Resumen

Las metaheurísticas son descripciones de heurísticas con un alto grado de generalidad. Son utilizadas frecuentemente, para obtener algoritmos eficientes que permitan obtener soluciones de cierta calidad para problemas de optimización y búsqueda. Suelen aplicarse en problemas cuya complejidad computacional provoca que los métodos exactos consuman una gran cantidad de recursos computacionales.

En las metaheurísticas se describen estrategias de búsqueda de soluciones de una forma lo suficientemente general como para que una misma metaheurística pueda ser aplicada en problemas muy distintos. Una determinada metaheurística puede ser utilizada para aplicaciones de ruteo, asignación de recursos, o secuenciamiento de tareas, por ejemplo.

La dificultad que se plantea reside en que el desarrollador de aplicaciones difícilmente puede reutilizar su código de una aplicación a otra. De forma general, no es posible distinguir entre el código propio de la aplicación concreta al código correspondiente a la metaheurística empleada.

El objetivo del presente proyecto es justamente aprovechar esta diferenciación teórica entre metaheurística y aplicación concreta. El desarrollador de aplicaciones debería poder beneficiarse de reutilizar el conocimiento de las metaheurísticas - encapsulado en componentes de software - para poder concentrarse en su aplicación concreta específica a resolver.

En este proyecto se identifican tres áreas de desarrollo: el código de las aplicaciones concretas desarrolladas, el código asociado a las metaheurísticas (código que debería ser reutilizable), y una tercer área de desarrollo que denominamos “framework”. El framework consiste en una capa de código que provee determinadas clases e interfases, estableciendo algunas “reglas de juego” orientadas a estandarizar algunos conceptos, facilitando los intercambios entre metaheurísticas y aplicaciones.

El resultado final es el desarrollo del framework propiamente dicho, y la implementación de dos metaheurísticas y dos problemas de optimización combinatoria muy populares. Ambos problemas son resueltos utilizando una y otra metaheurística, y realizando una combinación de ambas metaheurísticas. En esta experiencia es que se intenta mostrar las posibilidades de reutilización de los desarrollos realizados.

Adicionalmente el framework ofrece algunas herramientas de utilidad para el desarrollador de aplicaciones. Estas incluyen código ya desarrollado para ayudar a depurar aplicaciones y manejar estadísticas de ejecución, entre otros.

Índice

1 Introducción

El presente trabajo constituye un Proyecto de Grado del Departamento de Investigación Operativa del Instituto de Computación, perteneciente a la Facultad de Ingeniería de la República Oriental del Uruguay.

El objeto del proyecto fue planteado por el Departamento de Investigación Operativa con el fin de contar con una herramienta para aminorar los tiempos y costos de implementar las diferentes metaheurísticas que permita las resoluciones de Problemas de Optimización combinatoria [MHE01]. Además, se desea que esta herramienta permita la combinación e integración de las diferentes metaheurísticas entre sí.

A continuación se presentan diferentes definiciones a términos utilizados en el documento, para una mayor comprensión de los mismos en los siguientes capítulos.

Un *problema de optimización* es un problema de la forma [MHE05]:

$$\begin{cases} \min f(\underset{\circ}{x}) \\ \text{tal que } x \in D \end{cases}$$

Se suele utilizar la siguiente nomenclatura:

- $\breve{x}(x_1, x_2, ..., x_n)$ el conjunto de *variables del problema*
- D es el *espacio de soluciones factibles*
- $f : \breve{x} \sim f(\breve{x})$ es la *función objetivo*
- El *valor óptimo* de f es $f_0 = \min\{f(\breve{x}) : \breve{x} \in D\}$
- El *conjunto de soluciones óptimas* es $S_0 = \{\breve{x} \in D : f(\breve{x}) = f_0\}$

Un problema de *optimización combinatoria* es un problema de optimización en el que se cumple adicionalmente:

- Para todo i, $x_i \in D_i$ siendo D_i un conjunto discreto (finito o infinito).

La palabra *heurística* deriva del griego heuriskein, que significa "encontrar" o "descubrir". Las heurísticas son técnicas que buscan soluciones de buena calidad (de valor cercano al óptimo) a un costo computacional razonable, aunque sin garantizar la optimalidad de las mismas. En general, ni siquiera se conoce el grado de error.

Por otro lado las *metaheurísticas* [MHE02, MHE03] son heurísticas de "mas alto nivel", se pueden definir como estrategias que guían el proceso de búsqueda, incluyendo en general heurísticas subordinadas de uso genérico (no específicas para una clase de problemas).

Dichas estrategias son utilizadas frecuentemente para obtener algoritmos eficientes que permitan obtener soluciones de cierta calidad para problemas de optimización y búsqueda. Suelen aplicarse en problemas cuya complejidad computacional provoca que los métodos exactos consuman una gran cantidad de recursos computacionales.

Entre las metaheurísticas más conocidas se encuentran Simulated Annealing [RSI01..06], Tabu Search [TSE01..06], Algoritmos Genéticos [AGE01..10], Búsqueda con Vecindades Variables [BVV01, BVV02] y Sistemas de Hormigas [CHO01..11]. Estas descripciones de algoritmos tienen elementos en común. Por ejemplo, el uso de métodos constructivos, de búsqueda local, la presencia de una función objetivo, entre otros.

1.1 Contexto y antecedentes

Como se explica anteriormente el Departamento de Investigación Operativa requiere la creación de una herramienta la cual facilite el desarrollo de aplicaciones de metaheurísticas para problemas de Optimización combinatoria. Además se requiere que la herramienta a desarrollar permita la integración, conjunción y creación de diferentes metaheurísticas de forma sencilla y rápida. Teniendo en cuenta este contexto se procede a la investigación de las diferentes herramientas existentes con estos fines y las posibilidades de uso de las mismas.

Actualmente hay desarrolladas varias herramientas referentes a metaheurísticas. Dentro de estas se encuentran las más conocidas como la biblioteca MALLBA [MAL01], GALib [GAL01], EasyLocal++ [ELO01], TEA [TEA01..03], etc. Las diferentes herramientas encontradas, a pesar de contar con diversos diseños y formas de programación muy interesantes, en general se centraban sobre diferentes subgrupos de los tipos de Metaheurísticas. Por ejemplo se tiene la herramienta GALib que fue creada para el uso de Algoritmos Genéticos, o EasyLocal++ la cual fue creada con base a la búsqueda local. Si bien todas estas herramientas son válidas, lo que se pretende es crear una herramienta de uso más general y no tan centrado sobre un tipo de metaheurística en particular, por lo tanto se tomaron ideas de los distintos diseños para crear uno nuevo que - en principio - conformaría una herramienta más completa.

1.2 Definición del Objetivo del Proyecto

Como se puede observar de las definiciones anteriormente realizadas los conceptos de "metaheurística" y "aplicación de la metaheurística para resolver problemas de optimización combinatoria" tienen un alto grado de generalidad, por lo tanto se pueden notar claramente diferenciados. El objetivo de este proyecto es explotar esa diferenciación. Se desea diseñar e implementar un ambiente de trabajo para la implementación de este tipo de técnicas. El uso de este ambiente fija como objetivo reducir los tiempos dedicados a la programación, así como a la creación de código legible y portable, pero a la vez eficiente.

Adicionalmente se tienen dos restricciones planteadas por el cliente - el Departamento de Investigación Operativa - como por ejemplo se desea que la herramienta funcione tanto en ambientes Windows como Unix [Unix01], y todo el código desarrollado deberá ser en C/C++ [CPP01].

1.3 Requerimientos

El relevamiento de los requerimientos para la herramienta se realiza a través de reuniones con los docentes y con el estudio de los diferentes softwares similares existentes.

Al relevar los requerimientos, el foco de atención se centra en identificar los siguientes elementos:

- Los componentes principales de las soluciones.
- Los conceptos utilizados.
- Las distintas responsabilidades en juego dentro de las soluciones de metaheurísticas.
- La variedad en cuanto a distintas estrategias de resolver los problemas más comunes.

En particular, para el software desarrollado se seleccionan dos problemas muy populares de optmización combinatoria y dos metaheurísticas a aplicar. Los problemas seleccionados son Quadratic Assignment Problem [QAP01] y Traveling Salesman Problem [TSP01]. Las metaheurísticas seleccionadas son Tabu Search [TSE01..06] y GRASP [GRA01].

En particular se trabaja con los problemas Quadratic Assignment Problem y Traveling Salesman Problem, los cuales son utilizados tanto como para validar los requerimientos, como para la verificación de la herramienta.

También entre los requerimientos del proyecto es planteada la oportunidad de resolver los problemas seleccionados con una nueva metaheurística que consista en combinar GRASP con Tabu Search.

La siguiente tabla muestra en filas los problemas atacados (QAP y TSP) y en columnas los métodos de resolución utilizados:

	TabuSearch	GRASP	GRASP+TS
QAP	√	√	√
TSP	√	√	√

1.4 Diseño y método de solución

El proceso de diseño consistió en crear una estructura de clases e interfases, de forma tal que se respeten los límites de responsabilidades relevados en los requerimientos. El diseño incluye consideraciones sobre dependencias y visibilidad, así como especificar las operaciones públicas de las distintas clases e interfases.

1.5 Conclusiones

Se desarrollaron las soluciones de QAP y TSP, intercambiando sus componentes. De esta forma se verifica que realmente los solvers sean independientes del problema planteado. Adicionalmente se combinan los solvers GRASP y TabuSearch para crear un método de resolución que utilice ambos, mostrando de esta forma la factibilidad de la integración de diferentes solvers unos con otros.

En la sección de Conclusiones de nuestro informe se presentan:

- Un mapa de los componentes desarrollados y la cantidad de líneas de código como indicador del esfuerzo asociado.

- Un análisis que contrasta los objetivos definidos para el proyecto con el trabajo realizado
- Un análisis de nuestro desarrollo según atributos de calidad del software comúnmente aceptados.

1.6 Organización del informe

El presente informe consta de seis capítulos. El Capitulo 1 es el capítulo de introducción al proyecto. En este capítulo se hace una breve reseña al contexto, objetivos, metodología de trabajo y resultados obtenidos.

El Capitulo 2 muestra la descripción del proyecto a realizar. En el mismo se describen los aspectos más importantes a tener en cuenta al planificar el proyecto, presentando las motivaciones y objetivos del mismo.

En el Capítulo 3 se muestran los requerimientos relevados. Son relevados los distintos conceptos existentes y cómo son atribuidas las responsabilidades en juego, y se fija un alcance para nuestro proyecto.

En el Capítulo 4 el informe avanza a la especificación funcional y el diseño del proyecto. En este Capítulo se pueden observar con detenimiento cómo se trabajó sobre cada uno de los diferentes requerimientos funcionales y el diseño que se aplicó para la resolución del sistema a desarrollar.

Dentro del Capitulo 5 se presentan los ejemplos de problemas y metaheurísticas desarrollados.

Por último en el Capítulo 6 se presentan las conclusiones obtenidas y el trabajo a futuro sugerido.

1.7 Anexos

En la sección de Anexos se encuentra el relevamiento realizado anteriormente como Estado del Arte, el cual identifica y describe los diferentes trabajos existentes en la materia y da una breve comparación de los mismos, atacando sus puntos fuertes y débiles.

También en dicha sección se contiene un apartado con las referencias utilizadas en el presente documento, dando fechas y autores de las mismas.

2 Descripción del proyecto a realizar

2.1 Motivación

El objetivo consiste en facilitar el trabajo del desarrollador de aplicaciones de optimización combinatoria. Conceptualmente es posible diferenciar el código de software que corresponde a la resolución de problemas específicos del código que corresponde a los métodos generales de resolución

El enfoque adoptado tiene que responder a las necesidades de quien desarrolla aplicaciones, y de quien provee los métodos de resolución en forma de código reutilizable. Cómo estructurar estos elementos es parte de la presente investigación. En resumen, se tienen tres actores: el desarrollador de aplicaciones, el proveedor de lo que serán denominados "solvers", y quienes proveen el "framework" (en este caso los desarrolladores del presente proyecto).

2.2 Definición del problema

De forma más estricta, se plantea el problema como un conjunto de objetivos sujeto a determinadas restricciones. Los objetivos se corresponden con las necesidades de los tres agentes (desarrollador, proveedor de solvers, framework), pero en última instancia todo lo desarrollado está al servicio del desarrollador de aplicaciones.

En el siguiente diagrama se muestran los diferentes actores y sus objetivos.

Las siguientes restricciones fueron planteadas por el cliente, el Departamento de Investigación Operativa:

- Se desea que el framework funcione tanto en ambientes Windows como Unix.
- Todo el código desarrollado deberá ser en C/C++.

2.3 Objetivos

En esta sección se explican los diferentes objetivos del proyecto. Como se muestra en la sección anterior, los objetivos están asociados los tres actores identificados: el desarrollador de aplicaciones, el desarrollador de solvers, y un actor que representa a quien realiza el mantenimiento del framework.

2.3.1 Objetivos del desarrollador de aplicaciones

Los objetivos para el desarrollador de aplicaciones son los siguientes:

Objetivo	Porqué el objetivo
Bajo costo de desarrollo y mantenimiento	Este debería ser uno de los retornos esperados por quien utilice la solución. De cierta forma valida y resume otros objetivos más específicos.
Calidad en el código fuente y en el proceso de desarrollo	El presente enfoque debe facilitar el diseño y programación de código fuente de calidad. Se entiende por atributos de calidad la correctitud, confiabilidad, robustez, performance, amigabilidad, verificabilidad, mantenibilidad, reusabilidad, portabilidad, comprensibilidad e interoperabilidad. A nivel del proceso de desarrollo, tiene que facilitar un proceso de desarrollo productivo y visible.
Construir prototipos para evaluar alternativas	Se tiene que considerar que, durante el proceso de desarrollo de aplicaciones de optimización combinatoria, existe una fase en la que se realizan pruebas con distintas configuraciones. Esto puede ser tanto como para ajustar parámetros como para evaluar el uso de distintos componentes posibles.
Facilidad para intercambiar y combinar solvers	Similar a lo anterior, pero aquí se hace explicita la necesidad particular de cambiar una metaheurística por otra, o combinar dos metaheurísticas para resolver un problema.
Concentrarse en su problema específico	Se cree que si se logra darle al desarrollador algunos aspectos ya solucionados, para que se concentre en entender su realidad particular, entonces va a ser más productivo.
Poder reutilizar sus propias soluciones	Reducir esfuerzos de desarrollo.

2.3.2 Objetivos del desarrollador de solvers

Los objetivos para el desarrollador de solvers son los siguientes:

Objetivo	Porqué el objetivo
Atender las necesidades del desarrollador de aplicaciones	Es el desarrollador de aplicaciones quien va a utilizar el solver.
Ofrecer código reutilizable	La reutilizabilidad de los solvers es su propiedad esencial. Un solver debe encapsular el conocimiento asociado a un determinado método de resolución, de forma independiente de los problemas específicos a los que se vaya a aplicar.
Ofrecer soluciones adaptables a las diversidad de necesidades específicas que pueda tener el desarrollador	Este objetivo es la contrapartida de la necesidad que tiene el desarrollador de dar solución a problemas reales. La presente solución tiene que considerar esta necesidad como un problema de ambas partes.

2.3.3 Objetivos para mantenimiento del framework

Los objetivos para quien realiza el mantenimiento del framework son los siguientes:

Objetivo	Porqué el objetivo
Crear un conjunto de reglas mínimo para que sea viable el intercambio de solvers	El desarrollador de aplicaciones debería, por ejemplo, implementar una estructura de vecindad para utilizar luego en distintas metaheurísticas.
Proveer algunas funcionalidades comunes, tales como manejo de parámetros, estadísticas y criterios de finalización más comunes	Esto tiene dos ventajas. La primera, es que directamente se evita que el desarrollador de aplicaciones tenga que solucionar algunos problemas que se repiten. La segunda ventaja es que si hay una forma estándar de hacer las cosas, al migrar de un solver a otro, no necesita dedicar esfuerzo para resolver un mismo problema de otra forma. Estas funcionalidades si bien son “estándar” están pensadas para ser extendidas por los solvers.
No entorpecer el desarrollo de solvers ni su utilización	Las interfases especificadas por el framework así como sus funcionalidades ofrecidas, no deben limitar las posibilidades de los solvers. Además, el framework no debe ser demasiado exigente en cuanto a cómo deben estructurarse las aplicaciones.
Framework que permite desarrollar nuevos metaheurísticas y problemas	Hace viable el uso del framework.

3 Requerimientos

3.1 Introducción

De los capítulos anteriores surge la distinción entre desarrollador de aplicaciones, solvers, y framework. En este capítulo de requerimientos se relevan los distintos conceptos que aparecen en las metaheurísticas. Ejemplos de estos conceptos son “estructura de vecindad” y “criterios de finalización”. Luego para cada uno de estos conceptos, se establece qué responsabilidades corresponden a los distintos agentes.

3.2 Áreas de desarrollo y relaciones de dependencia

A las tres áreas de desarrollo identificadas anteriormente, en el siguiente diagrama se establecen sus relaciones de dependencia:

3.2.1 Dependencias entre aplicaciones y solvers

En el diagrama aparece una relación de dependencia desde las aplicaciones hacia los solvers, esto es porque las aplicaciones desarrolladas van a recurrir a solvers. Sin embargo no existe dependencia desde los solvers hacia las aplicaciones: los solvers son reutilizables, no pueden depender de alguna aplicación particular.

3.2.2 Dependencias entre solvers y framework

De la misma forma que cada solver es construido independientemente de las aplicaciones que lo van a utilizar, el framework es independiente de los solvers que van a ser desarrollados sobre él.

Sí existe dependencia desde los solvers hacia el framework, esto es porque los solvers desarrollados cumplen los estándares provistos por el framework.

3.2.3 Dependencias entre aplicaciones y framework

Las aplicaciones son dependientes del framework, esto se da de dos formas. Primeramente, las aplicaciones dependen de los solvers, y los solvers del framework – y la relación de dependencia es una relación transitiva. Pero también existe una dependencia de forma directa: las aplicaciones desarrolladas siguen reglas de juego del framework. Por ejemplo, el framework establece las operaciones asociadas a una estructura de vecindad. Las aplicaciones pueden definir estructuras de vecindad, que luego van a ser utilizadas por uno u otro solver.

Como es de esperar, no existe dependencia desde el framework hacia las aplicaciones: éste no puede depender de una aplicación en particular.

3.3 Alcance del proyecto

El proceso de diseño consistió en idear una estructura de clases e interfases, y especificar sus operaciones, de forma tal que se respeten los límites y responsabilidades relevados en los requerimientos.

El proyecto incluye en su alcance el desarrollo del framework y ejemplos de metaheurísticas y aplicaciones.

Como fue presentado anteriormente, el alcance del proyecto está dado por el siguiente cuadro:

	TabuSearch	GRASP	GRASP+TS
QAP	√	√	√
TSP	√	√	√

3.4 Componentes del software de metaheurísticas

Algunos conceptos son aplicables a todas las metaheurísticas en general, por ejemplo el concepto de "solución". Otros conceptos aparecen sólo en algunas metaheurísticas, como por ejemplo el concepto de "lista de elementos candidatos" en metaheurísticas constructivas.

En esta sección se presentan los distintos conceptos que aparecen al recurrir a metaheurísticas. Se construyen tablas de tres columnas: concepto analizado, responsabilidades específicas del problema instanciado, y responsabilidades de la metaheurística en general.

Esta división de responsabilidades es más clara de ver con un ejemplo. Consideremos el concepto de "solución". La estructura de una solución es algo que siempre depende del problema a resolver: un problema de ruteo puede no tener la misma estructura que un problema de asignación de recursos. Sin embargo, las metaheurísticas toman decisiones en el proceso de búsqueda según el valor asociado a las soluciones recorridas. Por lo tanto las soluciones, en un nivel de abstracción elevado tienen un valor objetivo, y en un nivel de abstracción más bajo tienen su estructura de datos.

A continuación se presentan las tablas de tres columnas mencionadas.

Metaheurísticas en general		
Concepto	Específico del problema	Método general de resolución
Estructura de una solución	La estructura de una solución está determinada por el problema a resolver	Los métodos generales de resolución observan el valor objetivo de las soluciones y su factibilidad.
Función objetivo	La función objetivo está determinada por el problema a resolver	Los métodos observan el valor objetivo para comparar soluciones.
Criterios de finalización	Pueden existir algunos criterios de finalización dependientes del problema específico	Existen algunos criterios muy comunes, como el tiempo máximo, tiempo máximo sin mejorar, cantidad máxima de iteraciones, cantidad de iteraciones sin mejorar, o alcanzar un determinado valor objetivo. Cada metaheurística puede traer sus propios criterios de finalización adicionales.
Estadísticas de ejecución	Pueden existir algunos datos a incluir en las estadísticas, que sean específicos del problema a resolver	Existen algunos datos que son muy generales, como el tiempo y cantidad de iteraciones transcurridas. Para cada metaheurística pueden aparecer datos adicionales.
Diversificación e intensificación de la búsqueda	Cuándo es conveniente diversificar e intensificar es decidido a nivel del problema específico. El usuario programador ajusta en tiempo de ejecución los valores de parámetros que crea convenientes.	Cómo se realiza la intensificación y diversificación depende de la metaheurística aplicada.

