Universidad de Granada

Departamento de Ciencias de la Computación e Inteligencia Artificial

Uncertain and Dynamic Optimization Problems: Solving Strategies and Applications

Doctoral Thesis

Doctoral Candidate:
Ignacio José García del Amo

Advisors:
David Alejandro Pelta
José Luis Verdegay Galdeano

Editor: Editorial de la Universidad de Granada
Autor: Ignacio José García del Amo
D.L.: GR 749-2013
ISBN: 978-84-9028-427-8

La memoria titulada **"Uncertain and Dynamic Optimization Problems: Solving Strategies and Applications"**, que presenta D. Ignacio José García del Amo para optar al grado de Doctor en Informática, ha sido realizada en el Departamento de Ciencias de la Computación e Inteligencia Artificial de la Universidad de Granada, bajo la dirección de los doctores D. David Alejandro Pelta y D. José Luis Verdegay Galdeano, del mismo departamento.

El doctorando y los directores de la tesis garantizamos, al firmar esta tesis doctoral, que el trabajo ha sido realizado por el doctorando bajo la dirección de los directores de la tesis, y hasta donde nuestro conocimiento alcanza, en la realización del trabajo se han respetado los derechos de otros autores a ser citados cuando se han utilizado sus resultados o publicaciones.

Granada, ... de 2012

Ignacio José García del Amo

David Alejandro Pelta　　　　José Luis Verdegay Galdeano

— And I will need the help of, oh, sixty apprentices and journeymen from the Guild of Cunning Artificers. Perhaps there should be a hundred. They will need to work round the clock.

— Apprentices? But I can see to it that the finest craftsmen…

Leonard held up a hand.

— Not craftsmen, my lord — he said —. I have no use for people who have learned the limits of the possible.

Terry Pratchett, *The Last Hero*

Contents

Agradecimientos

Esta tesis ha sido el resultado de muchos años de esfuerzo y de trabajo, y como cualquier empresa de larga duración y grandes aspiraciones, no ha estado exenta de dificultades. No será la primera ni la última que emprenda en mi vida, pero si he podido superar ésta en particular, ha sido no sólo porque me he subido a hombros de gigantes, sino porque me he apoyado en hombros de compañeros. Es justo por tanto reconocer y agradecer la ayuda que he recibido durante todo este tiempo. Así, pues, gracias...

A mis directores de tesis, David y Curro, por haberme guiado con sus enseñanzas en esta inolvidable etapa de mi vida que ha sido el doctorado. Esta tesis es fruto de los conocimientos y la experiencia investigadora que he adquirido en los últimos años gracias a la ayuda y consejos de mis tutores, no sólo en el ámbito académico, sino también en el personal.

A la Junta de Andalucía, por financiar mi formación doctoral a través de su programa de Proyectos de Excelencia. Dicha financiación también ha hecho posible la presentación de algunos de los trabajos de esta tesis en congresos internacionales de gran prestigio, así como mi estancia de investigación en la Universidad de Brunel, en Uxbridge, UK.

A Shengxiang Yang, por haber sido mi tutor y un excelente anfitrión durante dicha estancia, así como por animarme con sus ideas y su motivación en la investigación que allí realicé.

A mis compañeros del grupo MODO. Si en la elaboración de una tesis doctoral es crucial el apoyo de los directores e instituciones, no es menos importante el de los compañeros de trabajo. En este sentido, me siento orgulloso de haber formado parte del grupo MODO, cuyos miembros no sólo han colaborado activamente en muchas de las investigaciones llevadas a cabo en esta tesis, sino que han traspasado con creces las fronteras de lo laboral para convertirse en verdaderos amigos. Desde aquí quiero expresar mi cariño y respeto a Maite, Carlos, Antonio, Dago, Pablo, Quique, Socorro, y muy especialmente, a Juanra y Ricardo. Estos años han supuesto un crecimiento académico y profesional, pero nada puede compararse al crecimiento personal que he vivido con vosotros. Las conversaciones, pensamientos, ideas, ilusiones y aventuras que hemos compartido merecen un libro aparte.

A mis compañeros de la Universidad de La Laguna, a Pepe, a Belén, a Marcos, a Julio, a Cande, a Pino, a Miguel, a Pooky, a Rayco, a David... Y a todos aquéllos que, aunque no pertenezcan a la ULL, mi mente no puede evitar asociarlos con una sonrisa a los dos años que pasé allí: Kenneth, Silvija, Zrinka, Sergio... Tenerife marcó el comienzo de mi carrera investigadora, y siempre estará presente en mi memoria y en mi corazón.

A mis amigos. La vida es lo que pasa mientras estás ocupado haciendo el doctorado, y gracias a ellos, puedo decir que he vivido. No quiero olvidarme de los becarios de la UGR, de mis amigos de Madrid y Colmenar, de la gente de taekwondo, o de los compañeros que conocí (o reencontré) en Brunel. Demasiados como para nombrarlos a todos. Iván, Iker e Isra merecen mención aparte, porque siempre han estado ahí para lo bueno y para lo malo. Y con tantas coincidencias sería imposible pasar por alto a Núri.

A D. Manuel y a D. Lorenzo, porque muchos años después, todo lo que me enseñaron todavía sigue teniendo utilidad en mi vida. De ellos aprendí la importancia de hacer las cosas con la máxima calidad, dando siempre lo mejor de uno mismo.

A toda mi familia, que siempre me ha apoyado, y a la que tanto quiero. Me gustaría dar las gracias con especial cariño a mis "yeyes", Carmen y Mariano, y a mi abuela Carmen y mi tía Mª Carmen. A mis hermanos, Cristina, Carlos y Javier, a Carlos y a Mariví, y al pequeño Aarón, la alegría más reciente de mi familia. Y por supuesto, a Ester, quien con su cariño y apoyo incondicional ha llenado mi vida. No os podéis imaginar cuántas sonrisas os debo.

