienteant, *clienteact, *clientesig;
        struct Ruta *Vehicle,*VehicleAnt;
        int i,j;
        void cliente_aleatorio();
        void elimiarclientelejano();
        float eliminar_cliente(float cant,float dem);
        void retornar_proveedor(int i);
        void copiar_clientes();
        float minimo(float a, float b);

        int solucion_inicial(){
          struct ClientesEspera *ver;
          int vehiculos = F, v=0, otro=1, permitido = 0;
          float cant[N],q,falta;
          if(variable == 0)
             copiar_clientes(1);
          else{
             ver = clientes_utiliza;
             copiar_clientes(2);
          }
          for(i=0;i<N+1;i++){
              cant[i] = 0.0;  //cantidad a entregar para cada cliente (cant[i] <= demanda)
          }
          do{
            //Aun quedan clientes en la lista de clientes pero no hay vehiculos
            if(Clientes != NULL && vehiculos == 0){
                elimina_Ruta(1);
                eliminarclientes();
                falta = inicializar_clientes();
                elimiarclientelejano();
                solucion_inicial(1);
            }else{
              if(primero == 0){
                float semilla;
                semilla = time(NULL);
                srand(semilla); //obtenemos una semilla para generar aleatoriedad
                primero++;
              }
              otro = 1;
              cliente_aleatorio(); //escogemos un cliente
              i = clienteact->cliente;
              q = minimo(Q,(clienteact->demanda - cant[i]));
              cant[i] = cant[i] + minimo(Q,(clienteact->demanda - cant[i]));
              Vehicle = (struct Ruta *)malloc(sizeof(struct Ruta));
              Vehicle->cliente = i;
              Vehicle->demandaEntregada = cant[i];
              Vehicle->costocliente = costos[0][i];
              Vehicle->ant = NULL;
```

```
    Vehicle->sig = NULL;
    Sol_anterior[v] = Vehicle;
    VehicleAnt = Sol_anterior[v];
    Vehicle = NULL;
    free(Vehicle);
    cant[i] = eliminar_cliente(cant[i],q);
    if(q == Q || Clientes == NULL){
        retornar_proveedor(i);
        vehiculos--;
        v++;
        q=0.0;
    }else
       do{
        cliente_aleatorio();
        Vehicle = (struct Ruta *)malloc(sizeof(struct Ruta));
        i = clienteact->cliente;
        q = q + minimo(Q,(clienteact->demanda - cant[i]));
        cant[i] = cant[i] + minimo(Q,(clienteact->demanda - cant[i]));
        Vehicle->cliente = i;
        Vehicle->demandaEntregada = cant[i];
        j = VehicleAnt->cliente;
        Vehicle->costocliente = costos[j][i];
        Vehicle->ant = VehicleAnt;
        VehicleAnt->sig = Vehicle;
        Vehicle->sig = NULL;
        VehicleAnt = Vehicle;
        Vehicle = NULL;
        free(Vehicle);
        cant[i] = eliminar_cliente(cant[i],q);
        if(Clientes == NULL){
            retornar_proveedor(i);
            v++;
            vehiculos--;
            otro = 0;
        }else
          if(q >= Q){
            if(q==Q)
              retornar_proveedor(i);
            if(q>Q){
              Vehicle = VehicleAnt->ant;
              VehicleAnt = Vehicle;
              retornar_proveedor(j);
            }
            v++;
            vehiculos--;
            otro = 0;
            q=0;
            ver = Clientes;
            do{
               falta = falta + ver->demanda;
               ver = ver->sig;
            }while(ver != NULL);     }
       }while(otro!=0);
   }
  }while(Clientes!=NULL);
  return v;
}
```

```
float minimo(float a, float b){
   if(a<b)
     return(a);
   return(b);
}

void elimiarclientelejano(){
  struct ClientesEspera *ver,*anterior,*siguiente;
  ver = Clientes;
  siguiente = Clientes->sig;
  while(siguiente != NULL){
     if(ver->demanda < siguiente->demanda){
        anterior = ver;
     }else{
        anterior = siguiente;
        ver = siguiente;
     }
     siguiente = siguiente->sig;
  }
  if(anterior->ant!= NULL){
     ver = anterior->ant;
     if(anterior->sig != NULL){
        siguiente = anterior->sig;
        ver->sig = siguiente;
        siguiente->ant = ver;
     }else{
        siguiente = NULL;
        ver->sig = NULL;
     }
  }else{
     ver = NULL;
     siguiente = anterior->sig;
     siguiente->ant = NULL;
  }
  anterior->sig = NULL;
  anterior->ant = NULL;
  free(anterior);
  ver = NULL;
  siguiente = NULL;
  free(ver);
  free(siguiente);
}

/*****************************************************************************************************************/
/*Cliente Aleatorio: Función encargada de escoger un cliente de manera aleatoria de la lista de   */
/*clientes que esperan ser ingresados a una ruta.                                                  */
/*****************************************************************************************************************/

void cliente_aleatorio(){
    int random;
    random = rand() % N;
    clienteact = Clientes;
    clientesig = clienteact->sig;
    for(i=1;i<random;i++){
      if(clientesig!=NULL){
        clienteant = clienteact;
        clienteact = clientesig;
        clientesig = clienteact->sig;
```

```
            }
          }
        }

/*****************************************************************************************************************/
Sacar clientes: Función encargada de sacar los clientes que son asignados a una ruta de la lista de
clientes
/*****************************************************************************************************************/

        float eliminar_cliente(float cant,float dem){
            if(dem <= Q){
             if((cant == clienteact->demanda)){
                if(clienteact->ant == NULL){
                   if(clienteact->sig == NULL){
                         Clientes = NULL;
                         return cant;
                   }else{
                      Clientes = clienteact->sig;
                      Clientes->ant = NULL;
                      free(clienteact);
                      return cant;
                   }
                }else{
                   if(clienteact->sig == NULL){
                      clienteant->sig = NULL;
                      clientesig = NULL;
                      clienteact->ant = NULL;
                      clienteact->sig = NULL;
                      free(clienteact);
                      return cant;
                   }else{
                      clienteant->sig = clientesig;
                      clientesig->ant = clienteant;
                      clienteact->ant = NULL;
                      clienteact->sig = NULL;
                      free(clienteact);
                      return cant;
                   }
                }
             }
            }else
              return 0;
        }

