Biblioteca de clases para integrar algoritmos metaheuristicos basados en un punto

Class library to integrate metaheuristicos a point-based algorithms

Jenny Fajardo Calderín, David Paredes Miranda, Yasser Bardají López, Alejandro Rosete Suárez

Instituto Superior Politécnico José Antonio Echeverría

jfajardo@ceis.cujae.edu.cu

Resumen

En la actualidad existen diferentes bibliotecas de clases que implementan los algoritmos metaheurísticos basados en un punto, orientadas para cada algoritmo. Garantizan su extensibilidad permitiendo la incorporación de nuevos algoritmos o modificando los existentes. Ninguna combina las características similares entre los algoritmos, partiendo de que ninguno es mejor que otro en la totalidad de los problemas, según el Teorema No Free Lunch. Por tal motivo se decidió crear una biblioteca de clases que implementa un modelo de unificación de los algoritmos metaheurísticos basados en un punto, específicamente: Escalador de Colinas, Búsqueda Aleatoria, Recocido Simulado y Búsqueda Tabú.

En el presente trabajo se propone un diseño de clases empleando patrones de diseño, que mediante la configuración de los parámetros: tipo de aceptación, tipo de búsqueda y estrategia, se comporta como cada una de las Metaheurísticas, mencionadas anteriormente, e híbridos que combinan sus características. Además permite incorporar nuevos comportamientos y estrategias de búsqueda. Se logra la independencia entre problema y algoritmo, por lo que se ajusta a cualquier situación.

Para la implementación se empleó como lenguaje de programación Java, como herramienta CASE para la modelación y generación de código MagicDraw, y como entorno integrado de desarrollo Eclipse.

Palabras clave: Algoritmos Metaheúristicos, Biblioteca de Clases, integración.

Abstract

At present they exist different libraries that implement point-based metaheuristic algorithms, orientated for each algorithm. They guarantee extensibility permitting the incorporation of new algorithms or modifying the existent ones. None combines similar characteristics among them algorithms, using as a start point that no one is better than other in the problem's totality, according to the No Free Lunch Theory. For this reason it was decided to create a library that implements a unification model of the point-based metaheuristic algorithms, specifically: Hill Climbing, Random Search, Simulated Annealing and Tabu Search.

In the present work, proposed is a class design using design patterns, that through the configuration of parameters: type of acceptance, type search and strategy, which behaves like each of the metaheuristics mentioned previously , and hybrids that combine their characteristics. Besides, it permits incorporating new behaviors and search strategies. Independence is obtained between the problem and the algorithm allowing it to adjust to any situation.

For the implementation Java was used as the programming language, for the modeling and code generation a CASE tool, Magic Daraw was used, and Eclipse was used as the Integrated Development Environment Tool.

Key words: *Metaheuristic Algorithms, Library of Classes, integration.*

Introducción

En el siguiente trabajo se presenta la solución propuesta para llevar a la práctica un modelo conceptual de unificación de los algoritmos metaheurísticos basados en un punto, a través de la construcción de una biblioteca de clases. Se realiza una descripción

de las clases principales, patrones, mecanismos de diseño utilizados, y las clases empleadas propias de las tecnologías de Java, que permiten reducir considerablemente el tiempo de desarrollo y aumentar la calidad de las aplicaciones.

La biblioteca de clases implementa un modelo de integración de varios algoritmos metaheurísticos basados en un punto, concretamente: Escalador de Colinas [4], Búsqueda Tabú [5], Recocido Simulado [6] y procedimiento ávido-aleatorio de GRASP [7, 8]. Este modelo fue presentado en un trabajo previo [53], se centra en una estructura que combina el comportamiento de los mismos, a partir de determinar la similitud entre sus características.

Arquitectura de la biblioteca de clases

La arquitectura de la aplicación se encuentra proyectada en dos capas: Aplicación, que abarca todos los paquetes referentes a la biblioteca de clases y JDK (o Java Development Kit por su nombre en inglés), que refleja las clases e interfaces que brinda la plataforma Java utilizadas por los paquetes de la capa Aplicación. Ver figura 1.

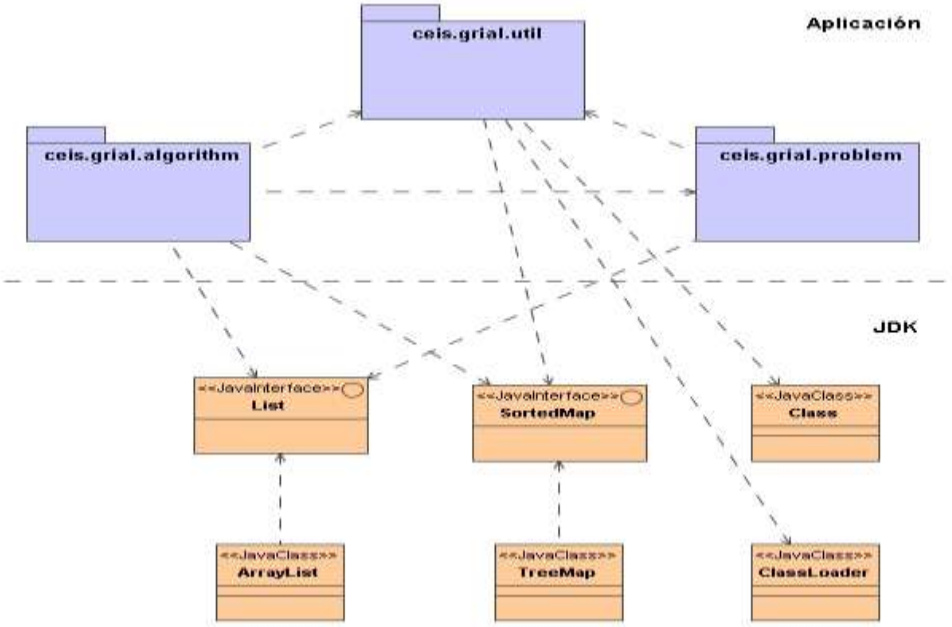

Fig. 1. Vista de la Arquitectura

La capa Aplicación está conformada por:

Paquete ceis.grial.algorithm:

Es el paquete más importante, puesto que agrupa las clases que implementan el modelo conceptual de unificación. Las clases que forman parte de él son: Search, AcceptableCandidate, AcceptAnyone, AcceptBest, AcceptNotBad, AcceptNotBadT, AcceptNotBadU, UpdateParameter, SolutionGreedy, StopSearch, CandidateValue, Neighborhood, TabuSolutions, SearchCandidate, GreaterCandidate, RandomCandidate y SmallerCandidate. Depende de los paquetes ceis.grial.util y ceis.grial.problem, así como de las interfaces List y SortedMap de la capa JDK.

Paquete ceis.grial.util:

Agrupa las clases que son reutilizadas por varias clases del paquete ceis.grial.algorithm y no son propiamente parte de la lógica del modelo, sino que garantizan su implementación. Contiene las interfaces y clases que implementan patrones de diseño, el mecanismo de diseño para la carga dinámica de clases y tipos enumerados. Las clases pertenecientes a este paquete son: FactoryLoader, FactoryAcceptCandidate, FactoryCandidate, Strategy, TypeCandidate, State, IFFactoryAcceptCandidate, IFFactoryCandidate, TypeProblem y TypeAcceptation. Depende de la clase Class y de la interface SortedMap de la capa JDK.

Paquete ceis.grial.problem:

Está conformado por las clases que abstraen el problema del algoritmo, permitiendo que estas sean empleadas para definir cualquier situación o problema. Las clases son: Problem, Domain, ObjetiveFunction, Operator, y Constrain. Depende del paquete ceis.grial.util y la interface List de la capa JDK.

Diseño de la estructura de clases

A continuación se muestra una descripción de las clases principales por paquetes:

Tabla 1. Clases del Paquete ceis.grial.algorithm

Paquete: ceis.grial.algorithm:	
Search	Clase que controla el proceso de búsqueda, teniendo la responsabilidad de manipular los datos del problema y los parámetros de entrada, a través de instancias de las clases: Problem, CandidateValue, AcceptableCandidate, UpdateParameter, StopSearch y SolutionGreedy. El constructor es privado, por lo que para ser instanciada se debe llamar a su método getSearch().
CandidateValue	Clase que tiene como responsabilidad determinar el estado candidato para cada iteración, así como manipular la vecindad a través de instancias de las clases: Neighborhood, SeachCandidate y TabuSolutions.
TabuSolutions	Clase que manipula la vecindad del estado actual, con el objetivo de filtrar la vecindad, o sea eliminar los estados de la vecindad que se encuentren en la lista tabú.
Neighborhood	Clase que tiene como responsabilidad generar la vecindad del estado actual.
SeachCandidate	Superclase abstracta, que delega sus responsabilidades en varias subclases: GreaterCandidate, SmallerCandidate y RandomCandidate.
GreaterCandidate	Clase que hereda de SeachCandidate, adquiriendo la responsabilidad de implementar el método stateSearch(), el cual evalúa los estados de la vecindad en la función objetivo, y determina el mayor estado de la vecindad.
SmallerCandidate:	Clase que hereda de SeachCandidate, adquiriendo la responsabilidad de implementar el método stateSearch(), el cual evalúa los estados de la vecindad en la función objetivo, y determina el menor estado de la vecindad.
RandomCandidate	Clase que hereda de SeachCandidate, adquiriendo la responsabilidad de implementar el método stateSearch(), el cual determina un estado aleatorio de la vecindad.
AcceptableCandidate	Superclase abstracta, que delega sus responsabilidades en varias subclases: AcceptBest, AcceptNotBad, AcceptNotBadU, AcceptNotBadT, y AcceptAnyone.
AcceptBest	Clase que hereda de AcceptableCandidate, adquiriendo la responsabilidad de implementar el método acceptCandidate(), el cual determina la aceptación de un estado candidato como nuevo estado actual, si la evaluación del estado candidato es mayor que la del estado actual.
AcceptNotBad	Clase que hereda de AcceptableCandidate, adquiriendo la responsabilidad de implementar el método acceptCandidate(), el cual determina la aceptación de un estado candidato como nuevo estado actual, si la evaluación del estado candidato es mayor o igual que la del estado actual.
AcceptNotBadU	Clase que hereda de AcceptableCandidate, adquiriendo la responsabilidad de implementar el método acceptCandidate(), el cual determina la aceptación de un estado candidato como nuevo estado actual, si la diferencia entre la evaluación del estado actual y la evaluación del estado candidato es menor que un Umbral.
AcceptNotBadT	Clase que hereda de AcceptableCandidate, adquiriendo la responsabilidad de implementar

	el método acceptCandidate(), el cual determina la aceptación de un estado candidato como nuevo estado actual, si la evaluación del estado candidato es mayor o igual que la del estado actual; o si determinada probabilidad aleatoria es menor el fp de Boltzman.
AcceptAnyone	Clase que hereda de AcceptableCandidate, adquiriendo la responsabilidad de implementar el método acceptCandidate(), el cual acepta cualquier estado candidato como nuevo estado actual.
StopSearch	Clase que controla cuando debe parar el proceso de búsqueda, ya sea por llegar a un número máximo de iteraciones o al valor final de temperatura.
UpdateParameter	Clase que tienen la responsabilidad de actualizar los parámetros del proceso de búsqueda para cada iteración, tales como: temperatura, cantidad de iteraciones y lista tabú.
SolutionGreedy	Definida como una clase abstracta, dado que para cada problema es específico la construcción del estado inicial. Su responsabilidad es construir un estado para iniciar el proceso de búsqueda. Los métodos que contiene son abstractos por lo que la clase que herede de ella debe implantarlos.