Y muy especialmente a mis padres, porque no es posible entender quién soy sin mencionarles de forma preeminente. Su amor, esfuerzo y dedicación han inspirado siempre en mí el mayor de los cariños. A ellos quisiera dedicarles esta tesis con todo mi agradecimiento.

Esta tesis ha sido financiada por la Junta de Andalucía con los proyectos P07-TIC-02970 y P11-TIC-8001, y por el Ministerio español de Ciencia e Innovación a través de los proyectos TIN2008-01948 y TIN2011-27696-C02-01.

Abstract

In the daily life we continuously face situations that can be formulated as dynamic optimization problems (DOPs), where the additional presence of uncertainty can make it even more difficult to solve them. Soft Computing techniques have some properties that turn them into ideal candidates for dealing with this type of problems. In this context, the thesis entitled *"Uncertain and Dynamic Optimization Problems: Solving Strategies and Applications"* focuses in the study, design and implementation of Soft Computing based methods to solve DOPs.

The research carried out in the thesis has produced new algorithms for DOPs, like Cooperative Strategies (CS), the Agents algorithm, and several improvements to a Particle Swarm Optimization (PSO) algorithm using a novel operator and heuristic rules. Additionally, because of the necessities that emerged when analyzing the experiments carried out, we developed a technique for comparing multiple algorithms over multiple DOP scenarios, named SRCS. This technique is specially well-suited for analyzing huge amounts of data. Lastly, we created a software tool for the configuration and analysis of algorithms for DOPs, as well as a framework for the implementation of this type of problems.

The results of this thesis have improved algorithms of the state of the art, obtaining interesting conclusions like the important role of cooperation for such purposes. Furthermore, the SRCS technique has allowed to discover which scenarios are most favorable for a set of algorithms, thus helping to identify general trends and behaviours. Finally, the experience obtained during the implementation of these experiments has derived in a collection of highly useful software for DOPs that has been made available to the research community as open source.

Resumen y conclusiones

Este capítulo presenta un resumen de la introducción (Capítulo 1) y conclusiones principales (Capítulo 6) de esta tesis en lengua española, a fin de cumplir con lo establecido en la normativa vigente de regulación de las enseñanzas oficiales de Doctorado y del título de Doctor por la Universidad de Granada aprobadas por Consejo de Gobierno de la Universidad de Granada en su sesión del 2 de Mayo del 2012.

Vivimos un momento de la historia en el que nuestras necesidades sociales, culturales y tecnológicas están fuertemente interrelacionadas. Ya no es posible pensar en resolver algunas de esas necesidades sin abordar, aunque sea de manera indirecta, alguna de las otras. Una aplicación informática que no tenga en cuenta los factores socioculturales de los usuarios a los que va dirigida está destinada al fracaso. Pero a la vez, y es aquí donde reside el enorme potencial de crecimiento que estamos experimentando, gracias a los recientes avances tecnológicos, podemos resolver necesidades que antes no era posible solucionar, o hacerlo de formas hasta hace poco impensables, abriendo un mundo de posibilidades a nuestro alcance.

En este contexto, muchas situaciones pueden ser modeladas como problemas de optimización, donde el objetivo es encontrar la mejor solución posible, con los recursos disponibles, que satisfaga un cierto criterio. Pensemos por ejemplo en la mejor forma de ordenar un conjunto de resultados de una búsqueda en internet en base a los términos introducidos y las preferencias de un usuario, o la forma óptima de servir una compra a un conjunto de clientes de una ciudad, o la exploración del grafo de amistades de una persona para sugerirle nuevos contactos lo más afines posible, etc. Este área, el de la optimización, clásica ya en el entorno de la investigación académica, se ha centrado históricamente en problemas estáticos y bien definidos.

Sin embargo, avanzamos hacia un mundo cada vez más integrado, más interrelacionado, más globalizado. Los problemas y necesidades a resolver no son ya algo perfectamente definido y acotado, sino que muchas veces implican dependencias entre elementos muy diferentes y complejos de por sí. Estas dependencias, a veces desconocidas, pueden desencadenar cambios muy rápidos a los que hay que dar respuesta, incluso sin disponer de toda la información posible. Los retos a abordar implican, pues, ser capaces de adaptarnos a entornos cambiantes (y lo que es más, que cambian cada vez más rápido), y de operar con información incompleta, difusa o a veces incluso, contradictoria.

Pensemos en el ejemplo anterior de la búsqueda: es evidente que las preferencias de un usuario no van a permanecer invariables a lo largo del tiempo, y lo que hoy puede ser una recomendación útil por parte del sistema, mañana podría dejar de serlo porque la persona ha cambiado sus intereses; el sistema debe ser capaz de adaptarse a este cambio. O podemos querer sugerir contactos en una red social a alguien que no ha rellenado todos los datos de su perfil, y de quien, por tanto, no disponemos de toda la información completa; la ausencia de información no debe ser obstáculo para sugerir contactos usando los datos de los que sí disponemos. Ejemplos de situaciones similares pueden encontrarse en otras áreas relevantes (Economía, Meteorología, Logística y Transporte, Bioquímica, Telecomunicaciones, etc.), y en todas ellas, los sistemas no sólo deben ser capaces de gestionar estos problemas, sino de proporcionar soluciones con la mayor calidad posible.

En esta situación, necesitamos Sistemas Inteligentes que sean capaces de lidiar con los problemas que surgen debido a la presencia de *dinamismo* e *incertidumbre.*

Contexto de la tesis

En el contexto de los Sistemas Inteligentes, una importante clase de problemas son los conocidos con el nombre de problemas de optimización, habitualmente asociados a tener que encontrar el máximo o mínimo valor que una determinada función puede alcanzar en un cierto conjunto previamente especificado. Todo lo relativo a estos problemas se enmarca dentro del cuerpo doctrinal denominado Programación Matemática, que incluye una enorme variedad de situaciones, según que se consideren casos lineales, no lineales, aleatoriedad, un solo decisor o varios decisores, etc.