/*****************************************************************************************************************/
/* Regresar al Proveedor: Función encargada de terminar una ruta regresando al vehículo al          */
/*proveedor, calcula los costos generados al realizar esta operación                                 */
/*****************************************************************************************************************/

        void retornar_proveedor(int i){
              Vehicle = (struct Ruta *)malloc(sizeof(struct Ruta));
              Vehicle->cliente = 0;
              Vehicle->demandaEntregada = 0.0;
              Vehicle->costocliente = costos[i][0];
              Vehicle->ant = VehicleAnt;
              VehicleAnt->sig = Vehicle;
              Vehicle->sig = NULL;
              Vehicle = NULL;
```

```
        free(Vehicle);
}

void copiar_clientes(int forma){
    int i, cliente, j;
    double demanda;
    struct ClientesEspera *auxiliar, *auxanterior, *canterior;
    if(forma == 1)
      canterior = Clientes;
    else
      canterior = clientes_utiliza;
    j = 0;
    do{
        auxiliar = (struct ClientesEspera *)malloc(sizeof(struct ClientesEspera));
        cliente = canterior->cliente;
        auxiliar->cliente = cliente;
        demanda = canterior->demanda;
        auxiliar->demanda = demanda;
        if(j == 0){
            auxiliar->ant = NULL;
            auxanterior = auxiliar;
            if(forma == 1)
                clientes_utiliza = auxiliar;
            else
                Clientes = auxiliar;
        }else{
            auxanterior->sig = auxiliar;
            auxiliar->ant = auxanterior;
            auxanterior = auxiliar;
        }
        canterior = canterior->sig;
        auxiliar = NULL;
        free(auxiliar);
        j++;
    }while(canterior!=NULL);
    auxanterior = NULL;
    free(auxanterior);
}
```

Tabu_search.c

```
/*****************************************************************************************************************/
 Archivo con las operaciones de la heurística Tabu Search.
/*****************************************************************************************************************/

        #include "cabecera.h"

        int n,espera = 0;
        int criterio_termino();
        void Evaluar_movimiento();
        int Criterio_Aspiracion();
        void solucion_aceptada();
        double Costo;

        void Tabu_Search(){
           int termino, cant = 1, rnd, fin,completo = 0,i,div=0;
           double Costo, costo2;
           clock_t comienzo;
           comienzo=clock();
           nrutas = solucion_inicial();  //inicializa la ruta para obtener una solución inicial
           mostrar_lista();
           copiar_ruta(0);
           copiar_ruta(1);
           costo2 = Costo_FObjetivo(3);
           printf("\nSOLUCION CON TABU SEARCH");
           printf("\n\nCosto Funcion Objetivo con una solucion inicial es %f",costo2);
           do{
             n = 0;
             do{
               rnd = rand() % 2;
               elegir_operacion(rnd);
               Evaluar_movimiento();  //comprueba si el movimiento es aceptado
               aumentar_tiempo_permanencia();  //de los movimiento tabu
               verifica_permanencia();  //ve si en la lista tabu hay elementos que se puede eliminar
               if(n == N*2)
                  completo = 1;
               if(espera == ((100*cant)+100*N))
                  completo = 1;
             }while(completo!=1);
             completo = 0;
             costo2 = Costo_FObjetivo(3);
             Costo = Costo_FObjetivo(2) - Costo_FObjetivo(3);
             if(Costo < 0){
                elimina_Ruta(2);
                copiar_ruta(1);
             }
             espera = 0;
             Costo = Costo_FObjetivo(3);
             if(costo2 == Costo)
                cant++;
             else
                cant = 1;
             termino = criterio_termino(div);
             if(termino == 0){
               if(cant == 5){ //Se escoge diversificación
                 if(VMI == 0) //se realiza otra ruta de solución inicial
                   nrutas = inicializar_clientes();
```

```
                else{
                  variable = 1;
                }
                  nrutas = solucion_inicial();
                  elimina_Ruta(1);
                  copiar_ruta(0);
                  Costo = Costo_FObjetivo(1);
                  elimina_lista_tabu();  //se elimina la lista tabu
                  div++;
                  cant = 1;
               }else{ //Se escoge intensificación
                  elimina_Ruta(0);
                  copiar_ruta(3);
                  reordena();    //se trata de mejorar la solución actual
                  elimina_Ruta(1);
                  copiar_ruta(0);
                  Costo = Costo_FObjetivo(1);
               }
             }else{
                fin = 1;
             }
          }while(fin != 1);
          costo2 = Costo_FObjetivo(3);
          printf("\n\nCosto Funcion Objetivo es %f",costo2);
          printf("\nTiempo de ejecución fue de %f",(clock()-comienzo)/(double)CLOCKS_PER_SEC);
          mostrar_lista();
       }

       int criterio_termino(int j){
          int resultado;
          //verifica si la solución mejor se ha repetido N veces (cantidad de cliente)
          //si es asi se da por terminado el algoritmo, sino continua
          if(j == 10)
             resultado = 1;
          else
             resultado = 0;
          return resultado;
       }

/*************************************************************************************************************/
/*Operación de Evaluación de movimiento: Función encargada de evaluar si la nueva solución es                */
/*aceptada o no lo es para esto verifica la función la lista tabu, criterios de aspiración y                 */
/*comparación de costos                                                                                      */
/*************************************************************************************************************/

       void Evaluar_movimiento(){
          int cliente, movimiento, eletabu, criterio, eletabu2;
          //Se comprueba si la solución encontrada disminuye el valor de la solución anterior
          Costo = Costo_FObjetivo(2) - Costo_FObjetivo(1);
          if(Costo<0){
             espera = 0;
             //verifica si el movimiento es un movimiento tabu
             eletabu = Es_tabu(cliente1,random4,posicion2);
             if(cliente2 == 0){  //dependiendo de la operacion verifica si es movimiento tabu
                //la operación es de ingresar un cliente de una ruta a otra
                eletabu2 = 0;
             }else{
                //la operación es de intercambio de cliente entre dos rutas
```