Metaheurísticas de Trayectoria		
Concepto	Específico del problema	Método general de resolución
Función objetivo	La función objetivo está determinada por el problema a resolver	Por motivos de performance, es deseable observar la variación del valor objetivo asociada a una solución vecina, para no calcular toda la función desde cero.
Estructura de vecindad	Está determinada por el problema a resolver, muy relacionada con la estructura de la solución	Los métodos generales pueden decidir hacia qué vecino avanzar, si seguir recorriendo o no la estructura de vecindad
Solución inicial	Generar una solución inicial es responsabilidad del problema específico	Los métodos de trayectoria reciben una solución inicial para empezar su búsqueda
Desplazarse a un vecino	Ejecutar esta operación requiere conocer estructuralmente las soluciones	Los métodos indican hacia qué vecino la búsqueda se va a desplazar

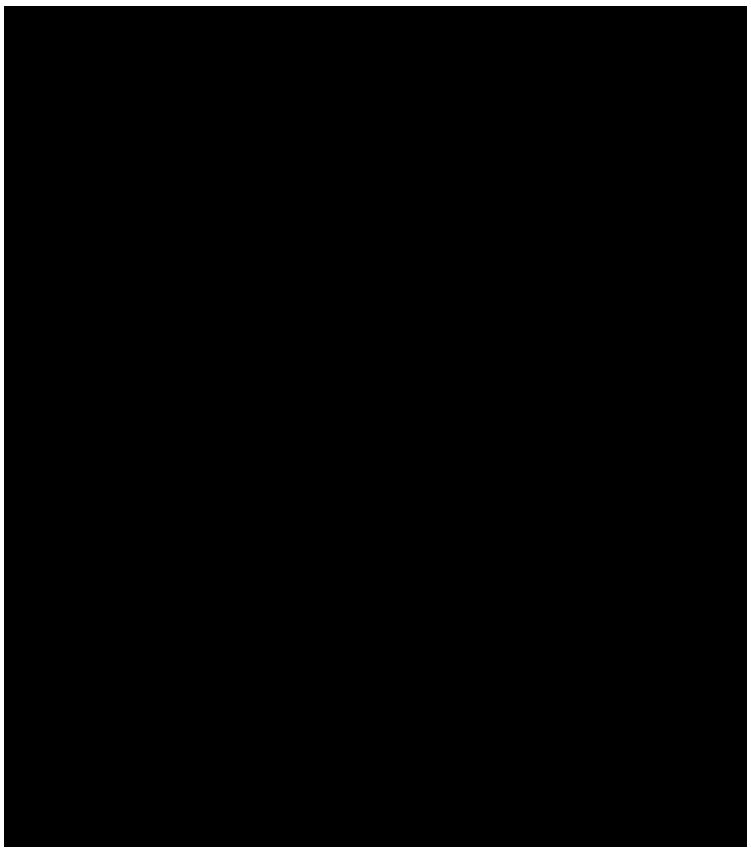
Metaheurísticas Constructivas (como GRASP)		
Concepto	Específico del problema	Método general de resolución
Componentes de una solución	En qué componentes se divide estructuralmente una solución, y qué elementos pueden utilizarse en cada componente	Los métodos generales observan los distintos componentes de la solución, evalúan candidatos, y deciden qué elementos agregar a la solución actualmente en construcción
Solución inicial vacía	Determinada por el problema específico	Los métodos reciben una solución inicial vacía como entrada para comenzar el proceso de construcción
Lista de elementos candidatos a solución actual	Esta lista debe ser provista por el código dependiente del problema específico, ya que es necesario conocer cómo se estructuran internamente las soluciones	Observando el costo incremental de los candidatos, los métodos seleccionan un el siguiente candidato para la construcción
Agregar un elemento a la solución actual	Quien conoce la estructura de soluciones es capaz de agregar un determinado elemento	Los métodos sólo deciden cuál elemento agregar
Evaluar si la solución está completa	Esto es posible evaluarlo a nivel de la estructura de la solución	Los métodos necesitan saber este dato de las soluciones que están construyendo
Estadísticas	-	Aparecen dos datos adicionales: el número de iteración global, y número de iteración de la búsqueda local

Metaheurística Tabu Search		
Concepto	Específico del problema	Método general de resolución
Lista tabú	Estructura de cada valor tabú. Cantidad de listas utilizadas. Decidir el largo de las listas tabú.	Las listas tabú tienen siempre, en un nivel de abstracción superior, el mismo comportamiento. Se producen inserciones, eliminaciones de tipo FIFO, y búsquedas.
Valor tabú	La estructura interna de datos de un valor tabú es definida a este nivel	Tabu Search utiliza valores tabú. Para estos valores necesita poder instanciarlos, copiar valores, verificar igualdades, y realizar búsquedas.
Vecino tabú	A nivel estructural, es posible ver cuáles son los "valores tabú" asociados a un vecino.	De forma general, se verifica si los valores tabú están presentes en sus respectivas listas tabú
Condiciones de aspiración	Las condiciones de aspiración son definidas a este nivel	El método necesita saber si un vecino cumple o no las condiciones de aspiración
Criterios de finalización	-	Aparece el nuevo criterio de finalización que consiste en terminar cuando todos los vecinos tienen el status tabú

4 Diseño y especificación funcional del Framework

El framework de por sí no incluye algoritmos de metaheurísticas. Sin embargo define y "estandariza" algunos conceptos a utilizar en el software de metaheurísticas. Sobre el framework es posible desarrollar componentes que son llamados "Solvers", y los desarrolladores de aplicaciones pueden reutilizar estos Solvers en sus aplicaciones.

El siguiente diagrama muestra la relación entre el framework, los Solvers y las aplicaciones:



Los Solvers - dado que son reutilizables - son independientes de las aplicaciones que los utilizan. De la misma forma se puede decir que el Framework es independiente de los Solvers, permitiendo que se desarrollen nuevos Solvers.

4.1 Elementos del Framework

El Framework está compuesto por clases e interfases reutilizables. Las interfases representan los conceptos estandarizados, tales como una solución o un explorador de vecindades. Las clases contienen código que pueden reutilizar tanto los desarrolladores de aplicaciones como los proveedores de Solvers.

Se ilustran a continuación los componentes del framework con el caso particular de una implementación de TSP:

En el diagrama anterior se muestra el framework, dos solvers (TabuSearch y GRASP) y una aplicación de TSP que utiliza ambos Solvers. Las interfases están en verde y las clases en naranja.

En las secciones siguientes se describe qué significan las interfases y clases del diagrama.

4.2 Proveedores y consumidores de metaheurísticas

El trabajo realizado está basado en el paradigma de componentes reutilizables. Los algoritmos de resolución de las metaheurísticas quedan encapsulados en "Solvers", donde un Solver es un componente que constituye una implementación de una metaheurística.

Un Solver va a ejecutar su algoritmo, y la ejecución del algoritmo va a alternar código propio del solver con código escrito por el desarrollador. Por ejemplo, cómo se calcula la función objetivo es algo que va a estar codificado por el desarrollador de la aplicación – mientras que las decisiones en el proceso de búsqueda suelen quedar del lado del solver.

Es posible identificar dos subsistemas: el código del solver, y el código escrito por el desarrollador. El intercambio entre estos dos subsistemas está inspirado en los patrones de diseño Façade Controller [Façade01] y Adapter [Adapter01]. El código del desarrollador tiene que contener una clase que implemente la interfase que se llama "Application": Esta interfase describe un conjunto de operaciones que los solvers pueden invocar para ejecutar las operaciones codificadas por el desarrollador.

El siguiente esquema muestra las interfases Solver y Application utilizadas en un caso de ejemplo.

En el diagrama anterior se aprecia cómo el framework tiene definidas las interfases Solver y Application. Luego un proveedor implementa su propio Solver, y especializa la interfase Application agregando algunas operaciones en TS_Application.

Los conceptos definidos en el framework son los siguientes:

Interfase	Descripción	A implementar por	Operaciones especificadas
Solver	Componente para ejecutar una metaheurística	Proveedor de solvers	Las instancias de solvers mantienen un link con una determinada clase creada por el usuario. Están especificadas las operaciones para establecer y consultar este link. Además los solvers tienen un método execute() para ejecutarse.
Application	Controlador de fachada del código del usuario programador	Desarrollador de aplicaciones	Un consumidor de un solver debe crear una clase que implemente esta interfase. Esta clase luego va a funcionar como controlador de fachada y adaptador entre el solver y el código del usuario. Las operaciones de Application van a ser invocadas por el solver que se utilice.

4.3 Soluciones

La interfase Solution describe las operaciones a ser implementadas por las soluciones.

El concepto definido es el siguiente:

Interfase	Descripción	A implementar por	Operaciones especificadas
Solution	Representa una solución	Desarrollador de aplicaciones	Obtener un string descriptivo, clonar y copiar instancias, verificar si es factible, y obtener el valor objetivo.

4.4 Exploración de estructuras de vecindad

Se dividen responsabilidades definiendo dos conceptos: explorador de vecindad, y movida. Un explorador de vecindad es una entidad que, dada una solución, puede identificar las distintas movidas posibles. "Explorer" es el nombre atribuido al explorador de vecindades y "Move" a una movida.

Los conceptos definidos son los siguientes:

Interfase	Descripción	A implementar por	Operaciones especificadas
Explorer	Explorador de vecindad	Desarrollador de aplicaciones	Operaciones para recorrer una estructura de vecindad. Los exploradores de vecindad devuelven posibles "movidas". El framework ya trae implementadas las estrategias first-improvement y best-improvement.
Move	Una movida hacia una solución vecina	Desarrollador de aplicaciones	Obtener el valor asociado a la movida, y aplicar a una solución. Además están las operaciones para clonar y copiar instancias.

4.5 Construcción de soluciones

Para construir soluciones se definen dos conceptos: "Builder", el constructor de vecindades, y "Element", que representa un elemento a agregar a la solución en construcción.

Los conceptos definidos son los siguientes:

Interfase	Descripción	A implementar por	Operaciones especificadas
Builder	Constructor de soluciones	Desarrollador de aplicaciones	Inicializar una solución nueva vacía, construir la lista de candidatos para la solución actual, consultar la lista de candidatos, aplicar un candidato, y verificar si la solución está completa.
Element	Elemento candidato de una solución	Desarrollador de aplicaciones	Obtener el costo incremental asociado. Además están las operaciones para clonar y copiar instancias.

4.6 Criterios de finalización

Es definido un concepto que es un "controlador" de condiciones de finalización. Cada solver tiene un controlador de este tipo asociado.

El concepto definido es el siguiente:

Interfase	Descripción	A implementar por	Operaciones especificadas
ExecutionLimits	Criterios de finalización	Proveedor de solver	Operaciones para establecer criterios de finalización

4.7 Estadísticas de ejecución

Aquí se definen dos interfases: una para manejo de estadísticas en general, y otra interfase que es utilizada cada vez que sea necesario extender el conjunto de datos manejado por las estadísticas.

Los conceptos definidos son los siguientes:

Interfase	Descripción	A implementar por	Operaciones especificadas
ExecutionInfo	Controlador de estadísticas	Proveedor de solver	Operaciones para manejar estadísticas
StatsAddIn	Complemento de estadísticas	Proveedor de solver y/o desarrollador de aplicaciones	Operaciones utilizadas para agregar más datos a las estadísticas estándar

4.8 Clases de utilidad provistas por el framework

El framework provee además las siguientes clases de uso general

Clase	Descripción
ParamReader	Clase utilizada para leer archivos de parámetros que cumplen con determinado formato
ExecutionInfoStd	Clase de ejemplo para utilizar, que implementa la interfase ExecutionInfo
ExecutionLimitsStd	Clase de ejemplo para utilizar, que implementa la interfase ExecutionLimits

4.9 Especificación de operaciones

Luego de ser presentados los distintos elementos que componen el framework, esta sección va un poco más en detalle especificando las operaciones de cada uno.

4.9.1 Interfase: Solver

La interfase Solver especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getApplication	-	Application	Retorna una referencia a la instancia del controlador de fachada del código de la aplicación desarrollada
setApplication	Application	-	Establece cuál es el controlador de fachada con quien va a interactuar el solver
execute	-	-	Inicia la ejecución del solver
getExecInfo	-	ExecutionInfo	Retorna una referencia al controlador de estadísticas utilizado por el solver
getExecLimits	-	ExecutionLimits	Retorna una referencia al controlador de criterios de finalización utilizados

En C++ la clase es la siguiente:

Las primeras tres operaciones son a implementar por el proveedor del solver. Las últimas dos operaciones, ya están implementadas y pueden ser redefinidas.

4.9.2 Interfase: Application

La interfase Application especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
initialize	-	-	El solver indica a la aplicación que va a comenzar su ejecución
afterStep	Referencia a controlador de estadísticas. Adicionalmente, un parámetro de salida booleano para terminar la ejecución.	-	El solver invoca esta operación luego de cada iteración realizada
afterImprovement	Igual a afterStep	-	El solver invoca esta operación cada vez que se encuentra una nueva mejor solución

En C++ la clase es la siguiente:

La interfase LocalSearchApplication hereda de Application. La operación adicional que es agregado es para proveer una solución inicial.

4.9.3 Interfase: Explorer

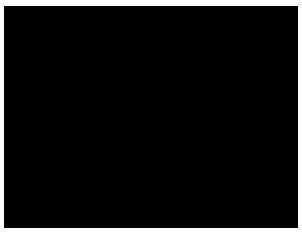
La interfase Explorer especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
initialize	Solución	-	El solver indica al explorador que se posicione en la solución dada
nextMove	-	Movida	Devuelve una movida, o NULL en caso de no haber más movidas
refresh	Movida	-	El solver indica al explorador que se ha aplicado la movida dada, y le pide que se actualice
getSolution	-	Solución	Devuelve la solución actual

Las siguientes operaciones ya vienen implementadas por el framework para la interfaz Explorer.

Nombre	Parámetros	Retorno	Descripción
getFirstMove	-	Movida	Devuelve la primer movida que mejora la solución actual, o NULL en caso de no encontrar ninguna
getBestMove	-	Movida	Devuelve la mejor movida
getDefaultMove	-	Movida	Invoca a getFirstMove. Esta operación se puede redefinir invocando a getBestMove, para predeterminar la búsqueda best-search.

En C++ la clase es la siguiente:



4.9.4 Interfase: Move

La interfase Move especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getValue	-	Valor	Retorna el valor asociado a la movida
copyFrom	Movida	-	Hace que la instancia actual tome los datos de la instancia referida
clone	-	Movida	Genera una nueva instancia idéntica
applyTo	Solución	-	Hace que la movida sea aplicada a la solución dada

En C++ la clase es la siguiente:

4.9.5 Interfase: Solution

La interfase Solution especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getDebugString	-	string	String que describe a la solución. Esta operación es a los efectos de facilitar la depuración.
clone	-	Solución	Genera una nueva instancia idéntica
copyFrom	Solución	-	Hace que la instancia actual tome los datos de la instancia referida
isFeasible	-	Booleano	Indica si la solución es factible o no
getValue	-	Valor	Retorna el valor asociado a la solución

En C++ la clase es la siguiente:

4.9.6 Interfase: Builder

La interfase Builder especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
startEmptySolution	-	-	El solver invoca esta operación para iniciar el proceso de construcción
buildElementsList	-	-	El solver solicita que se genere la lista con los elementos candidatos
getElementsCount	-	Entero	Retorna la cantidad de elementos de la lista de candidatos
getElement	Entero con el índice del elemento solicitado	Elemento	Retorna una referencia a un elemento de la lista de elementos candidatos
addElement	Elemento	-	Indica al constructor que se desea agregar el elemento dado
getSolution	-	Solución	Retorna una referencia a la solución del proceso de construcción
isComplete	-	Booleano	Indica si la solución está completa

En C++ la clase es la siguiente:

4.9.7 Interfase: Element

La interfase Element especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
clone	-	Elemento	Genera una nueva instancia idéntica
copyFrom	Elemento	-	Hace que la instancia actual tome los datos de la instancia referida
getCost	-	Valor	Devuelve el costo incremental asociado al elemento dado

En C++ la clase es la siguiente:

4.9.8 Interfase: ExecutionInfo

La interfase ExecutionInfo especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getCurrentStep	-	Entero	Retorna el número de iteración actual
getPartialStep	-	Entero	Retorna el número de iteración desde la última mejora
getTotalTime	-	Valor	Retorna la cantidad de segundos desde comenzada la ejecución
getPartialTime	-	Valor	Retorna la cantidad de segundos transcurridos desde la última mejora
getBestSolStepNumber	-	Entero	Devuelve el número de iteración asociado a la mejor solución encontrada
getBestSolTime	-	Valor	Devuelve la cantidad de segundos entre el comienzo y la mejor solución encontrada
getCurrentSolution	-	Solución	Retorna una referencia a una copia de la solución actual
getBestSolution	-	Solución	Retorna una referencia a una copia de la mejor solución encontrada
registerAddIn	StatsAddIn	-	Registra un complemento de estadísticas
setFileTrace	Ruta del archivo, nivel de trace	-	Establece un archivo de salida de estadísticas con un determinado nivel de detalle
setSOTrace	Nivel de trace	-	Establece un nivel de detalle para las estadísticas de salida estándar (Standard Output).

En C++ la clase es la siguiente:

El framework ya provee una implementación de esta interfase, la clase ExecutionInfoStd, que puede ser utilizada por los constructores de solvers. Las operaciones adicionales de esta clase son para actualizar los valores de las estadísticas.

Las estadísticas estándar incluyen los siguientes datos:

- Número de iteración actual
- Número de iteraciones desde última mejora global
- Tiempo total de ejecución
- Tiempo de ejecución desde última mejora global
- Valor actual

Los solvers y las aplicaciones de usuario pueden extender las estadísticas creando instancias que implementen la interfase StatsAddIn, que especifica el comportamiento de un complemento de estadísticas. Por ejemplo, el solver GRASP agrega un complemento de estadísticas que agrega dos datos adicionales: el número de iteración global, y el número de iteración dentro de la búsqueda local actual.

4.9.9 Interfase: ExecutionLimits

La interfase ExecutionLimits especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getTotalSteps	-	Entero	Retorna el máximo de iteraciones
getPartialSteps	-	Entero	Retorna el máximo de iteraciones consecutivas sin encontrar una mejor solución
getTotalTime	-	Valor	Retorna la cantidad de segundos máxima de ejecución
getPartialTime	-	Valor	Retorna la cantidad de segundos máxima sin que se encuentre una nueva mejor solución
getTargetValue	-	Valor	Retorna el valor objetivo con el cual se finaliza la ejecución
getTargetSense	-	-1 ó +1	Para uso del valor objetivo, especifica si se está minimizando o maximizando el valor
setTotalsteps	Entero	-	-
setPartialSteps	Entero	-	-
setTotalTime	Valor	-	-
setPartialTime	Valor	-	-
setTargetValue	Valor, sentido (-1 ó +1)	-	Establece el valor objetivo e indica si se está minimizando o maximizando.

En C++ la clase es la siguiente:

El framework ya provee una implementación de esta interfase, la clase ExecutionLimitsStd, que puede ser utilizada por los constructores de solvers. Las operaciones adicionales de esta clase son para chequear el cumplimiento de las condiciones de finalización.

4.9.10 Interfase: StatsAddIn

La interfase StatsAddIn especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
addHeader	Identificador de archivo abierto	-	Solicita que despliegue en el archivo las descripciones de los campos a agregar a las estadísticas
addStatsFile	Identificador de archivo abierto	-	Solicita que despliegue los valores de los datos en el archivo
addStatsSO	-	-	Solicita que despliegue los valores de los datos en la salida estándar

En C++ la clase es la siguiente:

4.9.11 Clase: ParamReader

La clase ParamReader ofrece las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
loadFile	Nombre de archivo	-	Carga los parámetros desde el archivo dado
hasParam	Nombre de parámetro	Booleano	Indica si existe un parámetro con el nombre dado
getValueInt	Nombre de parámetro	Entero	Devuelve un valor entero
getValueFloat	Nombre de parámetro	Decimal	Devuelve un valor decimal
getValueBool	Nombre de parámetro	Booleano	Devuelve un valor booleano
getValueStr	Nombre de parámetro	String	Devuelve un valor de tipo string
checkParam	Nombre de parámetro, más una parámetro de salida booleano	-	Se fija si existe el parámetro dado. En caso de no existir, despliega un mensaje de error en la salida estándar y establece el parámetro de salida en false. Esta operación sirve para simplificar el código del usuario cuando tiene que controlar la existencia de muchos parámetros.

En C++ la clase es la siguiente:

Esta clase tiene la capacidad de leer archivos de parámetros que cumplen un determinado formato. Este formato consiste en una sucesión del tipo <tipo, nombre, valor>. Un ejemplo de archivo es el siguiente:

Archivo: params.txt

```
int         quieto              1000000
int         maxiter             1000000
float       maxtime             30
int         iter_grasp_ts       100

string      problema    TSP
string      solver              grasp

string      archivo             QAP\Datos\lipa80a.dat
string      archivotsp          TSP\Datos\eil101.txt

int         semilla             1020
int         largo_tabu_qap      20
int         largo_tabu_tsp      5
float       alpha               0.5
bool        grasp_ts            true

string      salida              salida.txt
string      corridas            corridas\corridas.txt
string      stats               corridas\stats.txt
int         trace_pantalla      1
int         trace_archivo       1
```

5 Ejemplos de uso del Framework

5.1 Introducción

Están desarrollados como ejemplos dos Solvers que corresponden a las metaheurísticas GRASP [GRASP01] y Tabu Search [TS01] respectivamente, y dos aplicaciones de ejemplo que son los problemas QAP [QAP01] y TSP [TSP01] respectivamente.

Las combinaciones son las siguientes:

	TabuSearch	GRASP	GRASP+TS
QAP	√	√	√
TSP	√	√	√

La combinación GRASP+TS consiste en utilizar GRASP para iterar construyendo soluciones, pero la búsqueda local es realizada utilizando Tabu Search.

Esta tabla fue utilizada para verificar que el código desarrollado es reutilizable. Por ejemplo, consideremos la implementación de TSP con GRASP:

- Una parte del código corresponde al algoritmo de la metaheurística GRASP y es independiente de cualquier aplicación concreta, y ese código debería ser reutilizado al resolver QAP.
- De forma similar, otra parte del código corresponde a TSP independientemente de la metaheurística aplicada, y ese código debería ser reutilizado al resolver TSP con Tabu Search.
- Al intentar combinar GRASP y TabuSearch (tercer columna), debería ser posible reutilizar el código de TSP ya desarrollado para esas metaheurísticas.

En resumen, el software desarrollado ofrece las siguientes oportunidades de reutilización:

- El solver Tabu Search es independiente de los problemas que lo utilizan.
- Ídem para GRASP.
- Una parte del código de TSP es propio de TSP e independiente de cualquier solver: ese código es reutilizado en todas las implementaciones de TSP.
- Ídem para QAP.
- Al combinar GRASP y TabuSearch se reutiliza el código desarrollado correspondiente a las columnas anteriores de la tabla: código de QAP para TabuSearch, QAP para GRASP, TSP para Tabu Search, y TSP para GRASP.

5.2 Solver GRASP

5.2.1 Especificación funcional

El componente del solver GRASP provee las siguientes clases e interfases.

Tipo	Nombre	Descripción
Interfase	GRASP_Application	Heredada de la interfase Application, especifica las operaciones adicionales que debe implementar un consumidor del solver GRASP.
Clase	GRASP_Solver	Solver para la metaheurística GRASP
Clase	GRASP_Stats	Complemento que extiende las estadísticas básicas del framework, agregando el número de iteración global y número de iteración local.

El siguiente diagrama muestra las clases asociadas al solver GRASP.

La clase AplicacionTSP está incluida en el diagrama para mostrar un ejemplo de una clase que implemente la interfase GRASP_Application.

La clase GRASP_RCL es de uso interno del solver. Esta clase nunca es manipulada directamente por el desarrollador. La sigla RCL viene del concepto “Restricted Candidate List” de GRASP.

5.2.2 Especificación de operaciones

5.2.2.1 Interfase: GRASP_Application

Para la interfase GRASP_Application las operaciones especificadas son las siguientes.

Nombre	Parámetros	Retorno	Descripción
getBuilder	-	Builder	Retorna una referencia a la instancia de Builder que debe utilizar el solver.
getExplorer	-	Explorer	Retorna una referencia a la instancia de Explorer que debe utilizar el solver para realizar la búsqueda local.
afterNewElement	Solución actual, Número de componente agregado, Elemento agregado	-	Operación invocada por el solver luego de agregar cada elemento.

En C++ la clase es la siguiente:

5.2.2.2 Interfase: GRASP_Solver

Para la clase GRASP_Solver las operaciones especificadas son las siguientes. Además incluye las operaciones especificadas por la interfase Solver.