Los Problemas de Optimización Dinámicos (DOPs) es una categoría de problemas de optimización dentro de la que se encuadran aquellos problemas donde la función objetivo, las variables, las condiciones ambientales y/o la estructura del problema pueden cambiar mientras el mismo problema se está resolviendo. La formulación básica de un problema de este tipo es la siguiente:

$$DOP = \left\{ \begin{array}{l} \text{Optimizar } f(x,t) \\ \text{s.a. } x \in F(t) \subseteq S,\ t \in T \end{array} \right\} \tag{1}$$

donde

- $S \in \mathbb{R}^n$, S es el espacio de búsqueda.

- t es el tiempo (puede ser también medido en términos de *evaluaciones de la función objetivo*).
- $f : S \times T \rightarrow \mathbb{R}$, es la función objetivo que asigna a cada posible solución $x \in S$ en el instante t un valor numérico.
- $F(t)$, es el conjunto de soluciones factibles $x \in F(t) \subseteq S$ en el instante t.

En muchas ocasiones, encontrar la mejor solución posible a un problema de optimización suele ser una tarea compleja. Las causas de esto pueden ser múltiples, como que los problemas tengan una complejidad computacional elevada y no se pueda garantizar que hallemos la solución óptima en un tiempo razonable (e.g., problemas NP-duros), o bien que el coste de evaluar una solución, ya sea en términos económicos o temporales, sea elevado. La presencia de dinamismo supone una dificultad añadida a la hora de resolver estos problemas. **En este contexto, las técnicas proporcionadas por la Soft Computing emergen como candidatas ideales para afrontar estos problemas, especialmente las Metaheurísticas**.

El término "metaheurística" apareció por primera vez en un trabajo de Glover en 1986 [65], y resulta de la adición del término *meta* ("más allá" o "de un nivel superior") a la palabra *heurística* (del Griego, "heuriskein" — "descubrir" o "encontrar" [161]). Las heurísticas son métodos de optimización por búsqueda capaces de encontrar soluciones de alta calidad con un coste computacional razonable, aunque su optimalidad o factibilidad no estén garantizadas. Normalmente las heurísticas se consideran opuestas a los *métodos exactos*, ya que a éstos se les exije que proporcionen optimalidad y factibilidad. En este sentido, las metaheurísticas surgieron en un intento por estar "por encima de las heurísticas", con la idea de extraer las mejores partes de diversas heurísticas de éxito para crear métodos genéricos que pudieran ser aplicados a una variedad más amplia de problemas y conceptos, manteniendo a la vez las cualidades de "soluciones de alta calidad" y "con un coste computacional razonable".

Existe una enorme variedad de metaheurísticas con diferentes grados de complejidad y capacidad de optimización: Local Search, Multi-Start Local Search, Simulated Annealing, Tabu Search, Scatter Search, Greedy Randomize Adaptive Search Procedure (GRASP), Variable Neighbourhood Search (VNS), Evolutionary Algorithms (EA), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), etc. Cada una de estas metaheurísticas ha dado lugar a múltiples variaciones, y es común la creación de metaheurísticas híbridas combinando dos o más de ellas, utilizando la arquitectura maestro-subordinado inherente a su definición.

Todas estas metaheurísticas están basadas en diferentes paradigmas y poseen características únicas que las distinguen de las demás. Sin embargo, todas ellas en general (con la posible excepción de la sencilla Local Search) se basan en última instancia en combinaciones cuidadosamente calibradas de procesos de *diversificación* e *intensificación*.

La combinación de estos dos factores las hacen especialmente apropiadas para resolver problemas de optimización global. Sin embargo, y esto es lo realmente relevante para esta tesis, estas mismas características les confieren una capacidad de adaptación extraordinaria, a la vez que mantienen un gran potencial de búsqueda. Todo esto las convierte en candidatas ideales para resolver problemas de optimización que tengan presencia de *dinamismo* e *incertidumbre*.

Con el objetivo de abordar los retos que suponen este tipo de problemas y proporcionar métodos capaces de dar soluciones a los mismos, se plantea el Proyecto de Excelencia de la Junta de Andalucía *P07-TIC-02970*, titulado *"Modelos de Optimización Dinámicos e Imprecisos en Sistemas Inteligentes: Estudio de Métodos de Solución y Aplicaciones"*, dentro del cuál se enmarca esta tesis doctoral.

El núcleo del proyecto se estructura alrededor de los posibles escenarios que pueden surgir cuando se combinan las componentes de los problemas (básicamente, hablamos de función objetivo y restricciones), y las características de dinamismo e incertidumbre.

En el marco de este proyecto de investigación, **esta tesis se centra en uno de los escenarios estudiados: problemas de optimización con dinamismo en la función objetivo**.

Objetivos

Teniendo en cuenta la importancia de los DOPs, la necesidad de resolverlos, y la previsible adecuación de la técnicas de Soft Computing para ello — especialmente las Metaheurísticas — **se plantea como objetivo global de esta tesis el estudio, diseño e implementación de métodos basados en Soft Computing para la resolución de DOPs**. Para alcanzar este objetivo, hemos realizado un conjunto de tareas orientadas a responder las siguientes cuestiones:

1. **¿Es posible mejorar los algoritmos existentes, y, en caso afirmativo, qué técnicas se podrían usar para ello?** Para responder a esta pregunta será necesario ver qué tipos de algoritmos existen en el estado del arte que sean capaces de operar en presencia de dinamismo, e investigar su funcionamiento y los elementos que la forman. Pondremos especial interés en investigar el papel que juega la cooperación entre estos elementos de cara a mejorar la efectividad del algoritmo. Como objetivo secundario a la hora de contestar a esta pregunta, buscaremos técnicas para mejorar los algoritmos existentes que, en la medida de lo posible, no sean exclusivas de dichos algoritmos, sino que puedan aplicarse a otros de forma genérica.

