Un algoritmo distribuído de Ruin & Recreate para el problema de ruteo de vehículos con ventanas de tiempo

Martínez Quijano, Andrés - LU: 32/98

27 de diciembre de 2010

Directora: Dra. Irene Loiseau

Departamento de Computación

Facultad de Ciencias Exactas y Naturales

Universidad de Buenos Aires

Argentina

Resumen

En esta tesis se considera el problema de ruteo de vehículos con ventanas de tiempo (Vehicle Routing Problem with Time Windows, VRPTW) y se propone un algoritmo distribuído de la metaheurística Ruin and Recreate para el mismo. El problema de VRPTW es una variante del problema de ruteo de vehículos (VRP) donde cada cliente tiene una ventana de tiempo asociada durante la cual puede ser visitado por un vehículo. El algoritmo y la implementación propuestas se ejecutan de forma paralela y transparente en un número ilimitado de nodos computacionales. Los resultados obtenidos mediante la aplicación paralela del algoritmo son competitivos con las mejores soluciones obtenidas heurísticamente.

Abstract

This thesis deals with the vehicle routing problem with time windows (VRPTW) and it introduces a distributed algorithm based on the Ruin and Recreate metaheuristic. The VRPTW is a variant of the vehicle routing problem (VRP) where each client has a time window of the time when it can be served by a vehicle. The algorithm can be transparently executed in parallel in many cores across many computers on a network. The obtained results were found to be competetive with the best-known results.

Índice

Índice de algoritmos

Índice de figuras

1. Descripción del problema

1.1. Introducción

El ruteo de vehículos es un problema de logística muy presente en la vida real. Consiste en la optimización de asignación de rutas a vehículos en un orden y tiempo determinados, de forma de minimizar la distancia total recorrida, la cantidad de vehículos necesarios para cumplir los requisitos, o, en general, ambas cosas. En su instancia más simple, el VRP[1], existe un número fijo de clientes con demandas de mercadería que debe ser satisfecha con la visita de uno y sólo un vehículo (o camión). Dado que los camiones tienen capacidad limitada, en general no es viable que un sólo camión pueda visitar y satisfacer la demanda de todos los clientes, por lo que es necesario que se agreguen camiones a la solución para que ésta sea factible.

Una solución a una instancia de VRP consiste en un conjunto de rutas que debe recorrer cada vehículo, partiendo y terminando en el *depósito,* de forma que todos los clientes sean visitados y sus demandas y restricciones sean cumplidas. El objetivo del VRP es encontrar la mejor solución, según el parámetro de optimización deseado (cantidad de vehículos, distancia total recorrida, tiempos de espera, etc.). Existen diversas variantes al problema original: en VRPB[2] donde además del conjunto de clientes a quienes se les debe entregar mercadería, existe otro conjunto de *proveedores* a quienes se debe ir en busca de mercadería para llevar de vuelta al depósito. Además, todas las entregas deben hacerse antes que las recogidas, para evitar reacomodar la carga del vehículo. VRPPD[3], donde no hay un depósito único de donde obtener la carga a repartir, sino que está distribuída en distintos puntos, por lo que debe ser recogida primero para poder ser entregada. VRPSD[4] permite que un cliente sea satisfecho por más de un vehículo, dando lugar a entregas parciales.

En esta tesis se trata exclusivamente la variante VRPTW[5] donde cada cliente tiene asociada una ventana de tiempo que restringe el momento en que puede ser visitada por un vehículo y un tiempo necesario de servicio. Si bien un camión puede arribar a un cliente antes de tiempo, deberá esperar al inicio de su ventana de tiempo antes de poder iniciar la descarga, pero nunca podrá arribar luego de cerrada la ventana. El tiempo de servicio de cada cliente se impone como el tiempo que toma la descarga de mercadería. En el caso que plantea esta tesis, la ventana de tiempo aplica solamente a la llegada del camión y no al tiempo de servicio, por lo que, por ejemplo, se acepta que un camión llegue a una hora tal a un cliente de forma que el servicio termine después de la ventana de tiempo haya cerrado. Para una revisión más completa del problema de ruteo de vehículos, modelos, algoritmos, variantes y formulaciones ver [TOT/02].

En este trabajo se propone una solución metaheurística de tipo Ruin And Recreate [SCH/99] multiobjetivo, con implementación distribuída que permita utilizar varias computadoras en red para resolver el problema.

En el resto de la sección 1 se introduce más detalladamente el VRPTW. En la sección 2 se detalla la metaheurística *Ruin & Recreate*. En la sección 3 se

[1] del inglés *Vehicle Routing Problem*
[2] del inglés *Vehicle Routing Problem with Backhauls*
[3] del inglés *Vehicle Routing Problem with Pickup and Delivery*
[4] del inglés *Vehicle Routing Problem with Split Delivery*
[5] del inglés *Vehicle Routing Problem with Time Windows*

presenta en detalle la solución propuesta por esta tesis. La sección 4 expone los resultados obtenidos al aplicar el algoritmo, y finalmente en las secciones 5 y 6 se muestran las conclusiones y posible trabajo futuro, respectivamente.

1.2. Formulación de VRPTW

Sea $G = (V, A)$ un grafo completo donde $V = \{0, \ldots, n\}$ es el conjunto de vértices y A el conjunto de ejes. El nodo 0 (también notado como nodo $n+1$) corresponde al depósito de donde parten los vehículos y $i = 1 \ldots n$ son los clientes. $N = V \setminus \{0\}$ es el conjunto de clientes. Además:

$\Delta^+(i) = \{j/(i,j) \in A\}$ los nodos *salientes* del nodo i

$\Delta^-(i) = \{j/(j,i) \in A\}$ los nodos *entrantes* al nodo i [6]

Un costo no negativo c_{ij} es asociado a cada eje $(i,j) \in A$, representando el costo de viajar desde el nodo i hasta el nodo j. En el caso que plantea esta tesis, se asume que el grafo no es dirigido, o sea $c_{ij} = c_{ji}\, \forall i, j \in A,\ i \neq j$.

Cada cliente tiene asociada una demanda no negativa d_i (siendo $d_0 = 0$) y una ventana de tiempo $[a_i, b_i]$ con $a_i < b_i$. Los parámetros E y L indican el comienzo y fin, respectivamente, de la ventana de tiempo del depósito: $[a_0, b_0] = [E, L]$, donde $a_0 \leq a_i\ \forall i \in V$ y $b_0 \geq b_i\ \forall i \in V$. También se asocia a cada cliente un valor no negativo s_i indicando el tiempo de servicio, esto es, el tiempo mínimo que un vehículo debe permanecer en el cliente cargando o descargando mercadería antes de poder proseguir su ruta.

Se dispone de K vehículos idénticos entre sí, cada uno con capacidad de carga C.

Se definen además las siguientes variables:

$$x_{ijk} = \begin{cases} 1 & \text{si } (i,j) \in A \text{ es usado por el vehículo } k \in K \\ 0 & \text{si no} \end{cases}$$

$w_{ik}, i \in V, k \in K$ el comienzo del servicio en el nodo i por el vehículo k

En el caso en que el objetivo sea minimizar el costo, el problema se puede plantear como un problema de programación lineal entera, donde la solución óptima consiste en hallar:

$$\text{mín} \sum_{k \in K} \sum_{(i,j) \in A} c_{ij} x_{ijk} \tag{1}$$

sujeto a:

$$\sum_{k \in K} \sum_{j \in \Delta^+(i)} x_{ijk} = 1 \qquad \forall i \in N \tag{2}$$

$$\sum_{j \in \Delta^+(0)} x_{0jk} = 1 \qquad \forall k \in K \tag{3}$$

$$\sum_{i \in \Delta^-(j)} x_{ijk} - \sum_{i \in \Delta^+(j)} x_{jik} = 0 \qquad \forall k \in K, j \in N \tag{4}$$

$$\sum_{i \in \Delta^-(n+1)} x_{i,n+1,k} = 1 \qquad \forall k \in K \tag{5}$$

[6] dado que en esta tesis se trabaja únicamente con grafos completos, $\Delta^+(i) = \Delta^-(i) = V \setminus \{i\}$

$$x_{ijk}(w_{ik} + s_i + c_{ij} - w_{jk}) \leq 0 \qquad \forall k \in K, (i,j) \in A \tag{6}$$

$$a_i \left(\sum_{j \in \Delta^+(i)} x_{ijk} \right) \leq w_{ik} \leq b_i \left(\sum_{j \in \Delta^+(i)} x_{ijk} \right) \qquad \forall k \in K, i \in N \tag{7}$$

$$E \leq w_{ik} \leq L \qquad \forall k \in K, i \in \{0, n+1\} \tag{8}$$

$$\sum_{i \in N} d_i \sum_{j \in \Delta^+(i)} x_{ijk} \leq C \qquad \forall k \in K \tag{9}$$

$$x_{ijk} \geq 0 \qquad \forall k \in K, (i,j) \in A \tag{10}$$

$$x_{ijk} \in \{0,1\} \qquad \forall k \in K, (i,j) \in A \tag{11}$$

la condición 11 permite que la restricción 6 sea linealizable como

$$w_{ik} + s_i + c_{ij} - w_{jk} \leq (1 - x_{ijk}) M_{ij} \qquad \forall k \in K, (i,j) \in A \tag{12}$$

donde M_{ij} son constantes grandes

(2) restringe las visitas a sólo un vehículo por cliente

(3)(4)(5) modelan el tipo de recorridos que un vehículo puede hacer (salir del depósito, visitar clientes y volver al depósito)

(6)(7)(8)(9) aseguran factibilidad en cuanto a ventanas de tiempo y capacidad de vehículos

(7) además obliga que $w_{ik} = 0$ si el vehículo k no visita al cliente i

En esta tesis se trabaja con un problema multiobjetivo, donde se prioriza la reducción de la cantidad de vehículos de la solución, y en segunda instancia la distancia total recorrida.

1.3. Metaheurísticas

Dado que VRPTW es un problema NP-HARD (ver [TOT/02]), desde el punto de vista de las aplicaciones reales resulta impracticable buscar la optimalidad mediante algoritmos exactos en instancias grandes, por lo que se acude a heurísticas o metaheurísticas para obtener buenas soluciones factibles de los problemas en tiempos acotados.

