ICT-UNPA-148-2016
ISSN: 1852-4516
Aprobado por Resolución N° 0829/16-R-UNPA

Algoritmos metaheurísticos trayectoriales para optimizar problemas combinatorios

Trajectory metaheuristic algorithms to optimize problems combinatorics

Alancay, *N*[1]. Villagra, *S*[2]. Villagra, *A*[2].
[1]{*Becaria de Investigación*}
[2]{*Docentes Investigadores de la UNPA*}
alancaynatalia@hotmail.com, {svillagra, avillagra}@uaco.unpa.edu.ar

UNPA- UACO
Universidad Nacional de la Patagonia Austral - Unidad Académica Caleta Olivia
Departamento de Ciencias Exactas y Naturales
LabTEm- Laboratorio de Tecnologías Emergentes
Caleta Olivia, 2015

RESUMEN

La aplicación de los algoritmos metaheurísticos a problemas de optimización ha sido muy importante durante las últimas décadas. La principal ventaja de estas técnicas es su flexibilidad y robustez, lo que permite aplicarlas a un amplio conjunto de problemas. En este trabajo nos concentramos en metaheurísticas basadas en trayectoria *Simulated Annealing, Tabu Search* y *Variable Neighborhood Search* cuya principal característica es que parten de un punto y mediante la exploración del vecindario varían la solución actual, formando una trayectoria. Mediante las instancias de los problemas combinatorios seleccionados, se realiza una experimentación computacional que ilustra el comportamiento de los métodos algorítmicos para resolver los mismos. El objetivo principal de este trabajo es realizar el estudio y comparación de los resultados obtenidos para las metaheurísticas trayectoriales seleccionadas en su aplicación para la resolución de un conjunto de problemas académicos de optimización combinatoria.

Palabras clave: Metaheurísticas de trayectoria; Simulated Annealing; Tabu Search y Variable Neighborhood Search; Problemas de Optimización Combinatoria.

ABSTRACT

The application of metaheuristic algorithms to optimization problems has been very important during the last decades. The main advantage of these techniques is their flexibility and robustness, which allows them to be applied to a wide range of problems. In this work we concentrate on metaheuristics based on Simulated Annealing, Tabu Search and Variable Neighborhood Search trajectory whose main characteristic is that they start from a point and through the exploration of the neighborhood vary the current solution, forming a trajectory. By means of the instances of the selected combinatorial problems, a computational experimentation is carried out that illustrates the behavior of the algorithmic methods to solve them. The main objective of this work is to perform the study and comparison of the results obtained for the selected trajectories metaheuristics in its application for the resolution of a set of academic problems of combinatorial optimization.

Key words: Trajectory Metaheuristics; Simulated Annealing; Tabu Search; Variable Neighborhood Search; Combinatorial Optimization Problems.

1. INTRODUCCIÓN

En el campo de la optimización existe un particular interés en las metaheurísticas, las cuales se definen como estrategias inteligentes para diseñar o mejorar procedimientos heurísticos generales (Glover y Kochenberger 2003) para la resolución de problemas con un alto rendimiento. Hay un vasto número de aplicaciones reportadas y eventos que documentan el éxito e importancia en esta área.

La optimización viene dada por la reducción al mínimo de costo, tiempo, distancia y riesgo o la maximización de calidad, satisfacción y beneficios. Encontrar la mejor solución para determinados problemas de optimización con importancia científica o industrial deriva muchas veces en presentaciones computacionales que son de complejidad intratables. Sin embargo, el uso de algoritmos de aproximación es la principal alternativa para resolver esta clase de problemas ya que permite obtener una solución óptima o "casi" óptima en un tiempo razonable. Por esta razón, en este escenario de complejidad se presentan como una opción viable el uso de metaheurísticas proveyendo un resultado factible que satisface todas las restricciones del problema.

La mayoría de los problemas de optimización combinatoria son, en general, difíciles de resolver en la práctica. Estos problemas están incluidos en la clase de problemas *NP*-duros (Garey y Johnson 1979), ya que no se conocen algoritmos exactos con complejidad polinómica que permitan resolverlos. Debido a su intratabilidad, se han diseñado una gran cantidad de métodos aproximados, los cuales encuentran buenas soluciones en tiempos computacionales razonables. En esta clase de problemas, la búsqueda de una solución requiere una exploración organizada a través del espacio de búsqueda: una búsqueda sin guía es extremadamente ineficiente.

En el presente trabajo se intenta explicar al lector acerca del modelado e implementación de algunas metaheurísticas de trayectoria. Estas metaheurísticas se caracterizan porque parten de un punto para la mejora continua de la solución actual mediante la inspección de un vecindario donde la búsqueda finaliza cuando se alcanza un número máximo de iteraciones y se encuentra una solución con una calidad aceptable, o se detecta un estancamiento del proceso en la exploración del espacio de búsqueda.

En particular abordaremos las metaheurísticas de trayectoria denominadas Recocido Simulado (*Simulated Annealing*[1], SA Kirkpatrick et al. 1983), Búsqueda Tabú (*Tabu Search*, TS Glover y Laguna 1997) y búsqueda en vecindad variable (*Variable Neighborhood Search*, VNS Hansen y Mladenovíc 2001), aplicadas a la resolución de los diferentes problemas de optimización combinatoria abordados para determinar cuál de ellas es la de mejor desempeño. De acuerdo a esto, hemos utilizado un procedimiento estadístico para evaluar las metaheurísticas, y presentar las principales medidas de rendimiento así como los indicadores de calidad utilizados en problemas de optimización mono objetivo.

El trabajo está organizado de la siguiente manera: En la Sección 2 se presentan conceptos básicos relacionados a las metaheurísticas. En la Sección 3 se explica el método de búsqueda en el espacio de soluciones. La Sección 4 está dividida en diferentes subsecciones donde se describe con más detalle el proceso de los algoritmos SA, TS y VNS que forman parte las metaheurísticas de trayectoria utilizadas en este trabajo. En la Sección 5 describimos

[1] En este trabajo utilizamos los nombres y siglas en inglés.

los distintos problemas de optimización combinatoria que hemos utilizado. En la Sección 6 se detalla los parámetros aplicados y un análisis del comportamiento sobre la performance de los algoritmos. En la Sección 7 se realiza una conclusión del comportamiento que tuvieron los algoritmos y se proyectan las futuras investigaciones.

2. GENERALIZACIÓN DE CONCEPTOS BÁSICOS

Dada la dificultad práctica que implica resolver de forma exacta una serie importante de problemas combinatorios que requieren de alguna solución eficiente, han aparecido algoritmos que proporcionan soluciones factibles, es decir, que satisfacen todas las restricciones del problema. Este tipo de algoritmos se denominan heurísticas (Martí 2003).

Las heurísticas son procedimientos simples que a menudo están basados en el sentido común, y se supone ofrecen una buena solución, (aunque no necesariamente la óptima) a problemas difíciles, de un modo fácil y rápido.

Los procedimientos metaheurísticos (Glover 1986) fueron introducidos en 1986 por Glover y deriva de la composición de dos palabras griegas *heuristic* que significa "encontrar", y *meta*, que significa "más allá, en un nivel superior". Por esto las metaheurísticas se denominan como una clase de métodos de aproximación que están diseñados para resolver problemas difíciles de optimización combinatoria, así mismo también proporcionan un marco general para crear nuevos algoritmos híbridos combinando diferentes conceptos derivados de la inteligencia artificial, la evolución biológica y los mecanismos estadísticos.

En la literatura se pueden encontrar muchas clasificaciones de las metaheurísticas. La clasificación que se presenta en esta investigación es de principal interés, propuesta por Melián y Glover 2003. A continuación se definen el procedimiento que utilizan el algoritmo heurístico relacionado con las metaheurísticas de búsqueda.

Las metaheurísticas de búsqueda, establecen estrategias para recorrer el espacio de soluciones del problema transformando de forma iterativa una solución de partida. Este es el tipo de metaheurística más importante y abarca una gran variedad de métodos, entre los que están los métodos de búsqueda local y los de búsqueda global.