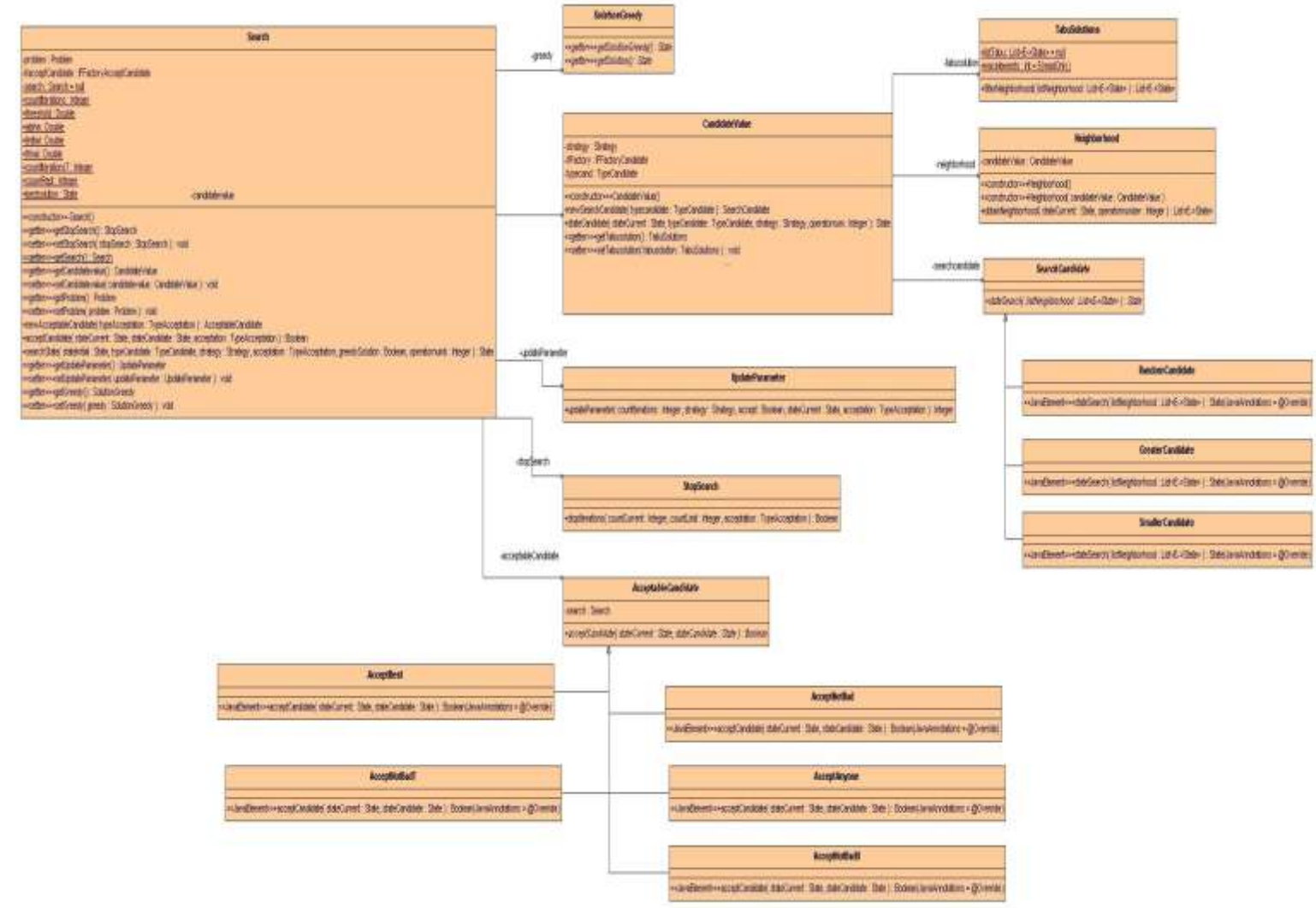

Fig. 2. Diagrama de Clases del Paquete ceis.grial.algorithm

reflejar el comportamiento de las instancias de los elementos contenidos en el diagrama de clases de la Figura 2, se emplean los diagramas de objetos en las Figura 3 y 4, que muestran un conjunto de objetos y sus relaciones en un momento concreto.

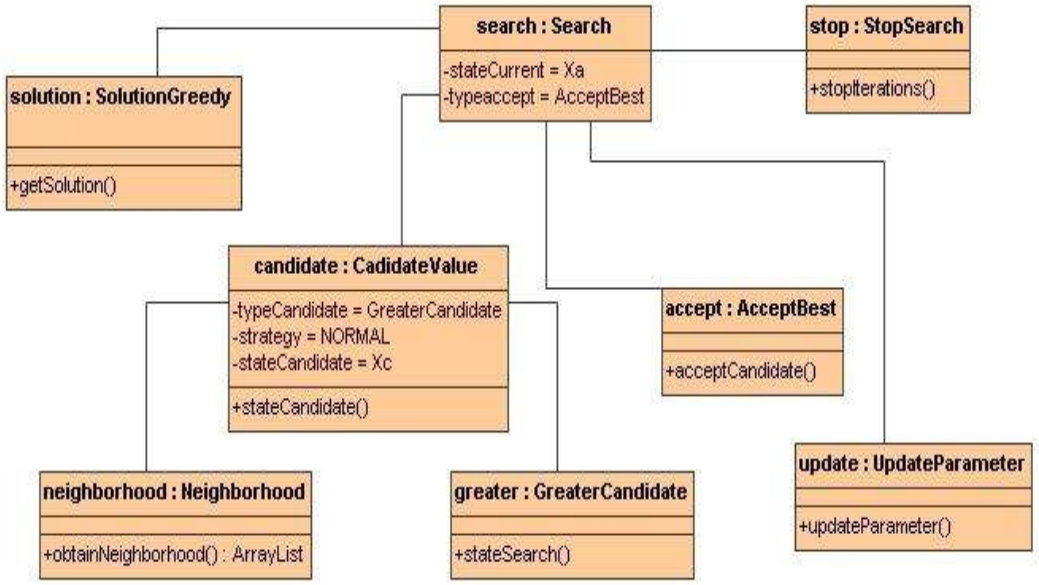

Fig. 3. Diagrama de Objetos con el comportamiento de un Escalador de Colinas Clásico.