Nombre	Parámetros	Retorno	Descripción
setAlpha	Valor alpha	-	Parámetro alpha propio de la metaheurística GRASP
getAlpha	-	Valor alpha	Parámetro alpha de GRASP

En C++ la clase es la siguiente:

5.2.2.3 Interfase: GRASP_Stats

Para la clase GRASP_Stats, las operaciones se limitan a las especificadas en la interfase StatsAddIn.

En C++ la clase es la siguiente:

5.3 Solver TabuSearch

5.3.1 Especificación funcional

El componente del solver Tabu Search provee las siguientes clases e interfases.

Tipo	Nombre	Descripción
Interfase	TS_Application	Heredada de la interfase LocalSearchApplication, especifica las operaciones adicionales que debe implementar un consumidor del solver TS.
Clase	TS_Solver	Solver para la metaheurística Tabu Search

El siguiente diagrama muestra las clases asociadas al solver Tabu Search.

Las clases TabuValueTSP y AplicacionTSP están incluidas en el diagrama como ejemplos de clases que implementan las interfases respectivas TabuValue y TS_Application.

Las clases TabuList y TabuValueNode son utilizadas internamente por el TS_Solver, y estas clases no son manipuladas directamente por el desarrollador de aplicaciones.

La clase TS_ExplorerWrapper es una funcionalidad adicional ofrecida por el solver de Tabu Search. Esta clase está pensada para combinar Tabu Search con otras metaheurísticas, en las cuales se desea realizar una búsqueda local aplicando Tabu Search. Esto está explicado más adelante.

5.3.2 Especificación de operaciones

5.3.2.1 Interfase: TS_Application

La interfase TS_Application especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getExplorer	-	Explorer	Retorna una referencia a la instancia de Explorer que debe utilizar el solver para realizar la búsqueda local.
readTabu	Movida, Solución, Número de lista, número de lectura. Está además el parámetro de salida que representa el valor tabú leído.	-	Lee un valor tabú de una movida candidata.
saveTabu	Movida, Solución, Número de lista, número de lectura. Está además el parámetro de salida que representa el valor tabú leído.	-	Lee el valor tabú de la movida realizada, para guardarlo en su lista correspondiente.

Las operaciones readTabu y saveTabu, si bien son similares, están separadas. La primer operación es invocada por el solver cuando está explorando las posibles movidas. La segunda operación es invocada luego de realizada una movida, para agregar los correspondientes valores tabú a las listas.

En algunas aplicaciones, para una lista tabú en cada movida se realiza más de una inserción. Por ejemplo, en QAP una movida puede consistir en intercambiar dos fábricas. Luego, hay dos emplazamientos que ingresan a la lista tabú. El parámetro “número de lectura” sirve justamente para que, en casos como este, el solver diga si está haciendo la primer o segunda lectura.

En C++ la clase es la siguiente:

5.3.3 Interfase: TabuValue

La interfase TabuValue especifica las siguientes operaciones.

Nombre	Parámetros	Retorno	Descripción
getKey	-	Long	Devuelve un entero largo utilizado para tablas internas de hash. Esta operación es para permitir que los valores tabú tengan búsquedas en tiempos de orden 1.
clone	-	TabuValue	Devuelve una instancia idéntica nueva
copyFrom	Referencia a la instancia desde la cuál se desea copiar	-	Copia datos de la instancia referida
isEqual	Referencia a la instancia a comparar	bool	Devuelve True si la instancia por referencia tiene los mismos datos

En C++ la clase es la siguiente:

5.3.4 Clase: TS_Solver

La clase TS_Solver posee las siguientes operaciones, además de las especificadas por la interfaz Solver.

Nombre	Parámetros	Retorno	Descripción
createTabuList	Largo inicial, Valores por movida, e instancia de ejemplo de los valores tabú a agregar	Número entero que identifica a la lista	Crea una lista tabú a ser mantenida por el solver. Luego de creada la lista, el solver va a solicitar a la instancia de Application que lea los valores tabú asociados a las movidas.
changeTabuSize	Número de lista tabú, nuevo tamaño	-	Modifica el largo de una lista tabú
isTabu	Movida, Solución	Booleano	Devuelve si la movida es o no tabú. No es necesario utilizar esta operación.

En C++ la clase es la siguiente:

5.3.5 Clase: TS_ExplorerWrapper

Esta clase fue creada con el propósito de combinar Tabu Search con otras metaheurísticas, por ejemplo, con GRASP.

La clase TS_ExplorerWrapper recibe como parámetros un explorador de vecindad y un Solver de Tabu Search, y se comporta hacia afuera como si fuera el explorador de vecindad original pero controlado por el tabu search.

Para que esto sea posible, TS_ExplorerWrapper implementa la intarfase Explorer (pues simula un explorador) y la interfase TS_Application (pues va a consumir un solver de tabu search).

La clase TS_ExplorerWrapper posee las siguientes operaciones:

Nombre	Parámetros	Retorno	Descripción
(Constructor)	Explorador de vecindades, Solver, y Aplicación del Solver	-	Establece los links a las clases a utilizar

Las demás operaciones de la clase son las correspondientes a las interfases implementadas.

En C++ la clase es la siguiente:

5.4 Resolución del problema QAP

La aplicación QAP resuelve el problema conocido como Quadratic Assignment Problem [QAP01].

5.4.1 Modelo conceptual

El siguiente es un diagrama UML de la resolución de QAP. En las secciones siguientes es explicado el rol de cada clase.

5.4.2 Clases implementadas

La siguiente tabla introduce las clases implementadas.

Clase	Hereda de	Descripción
AplicacionQAP	TS_Application, GRASP_Application	Cliente de los solvers GRASP y TabuSearch. Esta clase es el controlador de fachada del subsistema del desarrollador de aplicaciones. Esta clase posee operaciones que son invocadas por los solvers, para ejecutar las operaciones que son específicas de la aplicación desarrollada.
BuilderQAP	Builder	Constructor de soluciones. Una solución "vacía" es una solución en la que no hay fábricas emplazadas. El constructor itera emplazando fábricas en los nodos disponibles hasta que todas las fábricas estén emplazadas.
DatosQAP	(ninguna)	Es la clase que guarda los datos de un problema. En QAP los datos consisten en dos matrices: una matriz con las distancias entre los nodos, y una matriz con los flujos cuantificados entre fábricas. La matriz de distancias guarda en la celda (i,j) la distancia entre la fábrica i y j. La matriz de flujos guarda en la celda (i,j) el volumen de materiales que fluyen desde i hacia j.
ElementoQAP	Element	Elemento de una solución en construcción. Las instancias de ElementoQAP son creadas por el constructor BuilderQAP. Un elemento a agregar a la solución consiste en emplazar una determinada fábrica en un determinado nodo.
ExploraQAP	Explorer	Explorador de estructura de vecindad. La estructura de vecindad que utilizada consiste en, dada una solución actual, intercambiar la posición de dos de sus fábricas [VecindadQAP].
MovidaQAP	Move	Una "movida" es una posible modificación a la solución actual (que nos desplaza la solución actual hacia una solución vecina). La clase ExploraQAP es quien genera el conjunto de movidas posibles para una solución actual.
SolucionQAP	Solution	Contiene los datos de una solución. Una solución al problema QAP es una posible permutación de fábricas [QAP01].
TabuValueQAP	TabuValue	Las listas tabú del solver Tabu Search contienen valores tabú [TS01]. Un valor tabú en este caso consiste en un par <fábrica, nodo>, donde por ejemplo el par <3,7> significa que no está permitido emplazar la fábrica 3 en el nodo 7.

5.4.3 Reutilización

En la siguiente tabla se presentan las clases desarrolladas para QAP con su cantidad de líneas de código asociadas (LOCs = Lines Of Code). La cantidad de líneas de código es utilizada como un indicador de esfuerzo de codificación. La columna "utilización" muestra en qué contextos es utilizada cada clase.

Clase	Hereda de	LOCs	Utilización
AplicacionQAP	TS_Application, GRASP_Application	322	TS: QAP √, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
BuilderQAP	Builder	271	TS: QAP, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
DatosQAP	(ninguna)	87	TS: QAP √, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
ElementoQAP	Element	122	TS: QAP, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
ExploraQAP	Explorer	135	TS: QAP √, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
MovidaQAP	Move	77	TS: QAP √, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
SolucionQAP	Solution	48	TS: QAP √, TSP GRASP: QAP √, TSP GRASP+TS: QAP √, TSP
TabuValueQAP	TabuValue	94	TS: QAP √, TSP GRASP: QAP, TSP GRASP+TS: QAP √, TSP

5.4.4 Corridas

Fueron realizadas corridas de prueba con la aplicación desarrollada para QAP. Estas corridas no tienen como objetivo ajustar parámetros – son sólo para validar la aplicación desarrollada.

Las corridas realizadas corresponden al producto cartesiano de los siguientes conjuntos:

- 5 instancias de problemas QAP
- 3 semillas distintas
- 3 métodos: GRASP, Tabu Search, y la combinación de GRASP con Tabu Search

El criterio de finalización elegido consiste en aceptar un máximo de 30 segundos sin mejorar la mejor solución encontrada.

La siguiente tabla muestra un resumen de las corridas

Archivo	Tamaño	Óptimo	Mejor valor encontrado	Tiempo promedio* (segundos)	Exceso %
tho30	30	149936	150426	15.9	0.33%
chr25a	25	3796	3988	22.2	5.06%
tai80a	80	13557864	13814150	16.6	1.89%
lipa80a	80	253195	254880	7.6	0.67%
sko90	90	115534	116626	11.3	0.95%

* El *tiempo promedio* considera el tiempo en que se encontró la mejor solución de la corrida. Si a este valor se le agregan 30 segundos se obtiene el tiempo total de corrida.

En la siguiente tabla se detallan todas las corridas

Archivo	Método	Semilla	Alpha	Largo tabú	Valor	Tiempo	óptimo	Exceso
tho30	TS	1005	0.5	180	151858	2	149936	1.28%
tho30	GRASP	1005	0.5	180	150800	25	149936	0.58%
tho30	GRASPTS	1005	0.5	180	150454	17	149936	0.35%
tho30	TS	1010	0.5	180	150482	1	149936	0.36%
tho30	GRASP	1010	0.5	180	150426	29	149936	0.33%
tho30	GRASPTS	1010	0.5	180	150604	29	149936	0.45%
tho30	TS	1015	0.5	180	151506	2	149936	1.05%
tho30	GRASP	1015	0.5	180	150728	29	149936	0.53%
tho30	GRASPTS	1015	0.5	180	150974	9	149936	0.69%
chr25a	TS	1005	0.5	125	4066	6	3796	7.11%
chr25a	GRASP	1005	0.5	125	4330	4	3796	14.07%
chr25a	GRASPTS	1005	0.5	125	4236	21	3796	11.59%
chr25a	TS	1010	0.5	125	4158	15	3796	9.54%
chr25a	GRASP	1010	0.5	125	4214	59	3796	11.01%
chr25a	GRASPTS	1010	0.5	125	3988	37	3796	5.06%
chr25a	TS	1015	0.5	125	4134	4	3796	8.90%
chr25a	GRASP	1015	0.5	125	4242	35	3796	11.75%
chr25a	GRASPTS	1015	0.5	125	4266	19	3796	12.38%
tai80a	TS	1005	0.5	1280	13914786	0	13557864	2.63%
tai80a	GRASP	1005	0.5	1280	13910038	8	13557864	2.60%
tai80a	GRASPTS	1005	0.5	1280	13814150	30	13557864	1.89%
tai80a	TS	1010	0.5	1280	13890150	0	13557864	2.45%
tai80a	GRASP	1010	0.5	1280	13973670	61	13557864	3.07%
tai80a	GRASPTS	1010	0.5	1280	13860126	26	13557864	2.23%
tai80a	TS	1015	0.5	1280	13836938	5	13557864	2.06%
tai80a	GRASP	1015	0.5	1280	13934544	17	13557864	2.78%
tai80a	GRASPTS	1015	0.5	1280	13823264	2	13557864	1.96%
lipa80a	TS	1005	0.5	1280	254901	1	253195	0.67%
lipa80a	GRASP	1005	0.5	1280	255080	3	253195	0.74%
lipa80a	GRASPTS	1005	0.5	1280	254904	5	253195	0.67%
lipa80a	TS	1010	0.5	1280	255000	0	253195	0.71%
lipa80a	GRASP	1010	0.5	1280	255199	1	253195	0.79%
lipa80a	GRASPTS	1010	0.5	1280	254958	2	253195	0.70%
lipa80a	TS	1015	0.5	1280	254880	9	253195	0.67%
lipa80a	GRASP	1015	0.5	1280	255164	45	253195	0.78%
lipa80a	GRASPTS	1015	0.5	1280	254955	2	253195	0.70%
sko90	TS	1005	0.5	1620	116626	9	115534	0.95%
sko90	GRASP	1005	0.5	1620	117156	44	115534	1.40%
sko90	GRASPTS	1005	0.5	1620	117594	2	115534	1.78%
sko90	TS	1010	0.5	1620	117672	1	115534	1.85%
sko90	GRASP	1010	0.5	1620	117366	16	115534	1.59%
sko90	GRASPTS	1010	0.5	1620	116874	1	115534	1.16%
sko90	TS	1015	0.5	1620	117484	1	115534	1.69%
sko90	GRASP	1015	0.5	1620	117448	26	115534	1.66%
sko90	GRASPTS	1015	0.5	1620	117268	2	115534	1.50%

5.5 Resolución del problema TSP

La aplicación TSP resuelve el problema conocido como Travelling Salesman Problem [TSP01].

5.5.1 Modelo conceptual

El siguiente es un diagrama UML de la resolución de TSP. En las secciones siguientes es explicado el rol de cada clase.

5.5.2 Clases implementadas

Las clases implementadas son las siguientes:

Clase	Hereda de	Descripción
AplicacionTSP	GRASP_Application, TS_Application	Cliente de los solvers GRASP y TS. Esta clase es el controlador de fachada del subsistema del desarrollador de aplicaciones. Esta clase posee operaciones que son invocadas por los solvers, para ejecutar las operaciones que son específicas de la aplicación desarrollada.
BuilderTSP	Builder	Constructor de soluciones. Una solución inicial vacía es una solución en la que no hay ningún destino. Se utiliza el método de construcción de soluciones llamado "inserción más cercana" [InserTSP01].
DatosTSP	(ninguna)	Clase que contiene los datos del problema. Un problema TSP contiene un conjunto de nodos tal que para todo par de nodos la distancia entre estos nodos es conocida. La forma en que es representada esta información es con una matriz cuadrada donde la celda (i,j) representa la distancia entre i y j.
ElementoTSP	Element	Esta clase es utilizada por el constructor de soluciones BuilderTSP. Inserta un destino dentro del recorrido del tour. Por ejemplo, visitar el vértice 4 luego del 2º destino. Ver "inserción más cercana" [InserTSP01].
ExploraTSP	Explorer	Explorador de estructura de vecindad. La estructura de vecindad utilizada se conoce como "2-opt" [VecindadTSP].
MovidaTSP	Move	Realiza un intercambio según descrito en la estructura de vecindad "2-opt" [VecindadTSP].
SolucionTSP	Solution	Contiene los datos de una solución al problema TSP [TSP01].
TabuValueTSP	TabuValue	Las listas tabú del solver Tabu Search contienen valores tabú [TS01]. Un valor tabú es en esta caso una arista que no puede ser agregada (o quitada) del tour actual. Una arista puede identificarse de forma unívoca con un par <nodo1, nodo2> tal que (nodo1 < nodo2).

5.5.3 Reutilización

En la siguiente tabla se presentan las clases desarrolladas para QAP con su cantidad de líneas de código asociadas (LOCs = Lines Of Code). La cantidad de líneas de código es utilizada como un indicador de esfuerzo de codificación. La columna "utilización" muestra en qué contextos es utilizada cada clase.

Esta tabla es muy similar a la presentada para QAP.

Clase	Hereda de	LOCs	Utilización
AplicacionTSP	GRASP_Application, TS_Application	370	TS: TSP √ / QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
BuilderTSP	Builder	216	TS: QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
DatosTSP	-	137	TS: TSP √ / QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
ElementoTSP	Element	92	TS: QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
ExploraTSP	Explorer	132	TS: TSP √ / QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
MovidaTSP	Move	99	TS: TSP √ / QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
SolucionTSP	Solution	299	TS: TSP √ / QAP ; GRASP: TSP √ / QAP ; GRASP+TS: TSP √ / QAP
TabuValueTSP	TabuValue	108	TS: TSP √ / QAP ; GRASP: QAP ; GRASP+TS: TSP √ / QAP

5.5.4 Corridas

Análogamente a QAP, fueron realizadas corridas de prueba con la aplicación desarrollada para TSP. Estas corridas no tienen como objetivo ajustar parámetros – son sólo para validar la aplicación desarrollada.

Las corridas realizadas corresponden al producto cartesiano de los siguientes conjuntos:

- 5 instancias de problemas TSP
- 3 semillas distintas
- 3 métodos: GRASP, Tabu Search, y la combinación de GRASP con Tabu Search

El criterio de finalización elegido consiste en aceptar un máximo de 30 segundos sin mejorar la mejor solución encontrada.

La siguiente tabla muestra un resumen de las corridas

Archivo	Tamaño	Óptimo	Mejor valor encontrado	Tiempo promedio* (segundos)	Exceso %
a280	280	2579	2759.9919	4.9	7.02%
berlin52	52	7542	7544.3638	10.7	0.03%
ch130	130	6110	6175.0986	16.2	1.07%
ch150	150	6528	6724.0625	5.2	3.00%
eil101	101	629	651.6177	4.8	3.60%

* El *tiempo promedio* considera el tiempo en que se encontró la mejor solución de la corrida. Si a este valor se le agregan 30 segundos lo obtenido es el tiempo total de corrida.

En la siguiente tabla se detallan todas las corridas

Archivo	Método	Semilla	Alpha	Largo tabú	Valor	Tiempo	óptimo	Exceso
a280	TS	1005	0.5	470	2887.4849	2	2579	11.96%
a280	GRASP	1005	0.5	470	2764.6418	6	2579	7.20%
a280	GRASPTS	1005	0.5	470	2774.2671	11	2579	7.57%
a280	TS	1010	0.5	470	2783.5967	0	2579	7.93%
a280	GRASP	1010	0.5	470	2780.406	3	2579	7.81%
a280	GRASPTS	1010	0.5	470	2806.0991	0	2579	8.81%
a280	TS	1015	0.5	470	2953.9924	1	2579	14.54%
a280	GRASP	1015	0.5	470	2759.9919	20	2579	7.02%
a280	GRASPTS	1015	0.5	470	2783.0532	1	2579	7.91%
berlin52	TS	1005	0.5	16	8166.4048	1	7542	8.28%
berlin52	GRASP	1005	0.5	16	7544.3647	4	7542	0.03%
berlin52	GRASPTS	1005	0.5	16	7544.3647	0	7542	0.03%
berlin52	TS	1010	0.5	16	8048.9824	0	7542	6.72%
berlin52	GRASP	1010	0.5	16	7544.3638	2	7542	0.03%
berlin52	GRASPTS	1010	0.5	16	7544.3652	8	7542	0.03%
berlin52	TS	1015	0.5	16	8827.3027	0	7542	17.04%
berlin52	GRASP	1015	0.5	16	7544.3638	54	7542	0.03%
berlin52	GRASPTS	1015	0.5	16	7544.3638	27	7542	0.03%
ch130	TS	1005	0.5	101	6719.8169	0	6110	9.98%
ch130	GRASP	1005	0.5	101	6184.0571	60	6110	1.21%
ch130	GRASPTS	1005	0.5	101	6283.7446	4	6110	2.84%
ch130	TS	1010	0.5	101	6842.833	0	6110	11.99%
ch130	GRASP	1010	0.5	101	6188.3462	50	6110	1.28%
ch130	GRASPTS	1010	0.5	101	6297.9629	5	6110	3.08%
ch130	TS	1015	0.5	101	6950.2925	0	6110	13.75%
ch130	GRASP	1015	0.5	101	6175.0986	26	6110	1.07%
ch130	GRASPTS	1015	0.5	101	6190.9116	1	6110	1.32%
ch150	TS	1005	0.5	135	7180.6396	1	6528	10.00%
ch150	GRASP	1005	0.5	135	6737.8237	12	6528	3.21%
ch150	GRASPTS	1005	0.5	135	6799.8545	4	6528	4.16%
ch150	TS	1010	0.5	135	7390.5781	0	6528	13.21%
ch150	GRASP	1010	0.5	135	6724.0625	9	6528	3.00%
ch150	GRASPTS	1010	0.5	135	6900.8242	2	6528	5.71%
ch150	TS	1015	0.5	135	7171.248	0	6528	9.85%
ch150	GRASP	1015	0.5	135	6773.1577	11	6528	3.76%
ch150	GRASPTS	1015	0.5	135	6857.707	8	6528	5.05%
eil101	TS	1005	0.5	61	681.5616	0	629	8.36%
eil101	GRASP	1005	0.5	61	654.6043	1	629	4.07%
eil101	GRASPTS	1005	0.5	61	660.0533	3	629	4.94%
eil101	TS	1010	0.5	61	693.5427	0	629	10.26%
eil101	GRASP	1010	0.5	61	658.4572	6	629	4.68%
eil101	GRASPTS	1010	0.5	61	653.3839	11	629	3.88%
eil101	TS	1015	0.5	61	680.36	1	629	8.17%
eil101	GRASP	1015	0.5	61	651.6177	13	629	3.60%
eil101	GRASPTS	1015	0.5	61	658.9302	8	629	4.76%

5.6 Combinación de GRASP y Tabu Search

Tanto para el problema QAP como para TSP uno de los métodos de resolución consiste en combinar GRASP y Tabu Search.

La forma de combinar GRASP [GRASP01] con Tabu Search [TS01] consiste en idear una variante de GRASP en la cual la búsqueda local es realizada como si fuera una búsqueda tabú.

Esta combinación pudo ser realizada con un esfuerzo de codificación mínimo por parte del desarrollador de aplicaciones. Esto es posible porque el solver Tabu Search provee una clase llamada TS_ExplorerWrapper pensada para combinar Tabu Search con otras metaheurísticas, en el contexto en el cual se recurre a Tabu Search para realizar búsquedas locales.

La clase TS_ExplorerWrapper fue diseñada pensando en el patrón de diseño Wrapper [Wrapper01]. Esta clase recibe como parámetros un Explorer (explorador de vecindades) y un TS_Solver (solver de Tabu Search). Dados el explorador y el solver, la clase TS_ExplorerWrapper ejecuta el solver sobre el explorador dado, y se comporta hacia afuera como si fuera un Explorer común. El solver GRASP utiliza esta instancia de TS_ExplorerWrapper como explorador de vecindades.

6 Conclusiones y trabajo futuro

En esta sección es verificado el trabajo realizado siguiendo dos líneas de análisis:

- Lo realizado contra los objetivos y restricciones propuestos.
- Lo realizado contra los atributos de calidad de software comúnmente aceptados.