- En los métodos de **búsqueda local** se parte de una solución inicial e iterativamente se trata de mejorar la solución hasta que no sea posible obtener mejoras. La mejora de una solución se obtiene en base al análisis de soluciones similares o cercanas, denominadas soluciones vecinas. Tales métodos se conocen también como búsquedas monótonas (descendentes o ascendentes) y algoritmos escaladores (*hillclimbing*). El principal inconveniente de estas búsquedas locales es que se quedan atrapadas en un óptimo local; para escapar de óptimos locales se usan algoritmos que permiten seguir explorando el espacio de soluciones, haciendo uso de estructuras de memoria y técnicas probabilísticas. Por ello, el propósito de las primeras metaheurísticas era extender una búsqueda local para continuarla más allá de los óptimos locales, denominándose búsqueda global.
- En las metaheurísticas de **búsqueda global** se incorporan pautas para escapar de los óptimos locales de baja calidad. Estas pautas consideran tres formas básicas: volver a iniciar la búsqueda desde otra solución de arranque (metaheurísticas de arranque múltiple, *multistart*); modificar en forma sistemática la estructura de entornos que se está aplicando, es decir, la manera como se define una solución vecina (metaheurística VNS); y permitir movimientos o transformaciones de la solución de búsqueda que no sean de mejora (metaheurísticas no monótonas). En este último grupo se encuentran las metaheurísticas de estrategias probabilísticas, siendo SA la más representativa, y las estrategias con memoria,

representadas por TS.

De acuerdo a Melián y Glover 2003, todas las metaheurísticas de una u otra forma se pueden concebir como estrategias aplicadas a procesos de búsqueda, en las que para resolver el problema se van modificando elementos del espacio de búsqueda a medida que se aplican las distintas operaciones diseñadas para llegar a la solución definitiva. Es frecuente interpretar que el término metaheurística es aplicable esencialmente a los procedimientos de búsqueda sobre un espacio de soluciones alternativas. Por tal razón, y con el fin de facilitar la comprensión de las técnicas que serán tratadas aquí, en la siguiente sección se describe con más detalle el proceso de búsqueda SA, TS y VNS las cuales son las bases de muchas de las metaheurísticas de trayectoria para resolver una amplia gama de problemas de optimización combinatoria.

3. BÚSQUEDA EN EL ESPACIO DE SOLUCIONES

Los métodos de búsqueda conforman una clase general de las heurísticas basadas en el concepto de explorar el vecindario de la solución actual. Estos procedimientos comienzan con una solución inicial a partir de la cual recorren el espacio de soluciones haciendo un conjunto de modificaciones o movimientos. Las soluciones que se obtienen mediante uno de los posibles movimientos se denominan vecinas de ésta y constituyen su entorno. El conjunto de movimientos posibles da lugar a una relación de vecindad y una estructura de entornos en el espacio de soluciones.

En el contexto de la búsqueda por entorno, una solución del problema se puede definir como un vector *x* tal como se indica en la Figura 1 y el costo de una solución *x* se denota por *f(x)* llamada usualmente función objetivo.

x	0	0	1	0	1	0	0	1	0	1	0	0

Figura 1: Representación de una solución como un vector de variables binarias.

Para generar una solución vecina de *x* se pueden utilizar diferentes estrategias cuyo diseño depende de la naturaleza del problema y del ingenio del planificador. Cada solución vecina $x' \in N(x)$ (considerado como el vecindario de x) puede ser alcanzada directamente de *x* mediante una operación llamada movimiento, y se dice que *x* se mueve a *x'* cuando tal operación es ejecutada.

Solución actual X	0	0	1	0	1	0	0	1	0	1	0	0
Solución vecina X'	0	0	1	0	1	0	1	1	0	1	0	0

Figura 2: Representación de una solucón vecina generada mediante el cambio de alternativa.

Una estrategia sencilla para este caso es elegir aleatoriamente un valor del vector y cambiarlo por su opuesto, es decir, si el valor elegido al azar es 0 asignarle 1, y a la siguiente asignarle 1. Entonces una solución vecina *x'* es obtenida a partir de la solución *x* que es la que se muestra en la Figura 2.

Una vez diseñada la estrategia que se utilizará para generar movimientos que conducen a la obtención de soluciones vecinas se puede aplicar el proceso iterativo de búsqueda.

El esquema general de un procedimiento de búsqueda consiste en generar una solución inicial y, hasta que se cumpla un criterio de parada, seleccionar iterativamente un movimiento para modificar la solución. Las soluciones son evaluadas mientras se recorren y se propone la mejor solución encontrada. Otros aspectos que deben decidirse para aplicar este algoritmo, son la forma de generar la solución inicial y el criterio de parada.

Algunas metaheurísticas tales como SA, TS y VNS se basan en la inclusión de

procedimientos y estrategias que mejoran el proceso de búsqueda en el espacio de soluciones.

4. USO DE ALGORITMOS METAHEURÍSTICOS

Al enfrentar a diversos problemas de diferente índole, en algunas ocasiones se los puede resolver de forma manual usando nuestra razón pero cuando el tamaño de dichos problemas excede de nuestra capacidad nos vemos obligado a utilizar herramienta analítica que facilite su resolución. Es por esto que los algoritmos metaheurísticos son utilizados con gran éxito para la resolución de estos tipos de problemas.

En este trabajo hemos utilizado una batería de problemas que contienen características de optimización interesantes tales como multimodalidad, epistasis y deceptividad. Aplicando a diferentes metaheurísticas de trayectoria como SA, TS y VNS.

A continuación exponemos con detalle el concepto de los algoritmos propuestos con las instancias de los problemas a resolver.

4.1 Algoritmo *Tabu Search*

El término *Tabu Search* fue introducido en 1986 en el mismo artículo que introdujo el término metaheurística (Glover 1986). Los principios fundamentales de la búsqueda fueron elaborados en una serie de artículos a finales de los años 80 y principios de los 90, que fueron luego unificados en el libro "*Tabu Search*" en 1997 (Glover y Laguna 1997).

TS es un procedimiento heurístico de memorias adaptativa para la búsqueda de óptimos globales en problemas de optimización mono-objetivos. El método se basa en el principio que es mejor realizar un mal movimiento, cuando este es realizado de forma informada, que un buen movimiento ejecutado de forma aleatoria. TS está siendo aplicado exitosamente en las últimas décadas en multitud de problemas complejos (Sicilia et al. 2016; Misevičius 2015) (continuos y discretos, lineales y no lineales convexos y no convexos, etc.).

TS explora el espacio de soluciones a través de repetidos movimientos desde una solución a la mejor de sus vecinas tratando de evitar los óptimos locales. Para un problema mono-objetivo, TS realiza una búsqueda por entornos en la cual se desplaza en cada iteración a la mejor solución no tabú del vecindario de la solución actual. Los principales atributos de cada solución visitada son almacenados en una lista tabú por un determinado número de iteraciones para evitar que estas soluciones sean revisitadas, es decir, para evitar ciclos en la búsqueda por entornos. Así, un elemento del vecindario de la solución actual es declarado tabú (es decir, es prohibido) si alguno de sus atributos está en la lista tabú. En general, un método basado en búsqueda tabú requiere de los siguientes elementos:

1. **Solución inicial**. La búsqueda debe comenzar desde una solución inicial que podría ser cualquier solución admisible que satisfaga las restricciones del problema. Una buena solución inicial podría acelerar la búsqueda con el consiguiente ahorro de tiempo. Dicha solución puede ser generada aleatoriamente o utilizando funciones ávidas o *greedy functions* (funciones que incorporan información adicional del problema utilizadas como estrategias para generar puntos de mejor calidad).
2. **Movimiento**. Un movimiento es un procedimiento aleatorio o determinístico por el que se genera una solución admisible a partir de la solución inicial. Usualmente, este procedimiento es sencillo para el caso de problemas combinatorios, pero mucho más complejo para el caso de problemas de optimización continuos.
3. **Vecindad**. Dada una solución *s*, la vecindad *N(s)* es el conjunto de todas las soluciones admisibles que pueden ser generadas por la ejecución de un movimiento sobre la solución actual *s*. Este conjunto suele ser numerable para problemas combinatorios y, en aquellos casos en los que *N(s)* sea grande, se suele operar con un subconjunto de éste. Para problemas continuos, los posibles vecinos son no

numerables y se debe ser más creativo para definir *N(s)*.