Algunas de las metaheurísticas más utilizadas son:

1. *Simulated Annealing*: basada en un proceso metalurgico que consiste en calentar y dejar enfriar controladamente un material. Al calentarse los átomos salen de sus posiciones originales y se muevan al azar, al enfriarse lentamente tienen mayores posibilidades de encontrar configuraciones de menor energía que la inicial

2. *Tabú search*: una búsqueda local que guarda una lista de soluciones ya visitadas para permitir expander el entorno y no atorarse en un mínimo local

3. *Algoritmos genéticos*: basada en la evolución natural. En cada iteración una población de soluciones dada provee un conjunto de nuevas soluciones que son evaluadas y las *más aptas*, es decir, las de mejor función objetivo, tendrán más chances de ser parte de la nueva población, que se formará en base a mutaciones y combinaciones de las actuales

4. *Swarm optimization*: son algoritmos basados en comportamientos colectivos de algunos animales, por ejemplo: colonia de hormigas, colonia de abejas, de libélulas, etc

2. Ruin & Recreate

La metaheurística Ruin & Recreate fue desarrollada en el año 1998 por Schrimpf, Schneider, Stamm-Wilbrandt y Dueck y en un principio fue aplicada a VRPTW, a problemas de optimización de redes y al viajante de comercio [SCH/99]. La misma consiste en arruinar (*ruin*) una solución (generalmente factible) para luego reconstruirla (*recreate*) de forma de obtener una mejor solución de la que se partió (ver Algoritmo 1).

En el caso de VRPTW, arruinar la solución consiste en quitar t clientes según un criterio de arruine, dejando una solución infactible, donde sólo $n - t$ clientes son visitados por los vehículos. Luego el algoritmo intenta reintroducir los t clientes a la solución rota, tratando de obtener una mejor solución de la cual se partió. El proceso se repite hasta que una condición es satisfecha, sea porque se consiguió una solución suficientemente buena, porque se cumplió un número fijo de iteraciones, porque ocurrieron suficientes iteraciones sin obtener mejoras a la solución o cualquier otra condición de corte.

El *quid* del algoritmo es, claramente, la forma en que se arruina y recrea la solución.

2.1. Ruin

Arruinar la solución es computacionalmente sencillo, aunque algorítmicamente es tan importante como recrearla. Elegir qué nodos quitar de la solución es crucial a la hora de poder generar una mejora o no. En [SCH/99] se plantean diversas variantes básicas en lo referido a ruteo de vehículos, por ejemplo:

- Espacial: dado un nodo n se quitan de la solución n y todos los nodos cuya distancia a n sea menor a un parámetro dado

- Cadena: se quita una cadena consecutiva de nodos (correspondientes a un recorrido de un vehículo) de la solución, donde el largo de la cadena es parametrizable

- Azar: cada nodo de la solución se quita según un parámetro de probabilidad

2.2. Recreate

La reconstrucción de una solución consiste en volver a incluír los nodos quitados en la etapa "ruin" de forma de conseguir una mejor solución a la original.

Algorithm 1 Ruin & Recreate para VRPTW

```
repetir
sol ←solución inicial
arruinar sol quitando n clientes
nueva_sol ←reconstruir sol
si nueva_sol es mejor que sol
    sol ←nueva_sol
hasta que se cumpla un criterio de finalización
```

Idealmente, los nodos quitados se agregarían de forma óptima a la instancia rota, pero dado que la cantidad de posibles reinserciones crece exponencialmente en función de la cantidad de clientes quitados, esto es impracticable. Esta etapa suele ser la computacionalmente más complicada, en términos de recursos utilizados por la computadora, ya que en general se quiere que la solución reconstruída sea factible, por lo que las reinserciones deben satisfacer todos los requerimientos que el problema impone. Otro enfoque es permitir soluciones intermedias no factibles, de forma tal que puedan ser arruinadas nuevamente y en la subisiguiente recreación conseguir una solución factible que quizá no era accesible de no permitir la solución intermedia.

3. La solución

Esta tesis presenta un algoritmo paralelo, distribuído, escalable, con objetivo múltiple que apunta a disminuír tanto la longitud total recorrida por los vehículos como la cantidad de vehículos necesaria para visitar todos los clientes.

3.1. Objetivo Múltiple

Para lograr mayor efectividad al resolver VRPTW multiobjetivo, se ejecutan en paralelo dos algoritmos similares pero con distintos objetivos. *RAR* (Ruin & Recreate) intentará optimizar el largo total del recorrido de la solución mientras que *RARV* (Ruin & Recreate Vehicular) se enfoca exclusivamente en reducir la cantidad de vehículos necesarios. Ambos algoritmos utilizan la metaheurística Ruin & Recreate, pero con diferente tipo de rotura (como se detalla en 3.5). Las instancias se comunican entre sí a través de un árbitro (ver 3.2)

3.2. Arquitectura

La solución planteada en esta tesis está pensada para ser ejecutada concurrentemente en varias computadoras o núcleos conectados en red. Las instancias de los algoritmos RAR y RARV se ejecutan independientemente en cada núcleo o computadora, de esta forma, por ejemplo, una computadora con 4 núcleos podría ejecutar 4 instancias concurrentemente, y 2 computadoras de 4 núcleos 8 instancias. La solución fue diseñada de forma tal que puedan agregarse computadoras (ergo núcleos) a la red trivialmente y sin límites, incluso una vez iniciada la ejecución.

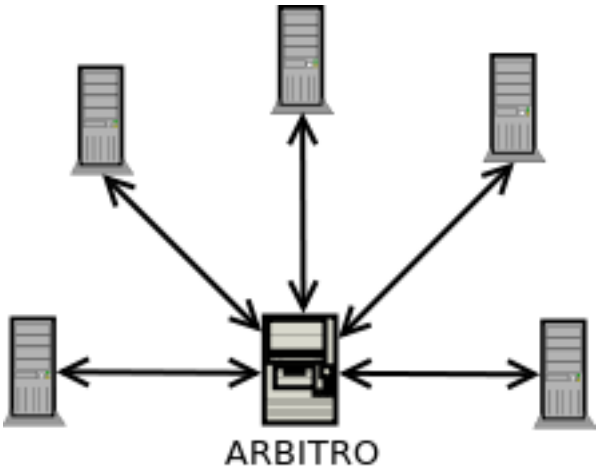


Figura 1: Diagrama de la arquitectura distribuída

Todas las instancias se conectan a un único árbitro, el cual envía parámetros de los algoritmos, el problema a resolver y la mejor solución hasta el momento a través de la red, para que cada algoritmo pueda empezar a trabajar. Enviar los parámetros de configuración y la instancia del problema facilita la ejecución cuando se cuenta con un gran número de instancias, ya que sólo hace falta modificar los parámetros del árbitro y ejecutarlo, y no hay necesidad de indicar por separado a cada instancia el problema sobre el cual se ha de trabajar o modificar parámetros de configuración de los algoritmos.

Cuando una instancia recibe el problema y la mejor solución, guarda esta última y comienza a buscar una mejor solución. Según el tipo de instancia, RAR buscará mejorar el largo del recorrido y $RARV$ buscará reducir la cantidad de vehículos necesarios. Cuando se encuentre una solución mejor a la mejor actual, se la envía al árbitro asincrónicamente y sigue buscando mejores soluciones. El árbitro se encarga de distribuír a todas las instancias cuando recibe una solución que es mejor a la actual (ver algoritmo 2)

3.2.1. Asincronía y latencia

Viendo en detalle el algoritmo 2, el árbitro chequea que la solución recibida sea mejor que la solución actual antes de guardarla como nueva mejor solución. Esto podría parecer redundante ya que todas las instancias tienen siempre la misma mejor solución, dado que el árbitro la envía a las instancias al actualizarla. Sin embargo, las instancias no detienen su procesamiento al recibir un mensaje del árbitro, y además existen latencias en la red que pueden provocar discrepancias, por lo que esa guarda es necesaria. Un ejemplo de por qué es necesaria esa condición es el siguiente:

1. El árbitro calcula la solución inicial en N
2. i_1 encuentra una mejor solución de $N-2$ y la envía
3. el árbitro la guarda como nueva mejor y redistribuye

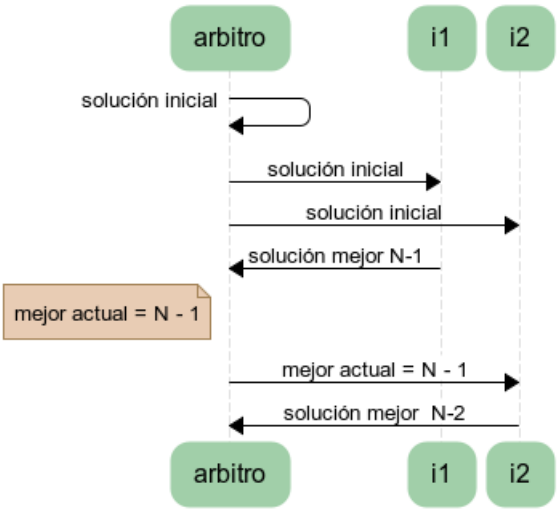


Figura 2: Ejemplo de distribución de soluciones

4. i_2 encuentra una mejor solución de $N-1$ y la envía
5. i_2 recibe la mejora de $N-2$ del árbitro

La instancia i_2 no cesa su búsqueda de mejor solución al recibir una solución del árbitro, y también puede pasar que por demoras en la red ésta llegue luego de haber encontrado la mejora del punto 4, por lo que sin la guarda mencionada se tomaría $N-1$ como mejor solución cuando i_1 había encontrado una de $N-2$, que sería descartada, como se puede ver en la Figura 2.

3.3. Solución inicial

Para poder arruinar y recrear una solución, se necesita en primera instancia una solución, factible, sobre la cual poder iterar. La solución inicial la construye el árbitro utilizando una variante de la heurística del vecino más próximo [LIN/65], la cual, partiendo del depósito, iterativamente agrega a la solución al cliente más cercano y factible (cuya demanda sea satisfacible por el vehículo y que la visita se produzca dentro de su ventana de tiempo) que aún no haya sido visitado. En caso de no haber clientes factibles para el vehículo actual, se agrega otro vehículo a la solución, agregando clientes de la misma manera, hasta que todos los clientes del problema hayan sido visitados. Se asume que la cantidad de vehículos disponible es ilimitado, por lo que siempre es posible agregar un nuevo vehículo para poder conseguir una solución inicial factible. Luego de la construcción de la solución inicial, se aplica un proceso de búsqueda local (ver 3.6).