2. **¿Qué metodología habría que aplicar para comparar el rendimiento de diferentes algoritmos sobre un DOP?** La presencia de dinamismo en la función objetivo obliga a redefinir cómo evaluamos el rendimiento de un algoritmo. No basta con encontrar una solución al problema una vez: el hecho de que el problema esté contínuamente cambiando obliga a reportar soluciones cada cierto tiempo. Intuitivamente, es mejor un algoritmo que devuelva soluciones razonablemente buenas de manera sostenida, que uno que encuentre la solución óptima una vez, pero que obtenga soluciones malas el resto del tiempo. Con esta forma de evaluar el rendimiento, la comparación de resultados se vuelve compleja, y para responder a la pregunta de este punto, es necesario buscar formas novedosas de mostrar la información para poder captar todo su significado e implicaciones.

3. **¿Qué dificultades podemos encontrarnos a la hora de implementar estos DOPs, algoritmos y medidas, y cómo podemos afrontarlas?** Esta cuestión, más técnica y aplicada que las anteriores, se basa en la experiencia acumulada durante la investigación, donde hemos podido comprobar que pasar de la teoría a la implementación práctica no siempre es fácil. La implementación de los algoritmos, los problemas, los mecanismos de dinamismo e incertidumbre, y las medidas de rendimiento presentan cada una sus propias particularidades. La respuesta a esta pregunta pretende ser un compendio de dicha experiencia y las conclusiones obtenidas.

Para ello, la tesis está estructurada de la siguiente manera:

- El Capítulo 2 presenta el contexto de investigación en el que se desarrolla esta tesis, con una breve revisión de la literatura en DOPs, centrándose en los problemas, algoritmos y medidas de rendimiento más relevantes para los objetivos de la tesis.

- El Capítulo 3 contiene las contribuciones algorítimicas más significativas de esta tesis, bien mediante mejoras a algoritmos existentes de la literatura, o directamente con nuevos algoritmos que han sido desarrollados durante esta investigación. Este capítulo se centra principalmente en buscar respuestas a la primera cuestión presentanda en los objetivos.

- En el Capítulo 4 introducimos una técnica nueva para comparar grandes cantidades de datos de una forma comprensible y aportando significado. Esta técnica, denominada SRCS, ha permitido extraer información muy valiosa sobre tendencias generales de algoritmos, así como recomendaciones sobre qué algoritmo utilizar en cada situación. Este capítulo está enfocado principalmente a tratar de responder la segunda cuestión presentada en los objetivos.

- El Capítulo 5 explora algunas de las dificultades y problemas a los que nos enfrentamos durante la implementación de los experimentos, y explica cómo los resolvimos. Este capítulo se ocupa de los aspectos del software y la experiencia que reunimos durante la investigación, y se centra por tanto en la tercera cuestión presentada en los objetivos.

- Finalmente, el Capítulo 6 presenta las conclusiones obtenidas de esta tesis, el conocimiento más relevante que hemos adquirido durante la experimentación, y responde a las cuestiones planteadas en los objetivos. Este capítulo también incluye un resumen de las publicaciones más relevantes obtenidas durante la realización de esta tesis.

Conclusiones

A raíz de las investigaciones llevadas a cabo en esta tesis y de las publicaciones a las que han dado lugar, organizaremos las conclusiones obtenidas a partir de las preguntas que se plantearon en los Objetivos.

¿Es posible mejorar los algoritmos existentes, y, en caso afirmativo, qué técnicas se podrían usar para ello?

Las investigaciones llevadas a cabo han producido, entre otros resultados, los siguientes algoritmos nuevos o mejoras de existentes:

- Un nuevo algoritmo para DOPs basado en Estrategias Cooperativas (CS) [69], así como una mejora para dicho algoritmo con reglas más efectivas [70].

- Un algoritmo basado en Agentes para DOPs, con resultados interesantes en problemas continuos [40], y muy prometedores en problemas combinatorios [68].

- Una serie de mejoras y estudios de propiedades de variantes del PSO para DOPs basadas en el uso de reglas heurísticas (mQSO-Change,, mQSO-Rand y mQSO-Both) [39, 41, 120].

Los algoritmos que hemos desarrollado abarcan un amplio espectro de métodos: PSO con reglas heurísticas, metaheurísticas de trayectoria cooperativas o agentes. Sin embargo, todos estos algoritmos tienen ciertas características comunes:

1. **Uso de poblaciones de soluciones**. Todos estos métodos mantienen un conjunto o conjuntos de soluciones que “conviven” a la vez en el transcurso

del proceso de búsqueda. Estas poblaciones de soluciones permiten diversificar la búsqueda, algo de por sí importante en un problema de optimización estático, y absolutamente imprescindible en un DOP.

2. **Existencia de algún tipo de cooperación entre los elementos constituyentes**. En el caso de las estrategias cooperativas esto es evidente, ya que dicha cooperación es explícita, y corre a cargo del coordinador central, quien intercambia información entre las diferentes metaheurísticas. En el caso de las diferentes variantes del PSO, las partículas de un mismo swarm están "conectadas" entre sí a través del *best* de dicho swarm, ya que sirve de referencia en el movimiento de las partículas. Además, en el caso del mQSO, que posee múltiples swarms, existe un método de exclusión que impide a dichos swarms acercarse demasiado unos a otros para evitar concentrarse en el mismo óptimo. Esta competición entre swarms a nivel local evita que el algoritmo desperdicie demasiados recursos en un área, por lo que, en cierto sentido, es una forma de cooperación a nivel global. Los agentes también cooperan de forma implícita, ya que explorar el espacio de búsqueda de forma indirecta a través de la matriz obliga a que los agentes mejoren soluciones que muy probablemente han sido ya modificadas por otros agentes anteriormente.