4. **Lista tabú**. Es un mecanismo de memoria adaptativa que trata de evitar que la búsqueda entre en un ciclo o quede atrapada en un óptimo local. Una vez que un movimiento, que genera una nueva solución, es aceptado, su movimiento inverso se añade a la lista tabú y permanece en ésta un número determinado de iteraciones. Si el tamaño de la lista tabú es pequeño, entonces la búsqueda se intensifica en una determinada área del espacio, mientras que si el tamaño de la lista es grande se enfatiza la búsqueda en diferentes regiones del espacio de soluciones.
5. **Criterio de parada**. En general, la búsqueda termina después de un número determinado de iteraciones, después de un tiempo de computación predefinido o cuando se alcanza un número dado de iteraciones sin mejorar la mejor solución.

A continuación mostramos los detalles del algoritmo TS que se basa en una búsqueda local mediante el uso de una lista tabú que inicializamos apropiadamente.

```
1 solución inicial ← s0
2 tabuList ← ([],s0.length)
3 Repetir
4       solución vecina s ← s0
5       if(not tabuList.contains(s))
6       {
7            if(Target.maximize){
8                 if ( f(s) >= f(s0) )
9                       s0 = s
10           }else{
11                if( f(s) <= f(s0))
12                      s0 = s
13           }
14           tabuList.grabarMovimientoMemoria(s);
15           if (tabuList.size() >= tabuList.getMaxListTabuSize())
16                tabuList.actualizarMemoria()
17      }
18 Hata condición de parada
19 return la mejor de todas s0 encontradas
```

Figura 3: Pseudocódigo de *Tabu Search*

El tamaño de la lista tabú afecta de manera significativa al funcionamiento del algoritmo. Un tamaño de lista pequeño, hará que el algoritmo se comporte de manera similar a una búsqueda local, por lo que puede suceder que el algoritmo termine probando cíclicamente una serie de soluciones poco diversas. Por otro lado, un tamaño de lista tabú muy grande, además de poder crear problemas de memoria y tiempo de cómputo, puede prohibir arbitrariamente el probar soluciones de calidad.

Pasamos a explicar la implementación del algoritmo TS que muestra la Figura 3. En la línea 1 y 2 creamos la solución inicial *s0* elegida al azar e inicializamos una lista tabú vacía con tamaño de acuerdo a la longitud del individuo del problema a resolver. La lista tabú es simplemente una estructura de memoria a corto plazo que contendrá un registro de los elementos de los estados visitados. En la línea 3 comienza el bucle que continuara la búsqueda de una solución optima hasta que se cumpla la condición de parada (tal condición puede ser de haber conseguido la solución optima o el límite de número de evaluaciones). En la línea 4, se establece la solución vecina, la cual se comprueba si está (solución vecina) forma parte o está contenida en los elementos de la lista tabú como se describe en la línea 5. En las líneas 7-14 analizamos si se trata de un problema de maximización o minimización para luego buscar la mejor solución en el espacio de soluciones, que no sea tabú (es decir que aun no esté en la lista tabú). La mejor solución siempre se añada a la lista tabú (línea 14). En general, los elementos duran o expiran en la lista tabú en el mismo orden en que se agregan,

por lo cual se actualiza la memoria de lista tabú cuando el tamaño de la misma llega al límite permitido (línea 16). Este proceso continúa hasta que cumpla un criterio específico de parada. Finalmente la mejor solución encontrada durante el proceso de búsqueda se devuelve (línea 19).

```
1 solución inicial ← s0 ; kmax ← nroEstructuraDeVecindad
2 Repetir
3       k ← 1
4       Repetir
5             solución vecina s ← s0
6             s ← s.busquedaLocal(k)
7             if( f(s) >= f(s0) )
8                   s0 ← s
9                   k←1
10            } else
11                  k ← k+1
12      Hasta k<= kmax
13 Hata condición de parada
14 return la mejor de todas s0 encontradas
```

Figura 4: Pseudocódigo de *Basic Variable Neighborhood Search*

4.2 Algoritmo *Variable Neighborhood Search*

Variable Neighborhood Search (Hansen y Mladenovíc 2001), es una metaheurística para resolver problemas de optimización (Alonso-Ayuso et al. 2015; Sicilia et al. 2016) cuya idea básica es el cambio sistemático de vecindario dentro de una búsqueda local, es decir, esta introduce como idea novedosa el manejo de un conjunto determinado de estructuras de vecindad. Por lo tanto, para cada solución *x* se define un conjunto de vecindades $N_k(x)$, donde $1 \leq k < k_{max}$. Cada una de estas vecindades $N_k(x)$ se puede construir mediante una o varias métricas (o cuasi-métricas) que lógicamente serán dependientes del problema.

VNS está basada en tres hechos simples:

1. Un mínimo local con respecto a una estructura de vecindad no lo es necesariamente con respecto a otra.
2. Un mínimo global es un mínimo local con respecto a todas las posibles estructuras de vecindades.
3. En muchos problemas el mínimo local con respecto a una o varias estructuras de vecindad están relativamente cerca.

La última observación es empírica e implica que un mínimo local muchas veces nos da información acerca del óptimo global.

En la literatura se pueden encontrar varias implementaciones distintas de VNS. Si tomamos en cuenta fundamentalmente las tres ideas introducidas con anterioridad se las pueden combinar para formar tres estrategias: (i) determinística, (ii) estocástica, y (iii) combinación de las dos.

La búsqueda del entorno variable descendente (*Variable Neighborhood Descent*, VND): es un método determinístico. La idea es buscar sistemáticamente en diferentes esquemas de vecindad hasta llegar a un mínimo local que es mínimo con respecto a todos los esquemas de vecindad.

La segunda combinación también puede ser estocástica, en este caso la implementación de VNS se llama reducido (*Reduced* VNS, RVNS): es un método aleatorio. Es exactamente igual que la anterior, sólo que los vecinos en cada vecindad son seleccionados aleatoriamente.

Por último, la combinación también puede ser mixta, es decir, combinando aspectos determinista y estocástico. En este caso, VNS recibe el nombre de VNS básico (*Basic* VNS,

BVNS). En esta implementación se selecciona un punto de forma aleatoria perteneciente a una vecindad, donde a esta se la mejora mediante una estrategia de búsqueda local. En el pseudocódigo presentado en la Figura 4 podremos ver en detalle la implementación de BVNS o VNS.

El algoritmo propuesto para VNS, empieza con una solución inicial *s0* (línea 1), en este caso es una solución aleatoria. En el pseudocódigo se configura k_{max} con un número máximo de vecindades a explorar (línea 1). En la línea 5 la solución vecina tiene asignado inicialmente una copia de la solución inicial. En cada iteración la solución vecina se perturba de manera aleatoria con *k* tamaño hasta llegar al k_{max} (línea 12), obteniendo una nueva solución. Se aplica el proceso de búsqueda local basado en VND obteniendo una solución mejorada (línea 6). En la línea 7 se analiza si la solución vecina es mejor que solución inicial, en tal caso *s0* se actualiza a *s* (línea 8) como la mejor solución encontrada hasta el momento y *k* se pone a 1 (línea 9); en caso contrario *k* se incrementa (línea 11). Este proceso se repite hasta que se alcance k_{max}, siempre que no se haya llegado al máximo número de iteraciones permitidas. Por lo tanto, k_{max} es un parámetro de la búsqueda.

4.3 Algoritmo *Simulated Annealing*

Esta metaheurística aparece a principios de los años 80 (Kirkpatrick et al. 1983). Se trata de un modelo de resolución para la optimización de problemas de tipo combinatorio con mínimos locales (Sen et al. 2016) (Wang et al. 2015). Consiste en generar aleatoriamente una solución cercana a la solución actual (o en el entorno de la solución) y la acepta como buena si consigue reducir una determinada función de costo, o con una determinada probabilidad de aceptación. Esta probabilidad de aceptación se irá reduciendo con el número de iteraciones y está relacionada también con el grado de empeoramiento del costo, es decir, en el algoritmo *Simulated Annealing* se pueden aceptar soluciones que empeoran la solución actual, solo que esta aceptación dependerá de una determinada probabilidad que depende de un parámetro, denominado temperatura, del estado del sistema.