En los ejemplos a los cuales se aplicó el algoritmo (ver la sección 4) siempre existe una solución inicial factible encontrable de esta forma. Si no se encuentra una solución inicial usando este algoritmo, se cancela la ejecución, dado que VRPTW es NP-HARD y hallar una solución (inicial) factible podría ser muy costoso.

Algorithm 2 Árbitro

```
arbitro(problema, tiempo) {
    mejor←vecino_mas_cercano(problema)
    mientras no pase tiempo {
        nueva←recibir_mejor_solucion()
        si nueva es mejor que mejor {
            mejor←nueva
            enviar nueva a las instancias
        }
    }
    enviar STOP a las instancias
}
```

3.4. Recreate

Recrear consiste en reinsertar los nodos que fueron quitados (arruinados) de la solución intentando obtener una solución que sea mejor de la que se partió. Se usó un algoritmo de *mejor inserción*, donde cada cliente quitado se intenta insertar en todas las posibles posiciones de todos los vehículos. De existir soluciones factibles tras la inserción, se elige la de menor largo y se procede al siguiente nodo hasta que no queden nodos por insertar. En el caso en que uno o más nodos sean ininsertables, es decir, no exista posición alguna en algún vehículo tal que la solución sea factible, se reintenta el procedimiento, con la solución rota original, pero reordenando los nodos rotos de forma que se intente primero agregar los ininsertables.

La idea detrás de este razonamiento es que, dados $R = v_1, v_2, \ldots, v_k \subseteq V$ nodos que hay que insertar en la solución, y $I = v_1^i, v_2^i, \ldots, v_t^i \subseteq R$ los nodos que no se pudieron insertar, es posible que un nodo v_j^i no sea posible insertar porque exista $v_p \in R \setminus I$ que anule las posibilidades de inserción de v_j^i (por capacidad de vehículos o ventana de tiempo). Al reordenar los nodos, se da lugar a la posibilidad que todos los nodos de I sean insertables y también los de $R \setminus I$, que tenían otras opciones de inserción que no bloqueaban los nodos de I pero no fueron elegidas porque fue priorizada la solución de menor largo. Este procedimiento se reintenta φ veces, donde φ es un parámetro de configuración.

Si en alguna iteración resulta que v_1^i es ininsertable (o sea el primer nodo es ininsertable) entonces se descarta esta solución rota como infactible, ya que no existe reordenamiento posible para que v_1^ipueda ser insertado.

Si la solución recreada no mejora el largo o la cantidad de vehículos (según el caso de RAR o $RARV$), entonces simplemente se descarta y se procede con la siguiente iteración de arruine.

3.5. RAR y RARV

Como ya se mencionó, RAR y $RARV$ son los algoritmos que intentan optimizar el largo total del recorrido y la cantidad de vehículos, respectivamente.

Ambos casos utilizan la metaheurística *Ruin & Recreate* [SCH/99] y en am-

Algorithm 3 recrear

```
recrear(solucionRota, clientes) {
ininsertables←{}
    por cada cliente en clientes {
        por cada vehiculo en solucionRota {

            nuevasSoluciones←obtener todas las

              soluciones factibles resultantes de
              insertar al cliente en la solucionRota
            si nuevasSoluciones es vacío

              agregar el cliente a ininsertables
            si no
              actualizar solucionRota con
              la solución de menor largo
              de nuevasSoluciones
        }
    }
    si es vacio {
        devolver solucionRota
    }
    si no {

        /* se intenta cambiar el orden de inserción */
        reintentar φ veces:

            recrear(solucionRota, ininsertables +
              (clientes - ininsertables))
    }
}
```

bos casos se recrea la solución con el mismo procedimiento, la diferencia radica en cómo se arruina la solución.

Sea $S = v_1^1, v_2^1, \ldots, v_{n_1}^1, v_1^2, v_2^2, \ldots, v_{n_2}^2, \ldots, v_1^k, v_2^k, \ldots, v_{n_k}^k$ una solución factible de un problema.

S tiene un total de k vehículos donde v^i visita n_i clientes. Por simplicidad, se omite en cada vehículo el primer y último cliente que en ambos casos es v_0, el depósito.

Como se detalló en la sección 3.4, el algoritmo de recreación nunca agrega nuevos vehículos, por lo que, exceptuando la etapa de búsqueda local (ver 3.6) que se efectúa en cada iteración, la única forma de reducir el número de vehículos necesarios sería si la rotura incluyera un vehículo completo. Es decir: dado $S \setminus R$ la solución rota, donde $R = v_1, v_2, \ldots, v_r$ son los clientes quitados de S, sólo si $vehiculos(S \setminus R) < vehiculos(S)$ se da la posibilidad que la reinserción de los R vehículos en $S \setminus R$ sea exitosa y se halle una solución todos los clientes puedan satisfechos con menos vehículos que en S.

Tomando esto en consideración, RAR y $RARV$ se diseñaron teniendo en cuenta específicamente:

- RAR no debe arruinar vehículos completos
- $RARV$ debe arruinar uno y sólo un vehículo completo

El objetivo de RAR es minimizar el largo del recorrido. Si RAR quitara un vehículo completo, estaría buscando entonces una solución con, al menos, un vehículo menos, lo cual es terreno de $RARV$, que está enfocado justamente a ese menester, con lo qué habrían más iteraciones intentando reducir vehículos y desbalanceando la configuración de la red.

Análogamente, si $RARV$ no arruina al menos un vehículo en cada iteración estaría buscando una solución con el número de vehículos actual, cosa no deseada. Por otro lado, si $RARV$ quitara más de un vehículo estaría intentando buscar una solución que, con alta probabilidad, sería infactible, a excepción de las primeras iteraciones, por lo que se limita a un y sólo un vehículo la rotura de $RARV$, para buscar soluciones factibles con un vehículo menos a la solución actual.

Para lograr los objetivos de diseño previamente mencionados, RAR aplica, independientemente del algoritmo de rotura utilizado, un procedimiento donde, en caso de haberse roto todo un vehículo, reinserta un cliente al azar de ese vehículo roto, de forma que dicho vehículo siga siendo parte de la solución y no sea quitado completamente.

Por su lado, $RARV$ quita siempre un vehículo completo al azar y luego aplica el algoritmo de rotura elegido, con su posterior procedimiento de verificación de que no se haya roto otro vehículo entero además del quitado, como se describió previamente. De esta forma, $RARV$ quita un y sólo un vehículo, concretamente el que se elige y quita al azar en primera instancia.

En esta tesis se utilizaron dos algoritmos de arruine: aleatorio y espacial. La decisión de usar uno u otro es aleatoria según un parámetro δ. En cada iteración se elige el mecanismo de rotura aleatoria con probabilidad δ, y el espacial con probabilidad $1 - \delta$. En particular, se utilizó $\delta = 3/4$.

Algorithm 4 RAR/RARV

```
RAR(mejor, vehicular?) {
    mientras no se reciba STOP:
        nueva←recibir_mejor_solucion()
        si nueva es mejor que mejor {
            mejor←nueva
        }
        si vehicular? ⇒ r←arruinarVehiculo(mejor)
        si no ⇒ r←arruinar(mejor)
        r←recrear(mejor - r, r)
        busqueda_local(r)
        si r es mejor que mejor {
            mejor←r
            enviar mejor al árbitro
        }
}
```

3.5.1. Arruine aleatorio

El arruine aleatorio, como su nombre lo indica, quita clientes de la solución aleatoriamente. Indistintamente del vehículo en que se encuentre, cada $v_j^i \in S$ es removido de S con probabilidad π.

Este algoritmo fue el primero en probarse en la etapa de prototipado, y si bien su simplicidad puede parecer excesiva y que los resultados dependan puramente del azar, al combinarlo con la búsqueda local (ver 3.6) y el proceso de inserción (ver 3.4) los resultados obtenidos fueron muy buenos (ver 4).

La elección de π influye directamente en al factibilidad de las soluciones a obtener. Si π fuese muy pequeña, se quitarían pocos vehículos de la solución, con lo cual sería difícil o improbable poder obtener nuevas soluciones distintas a la inicial, dado que los pocos clientes quitados tendrían, en general, pocas alternativas de reinserción. Por otro lado, si π fuese muy grande, se estarían quitando de la solución muchos clientes, lo cual dificultaría la obtención de soluciones factibles en el proceso de reinserción.

3.5.2. Arruine espacial

El arruine espacial intenta explotar la cercanía física de los clientes entre sí. La idea es que si se quitan clientes que están cerca entre ellos, será más fácil agregarlos a un vehículo que pueda aprovechar su ubicación y visitar a todos o varios de ellos en su viaje. Si bien esto no siempre es posible debido a capacidades de carga y ventanas de tiempo, en ocasiones es preferible arruinar a clientes que están cerca a hacer un arruine aleatorio. Por otro lado, el arruine espacial no tiene en cuenta ventanas de tiempo, sólo posiciones geográficas, por lo que usar *sólo* arruine espacial no da tan buenos resultados, motivo por el cual se lo combina con otras formas de arruine.

Conceptualmente, se elige un cliente al azar y se lo quita de la solución

junto con todos los clientes que estén dentro de un radio determinado de dicho cliente. La elección del radio es importante: un radio muy chico implicaría que se quitarán muy pocos clientes como para lograr una mejora, y un radio muy grande quitará demasiados, tendiendo a la infactibilidad de la solución.

La distancia de arruine que se utiliza en este trabajo se elije de forma aleatoria entre $d/2$ y $2/3 \cdot d$ donde d es la distancia promedio entre todos los clientes del problema.

3.6. Búsqueda local

La búsqueda local es un proceso mediante el cual se intenta, partiendo de una solución, variaciones y combinaciones de clientes y rutas dentro de un vecindario en pos de obtener una mejor solución. Cuanto mayor sea el entorno donde se realiza la búsqueda, más tiempo tomará el procedimiento de búsqueda local. En un extremo, la búsqueda local buscaría en todas las posibles combinaciones de clientes y vehículos, obteniendo fehacientemente la mejor solución posible, pero dado que esto toma tiempo exponencial según el tamaño del problema, se limita la búsqueda a entornos reducidos.