```
                eletabu2 = Es_tabu(cliente2, random3,posicion1);
            }
            //Si el movimiento resulta no ser tabu se agrega a la lista tabu y la solución es
        aceptada
            if(eletabu == 0 && eletabu2 == 0){
                agregar_elemento_tabu(cliente1,random4,posicion2);
                if(cliente2 != 0)
                    agregar_elemento_tabu(cliente2,random3,posicion1);
                solucion_aceptada();
            }else{ //Si el movimiento es tabú
                if(eletabu == 0 && eletabu2 != 0) //Comprueba los criterios de Aspiración
                    criterio = Criterio_Aspiracion(cliente2,random3,posicion1,0);
                if(eletabu2 == 0 && eletabu != 0)
                    criterio = Criterio_Aspiracion(cliente1,random4,posicion2,0);
                if(eletabu != 0 && eletabu2 != 0)
                    criterio = Criterio_Aspiracion(0,0,0,1);
                if(criterio==1) //el movimiento es revocado por un criterio de termino
                    solucion_aceptada();
            }
        }else{  //movimiento no aceptado ya que no disminuye el costo de la solución anterior
            elimina_Ruta(1);
            copiar_ruta(0);
            espera++;
        }
        verifica_permanencia();
    }

/*****************************************************************************************************************/
/*Operación de Criterio de Aspiración: Función encargada de evaluar los criterios de aspiración si */
/*un movimiento resulta ser Tabú para ver si será aceptada la solución o no                       */
/*****************************************************************************************************************/

    int Criterio_Aspiracion(int cliente, int movimiento, int posicion, int tipo){
        int aceptar;
        //1er Criterio de Aspiracion:
        // Solución encontrada disminuye el costo comparandola con la mejor solución
        Costo = Costo_FObjetivo(3) - Costo_FObjetivo(2);
        if(Costo < 0){
            if(tipo == 0){
                quitar_movimiento_Tabu(cliente,movimiento,posicion2);
            }else{
                quitar_movimiento_Tabu(cliente1,random4,posicion2);
                quitar_movimiento_Tabu(cliente2,random3,posicion1);
            }
            return 1;
        }
        //2da Criterio de Aspiración
        // Se comprueba si el movimiento cumplio su tiempo de permanencia en la lista Tabú
        verifica_permanencia();
        aceptar = Es_tabu(cliente1,random4,posicion2);
        if(aceptar == 0)
            aceptar = Es_tabu(cliente2,random3,posicion1);
        if(aceptar == 0)
            return 1; //se ha quitado el elemento tabu
        return 0;
    }

    void solucion_aceptada(){
```

```
    int i;
    elimina_Ruta(0);
    copiar_ruta(2);
    elimina_Ruta(1);
    copiar_ruta(0);
    n++;
}
```

Lista_tabu.c

```
/*****************************************************************************************************************/
Archivo con las operaciones encargadas para el manejo de la lista Tabú de la heurística de Tabu
Search
/*****************************************************************************************************************/

        #include "cabecera.h"

        int Es_tabu(int cliente,int movimiento,int posicion){
            struct ListaTabu *elesig, *elemento;
            elemento = Tabu;
            if(elemento == NULL)//No existen elementos en la lista Tabu
              return 0; //indica que no es un elemento tabu
            else //existe elementos en la lista tabu, buscar si el movimiento lo es
              elesig = elemento->sig;
              while(elemento!=NULL)
                if(elemento->cliente == cliente) //ve si en la lista hay un elemento con el mismo
        cliente
                    if(elemento->NumeroRuta == movimiento) //comprueba si la lista esta el
        movimiento
                        if(elemento->posicion == posicion)  //comprueba si efectua el mismo movimiento
                           return 1; //es un movimiento tabu
                        else
                           return 0;  //no es un movimiento tabu
                    else
                        return 0; //no es un movimiento tabu
                 else
                    return 0; //no es un movimiento tabu
        }

/*****************************************************************************************************************/
/*Operación de eliminar un elemento tabú: Función encargada de sacar de la lista tabú un          */
/*movimiento, para que  este deje de ser tabú.                                                    */
/*****************************************************************************************************************/

        void quitar_movimiento_Tabu(int cliente, int movimiento,int posicion){
            struct ListaTabu *eleant, *elesig, *elemento;
            int i=0;
            elemento = Tabu;   //Se posiciona en el comienzo de la lista Tabu
            eleant = elemento->ant;
            elesig = elemento->sig;
            while(elemento!=NULL){
              //verifica si el movimiento Tabu guardado es el del cliente
              if(elemento->cliente == cliente){
                //si es del cliente verifica si es el movimiento que estamos buscando
                if(elemento->NumeroRuta == movimiento){
                     if(elemento->posicion == posicion){
                        if(i==0){
                           Tabu = elesig;
                           if(elesig != NULL)
                              elesig->ant = eleant;
                        }else{
                           eleant->sig = elesig;
                           if(elesig != NULL)
                              elesig->ant = eleant;
                        }
                        elemento->ant = NULL;
```

```
                    elemento->sig = NULL;
                    elemento = NULL;
                    free(elemento);
                  }else{ //sino es así se sigue avanzando en la lista tabú
                    eleant = elemento;
                    elemento = elesig;
                    elesig = elemento->sig;
                    i++;
                  }
              }else{     //sino es así se sigue avanzando en la lista tabú
                eleant = elemento;
                elemento = elesig;
                elesig = elemento->sig;
                i++;
              }
             }
            }
            elesig = NULL;
            eleant = NULL;
            free(elesig);
            free(eleant);
          }

/*************************************************************************************************************/
/*Operación de ver permanencia de elementos en lista: Función encargada de ver si un elemento*/
/*tabú de  la lista tabú ha alcanzado el momento de salir de la lista tabú.                   */
/*************************************************************************************************************/

          void verifica_permanencia(){
            struct ListaTabu *eleant, *elesig, *elemento;
            int i = 0;
            if(Tabu != NULL){
              elemento = Tabu;
              eleant = elemento->ant;
              elesig = elemento->sig;
              while(elemento!=NULL){
                //verifica si el movimiento sobrepasa su limite de permanencia
                if(elemento->permanencia == 17){
                  if(i==0){ //Si alcanzo el límite
                      Tabu = elesig;
                      if(elesig != NULL)
                        elesig->ant = eleant;
                  }else{
                      eleant->sig = elesig;
                      if(elesig != NULL)
                        elesig->ant = eleant;
                  }
                  elemento->ant = NULL;
                  elemento->sig = NULL;
                  elemento = NULL;
                  free(elemento);
                }else{ //sino es así se sigue avanzando en la lista tabú
                  eleant = elemento;
                  elemento = elesig;
                  i++;
                  if(elesig != NULL)
                    elesig = elemento->sig;
                }
```