El nombre de SA viene de la idea en que está basado en un algoritmo diseñado en los años 50 para simular el enfriamiento de material (un proceso denominado "recocido"). El proceso de SA es el siguiente: parte de una solución inicial que de modo paulatino es transformada en otra que a su vez son mejoradas al introducirles pequeñas perturbaciones o cambios (tales como cambiar el valor de una variable o intercambiar los valores que tienen dos variables). Si este cambio da lugar a una solución "mejor" que la actual, se sustituye ésta por la nueva, continuando el proceso hasta que no es posible ninguna nueva mejora. Esto significa que la búsqueda finaliza en un óptimo local, que no tiene por qué ser forzosamente el global.

Un modo de evitar este problema es permitir que algunos movimientos sean hacia soluciones peores. Pero por si la búsqueda está realmente yendo hacia una buena solución, estos movimientos "de escape" deben realizarse de un modo controlado. En SA esto se realiza controlando la frecuencia de los movimientos de escape mediante una función de probabilidad que hará disminuir la probabilidad de esos movimientos hacia soluciones peores conforme avanza la búsqueda (y por tanto estamos previsiblemente más cerca del óptimo global).

La fundamentación de este control se basa en el trabajo de Metropolis (1953) en el campo de la termodinámica estadística. Básicamente, Metrópolis modeló el proceso de recocido mencionado anteriormente simulando los cambios energéticos en un sistema de partículas conforme decrece la temperatura, hasta que converge a un estado estable (congelado). La ley de la termodinámica dice que a una temperatura *t* la probabilidad de un incremento energético de magnitud δE se puede aproximar por $P[\delta E] = exp(-\delta E / kt)$ siendo *k* una constante física denominada de Boltzmann. En el algoritmo de Metrópolis se genera una perturbación aleatoria en el sistema y se calculan los cambios de energía resultantes: si hay

una caída energética, el cambio se acepta automáticamente; por el contrario, si se produce un incremento energético, el cambio será aceptado con una probabilidad dada por la formula anterior. El proceso se repite durante un número predefinido de iteraciones en series decrecientes de temperaturas, hasta que el sistema esté "frío".

En la Figura 5 se muestra la estructura básica de un algoritmo de enfriamiento lento simulado para el caso de problemas de maximización.

```
1 solución inicial ← s0
2 seleccionar temp.inicial t0 > 0
3 velocidad de enfriamiento ←  v
4 Repetir
5       Repetir
6           solución vecina s ← s0
7           if (f (s) >= f(s0))
8                       s0 ← s
9           else{
10              d ← f(s) - f(s0)
11              if(Math.random() < exp (-d/t))
12                      s0 ← s
13          }
14      Hata condición de parada
15      t ← t * v
16 Hata condición de parada
17 return la mejor de todas s0 encontradas
```

Figura 5: Pseudocódigo de S*imulated Annealing*

En este pseudocódigo se pueden diferenciar dos tipos de elementos que permiten afinar el algoritmo para que la ejecución del mismo sea la más eficaz de acuerdo con el problema que se desea resolver:

1. **Elementos Genéricos:** son elementos que no tienen una dependencia directa del problema, aunque hay que afinarlos para el problema concreto que se desea resolver.
   - Temperatura inicial: *t0*
   - Proceso de enfriamiento: *v*
   - Números de repeticiones.
   - Condición de parada.
2. **Elementos dependientes del problema.** Son los elementos que definen de forma directa el problema que se está resolviendo. Definen un modelo del problema y la estructura del espacio de soluciones para el mismo.
   - Espacio de soluciones: *S*
   - Solución inicial: *s0*
   - Estructura de vecindad: *s*
   - Función de costo: *f(s0)*

A continuación describiremos con más detalle los elementos mencionados.

a. **Temperatura inicial**

La temperatura inicial es la que se adopta para comenzar el proceso de enfriamiento y se irá bajando en tanto avance la evolución. Está debe ser lo suficientemente alta para permitir el libre cambio de soluciones vecinas y, sobre todo, que la solución final sea independiente de la solución inicial. Una regla práctica que se le suele poner a la temperatura inicial es que debe tener un valor tal que la proporción inicial de vecinos aceptados (tanto de mejora como de no mejora) tenga un alto valor, es decir, inicialmente todos los vecinos se aceptan con una probabilidad cercana a 1.

Lógicamente hay que elegir esta temperatura con cuidado porque si se elige una temperatura inicial demasiado alta se perderá tiempo realizando muchos movimientos

que se aceptan siempre, lo que no resulta de mucho interés ya que no se está persiguiendo ningún valor concreto. Por otra lado, si se elige una temperatura inicial demasiado baja el proceso se mueve poco desde la solución inicial por lo que se puede quedar atrapado desde un inicio en una región desde la que sólo se pueda llegar a un óptimo local, por lo que se perderá la posibilidad de explorar otras regiones donde se podría encontrar alguno de los óptimos globales del sistema.

***b.* Proceso de enfriamiento**

El proceso de enfriamiento es el mecanismo por el cual la temperatura va tendiendo a 0. Es decir, la probabilidad de aceptación de soluciones peores tiende a 0. El proceso de enfriamiento, por tanto, determina cómo se modifica la temperatura durante la ejecución del algoritmo.

Hay dos modelos básicos que se pueden utilizar para el proceso de enfriamiento. En el primero podríamos ir enfriando muy lentamente con cada iteración de la cadena de Markov de manera que en un número de pasos extremadamente alto la temperatura tienda a 0, ó utilizar un proceso de enfriamiento más rápido, pero antes de aplicar un cambio en la temperatura hay que dejar que el sistema llegue a un estado en el que se estabilice la solución.

Hay que intentar durante el proceso de enfriamiento que haya una alta aceptación al principio (de forma que se pueda realizar un proceso de exploración inicial en el espacio de búsqueda) e ir reduciendo está aceptación hacia el final.

***c.* Número de repeticiones**

El número de repeticiones del bucle interno nos va a dar el número de vecinos que se visitan antes de reducir la temperatura del sistema. El proceso debe ser dinámico, de forma que se aumente el número de repeticiones según se reduce la temperatura. De esta forma se conseguirá explotar la vecindad hacia el final del proceso.

Un mecanismo habitual para conseguirlo es repetir el bucle hasta que se cumpla una de las siguientes condiciones:

- Se aceptan un cierto número de evaluaciones. De esta forma el bucle termina en caso de que se hayan analizado un determinado número de evaluaciones del algoritmo.

- Se hayan obtenido la solución óptima. De esta forma el bucle termina si se da el caso que se ha conseguido la mejor solución.

Para que se pueda explorar con cierta completitud todos los vecinos.

***d.* Condición de parada**

Esta condición nos indica cuando damos por terminado el algoritmo de cálculo. Hay que tener en cuenta que al reducirse la temperatura, la probabilidad de aceptar soluciones peores tiende a 0. Así mismo, la probabilidad de encontrar soluciones mejores también tiende a 0. En este sentido las condiciones para terminar el algoritmo deberían tener en cuenta que ya se han realizado un determinado número de iteraciones sin mejorar la solución. La idea es evitar seguir insistiendo reiteradamente en la vecindad de una solución para la que ya se lleva tiempo intentando explorar.

***e.* Espacio de soluciones**

La definición del modelo del problema determinará la estructura del espacio de soluciones y, por tanto, la forma en que se va a poder recorrer, los algoritmos que se pueden utilizar para crear soluciones, etc.

Lógicamente para cada problema hay que estudiar de forma diferenciada cómo generar este espacio de soluciones, en relación con la facilidad para generar una vecindad y el

cálculo de la función de costo asociada a una determinada solución.

f. **Solución inicial**

La solución inicial, en general es cualquier solución válida del problema. Al ser un problema de optimización combinatoria, el modelo permite expresar una solución como una combinación de partes del mismo. De esta forma, la solución inicial más sencilla resulta de elegir una combinación aleatoria de elementos del problema que conformen una solución con la estructura del modelo elegido.

g. **Estructura de vecindad**

Dada una solución, hay que poder conseguir una nueva solución mediante un pequeño cambio en la solución original. Además la forma de generar una vecindad nos debe poder asegurar que a partir de una solución cualquiera se debe poder llegar a cualquier otra ya sea directa o indirectamente. Esta característica es vital para dar validez al proceso de convergencia del algoritmo. Asimismo, la solución vecina debería ser suficientemente pequeña como para poder realizar una búsqueda en pocas iteraciones, como suficientemente grande para generar mejoras sustanciales en pocos movimientos.