Para realizar la búsqueda local se aplican distintos *operadores* que alteran la solución actual, si la solución alterna es mejor a la actual se guarda y se repite el proceso hasta que no se obtenga mejora, de ahí el nombre "búsqueda local".

Los operadores se dividen en dos grandes grupos:

- uniruta: realizan cambios en un sólo vehículo
- multiruta: realizan cambios entre dos o más vehículos

Dentro de estos grupos se utilizaron los siguientes operadores:

- Uniruta
  - reverse(n)
  - relocate(n)
- Multiruta
  - tailExchange
  - relocate(n, m)

3.6.1. reverse(n)

El operador reverse toma un vehículo (una ruta) e invierte el sentido de una cadena de clientes, como se ilustra en la Figura 3. El valor n indica el tamaño de la cadena de clientes a invertir.

Dado n, por cada vehículo de la solución, se toman todas las cadenas de clientes de largo n y se comprueba si la solución consistente en invertir esa cadena es factible y de menor largo que la original.

En general, debido a las ventanas de tiempo, este operador no suele ser útil con valores grandes de n, pero al ser computacionalmente poco costoso (es O(n) sobre la cantidad de clientes), se aplicó este operador para $n = 2, 3, 4$

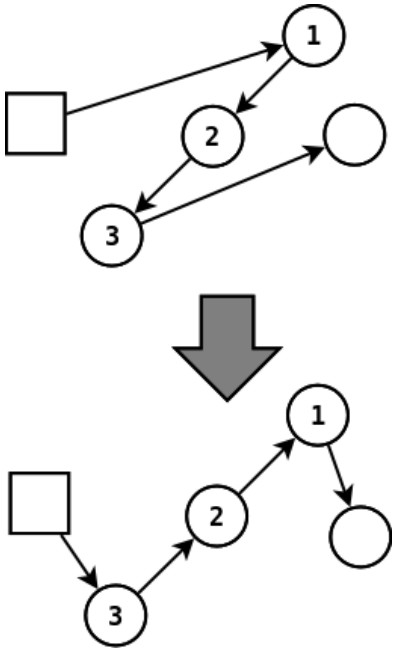


Figura 3: operador reverse(n)

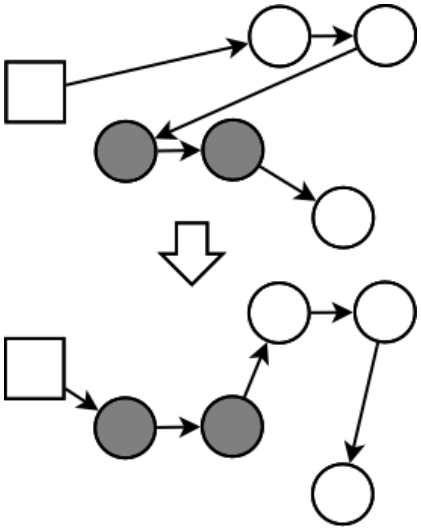


Figura 4: operador relocate(n)

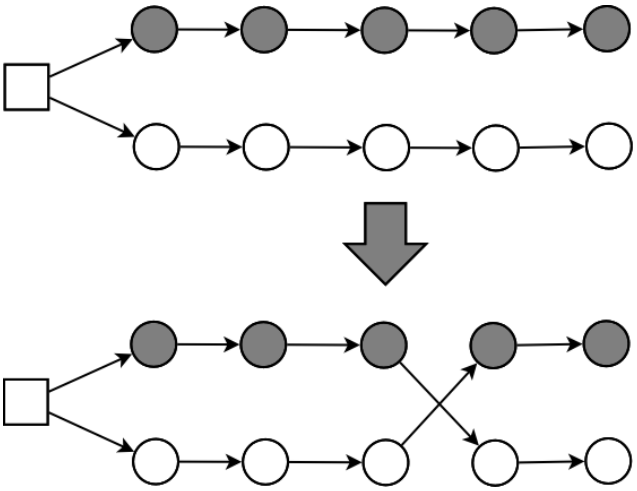


Figura 5: operador tailExchange

3.6.2. relocate(n)

El operador relocate(n) toma una cadena consecutiva de clientes de largo n dentro de un vehículo, e intenta reubicarla en otra posición dentro del mismo vehículo, como ilustra la Figura 4. Se aplicó este operador con $n = 1, 2, 3, 4$

3.6.3. tailExchange

tailExchange intercambia las colas de las rutas entre un par de vehículos, como muestra la Figura 5. Este operador es computacionalmente costoso ya que se aplica para todo par de rutas posible. Sea n el número total de vehículos, cada par de rutas implica $\binom{n}{2} = \frac{n\,(n-1)}{2} = O\,(n^2)$. Por cada par de rutas $v_i = \{v_1^i, v_2^i, \ldots, v_k^i\}, v_j = \{v_1^j, v_2^j, \ldots, v_q^j\}$ se calculan todos los posibles cambios de colas de todos los tamaños, incluyendo 0.

$$\forall x \in \{0 \ldots k\}, y \in \{0 \ldots q\}$$

$$v_i = \{v_1^i, v_2^i, \ldots, v_{k-x}^i, v_{q-y+1}^j, v_{q-y+2}^j, \ldots, v_q^j\}$$

$$v_j = \{v_1^j, v_2^j, \ldots, v_{q-y}^j, v_{k-x+1}^i, v_{k-x+2}^j, \ldots, v_q^j\}$$

En el caso particular donde se haga un intercambio de una cola de tamaño 0 contra una cola del tamaño total del vehículo y el resultado sea factible, se estaría reduciendo un vehículo, ya que el vehículo cuya cola de clientes era de hecho toda la lista de clientes y fue adosada exitosamente en el otro vehículo, sería ahora un vehículo vacío, sin clientes que visitar, y todos sus clientes serían visitados por el otro vehículo, como lo ilustra la Figura 6

3.6.4. relocate(n, m)

relocate(n, m) es el análogo multiruta de la versión *relocate(n)* uniruta. Al aplicar este operador, se intenta intercambios de cadenas de n clientes desde una

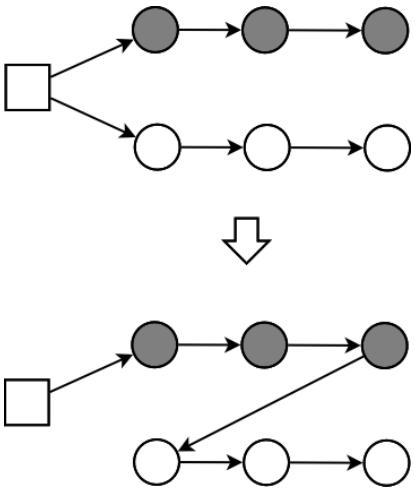


Figura 6: Reducción de un vehículo utilizando *tailExchange*

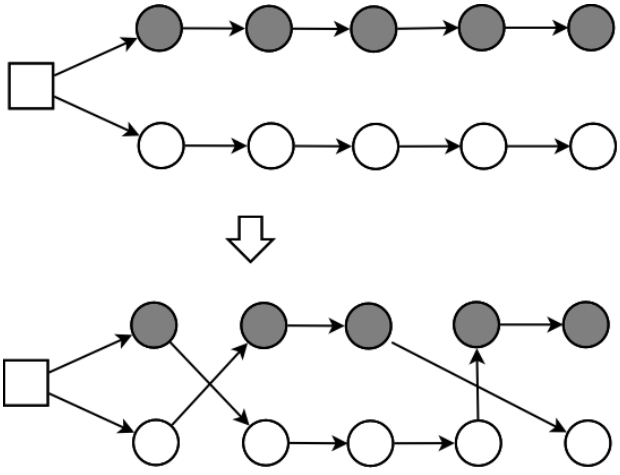


Figura 7: relocate(2,3)

ruta por cadenas de m clientes de la otra ruta, como se ilustra en la Figura 7. El operador intercambia las cadenas de clientes desde y hacia todas las posibles ubicaciones en la ruta, por lo que tiene $O(n^4)$.

relocate(n, m) podría aplicarse, dado un par de rutas de tamaños k y q, en todo par de valores $\{0 \dots k\}, \{0 \dots q\}$, pero en la práctica, dadas las limitaciones de carga y sobre todo las ventanas de tiempo, intercambiar rutas de muchos vehículos sería en general infactible, por lo que se sacrificaría importante tiempo de procesamiento. Este operador se aplicó para $n, m = 0, 1, 2, 3$.

3.6.5. Elección de mejoras

Independientemente del orden de aplicación de los operadores, un punto importante a determinar es cómo elegir las mejoras a aplicar a la solución. Aplicar una mejora provista por un operador podría impedir otra posible mejora a posteriori dada por otro operador que mejoraría aún más que si no se aplicaba la mejora inicial.

En un entorno ideal, la mejor opción sería armar un árbol con todas las posibles aplicaciones de mejoras de los distintos operadores, luego ubicar la hoja con la mejor solución y aplicar los operadores y elegir mejoras según el camino desde la raíz hasta la mejor solución. Este método es claramente impracticable. Otro opción más práctica es realizar esta misma operación pero limitar la profundidad del árbol, de forma de tener una visión de determinada cantidad de aplicaciones previsibles, lo cual no garantiza en absoluto que el mejor camino en este arbol acotado sea el camino óptimo a tomar. En esta tesis se omitió esta alternativa y directamente las mejoras se aplican en forma golosa: al encontrar un operador una solución mejor a la actual, inmediatemente se la guarda como nueva mejor.

3.7. Implementación

La implementación de la solución se realizó usando el lenguaje Scala[SCA], un lenguaje de programación relativamente nuevo (fue creado en 2001). La elección de este lenguaje se basó principalmente en sus cualidades de mezcla de paradigmas orientado a objetos y funcional, lo que permitió una alta expresividad con código conciso, y su librería de *actores remotos*, lo cual fue idóneo para la implementación de las comunicaciones entre los nodos y el árbitro. Esta combinación de factores permitió que todo el código de la tesis sea compacto: poco menos de 1000 líneas de código[7].