```
            }
            elesig = NULL;
            eleant = NULL;
            free(elesig);
            free(eleant);
          }
        }

/*****************************************************************************************************************/
/*Operación de agregar un elemento tabú: Función encargada de ingresar en la lista tabú un       */
/*movimiento, para este se convierta en un movimiento tabú.                                      */
/*****************************************************************************************************************/

        void agregar_elemento_tabu(int cliente, int movimiento, int posicion){
            struct ListaTabu *elemento, *auxtabu;
            //Nos creamos nuestro elemento y le agregamos los datos necesarios
            elemento = (struct ListaTabu *)malloc(sizeof(struct ListaTabu));
            elemento->cliente = cliente;
            elemento->NumeroRuta = movimiento;
            elemento->posicion = posicion;
            elemento->permanencia = 1;
            elemento->sig = NULL;
            if(Tabu == NULL){
               Tabu = elemento;
               elemento = NULL;
               free(elemento);
            }else{      //Nos posicionamos al final de la lista Tabu
               auxtabu = Tabu;
               while(auxtabu->sig!=NULL){
                  auxtabu = auxtabu->sig;
               }    //insertamos el nuevo elemento al final de la lista Tabu
               elemento->ant = auxtabu;
               auxtabu->sig = elemento;
               auxtabu = NULL;
               free(auxtabu);
               elemento = NULL;
               free(elemento);
            }
        }

/*****************************************************************************************************************/
/*Operación de aumento de tiempo de permanencia: Función encargada de ir aumentando el    */
/*contador  que nos indica el  tiempo en que el movimiento tabu permanece en la lista.               */
/*****************************************************************************************************************/

        void aumentar_tiempo_permanencia(){
            struct ListaTabu *auxtabu;
            int tiempo;
            if(Tabu != NULL){
              auxtabu = Tabu;
              do{
                 tiempo = auxtabu->permanencia;
                 tiempo++;
                 auxtabu->permanencia = tiempo;
                 auxtabu = auxtabu->sig;
              }while(auxtabu != NULL);
            }
        }
```

```
void elimina_lista_tabu(){
   struct ListaTabu *elesig, *auxiliar;
   if(Tabu != NULL)
      elesig = Tabu->sig;
   while(Tabu!=NULL){
      auxiliar = Tabu;
      Tabu = elesig;
      if(Tabu != NULL)
         elesig = Tabu->sig;
      auxiliar->sig = NULL;
      auxiliar->ant = NULL;
      free(auxiliar);
      }
   free(elesig);
}

void mostrar_lista_tabu(){
  int i;
  struct ListaTabu *Prueba;
  if(Tabu!=NULL){
     Prueba = Tabu;
     do{
        printf("cliente %d movimiento %d con posicion %d y permanencia %d\n",Prueba-
>cliente,Prueba->NumeroRuta, Prueba->posicion,Prueba->permanencia);
        Prueba = Prueba->sig;
     }while(Prueba!=NULL);
     system("pause");
  }
}
```

Simulated_annealing.c

```
/*****************************************************************************************************************/
Archivo con las operaciones de la heurística Simulated Annealing
/*****************************************************************************************************************/

        #include "cabecera.h"

        double temperatura_inicial(double probabilidad);

        void Simulated_Annealing(){
          int Operacion, i, n, j, r,l,s, salir = 0, termino1 = 0, termino2;
          double Costo, random, z, Temp[100];
          double alfa = 0.3;
          struct Ruta *auxiliar;
          clock_t comienzo;
          comienzo=clock();
          nrutas = solucion_inicial();  //Se genera una solución inicial, e indica cuantas rutas se
        utilizaron
          //Se realizan algunas copias de la solucion inicial
          copiar_ruta(0);
          copiar_ruta(1);
          Costo = Costo_FObjetivo(3);
          printf("\nSOLUCION CON SIMULATED ANNEALING");
          printf("\n\nCosto Funcion Objetivo etapa inicial es de %f",Costo);
          Temp[0] = temperatura_inicial(alfa); //Se calcula la temperatura inicial
          r=0;
          s=0;
          j=1; //indica la cantidad de veces que pasa por el ciclo de aceptaciones
          do{
            n=0; //indica la cantidad de aceptaciones por nivel de temperatura
            l=0;
            do{
              Operacion = rand() % 6; //inicializamos la variable que indica la cantidad de
        aceptaciones por T°
              elegir_operacion(Operacion);
              Costo = Costo_FObjetivo(2) - Costo_FObjetivo(3);
             //Comprueba si la nueva solución nos da un mejor valor final, si es así, se declara la
        solución
             //nueva como la mejor solución y también las modificaciones se realizarán sobre ésta
        para
             //encontrar nuevas soluciones
             if(Costo < 0){
                elimina_Ruta(0);
                copiar_ruta(2);
                elimina_Ruta(1);
                elimina_Ruta(2);
                copiar_ruta(0);
                copiar_ruta(1);
                n++;
                l=0;
             //si no es así se procede a comprobar cual es la probabilidad de aceptar la
             //solución generada para trabajar sobre ella
             }else{
                 random = rand()/(double)RAND_MAX;
                 Costo = Costo * -1;
                 z = exp(Costo/Temp[r]);
                 //si es aceptada la solución se incrementa el numero de soluciones
```

```
            //encontradas por temperatura y se acepta trabajar sobre la solucion
            if(random < z){
               n++;
               l=0;
               elimina_Ruta(0);
               copiar_ruta(2);
               elimina_Ruta(1);
               copiar_ruta(0);
            }else{
               //Si no es aceptada continua utilizando la misma solución anterior
               //para la generación de otra solución
               elimina_Ruta(1);
               copiar_ruta(0);
               l++;
            }
          }
          termino2 = (N*j);
          if(n >= termino2)
            termino1 = 1;
          termino2 = (N*100)+(j*100);
          if(l >= termino2)
            termino1 = 1;
        }while(termino1 != 1);
        termino1 = 0;
        j++;
        r++;
        Temp[r] = alfa*Temp[r-1];
        Costo = Costo_FObjetivo(2);
        if(Costo == Costo_FObjetivo(3))
           s++;
        else
           s = 0;
        if(Temp[r] < 0.0001)
           salir = 1;
        if(s == N)
           salir = 1;
      }while(salir!=1); //termina cuando la temperatura es menor que un cierto
      Costo = Costo_FObjetivo(3);
      printf("\n\nCosto de solucion final es: %f\n\n",Costo);
      printf("\nTiempo de ejecución fue de %f",(clock()-comienzo)/(double)CLOCKS_PER_SEC);
      mostrar_lista();
    }