Del análisis anterior surgen oportunidades de mejora de nuestro sistema que abren la puerta a trabajos futuros.

6.1 Objetivos y restricciones propuestos

El propósito de esta sección es contrastar los objetivos propuestos contra el trabajo realizado.

6.1.1 Desarrollador de aplicaciones

La siguiente tabla muestra los objetivos identificados para el desarrollador de aplicaciones.

Objetivo	Realizado	Limitaciones y trabajo futuro
Resolver problemas reales	Se trabajó junto a los docentes para que ellos validaran e indicaran limitaciones de las soluciones propuestas.	Se trabajó con dos problemas "idealizados" que son el TSP y QAP. Se podría intentar desarrollar aplicaciones con clientes finales.
Bajo costo de desarrollo y mantenimiento	Parte del conocimiento está disponible de forma reutilizable, lo que reduce la cantidad de software a desarrollar y mantener. Los arquietctura del framework junto con sus conceptos estandarizados facilitan una mejor modularidad del software desarrollado.	-
Calidad en el código fuente y en el proceso de desarrollo	(Ver tabla con atributos de calidad)	(Ver tabla con atributos de calidad)
Construir prototipos para evaluar alternativas	Fue posible intercambiar con facilidad los solvers reutilizando el código que es independiente del solver utilizado, como es posible ver en la sección "reutilización" de las aplicaciones QAP y TSP respectivamente. Las estadísticas, manejo de parámetros, y condiciones de finalización estándar permiten construir prototipos rápidamente.	-
Facilidad para	Al igual que en el ítem anterior, fue	-

intercambiar y combinar solvers	posible reutilizar código que es independiente del solver utilizado. El solver de Tabu Search trae implementado un "wrapper" que permite combinar tabu search con otras metaheurísticas con facilidad.	
Concentrarse en su problema específico	Los detalles de los algoritmos de ejecución quedan encapsulados dentro de los solvers. La reutilización con enfoque de caja negra ayuda a separar conocimiento y responsabilidades.	-
Poder reutilizar sus propias soluciones	Esto fue validado, resolviendo TSP y QAP con distintos solvers, y reutilizando siempre el código que es independiente del solver a ejecutar. (Ver sección de "reutilización" para QAP y TSP respectivamente)	-

6.1.2 Desarrollador de solvers

La siguiente tabla muestra los objetivos identificados para el desarrollador de solvers.

Objetivo	Realizado	Limitaciones y trabajo a futuro
Atender las necesidades del desarrollador de aplicaciones	Los solvers realizados fueron consumidos con problemas de ejemplo.	-
Ofrecer código reutilizable	Fue validado que los solvers sean reutilizables resolviendo más de un problema con cada solver.	-
Ofrecer soluciones adaptables a las diversidad de necesidades específicas que pueda tener el desarrollador	Se trabajó junto a los docentes para que ellos validaran e indicaran limitaciones de las soluciones propuestas.	Estudiar aplicaciones ya desarrolladas. En particular investigar benchmarks (algoritmos más rápidos conocidos). Ver si es posible implementar los algoritmos benchmark utilizando nuestro framework.

6.1.3 Mantenimiento del framework

La siguiente tabla muestra los objetivos identificados para el desarrollador de solvers.

Objetivo	Realizado	Limitaciones y trabajo a futuro
Crear un conjunto de reglas mínimo para que sea viable el intercambio de solvers	La posibilidad de intercambiar solvers fue validada intercambiando solvers en las soluciones desarrolladas. Las reglas del framework están en forma de "conceptos estandarizados". Los conceptos son intuitivos, no hay conceptos artificiales a eliminar. Estos "conceptos estándar" se traducen en interfases a ser implementadas. Cada interfase tiene las operaciones necesarias para el comportamiento que se espera de su entidad asociada.	Desarrollar nuevos solvers, en particular solvers de metaheurísticas poblacionales.
Proveer algunas funcionalidades comunes, tales como manejo de parámetros, estadísticas y criterios de finalización más comunes	Estas funcionalidades son provistas. Fueron utilizadas en las aplicaciones desarrolladas. Los criterios de finalización permiten incorporar criterios definidos por el desarrollador, y las estadísticas permiten ser extendidas tanto a nivel del solver como a nivel de la aplicación desarrollada.	-
No entorpecer el desarrollo de solvers ni su utilización	El diseño el framework fue hecho pensando en los patrones de diseño Adapter y Façade Controller, con el objetivo de no ser muy exigentes en cuanto a cómo deben ser estructurados los subsistemas que participan.	Desarrollar nuevos solvers y aplicaciones.
Framework que permite desarrollar nuevos metaheurísticas y problemas	Al no haber relaciones de dependencia desde el framework hacia los solvers desarrollados, es posible crear nuevos solvers conservando el framework. Análogamente, los solvers no tienen relaciones de dependencia hacia sus consumidores.	Incorporar nuevas metaheurísticas, en particular poblacionales, con las que seguramente sea a querer estandarizar más conceptos.

6.1.4 Restricciones

Restricción	Realizado	Limitaciones y trabajo a futuro
El código debe estar desarrollado en C/C++.	Cumplido.	-
Se desea que el framework funcione tanto en ambientes Unix como Windows.	Se codificó en C++ estándar incluyendo la Standard Template Library. No hay referencias a módulos dependientes de la plataforma.	No se intentó compilar y testear el framework en ambientes Unix.

6.2 Atributos de Calidad

Atributo	Realizado	Limitaciones y trabajo a futuro
Correctitud	Fueron realizadas pruebas funcionales automatizadas con lectura de datos de prueba desde archivos de texto.	No hay un documento que especifique matemáticamente el comportamiento de cada componente desarrollado. Los resultados obtenidos de las pruebas automatizadas sugieren que el código desarrollado es correcto en su conjunto.
Confiabilidad	En diseño fue definido el comportamiento esperado de los distintos componentes y fueron especificadas sus operaciones. Para que sea más confiable el manejo de la memoria, fue adoptado como criterio que quien crea nuevas instancias debe ser quien se encarga de destruirlas.	Para ver si el framework es confiable, habría que desarrollar muchas aplicaciones de usuario que reutilicen los solvers y el framework.
Robustez	-	Podría ser incluido en el código del framework y de los solvers, chequeos adicionales sobre los datos intercambiados con el código del usuario programador. Esto traería consecuencias de performance, pero es posible hacer que estos chequeos se habiliten y deshabiliten con una directiva de compilación.
Performance	El diseño tiene consideraciones de performance que toman en cuenta las prácticas más comunes al desarrollar aplicaciones de metaheurísticas. El framework está diseñado considerando los problemas de performance de la memoria dinámica: la solución adoptada es manejar pools de instancias a reutilizar para evitar operaciones new/delete. Muchas interfases especifican las operaciones clone() y copyFrom() con este objetivo.	Fue utilizado despacho dinámico de C++, propio de la programación orientada a objetos. No fue investigado el uso de múltiples hilos de ejecución (paralelismo).
Amigabilidad	Existen herramientas ya desarrolladas que ayudan al usuario programador, como las	Se podrían construir plantillas de código para ayudar a desarrollar

	estadísticas predefinidas, manejo de parámetros desde archivo de texto, y criterios de finalización más comunes ya implementados. El intercambio entre el código del solver código del usuario programador está ensado considerando los patrones de diseño dapter y Façade Controller, para darle más ertad al usuario programador en cómo be estructurar su código. Los conceptos definidos y estandarizados por el framework (solution, solver, application) son intuitivos en el sentido de no ser conceptos artificiales.	soluciones desde cero.
Verificabilidad	diseño modular ayuda a la verificabilidad e las soluciones desarrolladas.	-
Reparabilidad	Las estadísticas predefinidas ayudan al usuario programador a depurar sus soluciones. Como gran parte del código de la solución ya está desarrollado, hay menos código a verificar.	(ver ítem de robustez)
Evolucionabilidad	El framework tiene una arquitectura que permite desarrollar nuevos solvers y problemas sin necesidad de modificaciones al mismo.	-
Reusabilidad	El framework y los solvers están diseñados con el objetivo de ser reutilizables. En las soluciones desarrolladas, de hecho fueron reutilizaron componentes.	-
Portabilidad	El código fue desarrollado sólo con C++ estándar y llamadas a la Standard Template Library. Los fuentes están diseñados de forma de no incluir código dependiente de la plataforma utilizada.	No fue testeado el producto en plataformas UNIX. No se intentó compilar en UNIX.
Comprensibilidad	El diseño modular orientado a objetos, con entidades intuitivas, distribuye responsabilidades de forma tal que el código sea más comprensible.	Los mecanismos de herencia y sobrecarga de operaciones, propios de la programación orientada a objetos, pueden dificultar la depuración paso por paso.
Interoperabilidad	El software desarrollado consiste en bibliotecas a ser incluidas en cualquier proyecto de C++. Una biblioteca es el framework, y hay cada solver tiene su propia biblioteca. De esta forma el usuario programador elige qué solvers incluir, y el framework es independiente de los solvers que sean desarrollados.	-
Productividad (del proceso)	Esta solución permite un proceso más productivo, reutilizando componentes ya desarrollados.	-

	Es posible intercambiar solvers para experimentar distintos prototipos. Esto es posible porque los componentes intercambiables cumplen interfases comunes. Esto hace la fase inicial de exploración más productiva y de tiempos acotados.	
Oportunidad (entrega de producto en tiempo)	El framework, al definir conceptos estándar, ayuda al usuario programador a planificar qué clases y módulos debe implementar. Esto permite una mayor habilidad al estimar el tiempo de desarrollo.	-
Visibilidad (del proceso)	Cada solver tiene asociado un conjunto de interfases que el desarrollador debe implementar. Esto facilita la planificación inicial, lo que redunda en un proceso más visible.	-

6.3 Sobre las limitaciones

En la tabla del capítulo anterior fueron relevadas cuáles son las limitaciones de nuestro producto intentando utilizar un criterio objetivo y metódico, que consistió en verificar el cumplimiento de objetivos de nuestro proyecto y evaluar atributos de calidad.

En esta sección se desarrollan más algunas ideas sobre las siguientes limitaciones:

- Soporte para programación paralela
- Multiplataforma
- Falta de metaheurísticas de poblaciones
- Limitaciones del diseño

6.3.1 Eficiencia y programación paralela

Uno de los principales objetivos del presente proyecto es la eficiencia en la resolución de las diferentes Metaheurísticas creadas por el framework. Por esto, una cualidad importante del mismo seria que el mismo posea soporte para paralelismo, esto es que el mismo pueda, a la hora de realizar las diferentes corridas de los diferentes algoritmos, se pueda realizar en diferentes procesadores (los cuales pueden estar tanto en una supercomputadora o en clusters, etc). Este tipo de software esta empezando a tener un gran auge ya que se presentan las condiciones necesarias para que este tipo de sistemas tengas grandes prestaciones (existen librerías especializadas para el manejo de paralelismo, el hardware especializado a evolucionado y es más barato lo cual permite un acceso para más personas, las redes también han evolucionado de tal forma lo cual permite la realización de clústers de forma mas sencilla y veloz, etc.), por lo cual sería un gran apoyo a la eficiencia del framework que el mismo pudiera realizar sus operaciones de forma paralela. Este tipo sistemas (los sistemas paralelos), a pesar de que puede mejorar la eficiencia de las ejecuciones,

presenta la dificultad de que se debe planificar como un software de este estilo ya que la programación del mismo requiere un diseño especial, ya que en la mayoría de los casos los algoritmos para resoluciones lineales no son eficientes llevados a ambientes de programación paralela, por lo cual se puede ganar en eficiencia si el mismo esta pensado y diseñado para este tipo de ambientes.

En el caso particular del presente framework, el mismo no esta pensado ni diseñado de forma que soporte a este tipo de programación, ya que dentro de los requerimientos establecidos no se presentaba tal necesidad, por lo cual el mismo fue diseñado para el tipo de programación lineal. Las diferentes clases creadas están pensadas para dicho tipo de programación, además no se han incluido librerías de programación paralela ni se utilizan diferentes técnicas para este tipo de programación (uso de semáforos, utilización de fork, programación multihilado, etc.).

6.3.2 Multiplataforma

Otra de las limitaciones que presenta el presente sistema es que no posee soporte para multiplataforma. El soporte para multiplataforma permite al software poder correr en cualquier computador el cual contenga un sistema que contenga un compilador de C++, por lo cual permite que sea utilizado en casi en cualquier lado. En el presente caso dicho soporte no se presenta, a pesar de haber estado a priori en los requerimientos, aunque a nivel de diseño si fue utilizado para realizar el mismo.

A pesar que formalmente dicho soporte no esta presente, al diseñar y crear las diferentes estructuras se evitó utilizar llamadas al sistema. Este es el punto más crítico al considerar distintas plataformas.

Al no incluir llamados al sistema operativo, esto nos hace pensar que aunque el sistema haya sido desarrollado sobre Windows - utilizando el Visual Estudio .NET - el mismo no debería tener problemas con las diferentes plataformas. Pero al no ser debidamente testeado en las mismas no es posible asegurar que así sea. Por lo tanto no es posible afirmar su soporte para distintas plataformas.

6.3.3 Metaheurísticas poblacionales

Uno de los temas importantes de los requerimientos, representaba el hecho de que se pudiera realizar cualquier tipo de Metaheurística. No sólo las probadas en el presente caso, las cuales son Metaheurísticas de trayectoria, sino que también probar metaheurísticas poblacionales (algoritmos genéticos, colonia de hormigas), o híbridos entre poblacionales y de trayectoria.

El trabajo de extender el framework para metaheurísticas poblaciones consiste fuertemente en estandarizar conceptos que son aplicables a las metaheurísticas poblacionales. Así como hay conceptos definidos para estructuras de vecindad (Explorer, Move) seguramente se vayan a definir otros conceptos para los métodos poblacionales.

6.3.4 Limitaciones del diseño

Las limitaciones de diseño encontradas durante el proyecto fueron solucionadas. Sin embargo, es natural pensar que existan más limitaciones. Pensando en este riesgo es que fueron sugeridas algunas actividades de trabajo a futuro:

- Desarrollar aplicaciones con clientes reales
- Investigar benchmarks: para algunos problemas clásicos, existen algoritmos que tienen los mejores tiempos de respuesta. Intentar implementar esos algoritmos dentro del framework.
- Desarrollar nuevos solvers que utilicen otras metaheurísticas

Todo producto de software tiene un ciclo de vida. Solucionar limitaciones encontradas por los clientes, es uno de los varios tipos de mejoras que se realizan de una versión a otra.

6.4 Sobre el trabajo a futuro

Nuestro desarrollo consiste en un prototipo, por lo cual es natural pensar que tiene varios elementos a extender o mejorar. En un prototipo, típicamente se eligen qué aspectos se desean evaluar, y en base a las ideas presentes es que se priorizan y se ajusta el alcance del primer prototipo experimental.

El trabajo a futuro fue planteado intentando utilizar un criterio objetivo y metódico, y consiste en ver las limitaciones relevadas en relación con los objetivos propuestos. Todo trabajo a futuro debería responder a objetivos.

Aquí en esta sección se desarrollan un poco más las ideas sobre dos limitaciones importantes: paralelización y multiplataforma.

6.4.1 Paralelización

En lo que respecta a la paralelización actualmente el sistema no cuenta con soporte para el mismo. Como fue aclarado en la sección de limitaciones, para dar soporte a este tipo de programación no solo basta con incluir librerías de programación paralela o utilizar técnicas avanzadas de este tipo de programación, sino que se debe pensar todo como un software que corre en forma paralela.

Algunas metaheurísticas son paralelizables de forma natural, por ejemplo las poblacionales. Distintos hilos pueden calcular la función objetivo de distintos individuos, combinar distintos pares, etc.

GRASP es fácilmente paralelizable. Es posible hacer que cada hilo realice su propia construcción y búsqueda local.

Luego hay otras metaheurísticas con las que es más difícil aprovechar varias CPUs. Por ejemplo, Tabu Search, una de las dos metaheurísticas que fue implementada. Una posible idea es que al evaluar la vecindad de la solución actual, distintos hilos evalúen distintos vecinos.

Lo que se hace necesario analizar es lo siguiente:

- ¿Cómo afecta al framework la paralelización?
- ¿Es necesario fijar “reglas de juego” comunes para la paralelización, a nivel del framework?

- ¿Es necesario marcar ciertas pautas, para que distintos solvers paralelizables puedan ser combinados?
- ¿Hay que modificar interfases para tener en cuenta paralelización?
- ¿Quién instancia hilos nuevos? ¿El usuario programador, el framework, los solvers?
- ¿Cómo se resuelve el problema de multi-hilado que corra en distintas plataformas? ¿Es posible hacer una biblioteca con directivas de compilación condicional para que compile tanto en Windows como Unix, y que ofrezca un mismo servicio de multihilado del framework en ambas plataformas?

Este tipo de investigación escapa al alcance presentado para dicho proyecto. A nuestro criterio sería muy interesante poder desarrollar un segundo prototipo experimental que incluya multihilado. Cabe destacar que las computadoras de uso personal están tendiendo a traer posibilidades de multihilado: procesadores Hyper-Threading y Dual Core.

El esfuerzo realizado orientado a distribuir las responsabilidades dentro de nuestro software, por nuestro trabajo en modularización, debería tener algunas ventajas al querer migrar a un entorno de multihilado. A pesar de no ser un software diseñado para correr en forma paralela, se le puede aplicar diseños de programaciones paralelas en algunas de las estructuras que posee el sistema. Por lo cual podría ser, en un corto periodo de tiempo un software con una base de programación lineal, con algunas estructuras (aquellas que realizan iteraciones, por ejemplo) las cuales contienen programación paralelas.

6.4.2 Multiplataforma

En cuanto a la limitación del no soporte de multiplataforma, el presente software en sí no está atado con llamadas al sistema a ninguna plataforma en particular. Esto es algo que fue respetado al diseñar nuestra solución.

En la practica los desarrollos fueron realizados sobre Windows utilizando Visual Estudio .NET. En la configuración del entorno de desarrollo, se verificó con los docentes que la configuración del compilador no incluya características que sean exclusivas de Microsoft.

Para que nuestro framework funcione en ambientes Unix debería hacerse lo siguiente:

- Construir Makefiles para Unix
- Verificar que compile
- Utilizar los archivos para testeos automatizados que ya se tienen

6.4.3 Metaheurísticas Poblacionales

Un tema interesante de extender el presente sistema seria el de utilizar metaheurísticas basadas en poblaciones, ya que las metaheurísticas creadas y testeadas por el sistemas fueron GRASP y Tabu Search que son de trayectoria.

Este tipo de extensión permite observar como se puede comportar el sistema para los diferentes tipos de metaheurísticas existentes, aunque basándose en el diseño del framework, la creación de metaheurísticas de población no debería representar grandes cambios, ya que uno de los requerimientos (y uno de los puntos importantes del sistema) es que el framework se pudiera construir diferentes metaheurísticas. No obstante la construcción de dicho tipo de metaheurísticas

(Algoritmos genéticos, colonia de hormigas, etc.) escapan al alcance del presente proyecto, pero la construcción de las misma representan un paso fundamental en el avance de dicho proyecto en un futuro.

6.4.4 Plantillas de código

Se pueden hacer mejoras en lo referente a amigabilidad. A pesar de ser un software orientado a usuarios programadores, se podrían manejar de alguna forma "plantillas" con código a completar. Estas plantillas pueden estar como simples documentos, o sería posible hacer un pequeño programa que a partir de algunos parámetros "genere" el código a completar.

Al ver las diferencias y similitudes entre las dos aplicaciones desarrolladas – QAP y TSP – es posible identificar una estructura común que induce a pensar en plantillas. Cada solver podría venir con su plantilla. Las plantillas también podrían ayudar a reducir la curva de aprendizaje.

Informe final

Anexo 1: Estado Del Arte

1 Metaheurísticas

1.1 Introducción a las Metaheurísticas

Utilizamos Metaheurísticas para resolver problemas de Optimización Combinatoria. Un problema de optimización combinatoria es un problema de optimización en el cual las variables toman valores discretos, y nos interesa encontrar una "mejor combinación" de valores para nuestras variables, para alcanzar ciertos objetivos.

1.1.1 Definición de problema de optimización combinatoria

Sea un problema de Optimización Combinatoria P = (S, f), lo podemos definir por:

- Un conjunto de variables $X = \{x_1, \ldots, x_n\}$
- Dominios para las variables $D_1, \ldots, D_n$.
- Restricciones para las variables
- Una función objetivo f a minimizar, donde $f : D1 \times \ldots \times Dn \rightarrow R+$

El conjunto de todas las posibles soluciones factibles es

$S = \{s = \{(x_1, v_1), \ldots, (x_n, v_n)\} \mid v_i \in Di$, s satisface todas las restricciones$\}$

En otras palabras: S es el conjunto de soluciones s tales que los valores x_i que toman pertenecen a sus dominios respectivos, y además se satisfacen las restricciones sobre las variables.

Llamamos a S espacio de búsqueda, donde cada elemento de S es una solución candidata. Resolver el problema de optimización combinatoria consiste en encontrar una solución s* tal que pertenezca a S y minimice la función objetivo.

Algunos ejemplos de problemas de Optimización Combinatoria son el problema del agente viajero (TSP: Travelling Salesman Problem), el problema de asignación cuadrática (QAP: Quadratic Assignment Problem), y problemas de asignación de horarios y secuenciamiento de tareas.

1.1.2 Resolución de problemas de optimización combinatoria

Los algoritmos para atacar problemas de optimización combinatoria los podemos clasificar en dos categorías, *exactos* o *aproximados*.

Los algoritmos exactos garantizan que encuentran la solución óptima al problema en una cierta cota de tiempo. El problema es que muchos de los problemas de OC son NP-duros. Por lo tanto, no vamos a obtener una solución luego de esperar durante un tiempo polinomial.

Una posible forma de atacar este problema es con el enfoque Branch and Bound, que consiste en recorrer las combinaciones posibles por backtracking con prioridades, y recortar las ramas que no nos interesan. Esta técnica la nombramos por completitud, pero escapa al alcance de nuestro estudio.

En los métodos aproximados, no tengo garantías de encontrar la solución óptima global, pero sí puedo encontrar soluciones razonablemente buenas luego de un cierto tiempo de ejecución.

1.1.3 Construcción de soluciones aproximadas

Entre los métodos básicos de aproximación distinguimos entre los métodos *constructivos* y los métodos de *búsqueda local*.

Los métodos constructivos generan soluciones desde cero, agregando a un conjunto inicial vacío, los distintos componentes de la solución. Los métodos de búsqueda local hacen lo siguiente: dada una solución inicial, intento mejorarla modificando algunos componentes. Luego que obtengo una nueva solución, continúo iterativamente.

Los métodos de búsqueda local se desplazan de una solución actual a una solución *vecina*, y de antemano definimos una *estructura de vecindad*. La estructura de vecindad es una función que dada una solución, nos devuelve el conjunto de soluciones vecinas.