4. Resultados computacionales

En esta sección se presentan los resultados obtenidos al aplicar el algoritmo a distintas instancias de VRPTW. Las instancias utilizadas son las introducidas por Solomon [SOL/87] en 1987. Estas instancias se dividen en diversos grupos, según las ubicaciones de los clientes. En primer lugar se dividen en dos grandes grupos: el primer grupo (tipo 1) tiene ventanas pequeñas que sólo permite unos pocos vehículos por ruta; el segundo grupo (tipo 2) tiene ventanas mucho más amplias que permiten varios vehículos en una misma ruta. Por otro lado, hay 3

[7] 967 para ser exactos

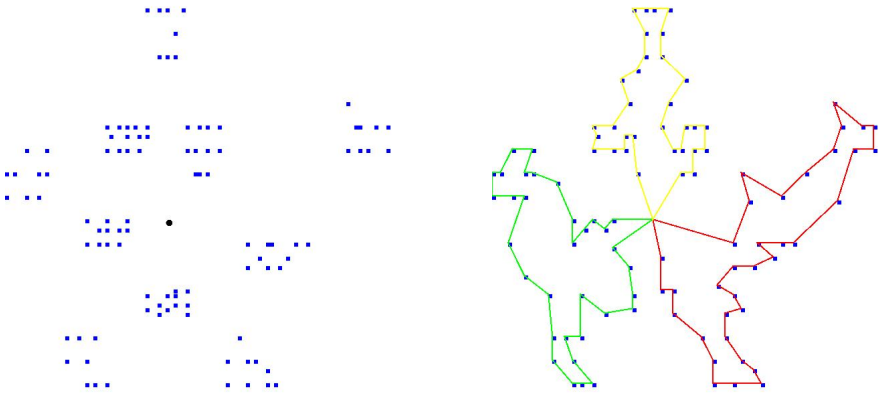


Figura 8: Instancia y solución de tipo C1

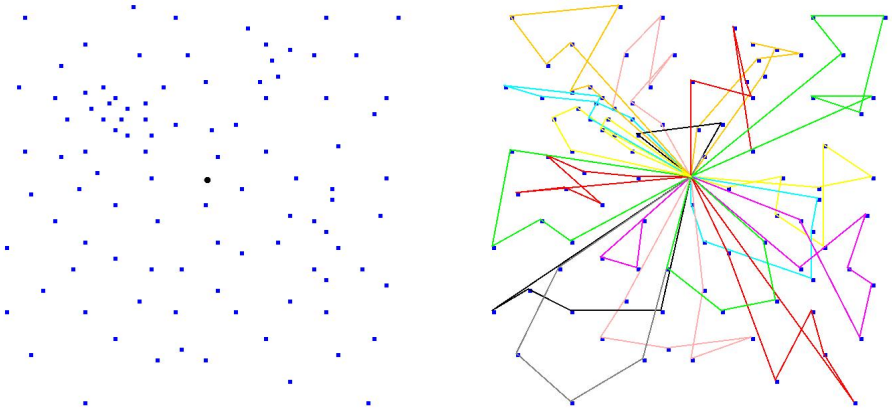


Figura 9: Instancia y solución de tipo R1

tipos de instancias, llamadas R, C y RC. El tipo R (del inglés *random*, aleatorio) posee todos los clientes esparcidos en forma aleatoria, en el tipo C (*clustered*, agrupados) los clientes se agrupan en conglomerados y por último el tipo RC mezcla clientes al azar con clientes agrupados.

4.1. Parámetros

Los parámetros de configuración del algoritmo, dada una instancia, son los siguientes:

- *núcleos*: si bien no es un parámetro *per se* que el algoritmo tome para su funcionamiento, la cantidad de computadoras (o núcleos, en computadoras multinúcleo) que resuelven el problema en paralelo es un parámetro a contemplar en la observación y análisis de los resultados
- t: tiempo de ejecución del algoritmo, en minutos

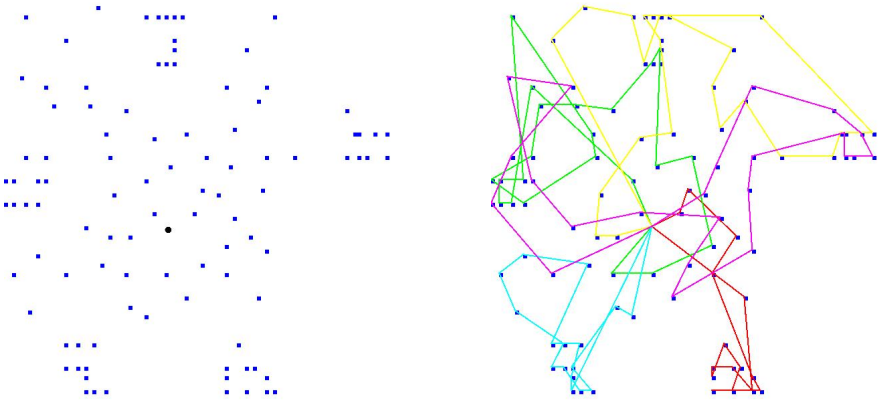


Figura 10: Instancia y solución de tipo RC2

- π: rotura de las instancias RAR
- $lsRAR$: si se ha de realizar búsqueda local luego de ejecutar RAR
- πv: rotura de las instancias $RARV$
- $lsRARv$: si se ha de realizar búsqueda local luego de ejecutar $RARV$
- δ: probabilidad de ejecutar rotura aleatoria o espacial en cada iteración

En un principio π *y* πv eran variables con valores entre 0 y 1, indicando probabilidad de rotura. Así, $\pi = 0{,}5$ por ejemplo indicaba que, estadísticamente, se quitaría de la solución la mitad de los clientes. En la etapa de calibración (ver 4.2) se comprobó que era más efectivo poder indicar un número de clientes a quitar de la solución, pero sin que sea un número exacto sino que pueda fluctuar aleatoriamente, por lo que internamente se mantuvieron π *y* πv como probabilidad de rotura, pero calculados tomando como parámetro la cantidad de clientes a quitar, según las fórmulas:

$$\pi = 1 - \left(\frac{clientes}{rotura}\right)^{-1}$$

$$\pi v = 1 - \left(\frac{clientes - \frac{clientes}{vehiculos}}{roturaV}\right)^{-1}$$

Donde *rotura* y *roturaV* indican la cantidad de clientes a quitar, según la instancia, *clientes* la cantidad de clientes del problema y *vehiculos* la cantidad de vehículos de la solución actual. En el caso de πv, éste es actualizado cada vez que se encuentra una solución que reduzca el número de vehículos, de forma de mantener, probabilísticamente, la cantidad de clientes a quitar en cada iteración.

4.2. Calibración

Se eligieron 5 problemas sobre los cuales realizar la calibración de los parámetros, sobre los que se basan el resto de las pruebas. Los problemas se eligieron

	Largo (NN)	Vehículos (NN)	Largo (mejor)	Vehículos (mejor)
R104	1188	14	1007	9
RC105	1103	5	1191	3
R1_2_2	4328	23	4040	18
R1_4_1	11258	43	11084	38
R1_6_1	23373	65	21131	59
R1_10_1	58356	103	58356	100

Cuadro 1: Instancias de calibración

en base a la diferencia en el número de vehículos provisto por la solución inicial (la heurística del vecino más cercano) contra la mejor solución conocida hasta el momento en primera instancia, y la diferencia en el largo total de ambas en segunda instancia (siguiendo el orden jerárquico que propone el algoritmo). La idea de esta elección es ver la capacidad de la solución propuesta de reducir la cantidad de vehículos necesarios, y en segunda medida el largo. De este proceso se tomaron las siguientes instancias para usar de base de calibración (ver detalle en Cuadro 1):

- R104 (100 clientes)
- RC105 (100 clientes)
- R1_2_2 (200 clientes)
- R1_4_1 (400 clientes)
- R1_6_1 (600 clientes)
- R1_10_1 (1000 clientes)

4.2.1. Repetibilidad

Como primera medida se evaluó la repetibilidad del algoritmo, esto es, cuán diferentes resultados producen diversas ejecuciones del mismo, manteniendo los parámetros. Si la repetibilidad fuera buena, o sea se obtuvieran resultados similares en distintas ejecuciones, no haría falta repetir ejecuciones en posteriores pruebas, ya que se podría asumir que se obtendrían, con alta probabilidad, resultados similares.

Para realizar esta prueba, se realizaron 10 ejecuciones de cada instancia, con parámetros:

$$\delta = 1/2 \qquad t = 2\,minutos \qquad cores = 2$$

$$\pi = 1/2 \qquad lsRAR = true$$

$$\pi v = 1/2 \qquad lsRARV = false$$

los resultados pueden verse en el Cuadro 2.

	Largo promedio	Veh. promedio	Desvío est. largo	Desvío est. veh.
R104	993.4	10	9.51 (0.957 %)	0 (0 %)
RC105	1552.31	14	10.26 (0.661 %)	0 (0 %)
R1_2_2	4468.16	18	58.85 (1.317 %)	0 (0 %)
R1_4_1	11030.9	40	64.41 (0.584 %)	0 (0 %)
R1_6_1	23335.15	59	210.59 (0.902 %)	0 (0 %)
R1_10_1	69236.42	100	894.54 (1.292 %)	0 (0 %)

Cuadro 2: Resultados de calibración

4.2.2. δ

δ es el parámetro que indica la probabilidad, en cada iteración, de utilizar rotura aleatoria; caso contrario se utiliza rotura espacial. Esto es:

$$\delta = 0 \Rightarrow \textit{siempre se utiliza rotura espacial}$$

$$\delta = 1 \Rightarrow \textit{siempre se utiliza rotura aleatoria}$$

En las pruebas realizadas, $\delta = 0$ mostró una muy mala performance, en ninguno de los casos logró reducir vehículos (ver Figura 11).

El otro extremo, $\delta = 1$ dió en general buenos resultados, pero en algunos casos, particularmente instancias de tipo RC, la rotura espacial suele aportar mejoras, por lo que se terminó utilizando $\delta = 3/4 = 0{,}75$, o sea que 1 de cada 4 iteraciones, en promedio, aplica rotura espacial y el resto rotura aleatoria.