/*****************************************************************************************************************/
/*Operación de Obtener costo de Ruta: Función encargada de obtener los costos finales de las     */
/*soluciones entregadas a la función                                                             */
/*****************************************************************************************************************/

    double Costo_FObjetivo(int tipo){
       struct Ruta *Total;
       double Costo;
       int i;
       Costo = 0.0;
       for(i=0;i<nrutas;i++){
          if(tipo == 1)
             Total = Sol_anterior[i];
          if(tipo == 2)
             Total = Sol_nueva[i];
```

```
            if(tipo == 3)
                Total = Sol_mejor[i];
            do{
              Costo = Costo + Total->costocliente;
              Total = Total->sig;
            }while(Total!=NULL);
          }
         return Costo;
        }

/*****************************************************************************************************************/
/*Operación de Temperatura inicial: Función encargada de obtener la temperatura inicial                          */
/*****************************************************************************************************************/

        double temperatura_inicial(double probabilidad){
            double variacion, resultado;
            variacion = N;
            variacion = variacion * -1;
            resultado = variacion/log(probabilidad);
            return resultado;
        }
```

Intercambiar.c

```
/*****************************************************************************************************************/
Archivo con las operaciones encargadas de formular una nueva solución a traves de una soluciòn
anterior, este archivo es ocupado por ambas heuísticas (Simulated Annealing y Tabu Search)
/*****************************************************************************************************************/

        #include "cabecera.h"

        struct Ruta *Aux,*siguiente, *aux2, *anterior, *Aux1,*nant;
        struct Ruta *nact,*nsig, *cant, *cant0, *Prueba;
        double costosolucion;
        int i,n,j, random1, random2;
        float cant1, cant2;

        void cambiar_nodo();
        void intercambio_nodos();

        void elegir_operacion(int Ope){
            int rnd;
            if(nrutas == 1)
               posicion_nodo(0);
            else
               if(Ope == 0){
                  rnd = rand() % nrutas;
                  posicion_nodo(rnd);
               }else if(Ope == 1 || Ope == 3)
                  intercambio_nodos();
               else
                  cambiar_nodo();
        }

/*****************************************************************************************************************/
/*Operación de intercambio de nodos: Función encargada de escoger de manera  aleatoria dos */
/*nodos de dos rutas aleatorias, se intercambia la posicion entre ellos (siempre y cuando no pase  */
/*las restricciones).                                                                                            */
/*****************************************************************************************************************/

        void intercambio_nodos(){
           int paso=0, contador1=0, contador2=0;
           do{
             //indica la cantidad de demanda de las rutas escogidas
             cant1=0;
             cant2=0;
             //numero aleatorios para elegir entre rutas y clientes
             random1 = rand() % N/nrutas;
             random2 = rand() % N/nrutas;
             random3 = rand() % nrutas;
             random4 = rand() % nrutas;
             //verifica que las rutas escogidas no sean las mismas
             if(random3 == random4){
                random3 = rand() % nrutas;
                if(random3 == random4){
                    random3++;
                    if(random3>=nrutas)
                       if(random4 == 0)
                          random3 = 1;
                       else
```

```
                random3 = 0;
            }
        }
        //realiza una copia de las rutas por si no cumple las restricciones
        preparar_listas(random3,1);
        preparar_listas(random4,2);
        aux2 = Aux; //busca al cliente escogido de esta ruta
        siguiente = aux2->sig;
        anterior = aux2->ant;
        for(i=1;i<random1;i++)
          if(random1 == 1 || siguiente->cliente!=0){
            anterior = aux2;
            aux2 = siguiente;
            siguiente = aux2->sig;
            contador1++;
          }
        cliente1 = aux2->cliente;
        posicion1 = contador1;
        cant0 = Aux;  //calcula la cantidad ocupada del vehículo de toda la ruta
        do{
          cant1 = cant1 + cant0->demandaEntregada;
          cant0 = cant0->sig;
        }while(cant0!=NULL);
        //se le quita la demanda a entregar del cliente a intercambiar
        cant1 = cant1 - aux2->demandaEntregada;
        nact = Aux1; //busca al cliente escogido de esta ruta
        nsig = nact->sig;
        nant = nact->ant;
        for(i=1;i<random2;i++)
          if(random2 == 1 || nsig->cliente!=0){
              nant = nact;
              nact = nsig;
              nsig = nact->sig;
              contador2++;
          }
        cliente2 = nact->cliente;
        posicion2 = contador2;
        cant = Aux1;  //calcula la cantidad ocupada del vehículo de toda la ruta
        do{
          cant2 = cant2 + cant->demandaEntregada;
          cant = cant->sig;
        }while(cant!=NULL);
        //se le quita la demanda a entregar del cliente a intercambiar
        cant2 = cant2 - nact->demandaEntregada;
        //se le suma la demanda de los clientes nuevos de la ruta
        cant1 = cant1 + nact->demandaEntregada;
        cant2 = cant2 + aux2->demandaEntregada;
        //verifica que cumpla la restriccion de capacidad del vehiculo, si se cumple realiza el
        //intercambio, sino se comienza a buscar de nuevo un par de rutas y clientes a
    intercambiar
        if(cant1 <= Q && cant2 <= Q){
          //caso en que el cliente escogido de la ruta a es el primer cliente que se visita
          if(anterior==NULL)
            if(nant==NULL){   //a su vez en la ruta b sucede lo mismo
              siguiente->ant = nact;
              nsig->ant = aux2;
              nact->sig = siguiente;
              aux2->sig = nsig;
```