Desde el punto de vista computacional la generación de un vecino debería ser rápida, de forma que se puedan calcular un mayor número de vecinos por unidad de tiempo. Así mismo, sería muy apropiado que el vecino generado siempre sea una solución factible, lo que puede resultar complicado.

h. **Función de costo**

La función de costo es una medida asociada a todos los elementos de una determinada solución. Uno de los objetivos que deben guiar el desarrollo de la función de costo es que esta sea rápida de calcular al generar una solución vecina, creando un cálculo incremental que tenga en cuenta sólo los cambios que generan la vecindad. De esta forma se podrían calcular más soluciones por unidad de tiempo. De hecho, el cálculo de la función de costo suele ser uno de los aspectos más costosos en tiempo de cómputo del algoritmo.

5. PROBLEMAS ESTUDIADOS

La efectividad de los algoritmos descriptos anteriormente fue probada inicialmente sobre algunos problemas clásicos de optimización combinatoria. Con el fin de analizar el comportamiento de la búsqueda de SA, TS y VNS se han elegido un conjunto de problemas representativos que poseen diversas características que son de particular interés en optimización.

Inicialmente se han utilizado los problemas abordados en (Alba y Troya 2000), que incluyen el problema masivamente multimodal (MMDP), el problema de la modulación en frecuencia de sonidos (FMS) y el generador multimodal P-PEAKS; luego hemos extendido esta base de tres problemas con los problemas COUNTSAT (una instancia de MAXSAT), diseño de códigos correctores de errores (ECC), máximo corte de un grafo (MAXCUT), el problema de OneMax y planificación de tareas con espera mínima (MTTP). A continuación presentamos la descripción de los ochos problemas abordados en este trabajo:

1) El problema **MMDP** fue específicamente diseñado para ser difícil de resolver para un EA (Goldberg et al.1992). Está compuesto de *k* subproblemas engañosos (s_i) formado cada uno de ellos por una cadena binaria de longitud seis, y su valor depende del número de unos que contenga dicha cadena (*fitness* s_i). Es fácil ver que estas funciones poseen dos máximos globales y un atractor engañoso en el punto intermedio. En MMDP, cada sub-

problema s_i contribuye al valor de fitness total en función del número de unos que tiene. El óptimo global del problema tiene el valor *k*, y se alcanza cuando todos los sub-problemas están compuestos de cero o seis unos. El número de óptimos locales es muy grande (22^k), mientras que sólo hay 2^k soluciones globales. Por tanto, el grado de multimodalidad viene dado por el parámetro *k*. En nuestro estudio, utilizaremos (mientras no se especifique lo contrario) una instancia considerablemente grande de $k = 40$ sub-problemas. La función que trataremos de maximizar corresponde con un valor máximo de 40.

2) El problema **FMS** (*Frequency Modulation Sounds*) (Tsutsui et al. 1997) consiste en determinar los 6 parámetros reales $\vec{x} = (a_1; w_1; a_2; w_2; a_3; w_3)$ (codificados con 32 bits en el rango [−6,4; +6,35]) del modelo de sonido en frecuencia modulada. El objetivo de este problema consiste en minimizar la suma del error cuadrático medio entre los datos de muestreo y los reales. Debido a la dificultad para resolver este problema sin operadores específicos, el algoritmo se detiene cuando el error cae por debajo de 10-2. La función de adecuación que tendremos que maximizar se corresponde su máximo valor cuando EFMS =0,0.

3) El problema **P-PEAK** (Kenneth et al. 1997) es un generador de problemas multimodales. Un generador de problemas es una tarea fácilmente parametrizable, con un grado de dificultad que se puede afinar, de manera que nos permita generar instancias con una complejidad tan grande como queramos. Adicionalmente, el uso de un generador de problemas elimina la posibilidad de afinar a mano los algoritmos para un problema particular, permitiendo de esta forma una mayor justicia al comparar los algoritmos. Utilizando un generador de problemas, evaluamos nuestros algoritmos sobre un elevado número de instancias de problema aleatorias, ya que cada vez que se ejecuta el algoritmo resolvemos una instancia distinta. Por tanto, se incrementa el poder de predicción de los resultados para la clase de problemas como tal, no para instancias concretas.
La idea de *P-PEAK* consiste en generar *P* cadenas aleatorias de *N bits* que representan la localización de *P* cimas en el espacio de búsqueda. Usando un mayor (o menor) número de cimas obtenemos problemas más (o menos) epistáticos. En nuestro caso hemos utilizado una instancia de *P = 100* cimas de longitud *N = 100 bits* cada una, representando un nivel de epistasis medio/alto (Alba y Troya 2000). El máximo valor de *fitness* que podremos conseguir es 1,0.

4) El problema **COUNTSAT** (Droste et al. 2000), es una instancia de MAXSAT. En COUNTSAT el valor de la solución se corresponde con el número de cláusulas (de entre todas las posibles cláusulas de *Horn* de tres variables) satisfechas por una entrada formada por *n* variables booleanas. Es fácil comprobar que el óptimo se consigue cuando todas las variables tienen el valor 1. En este estudio consideramos una instancia de *n = 20* variables, y el valor de la solución óptima al problema es 6860.
La función COUNTSAT esta extraída de MAXSAT con la intención de ser muy difícil de resolver con Algoritmos Genéticos (Droste et al. 2000). Las entradas aleatorias generadas uniformemente tendrán *n = 2* unos en término medio. Por tanto, los cambios locales decrementando el número de unos nos conducirán hacia entradas mejores, mientras que si se incrementa el número de unos se decrementará el valor de adecuación.

5) El problema **ECC** se presento en (MacWilliams y Sloane 1977). Consideremos una tupla (*n; M; d*) donde *n* es la longitud (número de bits) de cada palabra de código, *M* es el número de palabras de código, y *d* es la mínima distancia de Hamming entre dos palabras de código cualesquiera. El objetivo será obtener un código con el mayor valor posible de *d* (reflejando una mayor tolerancia al ruido y a los errores), y con valores de *n y M* fijados previamente. En este estudio consideramos una instancia donde *C* está formado por *M*=24

palabras de longitud n=12 bits. Lo que forma un espacio de búsqueda de tamaño $\binom{4096}{24}$, que es 10^{87} aproximadamente. La solución óptima para la instancia con M =24 y n = 12 tendría el valor de adecuación 0,0674.

6) El problema **MAXCUT** consiste en dividir un grafo ponderado $G = (V, E)$ en dos subgrafos disjuntos $G_0 = (V_0, E_0)$ y $G1 = (V_1, E1)$ de manera que la suma de los pesos de los ejes que posean un extremo en V_0 y el otro en V_1 sea máxima. Para codificar una partición de los vértices utilizaremos una cadena binaria $(x_1, x_2, \ldots, x_n)$ donde cada dígito se corresponde con un vértice. De manera que si un dígito tiene valor 1, su vértice correspondiente estaría en el conjunto V_1, y si tuviera valor 0 el vértice respectivo pertenecería a V_0. Aunque es posible generar instancias de grafos diferentes de forma aleatoria, hemos utilizado tres instancias distintas del problema. Dos de ellas se corresponden con grafos generados aleatoriamente: uno disperso "MAXCUT20_01" y otro denso "MAXCUT20_09", ambos formados por 20 vértices. La otra instancia es un grafo escalable de 100 vértices. Las soluciones óptimas para estas instancias son 10,11 para el caso de "MAXCUT20_01", 56,74 para "MAXCUT20_09" y 1077 para "MAXCUT100".

7) El problema **OneMax** o recuento de bits (Schaffer y Eshelman 1991), es un problema muy simple que consiste en maximizar el número de unos que contiene una cadena de bits. Formalmente, este problema se puede describir como encontrar una cadena $\vec{x} = \{x1, x2, \ldots, x_n\}$ con $x_i \in \{0, 1\}$ que maximice la función. La función f_{OneMax} tiene $(n+1)$ posibles valores diferentes, que están distribuidos multimodalmente. Para definir una instancia de este problema, sólo necesitamos definir la longitud de la cadena de *bits (n)*. Para un n dado, la solución óptima al problema es una cadena de n unos, es decir, se les fija el valor uno a todos los *bits* de la cadena.