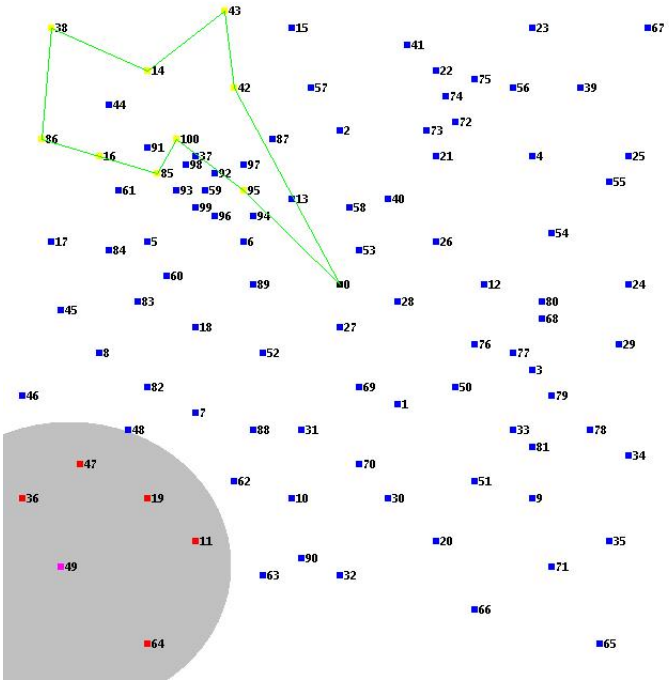
4.2.3. π y πv

π y πv determinan cuántos clientes se han de remover de la solución para luego intentar reconstruírla. Como se detalla en 4.1, si bien internamente ambos parámetros se utilizan como porcentajes, el programa recibe de parámetros los números de clientes aproximados a quitar, por lo que en esta sección se consideran π y πv no como porcentajes sino como números enteros de vehículos a quitar, aproximadamente, de la solución.

La elección de π y πv es la más importante de todos los parámetros. Si el valor fuese muy pequeño, se quitarían muy pocos vehículos de la solución, lo cual impediría generar buenas combinaciones para reinsertarlos, tendiendo a reconstruir la misma solución de la que se partió en la mayoría de los casos. Si, por el contrario, el valor fuera muy grande, quitar demasiados vehículos también es contraproducente ya que se utiliza mucho tiempo calculando las posibilidades de reinserción de cada uno de ellos, y dado que el algoritmo de reinserción es goloso (ver 3.4) no necesariamente quitar más nodos da una mejor solución, independientemente del tiempo tomado en reconstruirse.

Para calibrar πv se realizaron ejecuciones de cada instancia de calibración variando el valor de πv entre el 10 % y el 90 % de la cantidad de vehículos, durante 2 minutos, con 2 núcleos, ambos ejecutando *RARV*, y se tomó el tiempo en que se llegó al menor número de vehículos. Pueden verse gráficos de resultados en las Figuras 12 y 13.

En los casos en que en el tiempo establecido no se haya logrado el mejor número de vehículos (que corresponde al mejor número de vehículos logrado



En verde el trayecto del vehículo arruinado, el disco gris marca la zona arruinada espacialmente, con centro magenta y resto de clientes quitados en rojo

Figura 11: RARV con rotura espacial

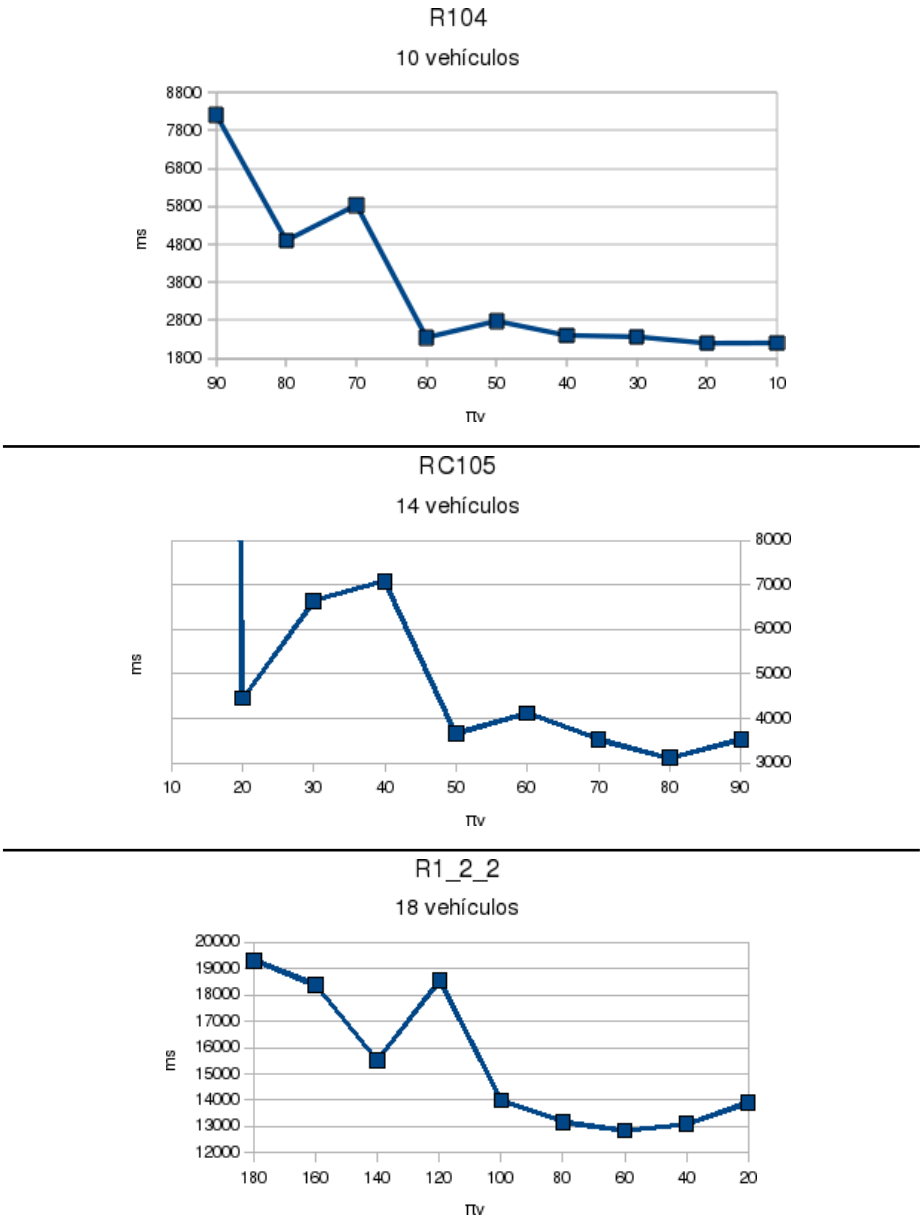


Figura 12: Calibración de πv

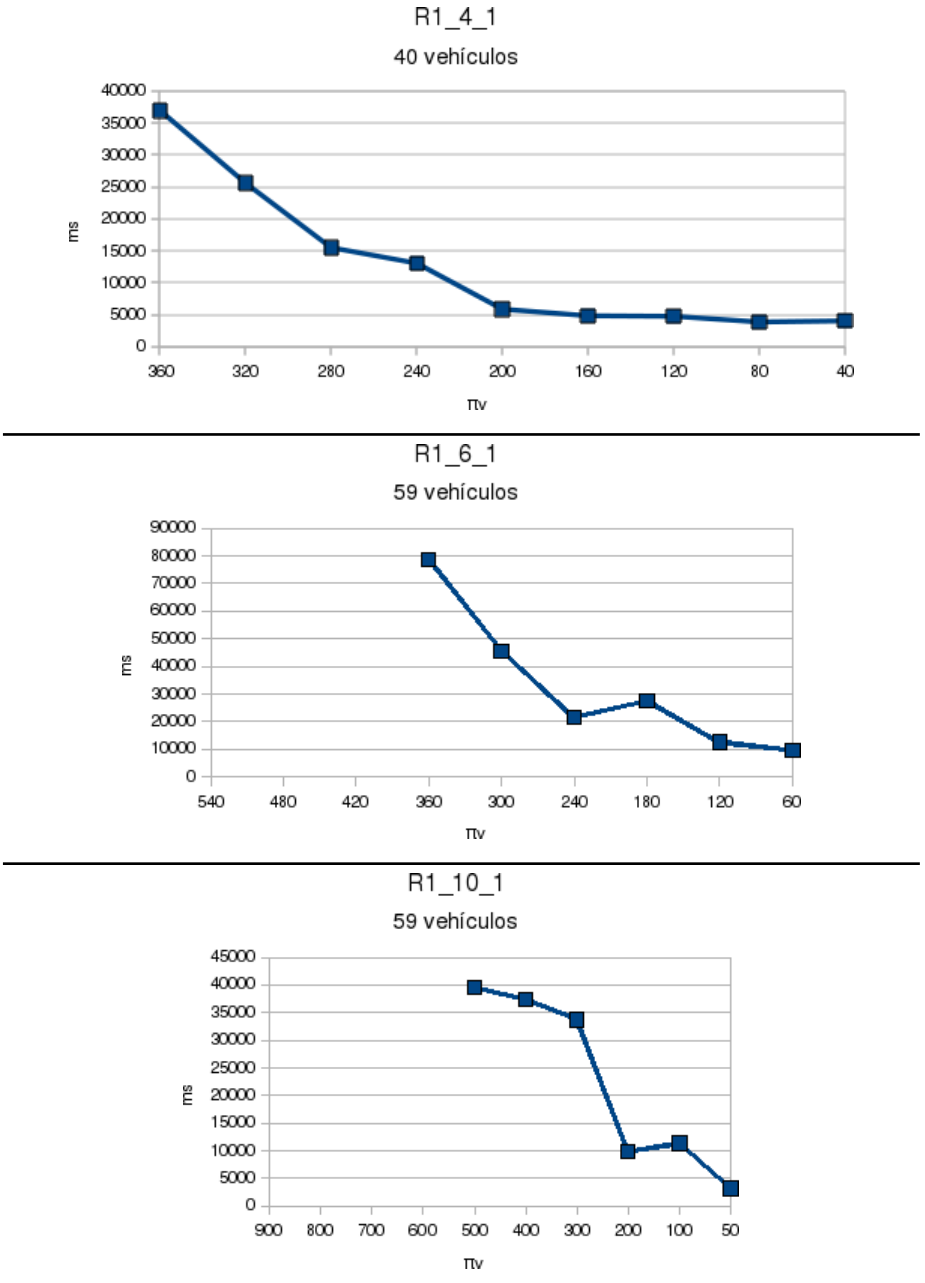


Figura 13: Calibración de πv (continuado)

por el algoritmo en estas ejecuciones, no necesariamente es el mejor conocido), no se proveen datos.