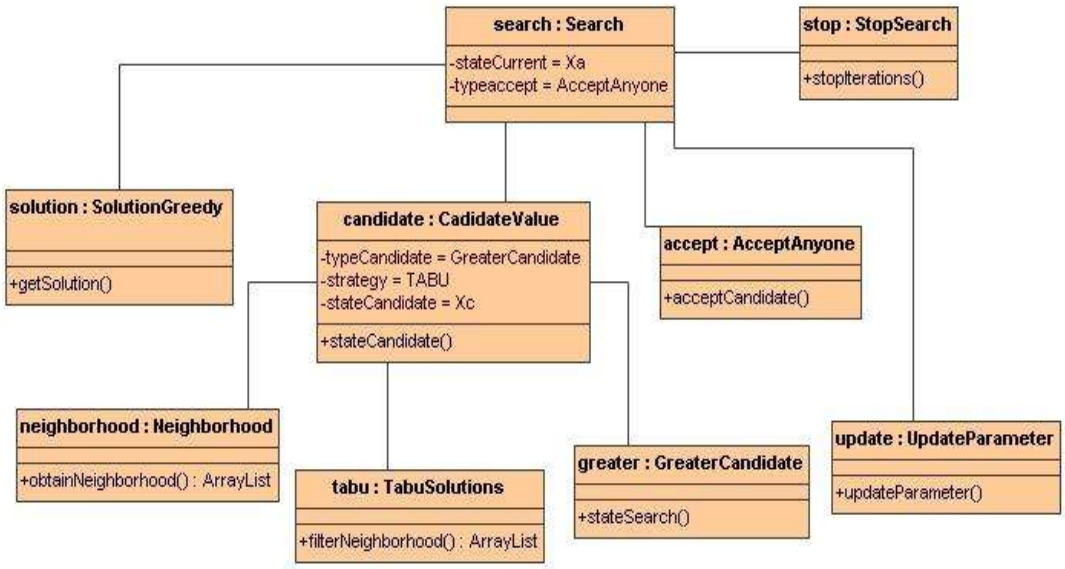

Fig. 4. Diagrama de Objetos con el comportamiento de la Búsqueda Tabú.

Tabla 2. Clases del Paquete ceis.grial.problem

Paquete: ceis.grial.problem	
Problem	Clase que controla el problema a solucionar, mediante el conjunto de elementos que lo caracterizan. Contiene instancias de las clases: ObjetiveFunction, Domain, Operador y Constrain.
Operator	Definida como una clase abstracta, dado que para cada problema los operadores a aplicar son específicos.
Constrain	Definida como una clase abstracta, dado que para cada problema las restricciones son específicas.
Domain	Definida como una clase abstracta, dado que para cada problema el dominio de cada variable es específico.
ObjetiveFunction	Definida como una clase abstracta, ya que representa la función que se desee optimizar,

	específica de cada problema.