Estas características vienen a confirmar el uso de algoritmos poblacionales como tendencia general entre los más usados para DOPs. La segunda característica es especialmente significativa en el contexto de esta tesis porque de alguna forma indica una posible vía a seguir de cara a mejorar algoritmos existentes. Todas las modificaciones que hemos introducido a lo largo de esta investigación se basan en fomentar la cooperación y aumentar el intercambio de información entre los elementos del algoritmo.

Especialmente interesante es el caso de la regla *Rand* del mQSO con reglas heurísticas [39]. La filosofía de esta regla es muy similar a la del PSO con el operador CPT [120] y al funcionamiento de las Estrategias Cooperativas (CS) [69, 70, 104]: observar el comportamiento de los elementos, y corregir los que lo estén haciendo peor, ya sea tratando de imitar a los mejores, o parándolos temporalmente para que no desperdicien recursos en momentos críticos. Cuando este esquema de cooperación se ha aplicado sobre un algoritmo, ha mejorado sus resultados en *todos* los casos.

Por otra parte, de los resultados obtenidos en [40], llama la atención la abrumadora supremacía de las CS en los problemas Ackley, Griewank y Rastrigin. Tal y como se explica en ese trabajo, parece que las estrategias cooperativas están especialmente indicadas cuando el DOP a resolver posee algún tipo de estructura en las posiciones relativas de los óptimos locales. En cambio, en problemas donde no existe dicha estructura, como el caso del MPB, los resultados están mucho más igualados, y se aprecian mejor los comportamientos y tendencias de cada algoritmo. SORIGA es la mejor opción para entornos con cambios muy rápidos, el mQSO + Regla Rand obtiene los mejores resultados para baja dimensionalidad, y el algoritmo Agentes se comporta mejor en alta dimensionalidad y con creciente severidad en los cambios.

Una interpretación que podemos hacer de esto es que el tipo de información del entorno que utiliza CS para decidir cómo cooperar les permite sacar el máximo partido a la estructura del problema. Esta información no es en cambio tan útil en el MPB porque no hay estructura de la que se pueda aprovechar, y pierden por tanto su ventaja. La cooperación impuesta por la regla *Rand* en el mQSO mejora su rendimiento, pero no puede sin embargo superar las limitaciones inherentes al propio algoritmo: las partículas tienden a explorar el entorno siguiendo una trayectoria y los swarms tienen un radio efectivo de acción, más allá del cual es difícil que exploren. Esto no supone un gran inconveniente en problemas de baja dimensionalidad, pero penaliza al algoritmo a medida que ésta aumenta, en favor del algoritmo Agentes. Finalmente, SORIGA posee un mecanismo de cooperación implícito y genérico, que no usa conocimiento del problema. Esto es una desventaja cuando las condiciones del problema son favorables a otras técnicas. Sin embargo, dota al algoritmo de una gran robustez, y le permite obtener buenos resultados en entornos con cambios muy rápidos, en situaciones en que los otros algoritmos no han tenido tiempo para optimizar adecuadamente.

En el caso de los problemas discretos, el uso del algoritmo Agentes combinado con un esquema de aprendizaje permite mejorar a uno de los métodos del estado del arte en este área, AHMA. El uso de cooperación explícita en este caso era complicado de utilizar debido a que la estructura del problema no estaba clara y era más bien aleatoria. Sin embargo, la cooperación implícita de los Agentes era apropiada para manejar estos escenarios, y el esquema de aprendizaje se pudo incorporar sin ningún problema con este algoritmo, permitiendo superar al AHMA.

Adicionalmente, a partir de la experiencia que hemos adquirido durante el desarrollo de los experimentos, hemos elaborado una clasificación de los algoritmos utilizados en esta tesis en función de dos criterios: *capacidad de optimización* y *flexibilidad*. Cuando hablamos de capacidad de optimización nos referimos a la habilidad de un algoritmo de producir soluciones de alta calidad, cercanas al óptimo. Cuando hablamos de flexibilidad, nos referimos a un doble concepto: la facilidad de un algoritmo, en *tiempo de ejecución*, para reaccionar a los cambios en el entorno, y a la facilidad, en *tiempo de diseño*, para adaptar el algoritmo a otros problemas (cómo de bien funciona el algoritmo en un problema nuevo con unos valores estándar de sus parámetros, cuánto código hay que modificar o crear para que funcione en otro problema, etc). Esta clasificación es un compendio de los resultados obtenidos en los diferentes trabajos (especialmente [40], presentado en la Sect. 4.4) y nuestra experiencia personal durante el desarrollo de los mismos.