1.1.4 Metaheurísticas

Las Metaheurísticas son descripciones de heurísticas con un alto grado de generalidad. Son métodos aproximados de resolución de problemas de Optimización Combinatoria, especificados de una forma suficientemente general como para que sean independientes del problema específico a resolver.

El término Metaheurísticas fue empleado por primera vez por Glover en 1986. Heurística viene de heuristikein del griego que significa “encontrar” mientras que el sufijo Meta significa “más allá, en un nivel superior”.

Esta clase de algoritmos incluye los Algoritmos de Hormigas (ACO: Ant Colony Optimization), Algoritmos Genéticos (GA: Genetic Algorithms), Búsqueda Local Iterativa (ILS: Iterated Local Search), Recocido Simulado (SA: Simulated Annealing), y Búsqueda Tabú (TS: Tabu Search).

En resumen, podemos decir que las metaheurísticas son estrategias de alto nivel para recorrer grandes espacios de búsqueda, utilizando distintos métodos. De aquí surge el concepto de *diversificación* o *intensificación* de la búsqueda. Decimos que intensificamos la búsqueda cuando nos detenemos a explorar en un espacio reducido.

Algunas características fundamentales de las metaheurísticas son las siguientes (según Christian Blum y Andrea Roli):

- Son estrategias que guían el proceso de exploración
- El objetivo es explorar el espacio de búsqueda de forma eficiente para encontrar soluciones cercanas a la óptima
- Las técnicas metaheurísticas van desde procedimientos simples de búsqueda local hasta complejos procesos de aprendizaje
- Los algoritmos metaheurísticos son aproximados y generalmente no determinísticos
- Proveen mecanismos para no quedar atrapados en óptimos locales
- Los conceptos básicos de las metaheurísticas permiten una descripción abstracta

- Las metaheurísticas pueden recurrir a heurísticas que resuelven problemas particulares del campo de aplicación, y estas heurísticas están controladas por la metaheurística.
- Las metaheurísticas más avanzadas tienen memoria, acumulan experiencia sobre el espacio recorrido, para guiar la búsqueda.

1.1.5 Clasificación de las Metaheurísticas

Hay diferentes criterios para clasificar a las metaheurísticas. Los criterios de clasificación que presentamos son de los autores Christian Blum y Andrea Roli.

1.1.5.1 Inspiradas en la naturaleza o no

Una forma intuitiva de clasificar a las metaheurísticas es según su origen, hay algunas inspiradas en comportamientos de la naturaleza como los algoritmos genéticos[AGE01..10] y algoritmos de hormigas [CHO01..11]. Hay otras no inspiradas en la naturaleza como lo es Tabu Search [TSE01..06] y Búsqueda Local Iterada [BLO01..10].

Hay algoritmos recientes que no pertenecen a ninguna de estas dos clases, que son híbridos. Tampoco es claro siempre a qué clase debe pertenecer cada metaheurística, por ejemplo se podría decir que Tabu Search al utilizar memoria está inspirada en la naturaleza.

1.1.5.2 Poblacionales y de trayectoria

Otra forma de clasificar a las metaheurísticas es ver la cantidad de soluciones manejadas al mismo tiempo. Hay algoritmos que trabajan con poblaciones de soluciones, mientras que hay otros que se desplazan de una solución actual a una nueva solución.

Ejemplos de metaheurísticas poblacionales son los algoritmos genéticos y hormigas. En las de trayectoria tenemos Tabu Search, Búsqueda Local Iterada, y Búsqueda en Vecindad Variable [BVV01, BVV02].

1.1.5.3 Función objetivo dinámica o estática

Aquí clasificamos a las Metaheurísticas según cómo utilizan su función objetivo. Mientras que muchos algoritmos mantienen la función objetivo intacta, hay otros que la modifican en tiempo de ejecución, como por ejemplo Búsqueda Local Guiada (GLS: Guided Local Search).

La idea detrás de modificar la función objetivo, es escapar a óptimos locales, modificando el espacio de búsqueda. Por ejemplo, si buscamos un mínimo en una superficie, podemos remplazar un pozo ya visitado por una montaña bien alta para evitar que el algoritmo vuelva a explorar esa zona. La función objetivo es modificada iterativamente para incorporar la información recolectada durante el proceso de búsqueda.

1.1.5.4 Una estructura de vecindad y múltiples estructuras

La mayoría de las Metaheurísticas trabajan con una única estructura de vecindad. Esto significa que la topología del espacio de búsqueda es estática, no cambia durante la ejecución. Hay otras metaheurísticas como Búsqueda en Vecindad Variable (VNS: Variable Search Neighborhood) que en vez de trabajar con una estructura de vecindad predefinida, trabaja con un conjunto de estructuras de vecindad predefinidas.

El uso de varias estructuras de vecindad permite:

- Intensificar o diversificar la búsqueda según la estructura de vecindad que estemos utilizando
- Escapar a óptimos locales: un óptimo local en una estructura no tiene porqué ser óptimo local en las demás estructuras de vecindad.

1.1.5.5 Uso de memoria

Algunas metaheurísticas utilizan memoria sobre el espacio de búsqueda recorrido. Los algoritmos que no trabajan con memoria se comportan como un proceso de Markov, ya que la única información que utilizan para determinar a qué estado desplazarse es el estado actual.

Hay diferentes formas de utilizar esta memoria. Usualmente podemos diferenciar entre memoria de corto y de largo plazo. La memoria de corto plazo registra generalmente los últimos movimientos realizados y las últimas soluciones realizadas. Para la memoria de largo plazo guardamos estadísticas acumuladas sobre la búsqueda que creemos que pueden ser útiles.

1.2 Algunas Metaheurísticas conocidas

El criterio de clasificación en el que tenemos metaheurísticas poblacionales y de trayectoria es importante a la hora de describir los algoritmos. Vamos a tener algoritmos muy distintos si se trata de una metaheurística de trayectoria o poblacional.

Metaheurísticas de Trayectoria

Llamamos Metaheurísticas de Trayectoria, en oposición a las poblacionales, a las metaheurísticas que trabajan con una única solución actual. El término “trayectoria” es porque el proceso de exploración se desplaza de una solución a la siguiente, generando un recorrido o trayectoria.

En algoritmos de trayectoria tenemos una solución inicial, y nos desplazamos a nuevas soluciones hasta terminar el proceso de búsqueda.

Llamamos “estructura de vecindad” al mecanismo empleado para, dada una solución actual, determinar cuáles son las soluciones “vecinas”. Formalmente se define estructura de vecindad como una función $N:\Omega \rightarrow 2^{\Omega}$. El conjunto N(s) se llama “vecindad” de s y cada elemento s’ de N(s) es una “solución vecina” de s.

Consideraciones al utilizar metaheurísticas de trayectoria

Los algoritmos de trayectoria se basan en el supuesto implícito de que generalmente una solución vecina tiene asociado un valor objetivo parecido al de la solución actual. Si esto no fuera así, si al desplazarnos a una solución vecina el valor nuevo es totalmente independiente del valor actual, entonces no tiene sentido aplicar los métodos de trayectoria. Una consecuencia de este concepto es que conviene modelar los problemas de forma tal que no hayan muchos valles y montañas, y que no hayan grandes llanuras.

En cada paso de la trayectoria necesitamos saber el valor de función objetivo asociado. Calcular la función objetivo puede ser costoso. Frecuentemente es posible calcular cuánto varía la función objetivo al desplazarme al vecino en vez de calcular toda la función desde cero. Ésta es una oportunidad que presentan las metaheurísticas de trayectoria.

Al recorrer una estructura de vecindad para desplazarme a una solución vecina, aparecen dos estrategias posibles: best-improvement, donde me desplazo hacia la mejor solución vecina, y first-improvement donde me desplazo hacia la primer solución vecina que mejora la solución actual. First-improvement es interesante cuando trabajamos con estructuras de vecindad muy grandes, donde recorrer a todos los vecinos implica un costo considerable.

Metaheurísticas Poblacionales

La diferencia de las metaheurísticas poblacionales respecto a las de trayectoria, es que en cada iteración manejamos un conjunto de soluciones en vez de una única solución.

1.2.1 Búsqueda Local Básica

El método más simple de trayectoria es la búsqueda local: dada una solución inicial nos desplazamos iterativamente a la mejor solución vecina, hasta quedar atrapados en un óptimo local.

El algoritmo es el siguiente:

```
s <- GenerarSoluciónInicial()
DO
        s <- Mejorar(N(s))          /* N(s) es la vecindad de s */
UNTIL no es posible mejorar
```

El problema de este algoritmo es que queda atrapado en un óptimo local, que puede no ser una solución lo suficientemente buena. Se han desarrollado técnicas que permiten escapar a este óptimo local, y esto tiene como consecuencia que las condiciones de terminación utilizadas son más complejas que el simple hecho de estar posicionados en un óptimo local.

Una primer estrategia para evitar quedar atrapados en óptimos locales sería recomenzar la búsqueda desde otro punto inicial. Esta es la idea de Búsqueda Local Iterativa (ILS: Iterated Local Search). En ILS se definen estrategias para establecer desde dónde voy a volver a aplicar la búsqueda local.

1.2.2 Recocido Simulado

Recocido simulado es una de las más viejas metaheurísticas y tiene una estrategia explícita para evadir óptimos locales. El origen de esta MH está en la mecánica estadística de cómo se enfrían los metales. La idea fundamental es que permitimos desplazarnos a soluciones vecinas peores que la solución actual con una cierta probabilidad p, y esta probabilidad decrece a lo largo de la ejecución.

El algoritmo comienza con una solución inicial, que puede ser creada de forma aleatoria, y se establece un parámetro global T que llamamos Temperatura. Luego se itera el siguiente procedimiento: elijo una solución vecina aleatoria, y la acepto si es mejor que la actual, y si es peor la acepto con una probabilidad que depende de qué tan peor sea y de la temperatura actual. Esta probabilidad se computa generalmente siguiendo la distribución de Boltzmann:

$$p = e^{-\frac{f(s')-f(s)}{T}}$$

(Donde f es la función objetivo, s y s' son las soluciones actual y vecina respectivamente, y T es la temperatura).

El algoritmo básico de recocido simulado es el siguiente:

```
s := GenerarSoluciónInicial();
T := T0;
WHILE no se cumplen condiciones de fin DO
       s' := ElegirVecinoAleatorio(s);
       IF f(s') < f(s) THEN
              s := s';
       ELSE
              p := CalcularProbabilidad(T, s', s);
              r := Aleatorio();     /* entre 0 y 1 */
              IF r <= p THEN
                     s := s'
              ENDIF;
       ENDIF;
       Actualizar(T);
END;
```

Está demostrado que Recocido Simulado converge a la solución óptima global si cumplimos con ciertas condiciones, en particular con la velocidad a la que disminuimos la temperatura. El problema es que necesitamos disminuir la temperatura de forma logarítmica, lo que representa un tiempo total de ejecución exponencial. Lo que se hace es elegir una velocidad para tener tiempos polinomiales, y con esto tenemos una probabilidad razonable de obtener soluciones cercanas al óptimo.

Las variantes de Recocido simulado más conocidas son probablemente Threshold Accepting, Great Deluge Algorithm, y Record-to-Record Travel.

Threshold Accepting

Threshold significa umbral. La idea es que el parámetro T se llama Threshold (umbral), y funcina de la siguiente manera: Sea D = f(s') – f(s) la distancia entre la solución actual y el vecino a aceptar.

Acepto este nuevo vecino si D < T. T representa cuánto acepto "empeorar" mi solución actual en términos absolutos. No necesito temperatura inicial no estrategia de enfriamiento, sólo el parámetro Threshold.

Great Deluge Algorithm

Estrategia del Gran Diluvio. La idea es limitar las soluciones vecinas que puedo aceptar, acotando su valor de función objetivo. Tengo un "techo" o "piso" del que no me puedo exceder (para minimización y maximización respectivamente). Esta cota se va desplazando a lo largo de la ejecución, limitando aún más las nuevas soluciones.

Si pensamos el problema de optimización como un problema de maximizar la función objetivo, entonces Gran Diluvio significa lo siguiente: la lluvia fija un cierto nivel de agua que es mi cota inferior para desplazarme. A medida que sigue lloviendo, el agua va subiendo. Los parámetros de este algoritmo son T (nivel del agua) y la velocidad de la lluvia.

Record-To-Record Travel

Para ver si acepto o no a mis nuevos vecinos, los comparo con la mejor solución que he obtenido. Si el vecino es peor a la mejor solución por una diferencia menor a cierto umbral D, entonces lo acepto. Sea R el valor registrado como mejor solución, y D el umbral aceptable. Si f(s') < R + D entonces acepto s'.

1.2.3 Tabu Search

Tabu Search es una reformulación de la búsqueda local, a la que se le agrega una estrategia para escapar de óptimos locales. Para explicar cómo funciona TS vamos a explicar cómo funciona en una estrategia bien simple y luego generalizamos.

En el caso más simple de TS tengo una "lista tabú" con las N soluciones visitadas recientemente. Las soluciones registradas en la lista tabú representan las soluciones hacia las que tengo prohibido desplazarme. Dada una solución actual, me desplazo hacia el mejor vecino (estrategia best-search). Este vecino puede ser peor que mi solución actual, pero cumple que no es tabú. La lista tabú es lo que permite escapar a óptimos locales: de no tenerla, el recorrido intentaría volver siempre al óptimo local luego de abandonarlo.

Generalizando, Tabu Search es una búsqueda con memoria, donde hay una o varias "listas tabú" que se mantienen a lo largo del recorrido. Estas listas tabú restringen los posibles vecinos a los cuales puedo acceder. Las listas Tabú pueden representar las soluciones hacia las que no me puedo desplazar, pero también pueden representar movimientos que no puedo efectuar, o componentes no admitidos para nuevas soluciones.

Cuando trabajamos con listas tabú de movimientos o componentes de soluciones, con un único registro en la tabla estamos restringiendo múltiples soluciones. Algunas soluciones que restringimos pueden ser muy buenas. Para solucionar este problema se definen "criterios de aspiración", que consiste en condiciones que permiten aceptar un vecino tabú. El caso más frecuente es aceptar soluciones vecinas mejores que la mejor solución encontrada hasta el momento.

La estrategia de desplazarme hacia el mejor vecino (best-improvement) la podemos remplazar por desplazarme hacia el primero que mejore la solución actual (first-improvement). Esto generalmente mejora el desempeño cuando trabajamos con vecindades muy grandes.

Ejemplo de Tabu Search

Sea un mapa de países, donde quiero pintar los países utilizando un máximo de k colores, y no puede haber dos países adyacentes con mismos colores. La solución inicial es una distribución arbitraria de colores. La función objetivo es la cantidad de países adyacentes con un mismo color, cantidad que voy a minimizar. Una solución vecina es, dada la solución actual, busco dos países adyacentes de mismo color y uno de ellos lo pinto de otro color.

En la lista tabú registro los pares (país, color) que pinto. La lista indica que por una cierta cantidad de iteraciones no voy a poder volver a pintar determinado país de determinado color.

Algoritmo de Tabu Search (ejemplo con estrategia best-improvement)

```
s := GenerarSoluciónInicial();
InicializarListasTabu(T1, …, Tr);

WHILE no se cumplen criterios de finalización DO
      /* busco mejor solución tal que no es tabú o bien cumple criterios de
       aspiración */
      s := MejorSolución(s, T1, …, Tr);
      ActualizarListasTabu(T1, …, Tr);
      ActualizarCondicionesAspiración();
END;
```

1.2.4 GRASP: Búsqueda Adaptativa, Aleatoria y Ávida

GRASP es una metaheurística multi-arranque. Cada iteración consiste en dos fases: construcción y búsqueda local. En la fase de construcción construimos una solución factible, y durante la fase de búsqueda local exploramos la vecindad hasta llegar a un óptimo local. La mejor solución encontrada hasta el momento es guardada como el resultado.

El algoritmo básico de GRASP es el siguiente:

```
FOR k = 1 TO MáxIteraciones DO
      s := ConstrucciónAleatoriaAvida(semilla);
      s := BúsquedaLocal(s);
      ActualizarSolución(s, MejorSol);  /*actualizo MejorSol si es mejor que s*/
END;
```

La construcción de la solución se basa en una estrategia ávida y aleatoria. Consideramos que una solución es el conjunto de los componentes de esta solución. Comenzamos entonces con una solución vacía a la que iterativamente le vamos a agregar sus componentes, sin destruir factibilidad.

En cada iteración de la construcción, para cada tipo de componente elegimos el componente de *costo incremental mínimo*, es decir, el que aumenta menos el valor de la función objetivo. Estos componentes cumplen además la propiedad de no romper la factibilidad de la solución. Nos queda entonces una lista que tiene, para cada tipo de componente, el componente factible de costo incremental mínimo. De forma aleatoria elegimos un componente a agregar de esta lista. Esta lista la llamamos Lista de Candidatos Restringidos (RCL: Restricted Candidate List).

El algoritmo básico para construir una solución es el siguiente:

```
s := SoluciónVacía();
EvaluarCostosIncrementales(s);

WHILE la solución no está completa DO
       RCL := ConstruirListaCandidatos(s);
       e := ElegirElementoAleatorio(RCL);
       s := Unir(s, e);
       EvaluarCostosIncrementales(s);
END;
```

Las variantes de GRASP se basan en distintos métodos para construir las soluciones.

1.2.5 Búsqueda en Vecindad Variable

La idea fundamental de VNS (Variable Search Neighborhood) es que cuando llegamos a un óptimo local, en realidad estamos en un óptimo local para la estructura de vecindad que estamos utilizando. Si tuviéramos otra estructura de vecindad mi solución actual probablemente no sería un óptimo local.

VNS es entonces una reformulación de la búsqueda local en la que recurrimos a varias estructuras de vecindad como estrategia para escapar a óptimos locales.

El algoritmo más simple de VNS es la variante llamada VNS descendente:

```
s := GenerarSoluciónInicial();

WHILE no se cumple condición de fin DO
       k := 1;

       WHILE k < k_max DO
              s'' := BúsquedaLocal( s');   /* búsqueda local sobre mi vecino*/
              IF f(s'') < f(s) THEN        /* me desplazo o no */
                     s := s'';
                     k := 1;               /* regreso a vecindad inicial */
              ELSE
                     k := k + 1;           /* avanzo a vecindad siguiente */
              ENDIF;
       END;
END;
```

En VNS descendente simplemente aplico búsqueda local con la estructura de vecindad actual, hasta llegar a un óptimo local. Al llegar a un óptimo local, empiezo a utilizar la siguiente estructura de vecindad. El detalle poco intuitivo es que cada vez que mejoro la solución actual, vuelvo a la vecindad inicial. Esto implica utilizar las vecindades según un orden de prioridad.

El algoritmo estándar o básico de VNS es el siguiente:

```
s := GenerarSoluciónInicial();

WHILE no se cumple condición de fin DO
       k := 1;
       WHILE k < k_max DO
              s' := SeleccionarAleatorio(N_k(s));/* Shaking: selecciono vecino
                                                   aleatorio */
              s'' := BúsquedaLocal( s');         /*búsqueda local sobre mi vecino*/
              IF f(s'') < f(s) THEN              /* me desplazo o no */
                     s := s'';
                     k := 1;                     /* regreso a vecindad inicial */
              ELSE
                     k := k + 1;                 /* avanzo a vecindad siguiente */
              ENDIF;
       END;
END;
```

En este algoritmo le agregamos la característica Shaking, que consiste en aplicar la búsqueda local sobre un vecino elegido arbitrariamente. Esta opción permite introducir aleatoriedad para obtener un algoritmo más estable, haciendo menos probable la posibilidad que la trayectoria se pierda en un ciclo.

Entre las demás variantes de VNS, algunas de las más conocidas son VNS sesgado y VNS con descomposición.

VNS sesgado es una reformulación del VNS básico en la que introducimos parámetros para manejar intensificación y diversificación. Nos desplazamos a una solución vecina cada vez que mejoramos la solución actual en un cierto valor, donde este valor depende de los parámetros.

VNS con descomposición consiste en restringir las estructuras de vecindad, de forma de considerar sólo vecinos que comparten una cierta cantidad de atributos con la solución actual.

1.2.6 Búsqueda Local Guiada

La idea básica de Búsqueda Local Guiada es modificar la función objetivo en tiempo de ejecución. De esta forma escapamos a mínimos locales modificando el "paisaje" que recorremos. En GLS (Guided Local Search, Búsqueda Local Guiada) mantenemos una estructura de vecindad estática mientras que la función objetivo es quien pasa a ser dinámica.

El mecanismo utilizado por GLS es basado en *características de soluciones*, donde cada característica de solución puede ser cualquier tipo de propiedad que una solución cumple o no.

Definimos la función $I_i(s)$ que indica si la característica i está presente en la solución s. $I_i(s) = 1$ si la característica está presente, $I_i(s) = 0$ en caso contrario.

Construimos una nueva función objetivo f' que considera las m características de soluciones:

$$f'(s) = f(s) + \lambda \sum_{i=1}^{m} p_i . I_i(s)$$

Los p_i se llaman *parámetros de penalización* y λ se llama *parámetro de regularización*. Los parámetros de penalización establecen los pesos respectivos de las características. El parámetro de regulación sirve para regular la relevancia de las penalizaciones frente a la función original.

El algoritmo GLS (Guided Local Search) es el siguiente:

```
s := GenerarSoluciónInicial();

WHILE no se cumplen condiciones de fin DO
       s := BúsquedaLocal(s, f');
       FOR todas las características i de utilidad máxima Util(s,i) DO
              pi := pi + 1;
       END;
       Actualizar(f', p);
END;
```

El algoritmo funciona de la siguiente manera: comienza por una solución inicial y aplica búsqueda local hasta llegar a un cierto mínimo local. Luego el vector p = (p_1, …, p_m) de penalidades es actualizado para incrementar algunas penalidades y la búsqueda local es ejecutada nuevamente. Las características penalizadas son aquellas que tienen una *utilidad* máxima:

$$Util(s,i) = I_i(s).\frac{c_i}{1+p_i}$$

Los valores c_i son los costos asignados a cada característica según su importancia relativa. Este costo es dividido por (1 + pi) para hacer que la utilidad considere la historia de la búsqueda.

El procedimiento de actualización de penalidades puede ser modificado para hacer que a medida que avanza la búsqueda, las modificaciones se hagan menos bruscas. Esto evita que el paisaje de búsqueda quede muy arrugado, y facilitar la convergencia.

1.2.7 Computación Evolutiva y Algoritmos Genéticos

La computación evolutiva intenta recrear artificialmente la capacidad de la naturaleza para evolucionar individuos que se adaptan a su entorno. En cada iteración, un conjunto de operadores es aplicado a los individuos de una determinada población para generar individuos de la población de siguiente generación (iteración).

Los algoritmos genéticos fueron desarrollados y aplicados por John Holland sobre fines de los años 60 en la universidad de Michigan, y publicó su libro en 1975. Estos algoritmos tienen como base el modelo evolutivo de la naturaleza.