Como se puede ver en los gráficos de las figuras 12 y 13, y quizá contrariamente a lo esperado, la cantidad de vehículos correspondientes a πv que obtiene mejores resultados no varía significativamente según la cantidad de clientes de la instancia, sino que es más bien constante. En los casos de las instancias de 100 clientes, donde se trabajó en primera medida, $\pi v = 50$ era lo que daba mejores resultados. Extrapolando estos datos, se usó $\pi v = 100$ para las instancias de 200 vehículos, asumiendo que romper la mitad de los clientes daría los mejores resultados, pero la etapa de calibración demostró que $\pi v = 50$ aproximadamente, fijo, sin importar la cantidad de clientes de la instancia es el valor que mejores resultados arroja.

Análogamente se procedió con π, y se concluyó, experimentalmente, que $\pi = 50$ arrojaba los mejores resultados.

4.3. Planes de ejecuciones

Como el objetivo de la tesis no es sólo comprobar la viabilidad del algoritmo sino tambien su paralelismo, se realizaron distintas ejecuciones para cada problema de los planteados. La idea es comprobar la escalabilidad de la solución: si al agregar nodos se obtienen mejores resultados. A los hechos, se realizaron 3 tipos de ejecuciones de cada problema

- 1 minuto en una computadora *dual core*, ejecutando *RAR* y *RARV* concurrentemente
- 5 minutos en la misma computadora *dual core*
- 5 minutos en 6 computadoras *quad core*, ejecutando 12 instancias de *RAR* y 12 de *RARV* concurrentemente

Se eligieron estos valores de tiempos para poder analizar la mejora obtenida en cada caso entre 1 y 5 minutos, ya que luego de 5 minutos de ejecución la cantidad de mejoras obtenidas disminuye drásticamente.

En ambos casos el árbitro se ejecutó *in situ* en alguna computadora que a la vez estaba utilizando todos sus núcleos en *RAR* o *RARV*, pero dado que el árbitro prácticamente no consume recursos del procesador (simplemente controla los valores recibidos y los reenvía), se considera que no afecta el rendimiento de la computadora en la cual se ejecuta.

Técnicamente, la computadora *dual core* es una AMD Athlon 64 X2 Dual Core Processor 6000+ @ 3GHz con 2 GB de memoria RAM, mientras que las *quad core* son Intel Core 2 Quad CPU Q9550 @ 2.83GHz con 4 GB de RAM.

La nomenclatura usada en las tablas de las secciones siguientes es la aquí descripta:

- en las columnas se indica el tipo de instancia (C1, C2, etc.) y los valores en cada fila corresponden al promedio obtenido para todas las instancias del tipo
- VMS y LMS se refiere a número de vehículos y largo total del recorrido, respectivamente, de la mejor solución conocida, tomando los valores de [TOP]

25 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	3	2	5	2.73	3.25	2.88
LMS	190.59	214.45	463.37	382.15	350.24	319.28
VNN	3	1.63	5.25	1.64	3.38	1.88
LNN	197.96	241.64	485.78	441.9	355.46	370.87
V 1' 2 cores	3	1,25	4.67	1.27	3.25	1.25
L 1' 2 cores	190.87	245.44	476	429.46	351.1	423.58

Cuadro 3: Resultados 25 clientes. $\pi = 12$, $\pi v = 12$, $\delta = 0{,}75$

- VNN es la cantidad de vehículos de la solución inicial, luego de haber aplicado búsqueda local
- LNN es el largo total del recorrido de la solución inicial, luego de haber aplicado búsqueda local
- V x' y cores es la cantidad de vehículos de la solución luego de aplicar el algoritmo durante x minutos, utilizando y núcleos de procesamiento
- L x' y cores es el largo total del recorrido de la solución luego de aplicar el algoritmo durante x minutos, utilizando y núcleos de procesamiento

4.4. 25 y 50 clientes

Las instancias de 25 y 50 clientes de Solomon corresponden a tomar los primeros 25 y 50, respectivamente, de los problemas de 100 clientes. El problema al comparar los resultados obtenidos en estos problemas reside a que no se encontró información sobre los mejores resultados conocidos calculados por heurísticas, quizá porque sean muy fáciles de resolver.

Todos estos problemas fueron resueltos óptimamente [SPR], por lo que no tiene mayor sentido utilizar heurísticas más allá de comparación o calibración de la misma.

El *quid* del asunto es que la resolución exacta obtenida por [SPR], contra las cuales se comparan los resultados, no usa un esquema jerárquico donde la cantidad de vehículos tiene mayor prioridad al largo total, sino que optimiza únicamente el largo total del recorrido, sin importar la cantidad de vehículos necesarios. Por otro lado, dichos resultados utilizan precisión aritmética limitada a un decimal, por lo que los casos donde el número de vehículos es igual tanto en el óptimo como en la solución producida por este trabajo, se nota una pequeña diferencia en el largo total, que presumiblemente sea producto de esta diferencia de precisiones.

Como puede verse en el Cuadro 3, la cantidad de vehículos fue reducida en la mayoría de los casos, por lo que en esos problemas el largo total no es directamente comparable. En los casos donde los vehículos no se redujeron (por ejemplo en todas las instancias de C1), la diferencia de largo fue menor al 0.29 %, diferencia que se asume es a causa de la precisión previamente mencionada.

50 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	5	2.75	7.75	4.11	6.5	4.43
LMS	361.69	357.5	766.13	634.03	730.31	585.24
VNN	5	2.13	8.67	2.27	7.63	2.86
LNN	381.5	414.31	813.53	706.32	798.69	701.48
V 1' 2 cores	5	2	7.33	2	6.5	2.14
L 1' 2 cores	362.52	403.86	798.31	674.98	731.83	763.85

Cuadro 4: Resultados 50 clientes. $\pi = 25$, $\pi v = 25$, $\delta = 0{,}75$

100 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	10	3	11.92	2.73	11.5	3.25
LMS	828.38	589.86	1209.89	951.02	1384.16	1119.17
VNN	10.11	3.38	14.67	3.64	13.88	4.25
LNN	852.41	608.88	1278.05	989.74	1458.55	1174.93
VRR	10	3	12.08	2.82	11.88	3.38
LRR	828.38	589.86	1211.53	958.05	1377.39	1097.63
V 1' 2 cores	10	3	12.42	2.91	12.13	3.38
L 1' 2 cores	828.38	589.86	1207.21	966.16	1373.36	1132.66
V 5' 2 cores	10	3	12.42	2.91	11.75	3.25
L 5' 2 cores	828.38	589.86	1195.79	950.04	1352.61	1149.91
V 5' 24 cores	10	3	12.25	2.91	11.63	3.25
L 5' 24 cores	828.38	589.86	1201.72	938.56	1381.1	1125.13

Cuadro 5: Resultados 100 clientes. $\pi = 50$, $\pi v = 50$, $\delta = 0{,}75$

Similar a lo acaecido con las instancias de 25, los problemas de 50 vehículos presentaron prácticamente los mismos resultados: reducción de vehículos y en el caso que se mantuvieron iguales, una diferencia muy pequeña que puede aducirse a errores de precisión (ver Cuadro 4).

Una notable excepción es el problema R106.50, donde no se pudo alcanzar el mínimo de vehículos de la mejor solución

R106.50	
VMS	**5**
LMS	**793**
V 1' 2 cores	7
L 1' 2 cores	857.98

4.5. 100 clientes

Los resultados de los problemas de 100 clientes pueden visualizarse en la Tabla 5. Se agregan además dos filas: VRR y LRR, que hacen referencia a la cantidad de vehículos y el largo total del recorrido de la mejor solución obtenida por [SCH/99].

Instancia	#veh RR	largo RR	#veh	largo	dif veh.	dif largo
R101	19	1650,8	19	1650,8	0	0 %
R102	17	1486,12	17	1486,12	0	0 %
R103	13	1296,19	13	1292,68	0	**-0,27 %**
R104	10	981,23	10	986,21	0	+0,51 %
R105	14	1377,11	14	1377,11	0	0 %
R106	12	1252,03	12	1252,03	0	0 %
R107	10	1119,93	10	1117,05	0	**-0,26 %**
R108	9	966,4	9	967,1	0	**-1,48 %**
R109	11	1210,66	11	1245,22	0	+2,85 %
R110	10	1121,46	11	1080,64	+1	-1,28 %
R111	10	1122,76	10	1096,73	0	**-2,32 %**
R112	10	953,63	10	953,63	0	0 %

Cuadro 6: Comparación con [SCH/99] para problemas de tipo R1

Instancia	#veh RR	largo RR	#veh	largo	dif veh.	dif largo
R201	4	1265,74	4	1253,02	0	**-1,00 %**
R202	3	1195,3	3	1191,7	0	**-0,30 %**
R203	3	947,63	3	948,83	0	+0,13 %
R204	2	848,91	2	835,27	0	**-1,61 %**
R205	3	1053,37	3	994,43	0	**-5,60 %**
R206	3	906,14	3	909,4	0	+0,36 %
R207	3	811,51	3	814,99	0	+0,43 %
R208	2	726,82	2	727,69	0	+0,12 %
R209	3	915,16	3	914,13	0	**-0,11 %**
R210	3	963,67	3	962,65	0	**-2,06 %**
R211	2	904,32	3	773,91	+1	-14,42 %

Cuadro 7: Comparación con [SCH/99] para problemas de tipo R2

Instancia	#veh RR	largo RR	#veh	largo	dif veh.	dif largo
RC101	15	1623,58	14	1696,95	**-1**	+3,55 %
RC102	13	1477,54	12	1554,75	**-1**	+1,62 %
RC103	11	1262,02	11	1262,02	0	0 %
RC104	10	1135,83	10	1135,52	0	**-0,03 %**
RC105	13	1733,56	13	1640,57	0	**-10,68 %**
RC106	12	1384,92	12	1237,26	0	**-10,66 %**
RC107	11	1230,95	11	1232,26	0	+0,11 %
RC108	10	1170,7	10	1139,82	0	**-2,64 %**

Cuadro 8: Comparación con [SCH/99] para problemas de tipo RC1

Instancia	#veh RR	largo RR	#veh	largo	dif veh.	dif largo
RC201	4	1415,33	4	1413,51	0	**-0,13 %**
RC202	4	1162,8	3	1365,65	**-1**	+17,44 %
RC203	3	1051,82	3	1052,01	0	0,02 %
RC204	3	798,46	3	799,57	0	0,14 %
RC205	4	1302,02	4	1297,65	0	**-0,34 %**
RC206	3	1152,03	3	1153,69	0	0,14 %
RC207	3	1068,86	3	1071,41	0	0,24 %
RC208	3	829,69	3	834,27	0	0,55 %

Cuadro 9: Comparación con [SCH/99] para problemas de tipo RC2

En las tablas 6, 7, 8 y 9 puede verse en detalle la comparación de los resultados individuales obtenidos por el algoritmo contra los resultados obtenidos por Schrimpf *et al* en [SCH/99] para las instancias de Solomon de tipo R1, R2, RC1 y RC2, respectivamente. Se resaltan en negrita los resultados que fueron mejorados. Se omitieron las comparaciones para las instancias de tipo C1 y C2 dado que los resultados obtenidos por ambos algoritmos fueron idénticos en todos los casos. Como referencia, Schrimpf obtuvo sus resultados tras aproximadamente 30 minutos de iteraciones en una computadora RS 6000 modelo 43P, 233 MHz [SCH/99].