En nuestra opinión, la familia del mQSO obtiene peores rendimientos en DOPs que otras familias que hemos analizado. Tan sólo la variante heurística del mQSO-Rand consigue buenos resultados en términos de *capacidad de optimización* en algunos escenarios, gracias al esquema de cooperación que utiliza. Sin embargo, este algoritmo contiene un número elevado de parámetros dependientes del problema, tales como el radio de las partículas quantum, distancia mínima entre swarms, los datos de configuración de las reglas, etc. Como vimos en [39,40], esto puede llevar a valores de parámetros que sólo funcionan bien en ciertos escenarios. Además, el mQSO está específicamente diseñado para optimización contínua, y contiene algunas dependencias en la propia formulación del método que hacen muy complicado portarlo a problemas discretos (las ecuaciones de movimiento, el concepto de "distancia", etc). Por todo ello, nuestra impresión es que esta familia es *poco flexible.*

Respecto a la familia de las CS, es evidente que son las que más *capacidad de optimización* tienen, a raíz de los resultados obtenidos en [40,69,70] . Además, son *razonablemente flexibles*, ya que el CS se ha utilizado con éxito en una gran variedad de problemas, incluyendo continuos y discretos. Sin embargo, el algoritmo CS es bastante complejo, con varias capas de abstracción incluyendo la implementación de los solvers, la implementación del coordinador, los parámetros de configuración, mensajes de comunicación, pizarras, sincronización, etc. Esto implica que puede ser necesario modificar una cantidad considerable de código para poder adaptar el CS de un problema a otro. Por ello, creemos que este método tiene una *flexibilidad moderada.*

Finalmente, la familia de Agentes y la de Algoritmos Evolutivos son, con diferencia, las *más flexibles* de todas. Apenas utilizan parámetros, o los parámetros poseen valores estándard muy buenos, lo que permite a ambas familias obtener un rendimiento aceptable en la mayoría de los problemas en sus primeras ejecuciones. Por ejemplo, SORIGA no tiene parámetros dependientes del problema, y la matriz por defecto 3x3 del algoritmo Agentes obtiene muy buenos resultados en todos los experimentos realizados. Además, son capaces de funcionar bien incluso en circunstancias extremas, como en DOPs con una elevada frecuencia de cambio o severidad (ver [40]). A cambio, no exhiben una *capacidad de optimización* tan elevada como las CS, pero son capaces de competir con el mQSO. Y en el caso del Agents-Adaptativo, la flexibilidad aplicada a la selección de operadores le permite obtener los mejores resultados de esta familia en términos de *capacidad de optimización.*

En conclusión, los resultados de todos estos trabajos permiten afirmar que **no sólo es posible mejorar los algoritmos existentes, sino que además hemos encontrado técnicas basadas en cooperación y esquemas de aprendiza-**

je para hacerlo de forma genérica, con gran efectividad y flexibilidad. Adicionalmente, hemos obtenido resultados que nos permiten conocer cuáles son los escenarios más favorables para utilizar cada algoritmo.

¿Qué metodología habría que aplicar para comparar el rendimiento de diferentes algoritmos sobre un DOP?

Los resultados obtenidos en diferentes fases de la investigación llevada a cabo en esta tesis demuestran que un determinado algoritmo puede resultar efectivo en un escenario de un DOP, y perder dicha efectividad en otro. Vimos un ejemplo de esto en [39] con la regla *Change* en escenarios distintos al Escenario 2 del MPB, o con el algoritmo Agentes al compararlo contra el AHMA en alguna configuración del problema Royal Road en [68]. Este resultado es ampliamente conocido en la literatura, normalmente conocido como el *teorema no-free-lunch* [171], i.e., ningún algoritmo puede ser el mejor en todas las situaciones posibles.

Con el objetivo de poder obtener una visión más completa del comportamiento, las fortalezas y las debilidades de un algoritmo, es necesario evaluar su rendimiento en múltiples escenarios. Por otra parte, como ya se ha justificado anteriormente, esta comparación puede producir demasiados resultados como para puedan ser comprendidos en toda su magnitud si sólamente se usan datos puramente numéricos.

Por todas estas razones, hemos desarrollado la técnica SRCS [38]. Esta técnica comprime la información a mostrar en 2 fases:

1. Transformar los datos *absolutos* de rendimiento de los algoritmos en datos *relativos* entre los algoritmos comparados, de manera que se pueda establecer un ránking que determine si un algoritmo es *mejor que* otro en un escenario dado.

2. Asignar colores a cada ránking, de manera que el rendimiento relativo de cada escenario analizado pueda ser visualizado en una imagen a color.

Esta técnica permite visualizar los resultados de un gran número de experimentos de manera comprensible, ayudando a identificar tendencias y patrones de comportamiento de los algoritmos analizados. Por ello, **gracias a SRCS se pueden sacar conclusiones muy prácticas, no sólo sobre qué algoritmo es el mejor, sino — y esto es más importante — sobre en qué escenarios es mejor utilizar uno u otro**. SRCS ha sido utilizada en los trabajos [68] y [40], donde su uso ha permitido extraer valiosas conclusiones.

¿Qué dificultades podemos encontrarnos a la hora de implementar estos DOPs, algoritmos y medidas, y cómo podemos afrontarlas?

La investigación llevada a cabo en esta tesis ha tenido una fuerte componente experimental. La implementación del sofware necesario para llevarla a cabo no ha estado exenta de problemas, y de ella hemos extraído las siguientes lecciones:

- Para obtener resultados útiles es necesario realizar un número potencialmente alto de ejecuciones de algoritmos, posiblemente sobre un gran número de variaciones de escenarios.
- En muchos casos los algoritmos están basados en metaheurísticas y tienen un factor aleatorio. Por ello, se debe repetir cada ejecución un cierto número de veces con diferentes semillas para obtener una muestra estadísticamente significativa de los resultados.
- Debido al carácter aleatorio de los algoritmos, en el análisis de los resultados es necesario utilizar herramientas estadísticas. El uso de paquetes software con estas capacidades es por tanto casi obligatorio.
- Los escenarios de pruebas suelen tener también una componente aleatoria en el dinamismo. Sin embargo, al contrario que los algoritmos, esta componente debe ser reproducible, para así garantizar que los experimentos de cada método se realizan en igualdad de condiciones, con exáctamente el mismo escenario.
- La mayoría de los algoritmos tienen un conjunto de parámetros que deben establecerse antes de la ejecución, con un amplio rango de valores posibles. Esto es especialmente complejo en los Sistemas de Optimización Cooperativos (COS), donde cada componente puede tener su propio conjunto de parámetros. Es por tanto muy recomendable disponer de mecanismos de configuración que faciliten este proceso.
- Una visualización del rendimiento de un algoritmo es extremadamente útil, especialmente en las primeras etapas de un experimento. Las visualizaciones ayudan a decidir si un problema está formulado correctamente y si el algoritmo está evolucionando razonablemente. Si las visualizaciones pueden realizarse en tiempo real, mejor aún.
- Aunque existe una gran variedad de algoritmos para DOPs, la mayoría de ellos se basan en poblaciones de elementos, con algún tipo de cooperación entre ellos, ya sea explícita o implícita. Esto abre la puerta a la creación de jerarquías de clases y módulos que permitan la reutilización de componentes.