En la naturaleza, cada individuo posee un código genético que condiciona sus características físicas. De esta forma queda condicionada también su aptitud a sobrevivir en el entorno. Los individuos menos aptos tienden a morir, y los más aptos a sobrevivir y tener descendencia. Los individuos nuevos heredan el código genético de sus padres, tienen 50% de cada uno, pero qué parte del código hereda de quién se produce de forma aleatoria. A este proceso se le llama Selección Natural, y contribuye a que las nuevas poblaciones hereden las mejores características de las poblaciones anteriores.

Al proceso de selección anterior la naturaleza le agrega el mecanismo de Mutación. El mecanismo de mutación consiste en que, al crear un nuevo individuo, existe una probabilidad muy baja de que un pequeño trozo de código genético sea alterado y diferente del de sus padres. Este mecanismo es crítico porque permite que evolucionen las poblaciones, adoptando características nuevas que sus antepasados no tenían.

Dado un cierto problema, creamos un conjunto inicial de soluciones. Cada solución es un individuo, y la función objetivo evalúa qué tan buena es. Recombinando soluciones vamos a obtener soluciones nuevas mejores, y podemos utilizar el mecanismo de mutación para no quedar atrapados en óptimos locales.

Ejemplo de un algoritmo genético simple:

```
Generar una población inicial;
Computar la función de evaluación de cada individuo;
WHILE NOT Terminado DO                  /* Producir nueva generación */
       FOR (Tamaño población / 2) DO    /* Ciclo reproductivo */
            Seleccionar dos individuos de la generación anterior, para el cruce
            (probabilidad de selección proporcional a la función de evaluación del
            individuo);
            Cruzar con cierta probabilidad los dos individuos obteniendo los
            descendientes;
            Mutar los dos descendientes con una cierta probabilidad;
            Computar la función de evaluación de los dos descendientes mutados;
            Insertar los dos descendientes mutados en la nueva generación;
       END;
       IF la población ha convergido THEN
              Terminado := TRUE
       ENDIF;
END;
```

1.2.8 Optimización de Colonia de Hormigas

La Optimización de Colonia de Hormigas surge de observar el comportamiento de las hormigas reales. Las hormigas son capaces de encontrar los caminos más cortos desde el hormiguero hacia la comida. Cada hormiga mientras se desplaza deposita en el suelo una sustancia llamada feromona. Para decidir hacia dónde avanzar, elige avanzar hacia donde siente un rastro de feromona más fuerte. Esta forma de colaborar lleva a que el hormiguero en su conjunto sea capaz de encontrar los caminos más cortos hacia las fuentes de comida.

Los algoritmos de Colonia de Hormigas (ACO: Ant Colony Optimization) se basan en un modelo probabilístico que modela los rastros de feromona. Las hormigas artificiales construyen soluciones de forma incremental, donde cada hormiga agrega de forma oportuna, un componente de la solución. Para hacer esto las hormigas realizan recorridos aleatorios en un grafo completamente conexo G = (C, L) cuyos vértices son componentes de solución y las aristas son llamadas *conexiones*. Este grafo se llama *Grafo de Construcción*. Cuando hay soluciones no factibles, es posible restringir este grafo quitando aristas. De todas formas a veces no es posible restringir tanto la búsqueda y es necesario relajar el problema para aceptar soluciones no factibles. Tanto los nodos como las aristas registran un valor de feromona.

El Algoritmo básico de ACO es el siguiente:

```
InicializarValoresFeromonas(T);

WHILE no se cumplen condiciones de fin DO
      FOR cada hormiga del hormiguero DO
            sa := ConstruirSolución(T, H);
      ENDFOR;
      ActualizarFeromonas(T, sa);    /* agrega feromona de hormiga y evapora */
END;
```

Al inicializar los valores de feromonas, se establece un mismo valor inicial a todas las aristas y nodos. Al construir una solución, la hormiga construye una solución agregando componentes. Cada nodo recorrido representa un componente a agregar. La hormiga elige aristas utilizando probabilidades de transición que se ajustan a la siguiente regla de transición:

$$p(c_r \mid s_a[c_i]) = \frac{[\eta_r]^\alpha [\tau_r]^\beta}{\sum_{c_u \in J(s_a[c_i])} [\eta_u]^\alpha [\tau_u]^\beta} \quad \text{para} \quad c_r \in J(s_a[c_i]), \text{ 0 en otro caso.}$$

Donde:

$c_i \in C$ son los componentes (nodos) de la solución.
τ_i es el valor de feromona en el nodo i.
τ_{ij} es el valor de feromona de la arista (i, j).
η_i es el *valor heurístico* del nodo i.
η_{ij} es el *valor heurístico* de la arista (i, j).
α y β son parámetros que representan el peso relativo de las feromonas frente a los valores heurísticos.
$J(s_a[c_i])$ denota el conjunto de componentes de solución que son admitidos para agregar a la solución actual.

Para simplificar, la regla de transición es tal que la hormiga prefiere ir por donde hay más feromona. α y β son parámetros para ajustar el peso relativo de la feromona y los valores heurísticos de los nodos.

Al actualizar feromonas, se agregan los trazos de feromonas de la hormiga y se evapora de forma general todo el grafo.

2 Ejemplos de problemas de optimización combinatoria

2.1 QAP: Quadratic Assignment Problem

2.1.1 Descripción

El problema QAP significa en Inglés "Quadratic Assignment Problem" y en Español se conoce como "Problema de Asignación Cuadrática".

El problema consiste en distribuir N fábricas entre los N nodos de un mapa, de forma tal de minimizar el costo total de transporte. Dadas dos fábricas i y j, el costo asociado se define como el producto de la distancia entre esas fábricas, por el flujo de materiales entre ellas.

Una solución al problema es una determinada forma de distribuir las fábricas, que se puede representar como un vector de largo N: si la celda 7 contiene el valor 3, significa que la fábrica 7 va emplazada en el nodo 3. El espacio de soluciones posibles son todas las posibles permutaciones de fábricas.

Las distancias entre los nodos las podemos representar como una matriz cuadrada de distancias donde la celda (i,j) representa la distancia entre los nodos i y j. Análogamente, representamos los flujos entre fábricas como una matriz cuadrada en la cual la celda (i,j) representa el volumen de materiales desde la fábrica i hacia la fábrica j.

2.1.2 Función objetivo

La función objetivo se puede evaluar de la siguiente forma:

$$\sum_{a,b \in P} w(a,b) \cdot d(f(a), f(b))$$

Donde:

- P es el conjunto de fábricas
- w(a,b) es el flujo desde la fábrica a hacia la fábrica b
- f(a) y f(b) representan los nodos respectivos de a y b
- d(f(a), f(b)) es la distancia entre los nodos respectivos de las fábricas a y b

Esta función objetivo tiene un tiempo de cálculo asociado de orden N^2.

2.1.3 Estructura de vecindad

La estructura de vecindad que utilizamos en nuestra implementación consiste en, dada una solución actual, intercambiar dos fábricas de lugar.

Al desplazarnos a un vecino, los costos que cambian son los costos asociados a las fábricas que cambian su posición. Los costos entre fábricas sin mover permanecen iguales. Esto hace que sea posible calcular cuánto varía la función objetivo al desplazarme a un vecino en tiempo de orden N.

2.1.4 Construcción de soluciones

Para QAP el método de construcción de soluciones que utilizamos es simple. Inicialmente tenemos un mapa vacío sin fábricas, e iterativamente vamos agregando fábricas hasta completar el mapa.

El costo incremental al agregar una nueva fábrica es el costo asociado a los nuevos flujos que introduce la fábrica.

2.2 TSP: Travelling Salesman Problem

2.2.1 Descripción

El problema TSP significa en Inglés Travelling Salesman Problem y en Español se conoce como Problema del Agente Viajero.

El viajero necesita recorrer un conjunto de ciudades en forma de “tour”, es decir, viajar por todas las ciudades pasando exactamente una vez por cada ciudad y volviendo a la ciudad de origen.

Una solución posible al problema es una secuencia de ciudades. El espacio de soluciones son todas las permutaciones de nodos posibles.

2.2.2 Función objetivo

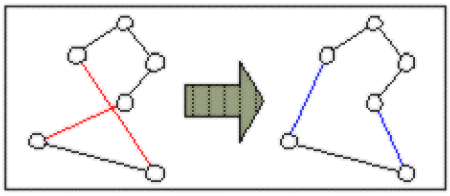
El costo asociado a una solución es la suma de las distancias de todos los trayectos recorridos, incluyendo en trayecto de regreso a la primer ciudad.

El objetivo es encontrar una solución de costo mínimo.

2.2.3 Estructura de vecindad

La estructura de vecindad que utilizamos se conoce como “2-opt” y consiste en, dada una solución actual, quitar dos aristas no consecutivas y agregar las únicas otras dos aristas que es posible agregar.

La siguiente imagen muestra un ejemplo:



2.2.4 Construcción de soluciones

El método que utilizamos para construir soluciones TSP se conoce como "inserción más cercana".

El método está definido por los siguientes pasos:

1. Elegir un vértice inicial (tour vacío)
2. Elegir 2º vértice para el tour
3. Elegir 3er vértice para el tour
4. Elegir un vértice j* que aún no esté incluido el tour. Para este vértice j* existe un vértice k* perteneciente al tour tal que la distancia entre k* y j* es mínima. Puedo insertar j* antes de k* o luego de k*, elijo la que sea mínima.

3 Herramientas de Metaheurísticas existentes

A continuación presentamos diez productos existentes que consideramos representativos - a grandes rasgos – de los productos de software disponibles, sus alcances, formatos, y estrategias. Incluimos para cada producto una breve descripción o síntesis de su arquitectura y diseño junto con explicaciones sobre cómo interactúan estos productos con el usuario programador.

Los productos de software que presentaremos son GAlib, EasyLocal ++, MALLBA, TEA, TSF, Open BEAGLE, ECJ, JDEAL, OpenTS y HOTFRAME. Luego de presentar estos diez productos incluimos en el documento una sección con descripciones breves de otros productos existentes similares a los que detallamos previamente.

3.1 GAlib: Genetic Algorithms Library

3.1.1 Introducción

GAlib [GAL01] está desarrollada en C++. La idea del autor era crear una biblioteca de algoritmos genéticos que contenga las diversas opciones y variantes de los algoritmos genéticos, que sea extensible, y que permitiera desarrollar aplicaciones razonablemente rápidas. Por rápidas se entiende eficientes en el consumo de recursos computacionales.

Una ventaja de utilizar esta biblioteca es que ya tiene muchos usuarios, y es posible ponerse en contacto con ellos a través de newsgroups y mailing lists existentes.

La idea básica que vemos en GAlib es que al desarrollar una solución de algoritmos genéticos, podemos dividir nuestro trabajo en los siguientes componentes: qué algoritmo utilizamos, cómo representamos cada solución, y cómo recombinamos soluciones. Estas tres partes las podemos pensar como “cajas negras” que podemos intercambiar. Para ello cada una implementa una cierta interfase.

3.1.2 Arquitectura

La arquitectura que nos presenta este framework es a través de clases, aprovechando las ventajas de la programación orientada a objetos.

Figura 1: Clases y subclases mas importantes de GAlib

Como se muestra en la figura 1, se tiene ciertas clases principales las cuales son las encargadas de abstraer las diferentes partes que contiene el framework. Dichas clases son:

- GAGeneticAlgorithm: de donde se derivan los algoritmos genéticos.
- GAScalingScheme: de donde se derivan los tipos de "escalado".
- GASelectionScheme: de donde se derivan los métodos de selección.
- GAGenome: de donde se derivan las estructuras que contienen los datos de un individuo.
- GAStatistics: que provee información relativa a la ejecución del algoritmo
- GAParameterList: la idea es tener una colección "genérica" en la que guardo todos mis parámetros del algoritmo
- GAPopulation: una instancia de esta clase contiene instancias de individuos

A partir de cada una de estas clases se derivan clases que implementan los diferentes tipos de implementaciones que pueden tener cada una de ellas. De esta forma se permite modularizar cada problema que se tiene a la hora de desarrollar una aplicación de este estilo, por lo que es mas fácil de entender y mas rápido para desarrollar.

3.1.3 Modo de uso de GAlib

El usuario programador trabaja principalmente con dos clases: un Genoma y un Algoritmo Genético. Cada instancia de genoma representa una solución al problema. El objeto Algoritmo Genético define cómo se va a llevar adelante la evolución. El usuario debe implementar para el Algoritmo Genético la función objetivo, los operadores para manipular el genoma, y las estrategias de selección / reemplazo.

El usuario programador debe hacer las siguientes tres cosas:
1. Definir una representación para las soluciones.
2. Definir los operadores genéticos.
3. Definir la función objetivo.

GAlib ya trae implementados ciertas representaciones (genomas) y operadores. Un ejemplo de un genoma podría ser un array de enteros, o un String.

3.2 EasyLocal++

3.2.1 Introducción

Es un framework orientado a objetos [ELO01] que puede ser usado como herramienta para construir algoritmos para búsquedas locales en C++. La idea de la arquitectura está basada en la esencia de las metaheurísticas de búsqueda local y en sus posibles composiciones. Una de sus principales virtudes es la posibilidad de crear e implementar nuevas heurísticas fácilmente y además permite la posibilidad reutilizar el código para la creación de las mismas.

El framework aún no está completamente implementado para el uso. Algunas de las partes del mismo las versiones actuales son inestables. Consideramos que la documentación no es tan desarrollada como la de otros productos.

3.2.2 Arquitectura

La arquitectura consiste en modulación por capas de cuatro niveles, en el cual cada nivel se basa en los servicios brindados por las capas o niveles inferiores, y provee operaciones más abstractas. Para una mejor compresión se muestra una imagen de la abstracción de los niveles utilizada por el EasyLocal++, en la Figura 2.

Figura 2: Niveles de abstracción de EasyLocal++

NIVEL 1: Solver ----- Estrategia del solver de la búsqueda local (Simple solver, Token ring solver)

NIVEL 2: Runners ------ Metaheurísticas de búsqueda local (Hill Climbing, Tabu Search, Simulated Annealing)

NIVEL 3: Helpers ------ Problema de búsqueda local (State manager, Neighborhood explorer)

NIVEL 4: Contiene 4 partes: Input, output, state y move. ------ Datos básicos

El problema de representación ocurre en le nivel 4, y debe ser definido en términos de los datos básicos. Estas clases son usadas por el framework como un template de instanciación para las demás clases y son utilizadas además para mantener los estados, la entrada/salida y los movimientos que realizará el algoritmo.

En el nivel 3 se define el problema de la búsqueda local donde se pueden especificar ciertos aspectos específicos de la búsqueda como por ejemplo la exploración de la vecindad.

En el nivel 2 se representa el algoritmo a utilizar. Aquí es donde se produce la corrida de la metaheurística de búsqueda local, empezando por una solución inicial y siguiendo el algoritmo hasta llegar a alguna solución que cumpla con ciertos requisitos definidos de antemano. Cada una de las clases que se utilice aquí tiene varios objetos que representan el estado de la búsqueda, y mantiene una "conexión" con cada un de los "helpers" (nivel 3) que se hayan invocado en las tareas de nuestros datos. El EasyLocal++ ya trae algunos de estos algoritmos, como por ejemplo Tabu Search.

El nivel 1 representa el software externo que será utilizado para generar soluciones iniciales y cómo crearlas, y en qué secuencia. Esta capa permite la creación de nuevas metaheurísticas a partir de la combinación de otras básicas y la posibilidad de tener algoritmos híbridos. Esta capa se comunica con el exterior a través del la entrada y salida antes descriptas, y mantiene una correspondencia con los "helpers" y "runner" invocados.

3.2.3 Modo de uso de EasyLocal++

El framework se utiliza creando las clases del nivel 1 y creando las clases derivadas de los "helpers" (capa 2) que contiene la especificación de cada problema. Los solvers y los runners son provistos por el framework y lo único que se debe hacer es setearlos en los "helpers". En síntesis lo que debe de crear un usuario para poder implementar un algoritmo para solucionar un problema es crear ciertas clases que definan las especificaciones del problema y definiendo como interactúan los diferentes componentes que ya trae el framework.

3.3 MALLBA

3.3.1 Introducción

Es una propuesta realizada en España por 3 grupos de investigadores, que surge con el objetivo de proporcionar una biblioteca de esqueletos algorítmicos para optimización combinatoria, y poder crear algoritmos híbridos de solución de los mismos. Es una librería realizada en C++ que pretende ser un código el cual se debe "rellenar" con las especificaciones particulares de cada problema. La idea de esta forma de trabajo es la de simplificar la tarea del usuario final proporcionándole la base del código y que el usuario sólo se limite a la tarea de solucionar los problemas particulares que representa la implementación de cada problema.

La Biblioteca MALLBA [MAL01] puede obtenerse gratuitamente, rellenando un documento de licencia, por lo cual a partir de esta librería cada usuario puede definir su propia librería. Además esta biblioteca tiene como particularidades las de centrarse en la eficiencia del código y la posibilidad de generar algoritmos paralelos.

3.3.2 Arquitectura

La arquitectura se muestra en la Figura 3, y las clases que se muestran se pueden catalogar en dos categorías: Clases provistas y clases requeridas.

Figura 3: Abstracción de clases para MALLBA para un problema particular

Las clases provistas son aquellas responsables de implementar toda la funcionalidad básica del algoritmo correspondiente. La clase Solver es la que encapsula el motor de optimización del algoritmo, luego la clase SetupParams es la encarga de de tener los parámetros propios de la ejecución. Después están las clases Statics la cual es la responsable de generar las estadísticas y las clases StateVariable y StateCenter que facilitan la hibridación de algoritmos.

Las clases requeridas son aquellas responsables de proporcionar al esqueleto los aspectos particulares de cada problema objetivo. Entre estas clases se encuentra la clase Problem la cual es la

responsable de proporcionar los métodos necesarios para manipular los datos del problema, la clase Solution que es la encargada de manejar las soluciones parciales del algoritmo, la clase UserStatics es la encargada de manejar las estadísticas deseadas que no estarían en la clase Statics.

3.3.3 Modo de uso de MALLBA

La forma de trabajo de la biblioteca MALLBA es la de implementar las clases requeridas y la de setear los parámetros en la clase SetupParams, de esta forma se pueden crear diferentes algoritmos partiendo siempre del esqueleto formado por las clases provistas, siendo así una forma sencilla de crear diferentes algoritmos.

3.4 TEA (Toolbox of Evolutionary Algorithms)

3.4.1 Introducción

Tea Library [TEA01..03] es una librería desarrollada en C++ con el fin de poder crear nuevos algoritmos o modificar los algoritmos ya existentes en el ámbito de los Algoritmos Evolutivos. Dentro de estos se pueden crear y/o modificar algoritmos genéticos y programación evolutiva.

A pesar de que se pueden crear nuevos algoritmos estos están atados a las clases ya realizadas por lo que los algoritmos creados serán modificaciones entre si ya que las partes que los componen son acotadas, pero este framework no queda solo con los módulos que presentes, sino que se mantiene en constante evolución ya que hay una pequeña comunidad la cual se dedica a utilizar dicho framework y a expandir la cantidad de módulos para poder crear mas cantidad de nuevos algoritmos, es mas, se puede crear cada usuario nuevas clases C++ para poder crear los algoritmos que cada uno quiera probar.

3.4.2 Arquitectura

La arquitectura de esta librería esta formada por tres niveles diferentes: Cromosomas, individuos y Poblaciones, cada una de ellas conteniendo su respectiva interfase (teaChromosome, teaIndividual y teaPopulation respectivamente).

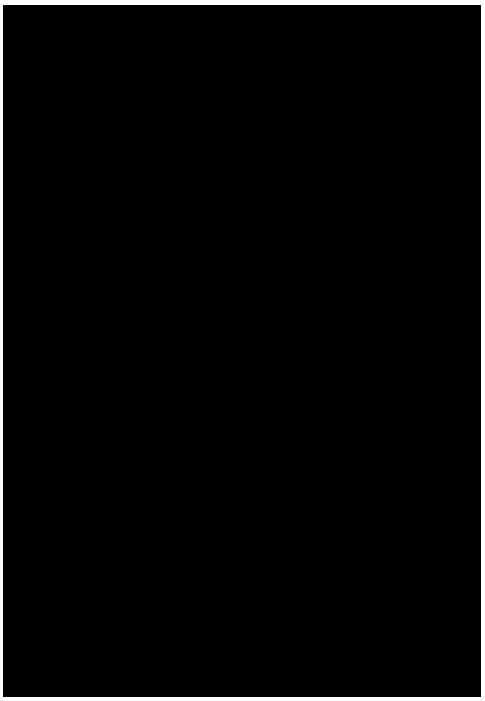


Figura 4: Clases mas importantes de TEA

Como se ve en la Figura 4 las poblaciones están formadas por uno o más individuos, mientras que cada individuo esta formado por uno o varios cromosomas, mientras que un genotipo esta formado también por uno más cromosomas. Los individuos contienen todos los cromosomas de un genotipo y además contienen información para calcular la función fitness. El manejo de los cromosomas y de la forma de mutación de los mismos está contenidas dentro de estas interfases, tanto en la de los individuos como en la de los cromosomas. Las poblaciones es la interfase que se encarga de manejar las transiciones de las distintas generaciones que se van creando en el algoritmo. Como se ven en la Figura 4 a cada interfase se tienen ciertas funciones:

- Poblaciones
  - Envolve: int -> P --Evoluciona un población en la cantidad de generaciones ingresada
  - Terminate: P -> Bool – Chequea el criterio de terminación
  - GetParners: P -> I* -- Elige los siguientes individuos para la precombinación de la población
  - Replace: P -> P –- Selecciona individuos para la creación de la nueva población
- Individuos
  - GetFitness: I -> val – Selecciona la cantidad de individuos para la función fitness
  - IsWorse: val x val -> Bool – Chequea la severidad de violar las reglas
  - Recombine: I* -> I* -- recombina un nuevo grupo de individuos a partir de un grupo de individuos
  - Mutate: I -> I – Realiza la mutación de un individuo
- Cromosoma
  - Recombine: C* -> C* -- Recombina un nuevo grupo de cromosomas a partir de un grupo de cromosomas.
  - Mutate: C -> C – Realiza la mutación de un cromosoma.

Vale recordar que ya se encuentran implementados en dicha librería varios modelos de poblaciones, individuos y cromosomas para poder trabajar con ellos.

3.4.3 Modo de uso de TEA

La forma de trabajo de esta librería es programando un main e ir creando a medida que se presicen nuevas instancias de las clases de poblaciones, individuos y cromosomas de los que ya se tienen implementados en la librería, creando así algoritmos que difieren en que tipo de población o individuo que se eligió para cada caso. Esta forma de trabajo permite crear varios algoritmos diferentes pero para crear nuevos modelos de poblaciones, individuos o cromosomas la única forma es implementando dentro de la librería nuevas clases que implementen cada uno de las interfases antes mencionadas.