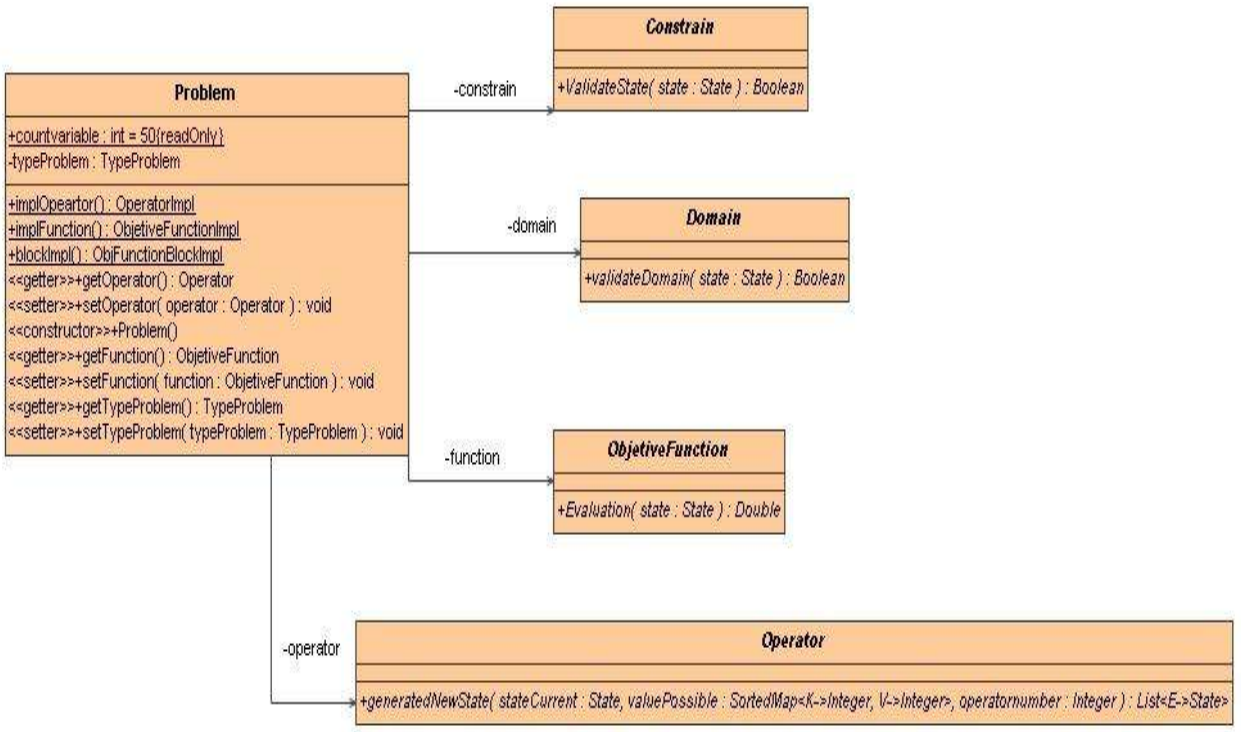

Fig. 5. Diagrama de Clases del Paquete ceis.grial.problem

Tabla 3. Clases del Paquete ceis.grial.util

Paquete: ceis.grial.util	
TypeAcceptation	Es una enumeración que define los tipos de aceptación existentes, utilizado por la clase Search para aceptar a un estado candidato como nuevo estado actual. Contiene las constantes: AcceptBest, AcceptAnyone, AcceptNotBadT, AcceptNotBadU y AcceptNotBad. Este recurso del lenguaje se emplea cuando se quiere que una variable tenga un solo valor, dentro de un conjunto de valores predefinidos.
TypeCandidate	Es una enumeración que define los tipos de candidatos existentes, utilizado por la clase CandidateValue para determinar qué tipo de estado candidato buscar. Contiene las constantes: SmallerCandidate, GreaterCandidate y RandomCandidate. Este recurso del lenguaje se emplea cuando se quiere que una variable tenga un solo valor, dentro de un conjunto de valores predefinidos.
Strategy	Es una enumeración que define las estrategias existentes, utilizado por la clase CandidateValue para determinar el filtrado o no de la vecindad del estado actual. Contiene las constantes: TABU y NORMAL. Este recurso del lenguaje se emplea cuando se quiere que una variable tenga un solo valor, dentro de un conjunto de valores predefinidos.
TypeProblem	Es una enumeración que define el tipo de objetivo en un problema. Contiene las constantes: Maximizar y Minimizar. Este recurso del lenguaje se emplea cuando se quiere que una variable tenga un solo valor, dentro de un conjunto de valores predefinidos.
State	Clase que manipula un estado del problema, las variables y sus valores correspondientes. La utilizan varias clases, entre ellas la clase Search, en el método searchState() que devuelve el estado al que se llega como resultado de aplicar un algoritmo a un problema. Tiene 2 constructores y en uno de ellos se redefinió el método compare() de la interface Comparator, que ordena ascendentemente las llaves de un SortedMap donde se almacenan las variables de un estado.

Patrones de diseño empleados

En la actualidad los patrones de diseño se han convertido en una parte muy importante de las Tecnologías Orientadas a Objetos, ya que hacen que los diseños sean más flexibles y reutilizables, solucionan problemas específicos del diseño, proporcionan una estandarización del modo en que se realiza el diseño, evitan la reiteración en la búsqueda de soluciones a problemas ya conocidos y solucionados anteriormente, y facilitan elementos reusables en el diseño de sistemas de software [48] [49] [50].

En el diseño de la biblioteca de clases se decide utilizar patrones de diseño, los cuales contribuyen en gran medida a garantizar su extensibilidad, flexibilidad y fiabilidad, por citar alguna de las ventajas más importantes. Los patrones empleados son: Método de Fábrica (o Factory Method por su nombre en inglés) y Solitario (o Singleton por su nombre en inglés), los cuales se clasifican como Patrones de Creación.

Los Patrones de Creación se conocen como patrones que proporcionan ayuda a la hora de crear objetos, principalmente cuando la creación requiere toma de decisiones, y la toma de decisiones pudiese ser dinámica; ayudan a estructurar y encapsular estas decisiones [48] [49] [50].

Singleton garantiza que solamente se cree una instancia de una clase, proporcionando un punto de acceso global a ella y todos los objetos que utilizan una instancia de esa clase usan la misma instancia, o sea controla el acceso a la única instancia; en otras palabras provee un mecanismo para limitar el número de instancias de una clase, por lo que el mismo objeto es siempre compartido por distintas partes del código [48] [49] [50].

Search es la clase controladora de la biblioteca, por lo que se decide crear un único punto de acceso global hacia ella y así se controla el número de instancias a crear de la misma clase. Implementa el patrón Singleton, poniendo su constructor con visibilidad privada, y para ser instanciada se debe invocar a su método estático getSearch, al cual no se le pasa parámetros. Éste método estático retorna un atributo estático y privado que es del mismo tipo que la clase. Ver figura 6.