8) El problema **MTTP (**Centre y Stinson 1985). Es un problema de planificación de tareas en el que cada tarea i del conjunto de tareas $T = \{1, 2, \ldots, n\}$ tiene una longitud l_i - el tiempo que tarda en ejecutarse- un límite de tiempo (o *deadline*) di -antes del cual debe ser planificada- y un peso w_i . El peso es una penalización que debe ser añadida a la función objetivo en el caso de que la tarea permanezca sin estar planificada. Las longitudes, pesos, y límites de tiempo de las tareas son todos números enteros positivos. Planificar la tarea de un sub-conjunto S de T consiste en encontrar el tiempo de comienzo de cada tarea en S, de forma que como mucho se realiza una tarea cada vez y que cada tarea termina de realizarse antes de su límite de tiempo.

El objetivo del problema es minimizar la suma de los pesos de las tareas no planificadas. Por tanto, la planificación óptima minimiza la función.

Hemos utilizado para nuestro estudio tres diferentes instancias (Khuri et al. 1994): "MTTP", "MTTP100" y "MTTP200", de tamaños 20, 100 y 200, y valores óptimos de 0,02439; 0,005 y 0,0025 respectivamente.

6. Experimentación

En esta sección se presentan y analizan los resultados obtenidos por los distintos algoritmos de trayectoria descriptos en la Sección 4 aplicado para resolver un conjunto de problemas académicos de optimización combinatoria. Utilizamos la herramienta Eclipse Kepler versión 2.0 con procesador AMD Athlon (tm) II P320 Dual-Core Processor 2.10 GHz y memoria RAM de 2.00 GB (en Windows 7 Starter) para implementar los algoritmos estudiados.

Hemos organizado esta sección en dos sub-secciones. En la primera parte exponemos los parámetros utilizados para la ejecución de los algoritmos, mientras que en la segunda estudiamos los resultados obtenidos por los diferentes algoritmos sobre todos los problemas.

A. Parametrización

La parametrización se realizo teniendo en cuenta una serie de pruebas experimentales para encontrar y establecer la configuración adecuada en cada algoritmo. Además, se tuvo en cuenta también los valores reportados en la literatura. Los detalles se resumen a continuación:

- Los algoritmos de TS, SA y VNS generan inicialmente 1 solución que es utilizado para realizar el análisis del problema a ejecutar.
- Cada solución se genera de manera aleatoria con valores binarios.
- El número máximo de evaluaciones utilizadas en los tres algoritmos (TS, SA y VNS) fue de 8000000.
- La longitud de la solución es una variable que se establece automáticamente de acuerdo al problema a ejecutar.
- En SA el parámetro de la temperatura inicial se ha establecido con el valor de 1000 buscando que a altas temperaturas en el enfriamiento simulado se diversifique aceptando las soluciones malas con altas probabilidades; y que a bajas temperaturas explorara esas soluciones encontradas a altas temperaturas, aceptando con una baja probabilidad pocas soluciones malas.
- En TS hemos inicializado el tamaño de la lista tabú de forma automática que se establece de acuerdo a la longitud de la solución del problema a resolver.
- En el algoritmo VNS se configura k_{max} con el 30% del tamaño de una solución que indica el número máximo de vecinos a explorar. En cada iteración, la solución vecina se perturba de manera aleatoria con tamaño k donde inicialmente comienza con el valor 1 y luego de cumplir determinadas condiciones va cambiando hasta alcanzar el valor de k_{max}, para obtener una solución mejorada.
- En VNS generemos una vecindad (para obtener la solución mejorada) con tamaño que es igual a 5 veces de la longitud de la solución.

B. Resultados Experimentales

En esta parte se ha hecho un análisis sobre la performance de los algoritmos para resolver la batería de problemas utilizada. Hemos realizado para cada algoritmo y problema 30 corridas independientes donde los mejores resultados se resaltan con **letra negrita**. Por otro lado, mediante la herramienta SPSS 15.0 para Windows se logro obtener resultados estadísticos relacionados a la media, mediana y pruebas de diferentes tipos (paramétricas y no paramétricas).

La Tabla 1 muestra el porcentaje (%) de éxito obtenido por cada algoritmo para cada problema. En dicha tabla podemos notar que SA ha obtenido en 8 de las 12 instancias un 100% de éxito, VNS ha obtenido en 6 de las 12 instancias un 100% de éxito y TS ha obtenido en 3 de las 12 instancias un 100% de éxito. Al final de la tabla hemos calculado el promedio de éxito general para cada uno de los algoritmos donde podemos observar que VNS cuenta con el mayor promedio de éxito (69%), le sigue SA (con 67%) y por último TS (con 33%).

INSTANCIA	TS	SA	VNS
COUNTSAT	10%	**100%**	7%
ECC	37%	**100%**	53%
FMS	0%	0%	0%
MAXCUT100	**100%**	**100%**	37%
MAXCUT20_01	17%	**100%**	**100%**
MAXCUT20_09	10%	**100%**	**100%**
MMDP	0%	0%	0%
MTTP	23%	**100%**	**100%**
MTTP100	0%	0%	**100%**
MTTP200	0%	0%	70%
OneMax	**100%**	**100%**	**100%**
P-PEAK	**100%**	**100%**	**100%**
Promedio	33%	67%	**69%**

Tabla 1: Porcentaje de éxito en los resultados obtenidos para TS, SA y VNS

A continuación, las siguientes tablas se agrupan de acuerdo a los algoritmos que encontraron un 100% de éxito para hacer un estudio más detallado de los mismos.

En base a la información mencionada anteriormente, tenemos entonces que las Tablas 2, 3 y 4 muestran el valor de la mediana para variables de rendimiento de evaluaciones para alcanzar el óptimo y el tiempo demandado por cada algoritmo en cada instancia de problema. Donde podemos observar en la Tabla 2 los resultados de algoritmos TS y SA para la instancia MAXCUT100 notamos que TS es más eficiente para las dos variables de rendimiento número de evaluaciones y tiempo. Sin embargo, en la Tabla 3 el algoritmo SA tiene los mejores valores para ambas variables de rendimiento en tres instancias (MAXCUT20_01, MAXCUT20_09 y MTTP). Por último, en la Tabla 4 se puede observar que TS obtiene en mediana el menor número de evaluaciones para las dos instancias analizadas (OneMax y P-PEAK). VNS mejora a TS y SA en cuanto a tiempo requerido para obtener el óptimo en las dos instancias analizadas.

	TS		SA	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
MAXCUT100	**23180**	**12027**	1488513	141828

Tabla 2: Medianas de las variables de rendimiento (evaluaciones y tiempo) obtenidas por TS y SA

	SA		VNS	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
MAXCUT20_01	**700**	**109**	811	235
MAXCUT20_09	**2237**	**188**	2712	805
MTTP	**915**	**133**	1550	454

Tabla 3: Medianas de las variables de rendimiento (evaluaciones y tiempo) obtenidas por SA y VNS

	TS		SA		VNS	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
OneMax	**2891**	16864	393661	12079	3073	**539**
P-PEAK	**404**	719	1561	391	467	**305**

Tabla 4: Medianas de las variables de rendimiento (evaluaciones y tiempo) obtenidas por TS, SA y VNS

Para hacer un análisis más profundo de los resultados obtenidos por los algoritmos aplicaremos *test* estadísticos. Recordemos que para determinar el tipo de *test* a aplicar (paramétrico o no paramétrico) debemos verificar que se cumplan tres condiciones: Independencia, Normalidad y Homocedasticidad. La independencia se cumple ya que se trata

de ejecuciones independientes de cada algoritmo. Para verificar la normalidad aplicaremos el Test de Kolmogorov-Smirnov y para la homocedasticidad el Test de Levene.

En caso de cumplir con estas tres condiciones se aplican Test Paramétricos (T-Test para los resultados de dos algoritmos y ANOVA para los resultados de más de 2 algoritmos). En caso de no cumplir alguna de las tres condiciones se aplican Test No Paramétricos U de Mann-Whitney (para dos algoritmos) o el Test de Kruskal-Wallis (para más de dos algoritmos). En todos los casos el nivel de significancia es $p=0,05$ usando SPSS.