4.6. 200 y 400 y 600 clientes

Los resultados para las instancias de 200, 400 y 600 clientes pueden verse en las tablas 10, 11 y 12 respectivamente. Al no haber resultados en [SCH/99] para esta cantidad de vehículos, se compara directamente contra los mejores resultados conocidos, obtenidos de [TOP].

En el caso de 600 clientes no se utilizó búsqueda local para RARV, dado que para esa cantidad de clientes realizar una búsqueda local es muy costosa

200 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	18,8	6	18	4	18	4,3
LMS	2713,28	1831,60	3636,20	2929,46	3178,91	2541,3
VNN	20	7,1	20,3	5	19,6	5,8
LNN	2774,65	1976,48	3956,22	3174,81	3499,39	2681,14
V 1' 2 cores	19	6	18,2	4,1	18,3	4,5
L 1' 2 cores	2795,95	1891,63	3916,21	3147,63	3507,53	2750,86
V 5' 2 cores	19	6	18,2	4,1	18,1	4,4
L 5' 2 cores	2749,00	1866,86	3800,47	3053,60	3497,99	2651,80
V 5' 24 cores	18,9	6	18,2	4,1	18,1	4,4
L 5' 24 cores	2738,62	1845,74	3756,41	2982,90	3338,50	2584,50

Cuadro 10: Resultados 200 clientes. $\pi = 50$, $\pi v = 50$, $\delta = 0{,}75$

400 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	37,6	11,7	36,2	8	36	8,5
LMS	7172,5	3949,82	8465,22	6188,50	7927,53	5265,19
VNN	40,5	14,2	39,1	9,7	38	10,9
LNN	7476,97	4298,83	9378,55	6809,66	8888,36	5739,15
V 1' 2 cores	39	12,6	36,6	8,1	37	9,7
L 1' 2 cores	7467,44	4830,05	9684,89	7540,50	8766,35	6102,25
V 5' 2 cores	38,4	12,2	36,5	8,1	36,5	9,7
L 5' 2 cores	7381,82	4104,13	9411,13	6789,46	8646,31	5596,96
V 5' 24 cores	38,2	12	36,4	8	36,4	9,1
L 5' 24 cores	7359,57	3988,39	9077,13	6677,42	8527,79	5608,56

Cuadro 11: Resultados 400 clientes. $\pi = 50$, $\pi v = 50$, $\delta = 0{,}75$

600 CLIENTES	C1	C2	R1	R2	RC1	RC2
VMS	57,4	17,5	54,5	11	55	11,6
LMS	14056,86	7595,61	18143,94	12483,66	16215,32	10704,70
VNN	61,1	20,9	58	14,1	57,8	15,2
LNN	15007,19	8368,93	20383,19	13766,77	18355,50	11532,44
V 5' 2 cores	58,6	18,6	55,4	11,1	55,7	12,8
L 5' 2 cores	14898,31	8612,08	19939,69	15357,56	18250,78	12141,22
V 5' 24 cores	58,2	18,4	55,4	11,1	55,5	12,5
L 5' 24 cores	14524,44	8263,90	19720,33	14255,56	17964,90	12179,26

Cuadro 12: Resultados 600 clientes. $\pi = 50$, $\pi v = 50$, $\delta = 0{,}75$

computacionalmente por la gran cantidad de combinaciones de rutas, y muy rara vez la búsqueda local logra reducir vehículos. Del mismo modo, al notar que en ejecuciones de 5 minutos el proceso de búsqueda local reducía la cantidad de iteraciones a 1 ó 2, debido al costo de la búsqueda local, se probó también con $lsRAR = false$, lo cual permitía un número de iteraciones muy superior, pero nunca alcanzando el nivel de optimización que provee la búsqueda local, por lo que fue conveniente ejecutar siempre con $lsRAR = true$.

5. Conclusiones

En esta tesis se presentó una variante del algoritmo *Ruin & Recreate* de Schrimpf *et al* [SCH/99], donde se utiliza un modelo multiobjetivo en el que se intenta reducir, paralelamente, tanto el largo total del recorrido como el vehículo, utilizando para ellos dos variantes de rotura ad hoc para cada caso. Además, se desarrolló una arquitectura distribuída que permite utilizar varios núcleos de una misma computadora (en caso de tenerlos) y varias computadoras en red, de forma de poder contar con un gran número de núcleos (ergo, instancias del algoritmo) ejecutándose a la vez en caso de contar con un cluster de computadoras o incluso en una red local de forma totalmente escalable, pudiendo agregar y quitar nodos a la arquitectura incluso durante la ejecución.

El algoritmo presentado demostró que, excepto contadas ocasiones, la resolución en paralelo del algoritmo multiobjetivo por varias computadoras simultaneamente arroja mejores resultados que utilizando sólo una instancia (de 1 ó 2 núcleos). En los casos que se podrían catalogar como *pequeños* (menores a 400 clientes), el algoritmo obtuvo muy buenos resultados, igualando en muchos casos a los mejores obtenidos mundialmente hasta la fecha. No ha sido posible, empero, mejorar un resultado mejor conocido, ni a nivel vehículos ni a nivel distancias. En los casos *no pequeños* (400 clientes o más), se obtuvieron resultados buenos considerando el poco tiempo utilizado para las pruebas. En los casos en que el algoritmo se corrió más de 5 minutos se obtuvieron mejoras, por lo que se intuye que en un caso real, disponer de un cluster de computadoras y tiempo suficiente sería idoneo para obtener mejores resultados aún.

La búsqueda local demostró en todos los casos ser crucial, mejorando drásticamente los resultados obtenidos comparando con ejecuciones sin esa optimización. En las ejecuciones de 400 y 600 vehículos, si bien la búsqueda local consumía el grueso del tiempo de ejecución, lo cual reducía el número de iteraciones considerablemente, igualmente en el total de los casos siempre fue conveniente incluír la búsqueda local, no así en el caso de RARV, dado que la búsqueda local muy rara vez reduce vehículos, por lo que se estarían reduciendo las iteraciones de rotura y recreación sin obtener casi nada a cambio. Otra opción que se probó fue la de quitar la búsqueda local, de forma de permitir varias mejoras pequeñas, y al finalizar el algoritmo, pasado el tiempo, realizar una búsqueda local sobre la mejor solución obtenida, pero los resultados obtenidos no alcanzaron aquellos que produjo la búsqueda local en cada iteración.

6. Trabajo futuro

Posibles mejoras al algoritmo:

- En la solución actual, cuando un nodo recibe una mejor solución del árbitro, primero debe terminar la iteración actual antes de proceder a reemplazar su solución por la nueva. En las instancias pequeñas este tiempo es despreciable, pero en instancias de 600 vehículos, por ejemplo, la búsqueda local toma varios minutos, por lo que sería conveniente cortar con la iteración para empezar a trabajar inmediatamente con la nueva solución
- La búsqueda local podría optimizarse de forma de no repetir la búsqueda en un par de rutas que no haya dado mejoras si las rutas no fueron modificadas, aunque en instancias grandes esto quizá requiera mucho espacio de memoria y no sea eficiente
- El algoritmo de inserción 3.4 intenta un número fijo de veces ubicando al principio el cliente que no pudo ser insertado en la iteración de inserción anterior, una posible mejora sería armar una tabla con *incompatibilidades*, donde se registren ordenes de clientes que hacen infactible la solución (por ejemplo que el cliente x por su ubicación y ventana hace imposible la visita al cliente y por el mismo vehículo) de forma de evitarlas
- Mezclar con tabú search o aceptar peores soluciones: el algoritmo siempre busca mejorar soluciones y descarta aquellas soluciones infactibles o las factibles pero peores. Una opción, que no se llegó a probar con detalle, sería usar tabú search de forma de guardar soluciones infactibles o no necesariamente mejores, de forma que eso pueda conllevar a salir de un mínimo local por arruinar siempre la misma solución

Referencias

[GAM/99] L. M. Gambardella, É. Taillard and G. Agazzi. **MACS-VRPTW: A Multiple Ant Colony System For Vehicle Routing Problems With Time Windows**. New Ideas in Optimization McGraw-Hill, London, UK, pp. 63-76, 1999.

[LIN/65] S. Lin. **Computer solutions of the traveling salesman problem**. Bell System Technical Journal No. 44, pp. 2245-2269, 1965.

[SCH/99] G. Schrimpf, J. Schneider, H. Stamm-Wilbrandt, and G. Dueck. **Record Breaking Optimization Results Using the Ruin and Recreate Principle**. Journal of Computational Physics 159, pp. 139–171, 1999.

[SOL/87] M. Solomon. **Algorithms for the vehicle routing and scheduling problem with time window constraints**. Operations Research Vol. 35, No. 2, pp. 254-265, 1987.

[TOT/02] P. Toth and D.Vigo. **The Vehicle Routing Problem**. SIAM, 2002.

[SPR] http://w.cba.neu.edu/~msolomon/problems.htm

[SCA] http://www.scala-lang.org/

[TOP] http://www.sintef.no/projectweb/top