```
            }else{  //cliente intermedio de la ruta b
              Aux = nact;
              Aux->ant = NULL;
              Aux->sig = siguiente;
              siguiente->ant = Aux;
              aux2->ant = nant;
              nant->sig = aux2;
              aux2->sig = nsig;
              nsig->ant = aux2;
            }
          else //es un cliente intermedio de la ruta a
            if(nant==NULL){  //primer cliente a visitar en la ruta b
              Aux1 = aux2;
              Aux1->ant = NULL;
              Aux1->sig = nsig;
              nsig->ant = Aux1;
              nact->ant = anterior;
              anterior->sig = nact;
              nact->sig = siguiente;
              siguiente->ant = nact;
            }else{ //cliente intermedio en la ruta a
              aux2->ant = nant;
              aux2->sig = nsig;
              nant->sig = aux2;
              nsig->ant = aux2;
              nact->ant = anterior;
              nact->sig = siguiente;
              anterior->sig = nact;
              siguiente->ant = nact;
            }
          for(i=0;i<nrutas;i++){        //realizado el intercambio se procede a entregar la nueva
        solución
            if(i==random3){
              Sol_nueva[i] = Aux;
            }
            if(i==random4){
              Sol_nueva[i] = Aux1;
            }
          }
          actualizar_costos(2); //modifica el cambio en los costos al realizarse el intercambio de
        clientes
          paso=1; //indica que se encontro una nueva solución que cumple con las restricciones
        }
       }while(paso!=1);
      }

/*****************************************************************************************************************/
/*Operación de cambiar nodo de ruta: Función encargada de escoger de manera aleatoria un */
/*nodos de una ruta aleatoria, y colocarlo en otra ruta aleatoria (siempre y cuando no pase las */
/*restricciones).                                                                                                */
/*****************************************************************************************************************/

      void cambiar_nodo(){
        int paso=0, contador = 0;
        do{
          cant2=0.0; //indicador de carga de un camión
          random1 = rand() % N/nrutas;  //número de cliente aleatorio
          random2 = rand() % N/nrutas;  //número de cliente aleatorio
```

```
    random3 = rand() % nrutas;  //número de ruta aleatoria
    random4 = rand() % nrutas;  //número de ruta aleatoria
    //Verifica que las rutas escogidas no sean las mismas
     if(random3 == random4){
        random3 = rand() % nrutas;
        if(random3 == random4){
            random3++;
            if(random3>=nrutas)
              if(random4 == 0)
                 random3 = 1;
              else
                 random3 = 0;
        }
     }
    //realiza una copia de las rutas por si no cumple las restricciones
    preparar_listas(random3,1);
    preparar_listas(random4,2);
    aux2 = Aux; //busca al cliente escogido de esta ruta
    siguiente = aux2->sig;
    anterior = aux2->ant;
    for(i=1;i<random1;i++)
      if(random1 == 1 || siguiente->cliente!=0){
         anterior = aux2;
         aux2 = siguiente;
         siguiente = aux2->sig;
      }
   cliente1 = aux2->cliente;
   nact = Aux1; //busca la posicion escogido de esta ruta
   nsig = nact->sig;
   nant = nact->ant;
   for(i=1;i<random2;i++)
      if(random2 == 1 || nsig->cliente!=0){
         nant = nact;
         nact = nsig;
         nsig = nact->sig;
         contador++;
      }
   cliente2 = 0;
   posicion2 = contador;
   cant = Aux1; //Calcula la cantidad de espacio ocupado por el vehículo
   do{
     cant2 = cant2 + cant->demandaEntregada;
     cant = cant->sig;
   }while(cant!=NULL);
   //Se verifica si la ruta tiene la capacidad de soportar otro cliente
   cant2 = cant2 + aux2->demandaEntregada;
   //Si se cumple la condición pasa a ingresar el cliente entre el cliente
   //escogido de la otra ruta, sino es asi pasa a reiniciar el proceso
   if(cant2 <= Q){
     if(anterior==NULL){  //Cliente a intercambiar es el primer cliente a visitar en la ruta
        siguiente->ant = anterior;
        Aux = siguiente;
        //si el lugar donde se va a ingresar el nuevo cliente en la ruta es al principio de la otra
ruta
        if(nant == NULL){
           random1 = rand() % 2;
           //se elige colocar el cliente nuevo como primer cliente a visitar en la ruta escogida
           if(random1 == 0){
```

```
                aux2->sig = nact;
                nact->ant = aux2;
                Aux1 = aux2;
            //se elige colocar el cliente nuevo después del primer cliente que
            //se visita en la ruta escogida
            }else{
                aux2->ant = nact;
                aux2->sig = nsig;
                nact->sig = aux2;
                nsig->ant = aux2;
            }
        //el lugar a colocar el cliente es entre medio de otros clientes en la ruta escogida
        }else{
            aux2->ant = nact;
            aux2->sig = nsig;
            nact->sig = aux2;
            nsig->ant = aux2;
        }
      }else{ //cliente a intercambiar es un cliente de entre medio
        anterior->sig = siguiente;
        siguiente->ant = anterior;
        //si el lugar donde se va a ingresar el nuevo cliente en la ruta es al principio de la otra
ruta
        if(nant == NULL){
            random1 = rand() % 2;
            //se elige colocar el cliente nuevo como primer cliente a visitar en la ruta escogida
            if(random1 == 0){
                aux2->ant = nant;
                aux2->sig = nact;
                nact->ant = aux2;
                Aux1 = aux2;
            //se elige colocar el cliente nuevo después del primer cliente que
            //se visita en la ruta escogida
            }else{
                aux2->ant = nact;
                aux2->sig = nsig;
                nsig->ant = aux2;
                nact->sig = aux2;
            }
        }else{ //el lugar a colocar el cliente es entre medio de otros clientes en la ruta
escogida
            aux2->ant = nact;
            aux2->sig = nsig;
            nsig->ant = aux2;
            nact->sig = aux2;
        }
      }
       for(i=0;i<nrutas;i++){ //realizado el intercambio se procede a entregar la nueva solución
        if(i==random3)
          Sol_nueva[i] = Aux;
        if(i==random4)
          Sol_nueva[i] = Aux1;
       }
       actualizar_costos(2); //modifica el cambio en los costos al realizarse el intercambio de
clientes
       paso=1; //indica que se encontro una nueva solución que cumple con las restricciones
      }
    }while(paso!=1);
```