3.5 TSF (Tabu Search Framework)

3.5.1 Introducción

Framework orientado a objetos realizado en C++ [TSF01] el cual utiliza interfaces para utilizar algoritmos basados en la estrategia de Tabu Search. Una de sus virtudes es que no esta restricto a una única representación y además provee una buena definición de la estructura de la búsqueda. Otra virtud con respecto a los demás frameworks de su estilo es su flexibilidad para incorporar varias estrategias de Tabu Search. La principal idea del mismo es la de ofrecer a los investigadores que lo utilicen se concentren en el diseño y el testeo de algoritmos y no se focalicen tanto en la programación del mismo.

3.5.2 Arquitectura

TSF esta compuesto por cuatro categorías de elementos: interfases, mecanismos de control, mecanismo de búsqueda y la librería de software de estrategias. Las interfases son las que definen los componentes básicos del Tabu Search. El mecanismo de control es utilizado por los usuarios para crear reglas para guiar la recorrida según pasen los eventos. Para solucionar los problemas de programación se creo el Engine Search, el mismo interactúa con las interfases para colectar la información y se la pasa al mecanismo de control dinámicamente para reajustar la próxima trayectoria. La librería de software de estrategias contiene diferentes paquetes para utilizar por el framework con distintas estrategias de búsqueda ya definidas.

Figura 5: Arquitectura de TSF

Interfases:

- contiene 7 interfases distintas
  - Solution (Solución): define la representación de la solución.
  - Objective function (Función objetivo): es la que evalúa el valor objetivo de la solución.
  - Neighborhood (Vecindario): genera una lista de vecindades usando las interfases Move y Constraint.
  - Move (Moverse): es la encargada de moverse a la siguiente solución.
  - Constraint: (Condiciones): es la que evalúa si la siguiente solución cumple con los requisitos para ir a ella.
  - Tabu list (Lista Tabu): es la que guarda los movimientos o las soluciones intermedias.
  - Aspiration Citeria (Criterio de aspiración): contiene las especificaciones de las soluciones.

Mecanismo de Control:

- consta de dos partes
  - Switch box (Caja de cambios): es la encargada de controlar las operaciones básicas de la Search Engine.
  - Event controller (Controlador de eventos): es la que hace reaccionar de diferentes formas al framework cuando ciertos eventos ocurren. Trabaja recolectando de las interfases la información para poder decirle a la Search engine que hacer partiendo de estrategias definidas por el usuario. Los eventos pueden ser definidos por el usuario, pero también existen algunos eventos ya incorporados por el framework (Tabu Search Start, Tabu Search Stop, etc.).

Librería de software de estrategias:

- a veces es necesario o se quiere utilizar estrategias mas complejas, y a través de esta librería es capaz de utilizar estrategias complejas. Algunas de estas estrategias son: Very Large-Scale Neighborhood (VLSN), Candidates List o Intensification search on Elite solutions.

3.5.3 Modo de uso de TSF

La forma de trabajar con este framework es implementando las interfases antes descriptas para poder definir los elementos a utilizar, las estrategias, los criterios, ver los movimientos, etc. También se pueden crear eventos dentro del Controlador de eventos para poder realizar diferentes estrategias a partir de diferentes lugares de disparos de ciertos procedimientos. También dentro de estos eventos se pueden utilizar diferentes estrategias llamando a la librería de software de estrategias. De esta forma se pueden crear varias diferentes estrategias para poder probar y encontrar mejores soluciones, concentrándose más en ver la teoría de la estrategia más que en la programación de la misma.

3.6 Open BEAGLE

3.6.1 Introducción

Open BEAGLE [OBE01] es un framework realizado en C++, para la resolución de problemas con técnicas de Computación Evolutiva. Dicho framework es de código libre por lo que se puede conseguir su código y también se puede realizar contribuciones al mismo, para la evolución del mismo.

Al ser realizado en C++ provee las virtudes que nos provee la programación orientada objetos, además contiene ciertas ventajas, a ser: contiene una interfase de programación amigable (por ejemplo el manejo de memoria utiliza garbage collection), es lo suficientemente genérico para poder aplicar cualquier tipo de algoritmo de computación evolutiva, contiene una optimización de código de las partes mas criticas para este tipo de algoritmos, contiene secciones de código para la corrección de la robustez del software.

Otra virtud de este software es la arquitectura en capas (que veremos en el siguiente punto) ya que permite que el mismo sea más entendible y más fácil de proveer mejoras y nuevos módulos al mismo.

3.6.2 Arquitectura

La arquitectura propuesta en este framework es de tres capas, como se muestra en la Figura 6. Las tres capas que se presentan son: una capa con los fundamentos de la programación orientada a objetos la cual se comunica con las librerías estándar de C++, otra capa la cual contiene un framework genérico para algoritmos de evolución computacional y la tercer capa que contiene una serie de módulos que permiten realizar los diferentes tipos de algoritmos que comprenden la evolución computacional.

Figura 6: Arquitectura de Open BEAGLE

La capa que contiene los fundamentos de la programación orientada objetos es la que se ocupa de todo el diseño que tendrá el framework, las clases que las compondrán y cuales serán sus funcionalidades. También define los formatos que tendrán las poblaciones, como serán representados los resultados y como pasar los parámetros pertinentes para cada problema particular. Estas representaciones se definen los archivos XML (eXtensible Markup Language) permitiendo de esta forma la flexibilidad y la amigabilidad para su entendimiento. Además esta capa permite utilizar las librerías estándar de C++, ya que ambas contienen el mismo formato.

La capa que contiene el framework genérico, el cual es una extensión de los fundamentos de la programación orientada objetos presentados anteriormente. Esta sección contiene las estructuras básicas de las poblaciones, sistemas de evolución y el conjunto de operadores. Los módulos que componen la sección genérica pueden ser remplazados o especializados para poder implementar un algoritmo específico.

Por ultimo se tiene los módulos que implementan los diferentes tipos de algoritmos que componen la evolución computacional. En estos momentos se tiene implementados solo tres tipos: Algoritmos genéticos, árboles GP y algoritmos co-evolutivos. Estos módulos reemplazan o especifican ciertos elementos del framework genérico para poder realizar cada uno de los algoritmos descriptos anteriormente.

3.6.3 Modo de uso de Open BEAGLE

La forma de trabajo con Open BEAGLE es bastante sencilla ya que para crear una aplicación del mismo solo se necesita crear un operador de evaluación y un procedimiento main para inicializar y definir los diferentes componentes del algoritmo. Esto puede se así ya que el framework trae por defecto definido todos los parámetros necesarios para la corrida de un algoritmo genérico, pero estos parámetros pueden ser modificados si se desean, modificando los archivos XML que provee el framework. Una vez creados y modificados dichos elementos ya se tiene todos los elementos para poder crear nuestro algoritmo.

3.7 ECJ (Evolutive Computering in Java)

3.7.1 Introducción

ECJ [ECJ01] es un producto para la creación de algoritmos para la resolución de problemas combinatorios utilizando técnica de evolución computacional. Dicho producto esta realizado en Java, por lo que su diseño esta creado en base a la programación a objetos y permite la funcionalidad en cualquier plataforma, y permite trabajar de forma flexible con sus clases ya que debido a su modularización son fáciles de modificar para crear clases propias que realicen lo que uno quiere, pero a pesar de su flexibilidad, una de sus principales razones de existir es la de poner énfasis en la eficiencia.

Otras de sus particularidades es que permite el multihilado para poder acelerar los cálculos, maneja los parámetros de cada algoritmo en un archivo de configuración por lo que para modificar alguno de estos parámetros solo es necesario modificar dicho archivo y además es un software de código libre, por lo que se puede conseguir el mismo de forma gratuita para poder estudiarlo y modificarlo a gusto de cada usuario.

3.7.2 Arquitectura

Las clases que componen dicha aplicación están modularizados de forma que cada paquete maneje cierta parte del algoritmo o sea utilizado según los requerimientos que cada algoritmo necesito. Los paquetes más importantes, y las clases que las componen y dichos paquetes son (para ver los todos los paquetes y sus clases remitirse a las referencias del producto):

- ec.util: este paquete contiene clases las cuales no están relacionadas propiamente con los algoritmos evolutivos, pero que son útiles para el buen funcionamiento de los mismos. Contiene por ejemplo el generador de números aleatorios, el manejo de los logs y el manejo del archivo de parámetros.

- ec: es el modulo que contiene todo el diseño abstracto del mecanismo de evolución. Contiene los patrones de diseño utilizados, las abstracciones de alto nivel de los objetos y de los procesos, y los puntos de entrada para los programas.

- ec.simple: contiene las implementaciones básicas para los objetos que participan en los algoritmos.

- ec.multiobjetive: provee las clases para crear algoritmos con objetivos múltiples.

- ec.select: contiene las implementaciones de los métodos de selección.

- ec.es: contiene las clases que hacen posible los procedimientos comunes de selección (mu,lambda – mu+lambda) en las estrategias de evolución.

- ec.experiment: aquí se pueden crear clases para experimentar nuevos algoritmos.

3.7.3 Modo de uso de ECJ

Una de las particularidades de este software es su forma de uso, ya que todo se remite a la realización de dos archivos: un archivo de parámetros y un archivo de extensión .java el cual define dos funciones. El archivo de configuración define todo lo necesario para definir todas las variables, procesos y objetos que se utilizaran para la resolución de los problemas. Dentro de este archivo se define una variable (eval.problem) la cual hace el enlace con el archivo que contra las operaciones evaluate y describe, los cuales definen el problema particular que queremos resolver. Una vez creados estos dos archivos se corre el programa y se obtiene una salida con los resultados que arroja el algoritmo seleccionado.

3.8 JDEAL (Java Distributed Evolutionary Algorithm Library)

3.8.1 Introducción

JDEAL[JDE01, JDE02] , tal como lo dice su nombre es una librería desarrollada en Java para la creación de algoritmos evolutivos de forma distribuida. Al ser desarrollada en Java cuenta con las ventajas que nos propone la programación orientada a objetos y de las librerías que este lenguaje nos provee.

Esta librería nos permite realizar implementación de algoritmos evolutivos de alta calidad, crear una fácil integración entre operadores, cromosomas y algoritmos, fácil extensión y reuso de componentes ya existentes. Además posee un claro análisis y un diseño fácil de entender, que se adapta a la posibilidad de crear algoritmos paralelos y distribuidos. Además de estas ventajas también es una librería que puede ser utilizada en otros algoritmos, tanto a nivel de línea de comandos como de interfaz grafica, y provee la facilidad de crear nuevos algoritmos, combinándolo con otros tipos de algoritmos.

3.8.2 Arquitectura

Dicha librería contiene una arquitectura muy amplia, por lo cual a continuación presentaremos los objetos más importantes, los cuales son los utilizados por el usuario a la hora de poder crear los algoritmos. Dentro de estos objetos tenemos as clases EvolutionayAlgorithm, GeneticAlgorithm, EvolutionStrategy, Chromosome, Mutator, Selector, ChromosomeEvaluator y PopulationEvaluator, las interfases Scaling y Termination, y los objetos Statics.

A continuación presentamos las clases:

- EvolutionaryAlgorithm todos los algoritmos evolutivos que queramos crear derivaran de dicha clases, la cual implementa no solo las tareas básicas de dichos algoritmos (Inicialización, evolución, terminación del test, finalización) , tares de manejo de los cromosomas y las poblaciones (inicialización, evaluación, selector, terminación) , sino que también genera un conjunto de eventos para los momentos críticos del algoritmo y como manejarlos.

- GeneticAlgorithm: es una derivación de la clase antes presentada EvolutionayAlgorithm, la cual agrega los operadores de CrossOver y Mutation para el manejo particular de los algoritmos genéticos.

- EvolutionStrategy: esta clase es utilizada para configurar que clase de estrategia de evolución se va a utilizar.

- Chromosome: en esta clase se define la representación que tendrá la posible solución de nuestro problema a resolver. Para definir una nueva representación se debe realizar una clase que derive de dicha clase y además definir nuevos operadores de CrossOver y Mutation para dicha representación.

- Crossover: esta clase presenta al operador que combina las soluciones que contiene una población en orden para poder generar nuevas soluciones para la siguiente generación. Esta clase contiene dos clases que derivan de ella, GACrossover y ESCrossover. Un operador a ser utilizado en el algoritmo debe derivar de alguna de estas clases, según que tipo de algoritmo se vaya a utilizar, si el algoritmo a utilizar es un algoritmo genético entonces el operador debe derivar de GACrossover, mientras que si el algoritmo es una estrategia de evolución entonces debe derivar de ESCrossover.

- Mutation: esta clase presenta otro operador combinatorio el cual es utilizado para crear cierta diversidad entre las soluciones de una población. Para realizar un nuevo operador se debe implementar una clase que derive de esta.

- Selector: esta clase es la responsable de elegir de entre la población los cromosomas que serán utilizados para la recombinación. En muchos casos esta selección esta basado en los valores de los cromosomas. Para crear nuevos selectores se debe crear una clase que derive de Selection, algunos selectores utilizan objetos Scaling (implementaciones de la interfaz Scaling) para calcular los valores de los cromosomas y las diferencias entre los valores. Esta clase cuando es pasada al algoritmo evolutivo queda registrada como un listener, y es notificada cuando el Scaling retorna el valor del cálculo realizado.

- ChromosomeEvaluator: es la encargada de evaluar que tan bueno es un cromosoma como solución del problema, también es el nexo entre el usuario y el algoritmo, ya que es lo que el usuario desea como solución del mismo.

- PopulationEvaluator: es la encarga de evaluar los cromosomas que contiene cierta población, utilizando la clase ChromosomeEvaluator descrito anteriormente. Los algoritmos para realizar estos debe estar en clase que deriven de dicha clase.

Ahora le presentamos las interfaces más importantes:

- Scaling: es la interfaz que permite definir las funciones que se utilizaran para calcular los valores de los cromosomas o para poder crear grupos de cromosomas según ciertas diferencias entre sus valores. Para poder utilizar estos objetos se deben crear clases que implementen esta interfaz.

- Termination: es la interfaz que permite definir clases las cuales implementen cuando terminar el algoritmo en ciertas condiciones. Para poder utilizar esto se deben crear clases que implementen esta interfaz.

Por ultimo queda definir los objetos Statistics, los cuales son utilizados para poder visualizar datos sobre el algoritmo, estos datos pueden ser tanto en tiempo de ejecución (ver los estados en que se encuentra, las soluciones parciales, los resultados finales, etc.), para poder ver estos datos se utiliza los objetos Statictics Writers, los cuales contienen funcionas con estos propósitos, los cuales permiten ir escribiendo en un archivo los distintos datos que se quieran visualizar.

3.8.3 Modo de uso de JDEAL

Para poder resolver un problema de optimización combinatoria con JDEAL se necesita, como mínimo, los siguientes pasos que describiremos a continuación. Primero se debe saber cual es la representación de nuestra solución, por lo cual debemos utilizar un Cromosoma, este cromosoma puede ser uno definido por el usuario (creando una clase derivada de Chromosome, como vimos anteriormente) o utilizando uno que ya provee la librería. Luego de crear o elegir el cromosoma lo cremaos dentro de nuestra rutina principal. Luego debemos definir que algoritmo utilizaremos, esto se hace utilizando las clases derivadas de EvolutioayAlgorithm, creadas por el usuario o ya definidas, y creando una instancia de esta. Después de esto se debe definir la función objetivo, utilizando, como vimos anteriormente, la clase ChromosomeEvaluator y creando una instancia de esta (siendo una clase ya provista por la librería o creando una nueva). Después de definir y crear estos objetos se define sobre el algoritmo los operadores (crossover y mutator, los cuales son instancias de clases derivadas de las clases del mismo nombre, las cuales pueden crearse o utilizar las provistas), la clase Selector (igual que con los operadores), la clase de terminación, el evaluador creado anteriormente y que clase de optimización se quiere utilizar. Además de estas cosas se le puede definir que estadísticas presentar del algoritmo, esto se realiza asociando al algoritmo las clases que contiene la presentación de estadísticas, que contiene la clase StatisticsWriter.
Con todos estos elementos creados ya tenemos todo lo necesario para correr nuestra aplicación para resolver un problema de optimización combinatoria con el algoritmo, los criterios y representaciones deseadas.

3.9 Open TS (OPEN Tabu Search)

3.9.1 Introducción

OpenTS [OTS01] es un framework realizado sobre Java, el cual es utilizado para la implementación de la metaheurística Tabu Search. Con este framework se puede trabajar sobre cualquier problema el cual pueda utilizarse para su solución dicha metaheurística, realizando para cada uno de estos problemas una solución particular basado en Tabu Search o aplicando la misma solución para varios problemas. También se puede crear diferente formas de aplicar el algoritmo Tabu Search.

Además permite tener el algoritmo Tabu Search bien estructurado, esto facilita la forma de exportar a otros usuarios la solución por lo cual permite que sea fácil de modificar, entender y exportar. También tiene la particular que al ser desarrollado en Java esta orientado a objetos, por lo cual permite una mejor modularización, es multiplataforma, por lo cual se puede correr tanto el ambientes Linux como Windows, y es fácil crear capas de presentación en ambientes Web. Otra particularidad de este framework es que permite trabajar con vario procesadores simultáneos ya que puede manejar multihilado.

3.9.2 Arquitectura

La arquitectura esta basada por las interfases las cuales el programador debe implementar para poder tener todos los elementos necesarios para poder desarrollar un algoritmo de Tabu Search.
Las interfases a implementar son las siguientes:

- AspirationCriteria: a través de esta interfase se implementa los criterios por los cuales se aceptara o no una solución como la siguiente mejor solución.
- Move: es la interfase la cual implementa que una solución pase a ser la siguiente mejor solución encontrada para seguir con la iteración.
- MoveManager: dicha interfase es la que maneja cuales serán los movimientos que se pueden realizar en cada momento.
- ObjetiveFunction: a través de esta interfase implementa la función objetivo, o sea la que evalúa la solución.
- Solution: aquí es donde se crean las definiciones de la solución.
- TabuSearch: implementa el código de control del algoritmo completo.
- TabuList: define la lista de movimientos que son Tabu y cuales no.
- TabuSearchListener: es la que implementa que hacer cuando suceden los eventos de inicio, fin de la búsqueda, movimiento, movimiento invalido, movimiento sin cambios, cuando se encontró una solución y cuando se encontró la mejor solución.

Estas interfases son implementadas por ciertas clases, las cuales estas clases deben ser extendidas para poder trabajar con el framework, siguiendo así una forma de trabajo la cual hace que la forma de trabajo este unificada por todos los usuarios lo que hace el código y cada algoritmo mas fácil de entender. Las clases que implementan dichas interfases y otras clases las cuales son muy importante su creación para un buen funcionamiento del framework se detallan aquí:

- Main: es la clase que contiene el código de inicialización de todos los elementos y el cual inicia el algoritmo y muestra la solución final.
- SingleThreadedTabuSearchBeanInfo: define la información del Bean para un hilo
- MultiThreadedTabuSearchBeanInfo: define la información de los Beans para multihilado.
- SimpleTabuList: es la clase que implementa la interfase TabuList
- SolutionAdapter: implementa la interfase Solution
- TabuSearchAdapter: implementa la interfase TabuSearchListener
- SingleThreadedTabuSearch: Implementa la interfase TabuSearch con la utilización de un solo hilo
- MultiThreadedTabuSearch: implementa la interfase TabuSearch para varios procesadores, creando varios hilos.
- NoCurrentSolutionException: es la que contiene que realizar cuando no hay soluciones en cierto paso del algoritmo.
- NoMovesGeneratedException: contiene la rutina de ejecución cuando no se generaron movimientos.

3.9.3 Modo de uso de OpenTS

La idea de trabajo de dicho framework es crear una clase la cual contenga el método main, el cual importe el paquete con las interfases y sus clases implementadas, y se trabaje instanciando los elementos que se vayan a usar. Estos elementos que se van a instanciar deben haber sido creado por el usuario, creando las clases que extienden las clases antes descriptas.

En forma general la forma del archivo será la siguiente:

```
import org.coinor.opents.*;

public class Main
{

    public static void main( String[] args )
    {
        // Inicializo las variables a utilizar
        // por ejemplo arreglos, listas, etc.

        // Creo el objeto TabuSearch el cual
         // crea los objetos necesarios para la
         // realización del algoritmo

         // Inicio el algoritmo, el cual a partir
         // del elemento creado (TabuSearch) realizara
         // hasta llegar a una solución.

        // Muestro la solución

    }

    ...

}
```

3.10 HOTFRAME (Heuristic OpTimization FRAMEwork)

3.10.1 Introducción

EL siguiente framework [HOT01], desarrollado en C++, permite usar metaheurísticas ya conocidas como Búsqueda Local Iterativa, Tabu Search, Recocido Simulado, métodos evolutivos, etc., para solucionar problemas específicos y además permite la creación de nuevas metaheurísticas, ya sea a partir de elementos de las ya mencionadas o creando nuevas. La forma de trabajo es en tener las metaheurísitcas, que serán clases creadas, y que estas colaboren con las clases creadas para el problema específico, llegando de esta forma a crear la solución al problema a partir de cierta metaheurística.

Para poder trabajar con otras metaheurísticas la forma seria creando nuevas clases las cuales simplemente ciertas interfases para poder comunicarse con el resto del framework. Luego de creadas las clases se trabaja de forma de generar un código que cree y haga interactuar las clases.

3.10.2 Arquitectura

La arquitectura se divide a grandes rasgos en dos partes: mecanismos de configuración básicos, componentes del problema específico y los componentes de las metaheurísticas. Los mecanismos de configuración básicos son aquellos que se necesitan para poder pasarle los parámetros a las metaheurísticas deseada a utilizar para resolver cierto problema. Cada metaheurística contiene

diferentes parámetros, dichos mecanismos son utilizados para según cada metaheurística se le pueda pasar los parámetros de configuración adecuados.

Los componentes del problema especifico son interfases las cuales implementándolas se tiene la forma en la que se debe de tratar el problema en cuestión. Dentro de estos componentes se tiene:

- Componente del problema: esta interfase define como será el acceso a los datos del problema.

- Componente de la solución: maneja la estructura y la forma de construirse las soluciones (movimientos, toma de vecindades, etc.).

- Componente de la información de la solución: maneja que información contendrá las soluciones encontradas.

- Componente de vecindades: maneja la estructura de las vecindades.

- Componente de atributos: maneja los atributos básicos en las soluciones.

- Componente de problemas específicos estándar: maneja las partes comunes de los problemas particulares con ciertos problemas conocidos de forma tal para poder acelerar el trabajo. (Ver diagramas en la documentación)

La componente de las metaheurísticas es una interfase implementada, de la cual de ella derivan cada una de las metaheurísticas implementadas. Cada una de ellas se implementa de forma diferente, en la Figura 7 se ve como deriva cada metaheurística de la clase Heuristic, y se ve un ejemplo de como es la estructura de la Búsqueda Local Iterativa:

Figura 7: Derivación de la metaherística de Búsqueda Local Iterativa para HOTFRAME

3.10.3 Modo de uso de HOTFRAME

El modo de utilizar el framework es creando algoritmos los cuales utilicen las clases creadas dentro del mismo, creando las clases pasándole los parámetros de configuración que se quiera tener en el mismo, etc. Estos algoritmos pueden ser híbridos, o sea utilizar diferentes metaheurísticas según la conveniencia, o pueden ser puros, utilizando las metaheurísticas ya definidas.