- De igual forma, aunque también existe una gran variedad de problemas dinámicos, cada uno de ellos con su propia función objetivo, espacio de búsqueda, etc., el proceso mediante el cuál se evalua una solución y se le asigna un valor de fitness es normalmente común a todos ellos. Más aún, las las características que definen el dinamismo (tales como la frecuencia de cambio, la severidad, el tipo de cambio, etc.), así como otro tipo de funcionalidad extendida (incluyendo medidas de rendimiento, GUIs, estadísticas, etc.), normalmente pueden intercambiarse entre los diferentes problemas. Al igual que sucedía con los algoritmos, esto sugiere de nuevo que la creación de un framework con estos componentes es posible.

A raíz de estos datos podemos sacar varias conclusiones. La primera de ellas es que **la configuración y ajuste inicial de algoritmos para DOPs no es una tarea sencilla**. Con el objetivo de aliviar esta situación, hemos creado una herramienta basada en modelos de software, DACOS [42], que permite configurar COS y visualizar su rendimiento en las fases iniciales de la experimentación. DACOS fue diseñado para cubrir una amplia variedad de algoritmos y problemas, pero para aquellos casos cuyas necesidades específicas requieran de su adaptación, el uso de modelos de software permite hacerlo de manera semi-automática.

Por otra parte, las características de los algoritmos usados en DOPs hacen que muchos de los elementos que los componen, y hasta su misma estructura interna, puedan ser reutilizados de un algoritmo a otro a la hora de implementarlo. La misma situación se da en los problemas, donde los mecanismos de dinamismo son compartidos entre ellos, y con frecuencia sólamente se diferencian en la función objetivo. Esta situación se da también en las medidas de rendimiento, que muchas veces sólo necesitan cierta información estándar de los algoritmos o de los problemas, como el número de soluciones de la población del algoritmo, el valor del óptimo, o el momento en que se produjo el último cambio, sim importar el algoritmo o problemas particulares que se estén utilizando en ese momento.

Estas circunstancias han propiciado **la creación de un framework para DOPs con diferentes algoritmos, problemas, propiedades dinámicas y medidas**, que hemos ido desarrollando y mejorando a lo largo de toda esta investigación. Este framework ha tomado ideas de una publicación anterior del autor [191], donde ya se diseñó una jerarquía de clases para metaheurísticas en problemas de optimización estáticos. El framework permite intercambiar diferentes problemas, utilizar múltiples algoritmos, usar diferentes medidas de rendimiento, y hasta visualizar resultados en tiempo de ejecución. Además, permite también la introducción de incertidumbre estocástica en diferentes puntos de la evaluación de una solución, que aunque no es el tema central de esta tesis, sí que es relevante para los objetivos del proyecto de investigación en el que se enmarca, y ya ha resultado de utilidad en otros trabajos en curso.

Chapter 1

Introduction

We live in a moment in history in which our social, cultural and technological necessities are strongly interrelated. It is no longer possible to think in solving any of these necessities without addressing some of the others, even in an indirect way. A computer application that does not consider the sociocultural factors of the users it targets, is doomed to fail. But at the same time, and here is where the enormous growing potential that we are experimenting resides, thanks to the recent technological advances, we can solve necessities that were not possible before, or do it in ways that were unthinkable until very recently, opening a world of opportunities.

In this context, many situations can be modeled as optimization problems, where the objective is to find the best possible solution that satisfies certain criteria, within the available resources. Let's think, for example, in the best way of sorting a set of results in an internet search according to the introduced query and the user preferences. Or the optimal way of serving a purchase to a set of customers in a city. Or how to explore the friendship graph of a person in order to suggest him new contacts as like-minded as possible. This field, optimization, classic in academical research, has historically focused in static, well-defined problems.

However, we are moving towards an increasingly integrated, interrelated, and globalized world. The problems and necessities to be solved are no longer something well-defined and bounded, but in many cases they imply dependencies between very different and complex elements. These dependencies, which can be unknown, may trigger very fast changes that must be addressed, even without all the information being available. The challenges we face therefore imply that we must be able to adapt to changing environments (moreover, that change at an increasingly fast rate), and to operate with incomplete, fuzzy, or sometimes even contradictory information.

Think about, e.g., the previous example about an internet search. It is obvious that user preferences are not going to remain static, and a useful recommendation today, could no be so tomorrow because the person has shifted his interests. The system must be able to adapt to this change. Or we may want to suggest contacts in a social network to someone who has not filled-in all the data of his profile; this absence of information should not prevent us from suggesting him contacts using the details that we *do* know. Examples of similar situations can be found in other relevant areas (Economics, Meteorology, Logistics and Transportation, Biochemistry, Telecommunications, etc.), and in all of them, systems must not only be able to manage these problems, but to provide solutions with the highest possible quality.

Given this situation, we need Intelligent Systems capable of dealing with the problems that arise due to presence of *dynamism* and *uncertainty*.