```
        }

/*****************************************************************************************************************/
/*Operación de copiar ruta: Función encargada de copiar las dos rutas escogidas aleatoriamente*/
/*en listas auxiliares para de esta manera no modificarlas si no mejora la    solucion.                          */
/*****************************************************************************************************************/

        void preparar_listas(int random, int tipo){
            aux2 = Sol_nueva[random];
            int i=0;
            do{
              siguiente = (struct Ruta *)malloc(sizeof(struct Ruta));
              siguiente->cliente = aux2->cliente;
              siguiente->demandaEntregada = aux2->demandaEntregada;
              siguiente->costocliente = aux2->costocliente;
              siguiente->ant = aux2->ant;
              if(i==0)
                 if(tipo==1)
                    Aux = siguiente;
                 else
                    Aux1 = siguiente;
              aux2 = aux2->sig;
              siguiente->sig = aux2;
              siguiente = NULL;
              free(siguiente);
              i++;
            }while(aux2!=NULL);
            aux2 = NULL;
        }


/*****************************************************************************************************************/
/*Operación de cambiar nodo de ruta: Función encargada de escoger un cliente de una ruta       */
/*y colocarlo en posición en la misma ruta.                                                    */
/*****************************************************************************************************************/

        int posicion_nodo(int nruta){
            //calcular el tamaño de clientes en la ruta dada
            int ncliente=0, numero; //variable donde se guardarán la cantidad de clientes
            struct Ruta *ver;
            preparar_listas(nruta,1);
            aux2 = Aux;
            siguiente = aux2;
            random4 = nruta;
            do{
              ncliente++;
              siguiente = siguiente->sig;
            }while(siguiente != NULL);
            numero = ncliente - 1;
            if(ncliente>=3){
              //escoger una posición de la ruta donde se realiza el intercambio y
              //escoger el cliente a intercambiar
              random1 = rand() % ncliente; //posicion
              random2 = rand() % numero; //cliente
              //comprobamos que el cliente realize el intercambio evitando posiciones
              //que lo deje en el mismo lugar
              if(random1 == random2 || random1==(random2+1)){
                 //se realiza un intento más para encontrar otra posición aleatoria
```

```
  random1 = rand() % ncliente;
  //si a pesar de esto la posicion sigue fallando
  if(random1 == random2 || random1==(random2+1)){
    if(random2 == 0) //comprueba que el cliente no es el primero
      random1 = 2;
    else
      if(random2 == random1)
        random1 = random2 - 1;
      else if(random2 == numero-1)
        random1 = numero - 2;
      else
        random1 = random1 + 1;
  }
}
cliente2 = 0;
aux2 = Aux; //nos posicionamos en la posición indicada
for(i=1;i<random1;i++)
   aux2 = aux2->sig;
posicion2 = random1;
nact = Aux; //nos posicionados en el cliente que se va a mover
for(i=0;i<random2;i++)
   nact = nact->sig;
cliente1 = nact->cliente;
//ingresamos al cliente en la posicion
if(random2 == 0){ //el cliente a mover es el primero
   siguiente = nact->sig;
   nact->sig = aux2->sig;
   nsig = aux2->sig;
   nsig->ant = nact;
   siguiente->ant = nact->ant;
   Aux = siguiente;
   aux2->sig = nact;
   nact->ant = aux2;
}else{
   if(random1 == 0){
     if(random2 == 1){ //se intercambia el 1° cliente con el 2°
       siguiente = nact->sig;
       aux2->sig = siguiente;
       siguiente->ant = aux2;
       aux2->ant = nact;
       nact->sig = aux2;
       nact->ant = NULL;
       Aux = nact;
     }else{   //cuando es otro cliente que toma la primera posicion
       siguiente = nact->sig;
       anterior = nact->ant;
       anterior->sig = siguiente;
       siguiente->ant = anterior;
       nact->sig = aux2;
       aux2->ant = nact;
       nact->ant = NULL;
       Aux = nact;
     }
  }else if(random2 == 1){
     siguiente = nact->sig;
     siguiente->ant = Aux;
     Aux->sig = siguiente;
     nsig = aux2->sig;
```

```
            aux2->sig = nact;
            nact->ant = aux2;
            nact->sig = nsig;
            nsig->ant = nact;
          }else{
            anterior = nact->ant;
            siguiente = nact->sig;
            anterior->sig = siguiente;
            siguiente->ant = anterior;
            nsig = aux2->sig;
            aux2->sig = nact;
            nact->ant = aux2;
            nsig->ant = nact;
            nact->sig = nsig;
          }
        }
        for(i=0;i<nrutas;i++){ //realizando el intercambio se procede a guardar la nueva solucion
          if(i==nruta){
            Sol_nueva[i] = Aux;
          }
        }
        actualizar_costos(2); //modifica el cambio en los costos al realizarse el intercambio de
    clientes
        return 0;
       }else
        return 1;
    }

/*****************************************************************************************************************/
/*Operación de arreglar los costos de las rutas: Función encargada de arreglar los costos que se    */
/*modifico al realizar las modificaciones en la ruta original, ingresa el costo actual.             */
/*****************************************************************************************************************/

    void actualizar_costos(int tipo){
      struct Ruta *auxiliar, *canterior;
      for(i=0;i<nrutas;i++){
        if(tipo==2){
          auxiliar = Sol_nueva[i];
          Sol_nueva[i]->costocliente = costos[0][auxiliar->cliente];
        }else{
          auxiliar = Sol_anterior[i];
          Sol_anterior[i]->costocliente = costos[0][auxiliar->cliente];
        }
        do{
          canterior = auxiliar;
          auxiliar = auxiliar->sig;
          auxiliar->costocliente = costos[canterior->cliente][auxiliar->cliente];
        }while(auxiliar->sig!=NULL);
      }
      auxiliar = NULL;
      canterior = NULL;
      free(auxiliar);
      free(canterior);
    }

    void mostrar_lista(){
      for(i=0;i<nrutas;i++){
        Prueba = Sol_anterior[i];
```

```
        printf("\n\n Sol anterior Ruta %d : \n",i+1);
        do{
          printf("cliente %d con demanda %f y costo de entrega es %f\n",Prueba->cliente,Prueba-
>demandaEntregada,Prueba->costocliente);
          Prueba = Prueba->sig;
        }while(Prueba!=NULL);
      }
    }