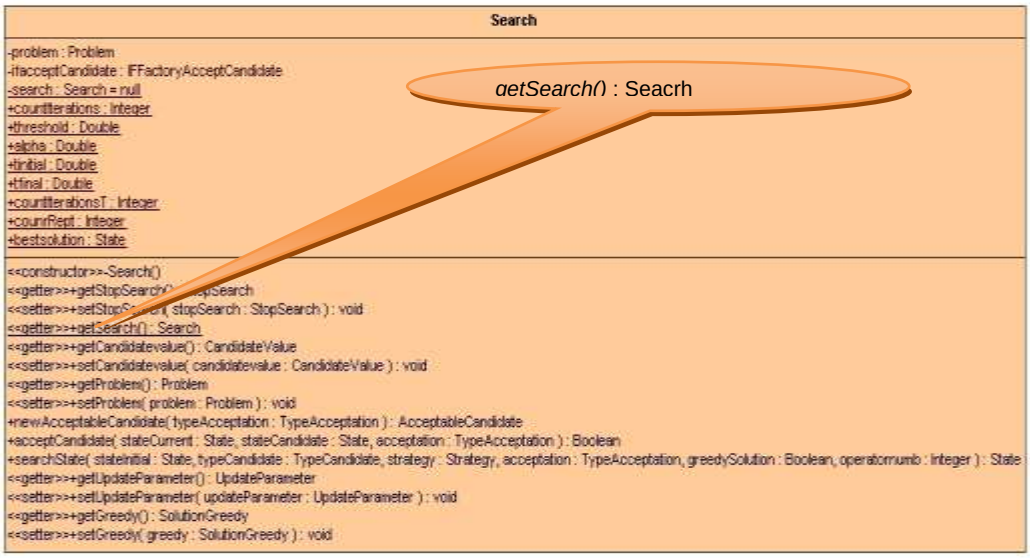

Fig. 6. Clase Search que implementa el Patrón de Diseño Singleton.

El patrón Método de Fábrica es empleado para construir instancias a objetos individuales con un propósito específico, sin que la petición de construcción conozca las clases específicas que serán instanciadas. Este patrón permite a las superclases abstractas delegar responsabilidades de instanciación a las subclases, así como crear objetos dentro de una clase con un método de fabricación, que es más flexible que hacerlo directamente [48] [49] [50].

El patrón Método de Fábrica es empleado para construir instancias a objetos individuales con un propósito específico, sin que la petición de construcción conozca las clases específicas que serán instanciadas. Este patrón permite a las superclases abstractas

delegar responsabilidades de instanciación a las subclases, así como crear objetos dentro de una clase con un método de fabricación, que es más flexible que hacerlo directamente [48] [49] [50].

La biblioteca de clases implementa el Método de Fábrica puesto que existen clases que no pueden anticipar que clases de objetos deben crear. Una de las implementaciones del patrón Método de Fábrica dentro de la biblioteca está dada porque la clase CandidateValue, necesita crear una instancia a una de las clases concretas de SeachCandidate. Esto se realiza indirectamente a través de una instancia de la clase FactoryCandidate, que implementa la interface IFFactoryCandidate. La interface tiene declarado un método createSearchCandidate() que tiene como argumento necesario un TypeCandidate para determinar que clase concreta instanciar.

Otra implementación del patrón Método de Fábrica está dada porque la clase Search, necesita crear una instancia a una de las clases concretas de AcceptableCandidate. Esto se realiza indirectamente a través de una instancia de la clase FactoryAcceptCandidate, que implementa la interface IFFactoryAcceptCandidate. La interface tiene declarado un método createSearchCandidate() que tiene como argumento necesario un TypeAcceptation para determinar que clase concreta instanciar.

Mecanismo de diseño para la carga dinámica de clases

El conjunto de clases concretas de SeachCandidate y de AcceptableCandidate que pueden ser instanciadas podrían cambiar dinámicamente, si se quisieran incorporar a la biblioteca nuevas estrategias de búsqueda y aceptación. Producto de ello se implementó el mecanismo de carga dinámica de clases para ser usado en el patrón de diseño Método de Fábrica, y así no se tienen que redefinir los métodos createSearchCandidate() y createAcceptCandidate(), de las clases FactoryCandidate y FactoryAcceptCandidate.

Para la implementación del mecanismo de diseño que garantiza la carga dinámica de clases se creó la clase FactoryLoader, la cual contiene el método estático getInstance(), al que se le pasa por parámetro el nombre de la clase que se desea instanciar. El comportamiento de este método se basa en verificar que exista una clase con el mismo nombre que el parámetro que se le pasa y devolver una instancia de la misma. Esta forma dinámica de cargar clases en tiempo de ejecución se realiza utilizando el API (o Application Program Interface por su nombre en inglés): Java Reflection.

Java Reflection permite a los programas examinar y hacer cambios a su estructura y comportamiento en tiempo de ejecución. Este API posibilita obtener la clase de un objeto, examinar los métodos de una clase, invocar métodos descubiertos en tiempo de ejecución y explorar la jerarquía de la herencia [51].

Gracias a la utilización de Java Reflection, los métodos createSearchCandidate y createAcceptCandidate de las clases FactoryCandidate y FactoryAcceptCandidate solo deben invocar al método getInstance() y pasarle como parámetro el nombre de la clase concreta que se desea instanciar, obteniendo como resultado una instancia de la misma.

Es válido aclarar que:

- Para incorporar alguna nueva estrategia de búsqueda de candidatos solamente se debe crear una clase que herede de SearchCandidate, y añadir el nombre de la clase dentro del enumerado TypeCandidate.
- Para incorporar un nuevo criterio de aceptación solamente se debe crear una clase que herede de AcceptableCandidate, y añadir el nombre de la clase dentro del enumerado TypeAcceptation.

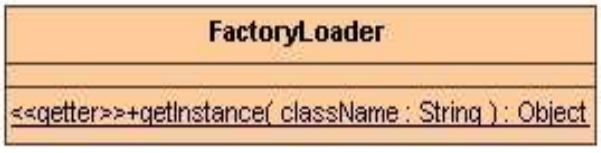

Fig. 7. Clase FactoryLoader

Clases reutilizadas propias de la tecnología

Java de conjunto con la máquina virtual proporciona un conjunto de clases que son reutilizables en la mayoría de las aplicaciones. Su empleo reduce considerablemente el tiempo de desarrollo y aumentan la calidad de las aplicaciones.

Particularmente se empleó:

Class: Objeto que accede dinámicamente a las clases e interfaces que están ejecutándose en la máquina virtual. Es creado por la máquina virtual y no tiene un constructor público [52].