En las Tablas 5, 6 y 7 se muestran el Test de Kolmogorov-Smirnov y su valor de *p*. Utilizamos el símbolo "*" para indicar que los resultados obtenidos por el algoritmo para cada problema no cumplen con la condición de normalidad. En la Tabla 5 vemos que ni los resultados de TS ni de SA para ambas variables de rendimiento tienen una distribución normal. En la Tabla 6 podemos observar que solamente para SA en la variable de rendimiento tiempo y en dos de las instancias (MAXCUT20_09 y MTTP) los resultados obtenidos tienen una distribución normal. En la Tabla 7 podemos observar que cinco de los resultados no tienen una distribución normal.

	TS		SA	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
MAXCUT100	(*)0,00	(*)0,03	(*)0,00	(*)0,00

Tabla 5: Test Kolmogorov-Smirnov aplicado a las variables evaluación y tiempo en TS y SA

	SA		VNS	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
MAXCUT20_01	(*)0,00	(*)0,00	(*)0,00	(*)0,00
MAXCUT20_09	(*)0,00	0,20	(*)0,00	(*)0,00
MTTP	(*)0,00	0,06	(*)0,00	(*)0,00

Tabla 6: Test Kolmogorov-Smirnov aplicado a las variables evaluación y tiempo en SA y VNS

	TS		SA		VNS	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
OneMax	0,20	0,20	0,11	0,08	0,20	(*)0,00
P-PEAK	0,17	(*)0,00	(*)0,00	(*)0,00	0,20	(*)0,00

Tabla 7: Test Kolmogorov-Smirnov aplicado a las variables evaluación y tiempo en TS, SA y VNS

En las Tablas 8, 9 y 10 muestran los resultados de la aplicación del Estadístico de Levene y utilizamos el símbolo "*" para indicar que el resultado obtenido para el conjunto de datos no cumple con la condición de homocedasticidad.

	Estadistico de Levene	
Instancia	**Evaluación**	**Tiempo**
MAXCUT100	(*)0,00	(*)0,00

Tabla 8: Test de Levene aplicado a variables de evaluación y tiempo de los algoritmos TS y SA

Instancia	Estadistico de Levene	
	Evaluación	Tiempo
MAXCUT20_01	(*)0,00	(*)0,00
MAXCUT20_09	(*)0,00	(*)0,00
MTTP	(*)0,00	(*)0,00

Tabla 9: Test de Levene aplicado a variables de evaluación y tiempo de los algoritmos SA y VNS

Instancia	Estadistico de Levene	
	Evaluación	Tiempo
OneMax	(*)0,00	(*)0,00
P-PEAK	(*)0,00	(*)0,00

Tabla 10: Test de Levene aplicado a variables de evaluación y tiempo de los algoritmos TS, SA y VNS

Las tablas anteriores nos han permitido determinar que los resultados de los algoritmos no cumplen con las condiciones requeridas para aplicar Test Paramétricos. Por tanto, fue necesario aplicar Test No Paramétricos para determinar la existencia de diferencias estadísticamente significativas, por esta razón utilizaremos los Test U de Mann-Whitney o Test de Kruskal-Wallis según corresponda.

En las Tablas 11, 12 y 13 utilizamos el signo (+) para señalar la existencia de diferencias estadísticamente significativas entre los resultados obtenidos por los algoritmos y (-) en caso opuesto. En la Tabla 11 se puede apreciar que TS presenta diferencias significativas con respecto a SA para las dos variables de rendimiento. En la Tabla 12 podemos observar que SA presenta diferencias estadísticamente significativas con respecto a VNS, en cuanto a la variable de rendimiento tiempo. No presenta diferencias estadísticamente significativas en cuanto a la variable número de evaluaciones necesarias para alcanzar el valor óptimo. Y por último, en la Tabla 13 los valores obtenidos demostraron que existen diferencias entre los tres algoritmos. TS y VNS tienen diferencias estadísticamente significativas con respecto a SA en cuanto a la variable de rendimiento número de evaluaciones. VNS tiene diferencias estadísticamente significativas con respecto a TS y SA en cuanto a la variable de rendimiento tiempo requerido en alcanzar el óptimo en las instancias OneMax y P-PEAK.

INSTANCIA	TS		SA		U de Mann-Whitney	
	Evaluación	Tiempo	Evaluación	Tiempo	Evaluación	Tiempo
MAXCUT100	**23180**	**12027**	1488513	141828	(+)	(+)

Tabla 11: Test U de Mann-Whitney aplicado a las variables de rendimiento evaluación y tiempo en TS y SA

INSTANCIA	SA		VNS		U de Mann-Whitney	
	Evaluación	Tiempo	Evaluación	Tiempo	Evaluación	Tiempo
MAXCUT20_01	**700**	**109**	811	235	(-)	(+)
MAXCUT20_09	**2237**	**188**	2712	805	(-)	(+)
MTTP	**915**	**133**	1550	454	(-)	(+)

Tabla 12: Test U de Mann-Whitney aplicado a las variables de rendimiento evaluación y tiempo en SA y VNS

	TS		SA		VNS		Kruskal-Wallis	
INSTANCIA	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**	**Evaluación**	**Tiempo**
OneMax	**2891**	16864	393661	12079	3073	**539**	(+)	(+)
P-PEAK	**404**	719	1561	391	467	**305**	(+)	(+)

Tabla 13: Test Kruskal-Wallis aplicado a las variables de rendimiento evaluación y tiempo en TS, SA y VNS

Desde otro punto de vista se muestran gráficos de Box-Plot de algunas instancias seleccionadas.

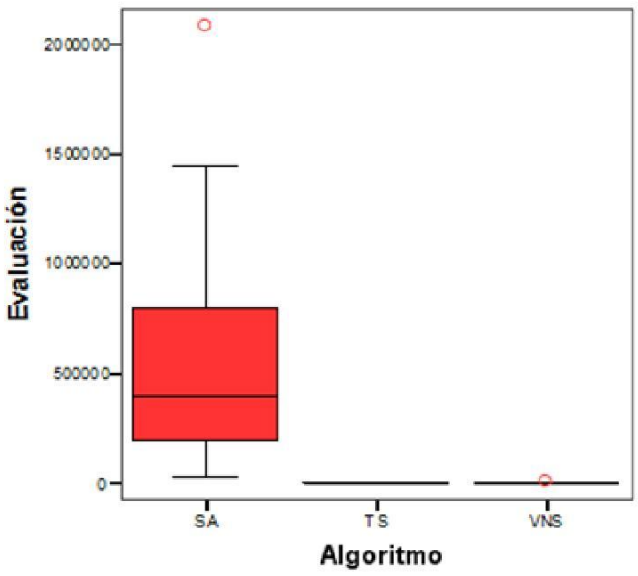

Figura 6 : Diagrama de caja(Boxplot) para la instancia OneMax de la variable evaluación en SA, TS y VNS

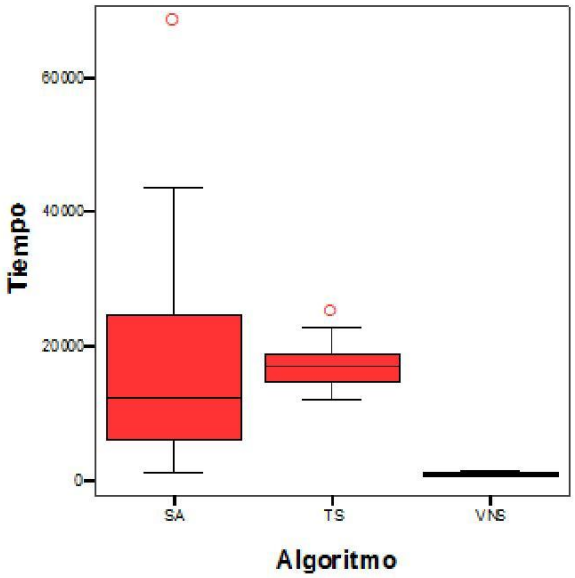

Figura 7: Diagrama de caja(Boxplot) para la instancia OneMax de la variable tiempo en SA, TS y VNS