/*****************************************************************************************************************/
/*Copiar Ruta: Función encargada de realizar unas copias de la solución inicial las cuales serán      */
/*Sol_nueva y Sol_mejor que se utilizarán después.                                                    */
/*****************************************************************************************************************/

    void copiar_ruta(int tipo){
      int i, cliente, j;
      double costo, demanda;
      struct Ruta *auxiliar, *auxanterior, *canterior;
      for(i=0;i<nrutas;i++){
        if(tipo == 3)
          canterior = Sol_mejor[i];
        if(tipo == 0)
          canterior = Sol_anterior[i];
        if(tipo == 1)
          canterior = Sol_anterior[i];
        if(tipo == 2)
          canterior = Sol_nueva[i];
        j=0;
        do{
          auxiliar = (struct Ruta *)malloc(sizeof(struct Ruta));
          cliente = canterior->cliente;
          auxiliar->cliente = cliente;
          demanda = canterior->demandaEntregada;
          auxiliar->demandaEntregada = demanda;
          costo = canterior->costocliente;
          auxiliar->costocliente = costo;
          auxiliar->sig = NULL;
          if(j == 0){
            auxiliar->ant = NULL;
            if(tipo == 0)
              Sol_nueva[i] = auxiliar;
            if(tipo == 1)
              Sol_mejor[i] = auxiliar;
            if(tipo == 3 || tipo == 2)
              Sol_anterior[i] = auxiliar;
            auxanterior = auxiliar;
          }else{
            auxanterior->sig = auxiliar;
            auxiliar->ant = auxanterior;
            auxanterior = auxiliar;
          }
          canterior = canterior->sig;
          auxiliar = NULL;
          free(auxiliar);
          j++;
        }while(canterior!=NULL);
        auxanterior = NULL;
```

```
            free(auxanterior);
        }
    }


/*****************************************************************************************************************/
/*Elimina ruta: Función encargada de eliminar los clientes de la solución elegida                                */
/*****************************************************************************************************************/

    void elimina_Ruta(int tipo){
        struct Ruta *ruta1, *rutasig, *auxiliar;
        int i;
        for(i=0;i<nrutas;i++){
            if(tipo == 0)
                ruta1 = Sol_anterior[i];
            if(tipo == 1)
                ruta1 = Sol_nueva[i];
            if(tipo == 2)
                ruta1 = Sol_mejor[i];
            if(ruta1 != NULL)
                rutasig = ruta1->sig;
            while(ruta1!=NULL){
                auxiliar = ruta1;
                ruta1 = rutasig;
                if(ruta1 != NULL)
                    rutasig = ruta1->sig;
                auxiliar->sig = NULL;
                auxiliar->ant = NULL;
                free(auxiliar);
            }
            if(tipo == 0)
                Sol_anterior[i] = NULL;
            if(tipo == 1)
                Sol_nueva[i] = NULL;
            if(tipo == 2)
                Sol_mejor[i] = NULL;
        }
        free(rutasig);
        ruta1 = NULL;
        free(ruta1);
    }
```

Reordenar.c

```
/*****************************************************************************************************************/
Archivo con las operaciones encargadas para el proceso de intensificación de la heurística de Tabu Search
/*****************************************************************************************************************/

        #include "cabecera.h"

        struct Ruta *orden;

        void reordena(){
           struct Ruta *auxiliar1, *auxiliar2, *auxiliar3, *auxiliar4, *auxiliar;
           int cliente1, cliente2, cliente3,i, n;
           double mincosto,costo;
           for(i=0;i<nrutas;i++){
               auxiliar1 = Sol_anterior[i];
               do{
                 if(auxiliar1->ant == NULL){
                    auxiliar4 = auxiliar1;
                    cliente3 = auxiliar4->cliente;
                    cliente2 = auxiliar4->cliente;
                    mincosto = costos[0][cliente3];
                    auxiliar2 = auxiliar4->sig;
                    while(auxiliar2->cliente != 0){
                      n = auxiliar2->cliente;
                      costo = costos[0][n];
                      if(costo < mincosto){
                         mincosto = costo;
                         cliente2 = auxiliar2->cliente;
                         auxiliar3 = auxiliar2;
                      }
                      auxiliar2 = auxiliar2->sig;
                    }
                 }else{
                    auxiliar4 = auxiliar1->ant;
                    cliente1 = auxiliar4->cliente;
                    n = auxiliar1->cliente;
                    mincosto = costos[cliente1][n];
                    cliente3 = auxiliar1->cliente;
                    cliente2 = auxiliar1->cliente;
                    auxiliar4 = auxiliar1;
                    auxiliar2 = auxiliar1->sig;
                    while(auxiliar2->cliente != 0){
                       n = auxiliar2->cliente;
                       costo = costos[cliente1][n];
                       if(costo < mincosto){
                          mincosto = costo;
                          cliente2 = auxiliar2->cliente;
                          auxiliar3 = auxiliar2;
                       }
                       auxiliar2 = auxiliar2->sig;
                    }
                 }
                 if(cliente3 != cliente2){
                    auxiliar2 = auxiliar4->ant;
                    auxiliar = auxiliar3->ant;
```

```
            //los clientes a intercambiar estan seguidos, por lo que auxiliar que es el cliente
anterior a
            //que se escodio para cambiar es igual a el cliente que se esta comparando
            if(auxiliar == auxiliar4)
               auxiliar4->ant = auxiliar3;
            else
               auxiliar4->ant = auxiliar;
            auxiliar3->ant = auxiliar2;
            auxiliar2 = auxiliar4->sig;
            auxiliar = auxiliar3->sig;
            auxiliar4->sig = auxiliar;
            if(auxiliar3 == auxiliar2) //los clientes a intercambiar estan seguidos
               auxiliar3->sig = auxiliar4;
            else
               auxiliar3->sig = auxiliar2;
            auxiliar->ant = auxiliar4;
            if(auxiliar2 != auxiliar3)
               auxiliar2->ant = auxiliar3;
            auxiliar = auxiliar4->ant;
            if(auxiliar != auxiliar3)
               auxiliar->sig = auxiliar4;
            if(auxiliar3->ant == NULL){
              Sol_anterior[i] = auxiliar3;
              auxiliar1 = auxiliar3;
            }else{
              auxiliar1 = auxiliar3;
              auxiliar = auxiliar3->ant;
              auxiliar->sig = auxiliar3;
            }
          }
          auxiliar1 = auxiliar1->sig;
        }while(auxiliar1->cliente != 0);
    }
    actualizar_costos(0);
}
```