ClassLoader: Objeto responsable de la carga dinámica de clases. A partir del nombre de la clase de Java que se cargará, que corresponde con el nombre del fichero, se procede a llenar los datos del objeto Class que lo invoca mediante el acceso a su referencia [52].

List: Interface que permite la manipulación de una colección de objetos ordenados. Se tiene control sobre todos los elementos que la conforman [52].

ArrayList: Clase que implementa la interface List, cuyo tamaño es variable. Es un arreglo secuencial [52].

SortedMap: Interface que permite la manipulación de una colección de pares llave – valor, donde la llave debe ser un valor único. Hereda de la interface Map. Garantiza el orden ascendente de sus elementos por el valor de la llave [52].

TreeMap: Clase que implementa la interface SortedMap, basándose en la estructura de dato árbol Rojo-Negro. La inserción de las llaves depende de la implementación de la interface Comparator [52].

CONFIGURACIÓN DE PARÁMETROS

Para combinar comportamientos en la biblioteca de clases se deben configurar los siguientes parámetros:

Inicialización: Si o No, indica si se generó o no, el estado inicial por un procedimiento ávido-aleatorio.

Estrategia: Normal o Tabú, indica si se emplea o no la lista tabú para filtrar la vecindad del estado actual.

Tipo de Búsqueda: Mayor, Menor o Aleatoria, indica el criterio de selección en la vecindad del estado actual.

Tipo de Aceptación: Mejor, Cualquiera, No tan malo, No tan malo T o No tan malo U, indica el criterio de aceptación de un estado candidato como nuevo estado actual.

La combinación de los diferentes valores que pueden tomar los parámetros, permite que se apliquen las Metaheurística que se muestra en la Tabla 4.

Tabla 4. Ejemplos de Metaheurísticas.

Algoritmo	Inicialización	Estrategia	Tipo de Búsqueda	Tipo de Aceptación
Escalador de Colinas Clásico	No	Normal	Mayor	Mejor
Búsqueda Tabú	No	Tabú	Mayor	Cualquiera
Recocido Simulado	No	Normal	Aleatoria	No tan malo T
Híbrido 1	No	Normal	Aleatoria	Cualquiera
Híbrido 2	No	Tabú	Mayor	Mejor
Híbrido 3	Si	Normal	Mayor	No tan malo U
Híbrido 4	Si	Tabú	Aleatoria	No tan malo

Conclusiones

A partir de los resultados obtenidos en el presente artículo, se puede concluir que se desarrolló una biblioteca de clases que implementa un modelo de integración de algoritmos metaheurísticos basados en un punto, haciendo uso de patrones y

mecanismos de diseño, que conciben el diseño más flexible y reutilizable. Permite la configuración de diferentes parámetros para combinar el comportamiento de varias Metaheurísticas.

Referencias bibliográficas

1. Winston, P.H., Inteligencia Artificial. Tercera ed.
2. Frederick Hillier, G.L., Introducción a la investigación de operaciones. 4ta ed. 1997: McGraw - Hill. 998.
3. Martí, R. (2003) Procedimientos Metaheurísticos en Optimización Combinatoria. Volume, 60
4. Rosete, A., Una solución flexible y eficiente para el trazado de grafos basada en el Escalador de Colinas Estocástico, in FACULTAD DE INGENIERÍA INDUSTRIAL, CEIS. 2000, ISPJAE: La Habana p. 140.
5. Riojas Cañari, A.C., Búsqueda Tabú: conceptos, algoritmo y aplicación al problema de las N-reinas, in Facultad de Ciencias Matemáticas. 2005: Lima, Peru.
6. Morgan, W., Simulated Annealing & other Combinatorial Approximation Algorithms. Concurrent Systems Engineering, 2006.
7. Olga Gómez, S.C., Laura Núñez, Joaquín Pacheco, Resolución del problema de selección de variables cuantitativas mediante GRASP. Aplicación a ratios financieros.
8. Leonidas S. Pitsoulis, M.G.C.R., Greedy Randomized Adaptive Search Procedures. 2001, AT&T Labs Research.
9. Konar, A., Artificial Intelligence and Soft Computing, in Behavioral and Cognitive Modeling of the Human Brain, D.o.E.a.T.-c.E.J. University, Editor. 2000, CRC Press: Calcutta, India. p. 788.
10. Larrañaga, J.A.L., Heinz Muhlenbein, Algoritmos de Estimación de Distribuciones en Problemas de Optimización Combinatoria. Revista Iberoamericana de Inteligencia Artificial, 2003. 19: p. 149-168.
11. Sergio Alonso, O.C., Iñaki Fernández de Viana, Francisco Herrera, La Metaheurística de Optimización Basada en Colonias de Hormigas: Modelos y Nuevos Enfoques.
12. David Keil, D.G., Adaptation and evolution in dynamic persistent environments. 2005, Computer Science & Engineering Dept. University of Connecticut Storrs.
13. Maischberger, M. METSlib Trac. 2007 [cited 2008 24/03]; Available from: http://code.100allora.it/metslib.
14. Jaszkiewicz's, A. MOMHlib++. 2007 [cited; Available from: http://www-idss.cs.put.poznan.pl/~jaszkiewicz/MOMHLib/.
15. Johann Dréo, J.-P.A., Walid Tfaili Main Page. Open Metaheuristics. 2006 [cited 2008 27/02]; Available from: http://ometah.berlios.de/index.php/Main_Page.
16. Stefan Bleuler, M.L., Lothar Thiele, Eckart Zitzler, PISA — A Platform and Programming Language Independent Interface for Search Algorithms. 2002, Swiss Federal Institute of Technology: Zurich, Switzerland.
17. Diana Holstein, P.M., Una Metaheurística Co-evolutiva para el Problema del Viajante de Comercio. 2001.
18. Bocca, P.R., Discusión y Análisis de la metaheurística SN. 2003.
19. Formella, D.A., Técnicas avanzadas de optimización. 2006, Universidad de Vigo, Departamento de Informática Área de Lenguajes y Sistemas Informáticos.
20. Lóvasz, M.G.y.L., Combinatorial Optimization: A Survey. 1993, DIMACS.
21. Thomas H. Cormen, C.E.L., Ronald L. Rivest, Clifford Stein, Capítulo 34 NP-Completeness in Introduction to Algorithms, McGraw-Hill, Editor. 2002.
22. Wilf, H.S., Capítulo 5. NP-completeness, in Algorithms and Complexity. 1994, University of Pennsylvania: Philadelphia.
23. Martí, R. (2003) Algoritmos Heurísticos en Optimización Combinatoria. Revista Iberoamericana de Inteligencia Artificial Volume, 123-130