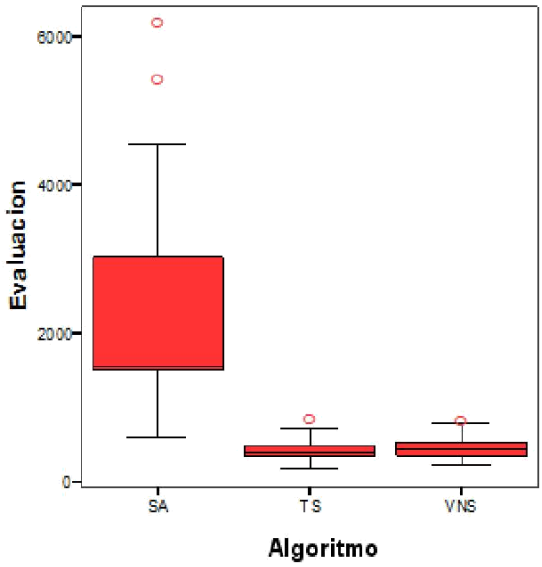

Figura 8: Diagrama de caja(Boxplot) para la instancia P-PEAK de la variable evaluación en SA, TS y VNS

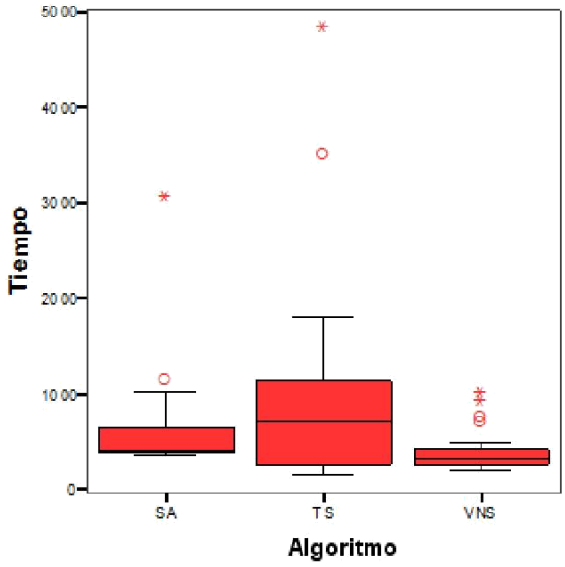

Figura 9: Diagrama de caja(Boxplot) para la instancia P-PEAK de la variable tiempo en SA, TS y VNS

La representación grafica de la Figura 6 muestra que los algoritmos TS y VNS

obtuvieron valores muy cercanos y compactos en cuanto al número de evaluaciones requerido para alcanzar el valor óptimo en la instancia OneMax comparados a los obtenidos por SA. Sin embargo en la Figura 7 podemos notar que VNS tiene los mejores resultados a diferencias de SA y TS que presenta una densidad notable en relación a la variable tiempo.

Las Figuras 8 y 9 muestran valores vinculados a la instancia P-PEAK de la evaluación y tiempo respectivamente, donde podemos observar que TS (en Figura 8) y VNS (en Figura 9) han logrado devolver resultados más robustos y compactos alrededor de la mediana.

7. CONCLUSIONES

En el desarrollo de esta investigación hemos realizado una comparación entre *Simulated Annealing, Tabu Search y Variable Neighborhood Search* aplicados a diversos problemas de optimización combinatoria.

Se analizo el porcentaje de éxito de los tres algoritmos para el conjunto de instancias de problemas seleccionados. De los experimentos realizados obtuvimos que, VNS tiene el mayor porcentaje de éxito. Además VNS logro tener diferencias estadísticamente significativas con respecto a las variables de rendimiento tiempo y evaluación en instancias como OneMax y P-PEAK . En instancias grandes como MTTP100 y MTTP200 VNS fue el que obtuvo un mayor porcentaje de éxito con respecto a los otros dos algoritmos.

No obstante, para instancias pequeñas SA tuvo diferencias estadísticamente significativas con respecto a VNS.

Podemos concluir con un nivel de confianza del 95% que para este conjunto de instancias, el algoritmo VNS tiene eliminar un mejor desempeño que TS y SA en instancias complejas. Y con un nivel de confianza del 95% podemos concluir que para el conjunto de instancias pequeños SA es mejor que VNS.

En trabajos futuros aplicaremos otras metaheurísticas de trayectoria sobre un conjunto de instancias ampliado.

8. AGRADECIMIENTOS

Se agradece la cooperación del equipo de proyecto del LabTEm y la Universidad Nacional de la Patagonia Austral, por el continuo apoyo y los valiosos aportes para el desarrollo de este trabajo.

9. REFERENCIAS

ALBA E. y Troya J. M, 2000. Cellular evolutionary algorithms: Evaluating the influence of ratio. In Parallel Problem Solving from Nature PPSN VI, pages 29–38. Springer.

ALONSO-Ayuso, A., Escudero, L. F., Martín-Campo, F. J., & Mladenović, N. (2015). A VNS metaheuristic for solving the aircraft conflict detection and resolution problem by performing turn changes. Journal of Global Optimization, 63(3), 583-596.

CENTRE Ch. B. R. y Stinson D. R, 1985. An introduction to the Design and Analysis of Algorithms. Charles Babbage Research Centre.

DROSTE S., Jansen T. y Wegener I, 2000. A natural and simple function which is hard for all evolutionary algorithms, volume 4. IEEE.

GAREY M. y Johnson D. Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H. Freeman, New York, 1979.

GLOVER F. y Kochenberger G, 2003. Handbook of Metaheuristics , Kluwer Academic Publishers.

GLOVER, F. y Laguna M. 1997. Tabu Search, Kluwer Academic Publishers.

GLOVER, F. 1986 "Future Paths for Integer Programming and Links to Artificial Intelligence", Computers and Operations Research. 5. 533-549.

GOLDBERG D. E., Deb K., y Horn J., "Massively multimodality, deception and genetic algorithms," in Proceedings of the Parallel Problem Solving from Nature, PPSN II, R.

MÄNNER and B. Manderick, Eds. 1992, pp. 37–46, North-Holland.

HANSEN, P. y Mladenovic, N.: Variable neighborhood search: Principles and applications. European Journal of Operational Research. 130 (3), 449–467 (2001).

KENNETH A De J., Mitchell A P. y Spears W. M, 1997. Using problem generators to explore the effects of epistasis. In ICGA, pages 338–345. Citeseer.

KHURI S., Bäck T. y Heitkötter J, 1994. An evolutionary approach to combinatorial optimization problems. In ACM Conference on Computer Science, pages 66–73.Citeseer.

KIRKPATRICK S., Gelatt C.D. y Vecchi M.P, 1983. Optimization by Simulated Annealing, Science 220, pp. 671.

MACWILLIAMS F. J. y Sloane NJ N. J. A, 1977. The Theory of Error-correcting Codes: Part 2, volume 16. Elsevier.

MARTÍ R. Algoritmos heurísticos en optimización combinatoria. Departamento de Estadística e Investigación Operativa, Facultad de Ciencias Matemáticas. Universidad de Valencia, España 2003.

MELIÁN B. y Glover F. 2003. Búsqueda Tabú. Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial 19: 29-48.

MISEVIČIUS, A. (2015). Using iterated tabu search for the traveling salesman problem. Information technology and control, 32(3).

SCHAFFER D. J. y Eshelman L. J, 1991. On crossover as an evolutionarily viable strategy. In ICGA, volume 91, pages 61–68.

SEN, G., Krishnamoorthy, M., Rangaraj, N., & Narayanan, V. (2016). Mathematical models and empirical analysis of a simulated annealing approach for two variants of the static data segment allocation problem. Networks.

SICILIA, J. A., Quemada, C., Royo, B., & Escuín, D. (2016). An optimization algorithm for solving the rich vehicle routing problem based on Variable Neighborhood Search and Tabu Search metaheuristics. Journal of Computational and Applied Mathematics, 291, 468-477.

TSUTSUI S., Ghosh A., Corne D. y Fujimoto Y, 1997. A real coded genetic algorithm with an explorer and an exploiter populations. In ICGA, pages 238–245, 1997.

WANG, S., Zuo, X., Liu, X., Zhao, X., & Li, J. (2015). Solving dynamic double row layout problem via combining simulated annealing and mathematical programming. Applied Soft Computing, 37, 303-310.