También se puede extender el mismo creando nuevas metaheurísticas, aunque esto ya es integrando nuevas clases al framework, por lo cual requiera mas trabajo de programación para el usuario.

3.11 Otros productos

Dentro de esta sección describiremos algunos software los cuales la documentación encontrada sobre los mismos no fue la suficiente como para poder desarrollar una extensa descripción del mismo o porque su forma de trabajo o sus fines son muy parecidos a alguno de los productos de software antes descriptos, por lo cual los presentamos haciendo una pequeña descripción de los mismos.

Localizer++ [LO+01] es una librería la cual su formato y sus objetivos son muy parecidos a los ofrecidos por EasyLocal++, ya que es una librería especializada en Simulado Recocido, búsqueda local, etc. Además la estructura es muy parecida a la descripta en EasyLocal++.

EasyLocal++ en realidad es el siguiente paso que se dio luego de la librería llamada Local++, ya que los autores de dichas librerías son los mismos y sus arquitecturas difieren en muy pocos aspectos.

Dentro los productos los cuales no se encontró una documentación acorde como para realizar una descripción mas a fondo, podemos encontrar: NUS [NUS01], ACF, MDF, y SDSC EBSA [SDS01]. Los productos NUS, ACF, MDF y el antes presentado TSF son productos desarrollados por la Universidad Nacional de Singapur. NUS es un framework basado en métodos de Tabu Search al igual que TSF (Tabu Search Framework), mientras que ACF (Ants Colony Framework) es el único producto encontrado el cual implementa métodos de colonia de hormigas, por desgracias la documentación del mismo es casi nula. El último de los productos desarrollados en esta universidad es MDF (Metaheuristic Development Framework) el cual intenta ser un framework, no solo basado en una sola metaheurística, sino que sea un framework en el cual se puede implementar y crear metaheurísticas nuevas o ya existentes. Por ultimo tenemos el producto SDSC EBSA el cual es una librería que implementa métodos basados en el recocido simulado.

4 Conclusiones de los productos presentados

Como se puede apreciar en las descripciones de los productos anteriormente descriptos, todos basan su estilo en la programación orientada a objetos, ya sea utilizando Java o C++ como lenguaje, esto se debe a la fácil modularización que ofrece, lo fácil que es entender y lo fácil que permite reusar y extender el código. También este tipo de programación facilita que se utilicen librerías ya creadas, ya que ambos lenguajes contienen grandes cantidades de librerías que implementan varias funciones que son utilizadas por los diferentes productos. Dentro de la programación a objetos se diferencias dos lenguajes principales: Java y C++. La mayoría de los productos se inclinan por la utilización de C++ ya que permite trabajar de forma más eficiente al ser un lenguaje de un nivel "más bajo" que Java al no ser un lenguaje interpretado. Esto es una gran ventaja a la hora de pensar en un framework de estas características ya que los algoritmos propuestos generalmente utilizan muchas instrucciones matemáticas y el lenguaje C es más eficiente en este tipo de cálculos. En cambio los productos que se inclinaron por el lenguaje Java (ECJ, JDEAL, OpenTS) tiene n la particularidad de poder ser utilizados en cualquier plataforma, ya sea Linux o Windows sin ningún problema, cosa que si se quiere realizar en C se debe tener una seria de consideraciones que no siempre son fáciles de implementar o el esfuerzo de crear soluciones para una u otra plataforma seria muy grande.

Un punto en el cual los diferentes productos difieren es en cual tipo de metaheurística aplican. No solo por lo que se puede observar en los productos presentados detenidamente, sino en una visión mas general de los productos de este estilo, se puede notar que hay un gran auge de las Metaheurísticas basadas en la evolución computacional, Tabu Search, búsqueda local y recocido simulado. Dentro de los productos presentados tenemos OpenTS y TSF que se basan en Tabu Search y EasyLocal++ se basa en todo tipo de búsquedas locales ,Tabu Search, Simulado Recocido, etc. Después tenemos GAlib que se basa en algoritmos genéticos (que es un tipo particular de la evolución computacional), TEA implementa la metaheurística de algoritmos evolutivos al igual Open BEAGLE y JDEAL, mientras que ECJ implementa algoritmos basados en evolución computacional. Si vemos también los productos vistos en la sección de "otros productos" podemos observar que también en estos la mayoría están dentro de las metaheurísticas de Tabu Search, recocido simulado, algoritmos genéticos, etc., salvo el framework ACF (Ants Colony Framework) que utiliza la metaheurística de colonia de hormigas. Además de estos productos basados en ciertas metaheurísticas también encontramos productos los cuales no se basan solo en una metaheurística, sino los cuales permiten trabajar con distintas clases de metaheurísticas y también permiten realizar nuevas combinando secciones de las distintas metaheurísticas ya conocidas. Estos productos que permiten este tipo de manejo son MALLBA y HOTFRAME. Estos, como vimos en la explicación anterior, permiten trabajar con mas metaheurísticas, pero contienen la contra que requieren de mas trabajo de programación ya que si se quiere agregar alguna nueva metaheurística se deben crear las clases que las compondrá, y se debe realizar el diseño de las mismas de forma que no destruyan las metaheurísticas que ya tiene integradas.

Otro punto de comparación entre los productos es el tipo de arquitectura que contienen. La mayoría de los productos definen las clases y las interfases, pero solo dos productos definen una arquitectura de capas. Estos productos son OpenBEAGLE, TEA y EasyLocal++. La ventaja de tener la arquitectura en capas es que permite dividir que pertenece a cada una y de ese modo poder trabajar más específicamente en cada una de ellas., pero también permite tener una mejor visión de que hace cada una de ella, por lo que mejora el entendimiento del software para su reuso y su extensibilidad.

La forma de utilización de los productos también es un tema importante, ya que es lo primero que vera un usuario a la hora de enfrentarse al producto. En general la forma de trabajo es bastante

similar en todos, ya que para poder utilizar algoritmos nuevos se deben implementar las clases para poder utilizar los elementos que implementaran dicho algoritmo, se debe crear un procedimiento main que inicialice e instancie los elementos que se van a utilizar para armar el algoritmo y por ultimo, y es donde mas varia entre productos, es el pasaje de los parámetros a utilizar. Mientras que en algunos productos se setean los parámetros en archivos especiales (en Open BEAGLE en archivos XML, por ejemplo) en otros productos se definen clases en los cuales definir los parámetros que se utilizarán dentro del algoritmo.

Por ultimo podemos ver que hay algunos productos que permite la posibilidad de trabajar en varios procesadores a la vez para poder realizar los cálculos. Estos productos, que permiten el multihilado, son ECJ y OpenTS y uno de sus principales propósitos por lo que cumplen con esta propiedad es que tratan de tener ejecuciones eficientes para el manejo de los cálculos.

En conclusión los productos presentados tiene varios elementos en común por lo que nos deja una visión de como es mas fácil, o por lo menos como es la mejor forma, de realizar productos de la misma índole. En cambio nos define distintas formas en las que se puede enfocar el método de resolución (si multihilado, multiplataforma, etc.) por lo cual nos deja varios ejemplos a seguir según sea nuestra elección de que forma enfocar nuestra futura aplicación. Otro puntos que vemos es que en general los productos se especializan en alguna metaheurística en particular, por lo que nos enfocaremos más en aquellos productos que presentan la posibilidad de manejar varias metaheurísticas, ya que es lo que se quiere desarrollar en un futuro.

Informe final

Anexo 2: Referencias y Bibliografía

Referencias y Bibliografía

El presente Anexo muestra la bibliografía y las distintas referencias utilizadas en la realización del presente proyecto. Dentro de las presente referencias se pueden diferencias aquellas las cuales fueron sometidas a un proceso de validación por referato y aquellas que no. Aquellas las cuales si fueron validadas por referato son marcadas con un asterisco (*) al terminar la referencia.

A

[AGE01](*) A Genetic Algorithm Tutorial – D. Whitley - Statistics and Computing, volume 4, 65—85 - 1994. http://citeseer.nj.nec.com/whitley93genetic.html

[AGE02](*) Genetic Algorithms for Combinatorial Optimization: The Assembly Line Balancing Problem - E. Anderson y M. Ferris - ORSA Journal on Computing , Vol. 6, 161- 173 - 1994. http://citeseer.nj.nec.com/anderson94genetic.html

[AGE03](*) International Journal on Evolutionary Optimization – S. Kundu – 27/01/2002 http://www.jeo.org/

{AGE04](*) An Introduction to Genetic Algorithms - M Mitchell - MIT Press, ISBN: 0262631857 -1998

{AGE05](*) An Introduction to Genetic Algorithms for Scientists and Engineers - D Coley – World Scientific, ISBN: 9810236026 - 1999

[AGE06](*) Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms - T Baeck - Oxford University Press, ISBN: 0195099710 - 1996

[AGE07](*) Evolutionary Computation 1: Basic Algorithms and Operators – T. Baeck, D. Fogel, y Z. Michalewicz - Institute of Physics, ISBN: 0750306645 - 2000

[AGE08](*) Evolutionary Computation 2 : Advanced Algorithms and Operators – T. Baeck, D. Fogel y Z. Michalewicz - Institute of Physics, ISBN: 0750306653 - 2000

[AGE09](*) Genetic Algorithms + Data Structures = Evolution Programs – Z. Michalewicz - Springer-Verlag, ISBN: 3540606769 - 1993

[AGE10](*) Genetic Algorithms in Search, Optimization and Machine Learning – D. Goldberg - Addison-Wesley, ISBN: 0201157675 - 1989

B

[BDI01](*) Fundamentals of Scatter Search and Path Relinking – F. Glover, M. Laguna, R. Martí - 2000. http://www-bus.colorado.edu/faculty/glover/ssandprfundamentals.pdf

[BLO01](*) Iterated Local Search - H.R. Lourenco, O. Martin y T. Stuetzle. - F. Glover and G. Kochenberger, editors, Handbook of Metaheuristics. 321-353, Kluwer Academic Publishers, Norwell, MA - 2002.

[BLO02](*) Towards practical neural computation for combinatorial optimization problems - E. B. Baum - J. Denker, editor, Neural Networks for Computing, 53--64, AIP conference proceedings – 1986.

[BLO03](*) Local optima avoidance in depot location. Journal of the Operational Research - J. Baxter. Society, 32:815—819 – 1981.

[BLO04](*) An iterated dynasearch algorithm for the single--machine total weighted tardiness scheduling problem - R. K. Congram, C.N. Potts y S. L. Van de Velde - INFORMS Journal on Computing, 14(1):52-67 – 2002.

[BLO05](*) The travelling salesman problem: A case study in local optimization - D. S. Johnson y L. A. McGeoch - E.H.L. Aarts and J.K. Lenstra, editors, Local Search in Combinatorial Optimization, 215--310. John Wiley & Sons, Chichester, England - 1997.

[BLO05](*) A large step random walk for minimizing total weighted tardiness in a job shop - S. Kreipl - Journal of Scheduling, 3(3):125—138 - 2000.

[BLO06](*) Combining simulated annealing with local search heuristics - O. Martin y S. W. Otto - Annals of Operations Research, 63:57—75 - 1996.

{BLO07](*) Large-step Markov chains for the traveling salesman problem - O. Martin, S. W. Otto y E. W. Felten - Complex Systems, 5(3):299—326 - 1991.

[BLO08] (*) Large-step Markov chains for the TSP incorporating local search heuristics - O. Martin, S. W. Otto y E. W. Felten - Operations Research Letters, 11:219—224 - 1992.

[BLO09](*) Iterated Local Search for the Quadratic Assignment Problem - T. Stuetzle - Technical Report AIDA-99-03, Darmstadt University of Technology, Computer Science Department, Intellectics Group - 1999

[BLO10](*) Local Search Algorithms for Combinatorial Problems --- Analysis, Improvements, and New Applications - T. Stuetzle - PhD thesis, Darmstadt University of Technology, Department of Computer Science - 1998.

[BVV01](*) A variable neighborhood search for graph coloring – C. Avanthay, A. Hertz y N. Zu - European Journal of Operational Research 151 (2003) 379-388 - 2006
http://www.gerad.ca/~alainh/VNSEJOR.pdf

[BVV02] A Reactive Variable Neighborhood Search for the Vehicle Routing Problem with Time Windows - Olli Bräysy – 2006

C

[CHO01](*) Ant Colony Optimization: A New Meta-Heuristic - M. Dorigo, G. Di Caro y L. Gambardella - Proceedings of the Congress on Evolutionary Computation, 2, IEEE Press - 1999
http://citeseer.ist.psu.edu/606667.html

[CHO02] Ant Colony Optimization – M. Dorigo y T. Stützle - ISBN 0-262-04219-3 - 2006

http://mitpress.mit.edu/catalog/item/default.asp?sid=10FAC773-0F66-4B60-9A6B-803F38E7BEA9&ttype=2&tid=10139

[CHO03] The Ant Colony Optimization Website – P. Balaprakash y M. Montes - 2006
http://iridia.ulb.ac.be/~mdorigo/ACO/ACO.html

[CHO04](*) Optimization, Learning and Natural Algorithms - M. Dorigo - Ph.D. thesis, DEI, Politecnico di Milano, Italy - 1992.

[CHO05] The Ant Colony Optimization metaheuristic - M. Dorigo y G. Di Caro - D. Corne, M. Dorigo y F. Glover - New Ideas in Optimization, 11-32. McGraw-Hill - 1999.

[CHO06](*) Ant algorithms for discrete optimization - M. Dorigo, G. Di Caro y L. M. Gambardella - Artificial Life, 5, 2, 137-172 - 1999

[CHO07](*) Colony system: A cooperative learning approach to the travelling salesman problem - M. Dorigo y L. M. Gambardella.- IEEE Transactions on Evolutionary Computation, 1, 1, 53-66 - 1997

[CHO08](*) Ant System: Optimization by a colony of cooperating agents - M. Dorigo, V. Maniezzo y A. Colorni - IEEE Transactions on Systems, Man and Cybernetics - Part B, 26, 1, 29-41 - 1996

[CH09] The ant colony optimization metaheuristic: Algorithms, applications and advances - M. Dorigo y T. Stützle- F. Glover and G. Kochenberger editors, Handbook of Metaheuristics, volume 57 of International Series in Operations Research & Management Science, 251-285. Kluwer Academic Publishers, Norwell, MA - 2002

[CHO10](*) Ant Colony Optimization - M. Dorigo y T. Stützle - MIT Press, Boston, MA - 2004

[CHO11](*) MAX-MIN Ant System - T. Stützle y H. H. Hoos - Future Generation Computer Systems, 16, 8, 889-914 - 2004

[CPP01] Referencias C++ - Wikipedia
http://es.wikipedia.org/wiki/C_m%C3%A1s_m%C3%A1s

E

[ECJ01](*) A Java-based Evolutionary Computation Research System – S. Luke, L. Panait, G. Balan, S. Paus, Z. Skolicki, J. Bassett, R. Hubley y A. Chircop - George Mason University's ECLab Evolutionary Computation Laboratory - 2006
http://cs.gmu.edu/~eclab/projects/ecj/

[ELO01](*) Easy Local ++ Home Page – A. Shaerf y L. Di Gaspero - Dipartimento di Ingegneria Elettrica, Gestionale e Mecánica, Università degli Studi di Udine - 31/05/2004
http://tabu.diegm.uniud.it/EasyLocal/

F

[Frame01] Concepto de Framework de software – Wikipedia
http://en.wikipedia.org/wiki/Framework

G

[GAL01] GALib: Mathew's Genetics Algorithms Library - LFM-SDM Virtual Community - 2006
http://lancet.mit.edu/ga/

[GRA01](*) Greedy Randomized Adaptive Search Procedures - M. G.C. Resende , C. C. Ribeiro - AT&T Labs Research Technical Report. State-of-the-Art Handbook in Metaheuristics, F. Glover and G. Kochenberger, editores., Kluwer Academic Publishers – 13/01/2001
http://www.optimization-online.org/DB_FILE/2001/09/371.pdf

H

[HOT01] Heuristic OpTimization FRAMEwork - A. Fink y S. Vob – 01/06/2003
http://www.rrz.uni-hamburg.de/IWI/hotframe/hotframe.html

J

[JDE01] EVOWeb Software JDEAL - Universidade Tecnica de Lisboa, LASEEB - 2006
http://evonet.lri.fr/evoweb/resources/software/record.php?id=44

[JDE02] The Java Distributed Evolutionary Algorithms Library – J. Costa, N. Lopes y P. Silva - 2006
http://laseeb.isr.ist.utl.pt/sw/jdeal/home.html

L

[LO+01] Localizer++: An open library for Local Search - L. Michel, y P. Van Hentendrick - Brown University - 2004

[LOC01] EasyLocal Project Home Page – A. Schaerf, L. Di Gaspero y M. Cadoli – 03/11/2005
http://www.diegm.uniud.it/~aschaerf/projects/local++/

M

[MAL01] MALLBA Project – M.J. Blesa y J. Petit - Departmento de Estadística, Investigación Operativa y Computación (Universidad de La Laguna) - 2001
http://www.lsi.upc.edu/~mallba/

[MHE01] Metaheuristics in Combinatorial Optimization: Overview and Conceptual Comparison - C. Blum (Université de Bruxelles) y A. Roli (Universitá degli Studi di Bologna) – 23/06/2004.
http://iridia.ulb.ac.be/~meta/newsite/downloads/ACSUR-blum-roli.pdf

[MHE02](*) Metaheurísticas: Una visión global – B. Melián, J.A. Moreno Pérez y J. Marcos Moreno-Vega - Inteligencia Artificial, Revista Iberoamericana de Inteligencia Artificial.. Numero 19, Volumen 2, páginas 7-28 - 04/06/2003.
http://sensei.ieec.uned.es/cgi-bin/aepia/contenidoNum.pl?numero=19

[MHE03] Optimización Online - Mathematical Programming Society - 2006
http://www.optimization-online.org

[MHE04](*) Journal of Optimization Theory and Applications – A. Miele - Aero-Astronautics Group, Rice University, Houston, TX, USA – 02/01/2006.
http://springerlink.metapress.com/(5mv0hp55cqtyclbx5remjcb5)/app/home/journal.asp?referrer=parent&backto=linkingpublicationresults,1:104801,1

[MHE05] Curso Seminario: Elementos de metaheurísticas – H. Cancela, G. Ferreira y P. Rodríguez Bocca – Facultad de Ingeniería de la Universidad de la República, Uruguay - 2003
http://www.fing.edu.uy/inco/grupos/invop/mh/

N

[NUS01]. Honours Year Project Report Tabu Search Framework - W.C. Wan - 2002
http://www.comp.nus.edu.sg/~stevenha/hyp/WanWC_HYP_Report.zip

O

[OBE01] Open BEAGLE, a versatile EC framework – C. Gagné y J. Beaulieu – 06/02/2006
http://beagle.gel.ulaval.ca/index.html

[OTS01] Java Tabu Search Framework Page – R. Harder - COmputational INfrastructure for Operations Research - 2006
http://www.coin-or.org/OpenTS/

Q

[QAP01] Quadratic Assignment Problem (Problema de Asignación Cuadrática) - Wikipedia
http://en.wikipedia.org/wiki/Quadratic_assignment_problem

R

[RSI01](*) Heuristics from nature for hard combinatorial optimization problems - A. Colorni, M. Dorigo, F. Maffioli, V. Maniezzo, G. Righini y M. Trubian - International Transactions in Operational Research 3(1):1-21 - 1996.
http://citeseer.nj.nec.com/colorni96heuristics.html

[RSI02](*) Simulated Annealing Information - Taygeta Scientific Inc - 2001
http://www.taygeta.com/annealing/simanneal.html

[RSI03](*) Simulated Annealing: Theory and Applications - P.J.M. van Laarhoven y E.H.L. Aarts - D.Reidel Publishing Company, Kluwer - 1987

[RSI04](*) Approach to the Traveling Salesman Problem: An Efficient Simulation Algorithm - V. Cerny - Thermodynamical, J. Opt. Theory Appl., 45, 1, 41-51 - 1985

[RSI05](*) Optimization by Simulated Annealing - S. Kirkpatrick, C. D. Gelatt Jr. y M. P. Vecchi - Science, 220, 4598, 671-680 - 1983.

[RSI06](*) Equation of State Calculations by Fast Computing Machines - N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller y E. Teller - J. Chem. Phys.,21, 6, 1087-1092 - 1953.

S

[SDS01] SDSC EBSA. FTP donde – Anonimo - 1993
ftp://ftp.sdsc.edu/

T

[TEA01] C++ Library for Evolutionary Algorithms – M. Emmerich, M. Laumanns, R. Hosenberg y L. Shonemann – Universidad de Dortmun, TEA Team - 2006
http://sfbci.uni-dortmund.de/index.php?option=com_content&task=view&id=28&Itemid=160&lang=en

[TEA02] TEA: Toolbox for Evolutionary Algorithms - Nikos Drakos - Computer Based Learning Unit, University of Leeds – 31/03/1998.
http://ls11-www.cs.uni-dortmund.de/people/tea/teaDoc.html

[TEA03] TEA - A Toolbox for the Design of Parallel Evolutionary Algorithms in C++ technical report – M. Emmerich y R. Hosenberg - CI-106/01 Collaborative Research Center (Sonderforschungsbereich) SFB 531 Design and Management of Complex Technical Processes and Systems by Means of Computational Intelligence Methods , University of Dortmund - 2000

[TSE01] Tabu Search Website – C. Rego y F. Glover – 03/06/2004
http://www.tabusearch.net/

[TSE02](*) A Tutorial on Tabu Search - A. Hertz, E. Taillard y D. de Werra - Proc. of Giornate di Lavoro AIRO'95 (Enterprise Systems: Management of Technological and Organizational Changes), 13-24 – 1995
http://citeseer.ist.psu.edu/hertz92tutorial.html

[TSE03](*) Tabu search - Part I, II - Glover, F - ORSA Journal on Computing 1, 190-206 - 1990

[TSE04](*) Tabu search, Annals of Operations Research - F. Glover, E. Taillard, A. Laguna, D. De Werra - 41, Baltzer, Basel - 1993

[TSE05](*) Tabu Search - F. Glover y A. Laguna - Kluwer Academic Publishers - 1997

[TSE06] Página con referencias de Tabu Search – M. Mastrolilli - 2001
http://www.idsia.ch/~monaldo/tabusearch/ref.html

[TSF01](*) A Generic Object-Oriented Tabu Search Framework - Hoong Chouin Lau, Wee Chong Wan y Xiaomin Jia - The Fifth Metaheuristics International Conference, Kyoto, Japon

[TSP01] Travelling Salesman Problem (Problema del Viajero) - Wikipedia
http://en.wikipedia.org/wiki/Traveling_salesman_problem

U

[Unix01] Referencias Sistemas Unix – Wikipedia
http://es.wikipedia.org/wiki/Unix