24. Herrera, F., Introducción a los Algoritmos Metaheurísticos, in Ciencias de la Computación e I.A. 2006, Universidad de Granada.
25. Granada, G.d.T.s.M.d.D.y.O.U.d. ¿Qué es Soft Computing? . [cited 2008 2/5]; Available from: http://modo.ugr.es/es/soft_computing.
26. Carlos Castro, M.V., Jairo Maya, Algoritmo de búsqueda aleatoria para la programación de la producción en un taller de fabricación. Universidad Eafit, 2003. vol. 39(131): p. 76 - 86.
27. Juan J. del Coz Velasco, J.D.P., Sistemas Inteligentes. Búsqueda Heurística: A*. 2008: Escuela Politécnica Superior de Ingeniería de Gijón. Universidad de Oviedo.
28. Rosen, K.H., A Library of Graph Algorithms and Optimization", Discrete Mathematics and its Applications. 2007, Taylor & Francis Group.
29. Jones, T., Evolutionary Algorithms, Fitness Landscapes and Search. 1995, The University of New Mexico. p. 224.
30. Fausto García Márquez, M.L., Optimización: Conceptos Fundamentales y Tendencias Actuales.
31. Coello, C.A.C., Búsqueda Tabú: Evitando lo Prohibido in Soluciones Avanzadas. Tecnologías de Información y Estrategias de Negocios. 1997. p. 9.
32. Woodruff, D.L.Z., E. , Hashing vectors for tabu search. Annals of Operations Research, 1993. 41: p. 16.
33. Pierre Hansen, N.M.y.J.A.M.P., Búsqueda de Entorno Variable. 2003.
34. Pierre Hansen, N.M.y.J.A.M.P., A Tutorial on Variable Neighborhood Search. Les Cahiers du GERAD, 2003. G–2003–46.
35. Michael Polacek, K.F.D., Richard F. Hartl, Vittorio Maniezzo, A Variable Neighborhood Search for the Capacitated Arc Routing Problem with Intermediate Facilities, University of Vienna: Vienna, Austria.
36. Ricardo Alberto Hincapié Isaza, C.A.R.P., Ramón Alfonso Gallego R, Técnicas Heurísticas Aplicadas, in Grupo de investigación en planteamientos de Sistemas Eléctricos 2006, Universidad Tecnológica de Pereira.
37. Morales, E., Capítulo 6. Recocido Simulado, in Búsqueda, Optimización y Aprendizaje. 2004.
38. Cerny, V., Thermodynamical Approach to the Traveling Salesman Problem: An Efficient Simulation Annealing Algorithm. Opt. Theory Appl, 1985. 45 (1 / enero).
39. Ingber, L., Adaptative Simulated Annealing (ASA). 2008.
40. King-Wai Chu, Y.D., John Reinitzy, Parallel Simulated Annealing by Mixing of States. Journal of Computational Physics, 1999. 148(2): p. 17.
41. Page, C.H. OpenTS - Java Tabu Search. 2008 [cited; Available from: http://www.coin-or.org/Ots/index.html.
42. OR, C.O.I.f. Computational Infrastructure for Operations Research. 2008 [cited 2008 20/01]; Available from: http://www.coin-or.org/.
43. Developers.net. Rapid Application Development. 2005 [cited 2008 17/03]; Available from: http://www.blueink.biz/RapidApplicationDevelopment.aspx.
44. Ltd., S.S.P. The UML Tool for the Enterprise. 2008 [cited 2008 16/03]; Available from: http://www.sparxsystems.com.au/platforms/uml_tool.html.
45. Institute, S.E. What is a CASE Environment? 2007 11/01/2007 [cited 2008 24/03]; Available from: http://www.sei.cmu.edu/legacy/case/case_whatis.html.
46. Trottier, A., Java 2 Core Language Little Black Book. 2002, Paraglyph Press.
47. Eckel, B., Piensa en Java, F.d.I.U.d. Deusto, Editor. 2001, Pearson Education.
48. Eckel, B., Thinking in Patterns with Java. 2003: President, MindView.

49. Francisco Javier Martínez, J.M.C.L., Guía de Construcción de Software en Java con Patrones de Diseño. 2000, Escuela Universitaria de Ingeniería Técnica en Informática de Oviedo.

50. Erich Gamma, R.H., Ralph Johnson y John Vlissides Design Patterns: Elements of Reusable Object-Oriented Software. 1994: Addison-Wesley Professional Computing Series.

51. Forman, I.R. and N. Forman, Java Reflection in Action. 2005: Manning Publications Co.

52. Microsistem, S., JDK™ 5.0 Documentation, I. Sun Microsistems, Editor. 2004.

53 Y. Bardají , D. Paredes, J. Fajardo, A. Rosete: INTEGRACIÓN DE ALGORITMOS METAHEURÍSTICOS DE UN SOLO PUNTO, XIV Latin Ibero-American Congress on Operations Research (CLAIO 2008), Cartagena de Indias, Colombia, 8-12 Septiembre, 2008.