1.1. Context of the thesis

In the context of Intelligent Systems, an important class of problems are those known under the name of *optimization problems*, usually associated to finding the maximum or the minimum value that a certain function can take among some previously specified domain. Everything related to these problems is framed within the doctrinal body denoted Mathematical Programming, which includes an enormous variety of situations, depending on whether we are considering linear cases, non-linear cases, randomness, a single or multiple decision-makers, etc.

Dynamic Optimization Problems (DOPs) is a category of optimization problems that groups those in which the objective function, the constraints, the variables, the environment conditions and/or the structure of the problem itself may change while it is being solved. The basic formulation of these type of problems is as follows:

$$DOP = \left\{ \begin{array}{l} \text{Optimize } f(x,t) \\ \text{s.t. } x \in F(t) \subseteq S,\ t \in T \end{array} \right\} \tag{1.1}$$

where

- $S \in \mathbb{R}^n$, S is the search space.
- t is the time (it may also be measured in terms of *objective function evaluations*).
- $f : S \times T \to \mathbb{R}$, is the objective function that assigns a numerical value to each possible solution $x \in S$ at time t.
- $F(t)$, is the set of feasible solutions $x \in F(t) \subseteq S$ at time t.

In many occasions, finding the best possible solution of an optimization problem is a complex task. There can be multiple causes for this, like, e.g., that finding this optimal solution in a reasonable time cannot be guaranteed because of the computational complexity of the problem (e.g., NP-hard problems), or that evaluating a solution is very expensive, either in economical or temporal terms. The presence of dynamism in these problems implies another extra difficulty when trying to solve them. **In this context, the techniques provided by Soft Computing emerge as ideal candidates for facing these problems**.

In 1965, Lofti A. Zadeh introduced in [189] the concept of *fuzzy set*, allowing an element to be member of a set in a gradual way, and not in an absolute way as stablished by the classic set theory. In other words, in fuzzy sets theory, membership functions were allowed to take values in the $[0, 1]$ interval, instead of the classical $(0, 1)$ set. Since then, applications and developments based in this simple concept have evolved extraordinarily in all knowledge fields, giving theoretical fundament, structure and contents to a new area denominated Soft Computing, which has become the founding seed of modern Intelligent Systems.

Zadeh himself proposed the first definition of Soft Computing in 1994 [188], although in [190] he claimed that this idea dates back to 1990. The definition proposed by Zadeh was the following:

> Basically, soft computing is not a homogeneous body of concepts and techniques. Rather, it is a partnership of distinct methods that in one way or another conform to its guiding principle. At this juncture, the dominant aim of soft computing is to exploit the tolerance for imprecision and uncertainty to achieve tractability, robustness, and low solution cost. The principal constituents of soft computing are *fuzzy logic*, *neurocomputing*, and *probabilistic reasoning*, with the latter subsuming genetic algorithms, belief networks, chaotic systems, and parts of learning theory. In the partnership of fuzzy logic, neurocomputing and probabilistic reasoning, fuzzy logic is mainly concerned with imprecision and approximate reasoning; neurocomputing with learning and curve-fitting; and probabilistic reasoning with uncertainty and belief propagation.

As it usually happens in emerging areas, this first definition served the purpose of categorizing a concept that was beginning to take shape, although this definition was not too precise. It was rather an attempt of grouping several techniques and concepts that aimed at dealing with the inherent uncertainty and imprecision of real world problems.

This definition has evolved with years, and in 2008, Verdegay, Yager and Bonissone [161] proposed a more precise and illustrative definition of what is Soft Computing currently:

> The viewpoint that we will consider here (and which we will adopt in the future) is another way of defining soft computing, whereby it is considered to be the antithesis of what we might call hard computing. This viewpoint is consistent with the one in [188, 190]. Soft computing could therefore be seen as a series of techniques and methods so that real practical situations could be dealt with in the same way as humans deal with them, i.e. on the basis of intelligence, common sense, consideration of analogies, approaches, etc. In this sense, soft computing is a family of problem-resolution methods headed by *approximate reasoning* and *functional and optimization approximation methods*, including search methods. Soft computing is therefore in the theoretical basis for the area of intelligent systems.

Verdegay et al. decompose Soft Computing into two big groups of problem-solving methods: approximate reasoning, and functional approximation and randomized search. In that same work, they introduce a second level of decomposition, compatible with the categories initially proposed by Zadeh in [188]. With the perspective that time gives, it has been seen that some of these components are more important than others, with four of them clearly standing out: probabilistic reasoning, fuzzy logic and fuzzy sets, neural networks and genetic algorithms (GA), with GA being eventually superseded by the more generic class of *evolutionary algorithms* (EA).

The authors also show that the combination of these second-level componentes produces other emergent research areas that, by extension, are also grouped under the definition of Soft Computing. Some examples of these combinations are: hybrid probabilistic models, fuzzy event belief models, fuzzy neural systems, fuzzy logic-based controllers adjusted with EA, neural systems with fuzzy-controlled parameters, fuzzy genetic systems, etc. A more detailed analysis of these combinations can be seen in [17, 18].

Finally, Verdegay et al. deepen in the role that EAs play in the context of Soft Computing as search-based optimization methods. It is easy to observe that these algorithms belong to the family of *metaheuristics*, a class of methods that act on the basis that "satisfaction is better than optimization". In this sense, metaheuristics may not always find optimal solutions, but they usually return solutions that largely satisfy the decision-maker's expectations, both in terms of quality and computational time. The authors end up justifying the inclusion of metaheuristics as the fourth second-level component of Soft Computing, replacing EAs, since metaheuristics encompass a wider and more generic set of algorithms that fulfill the same purpose than EAs. This last component, metaheuristics, is a key concept in this thesis, and we will discuss it with more detail. Figure 1.1 shows a diagram that summarizes the main components of Soft Computing according

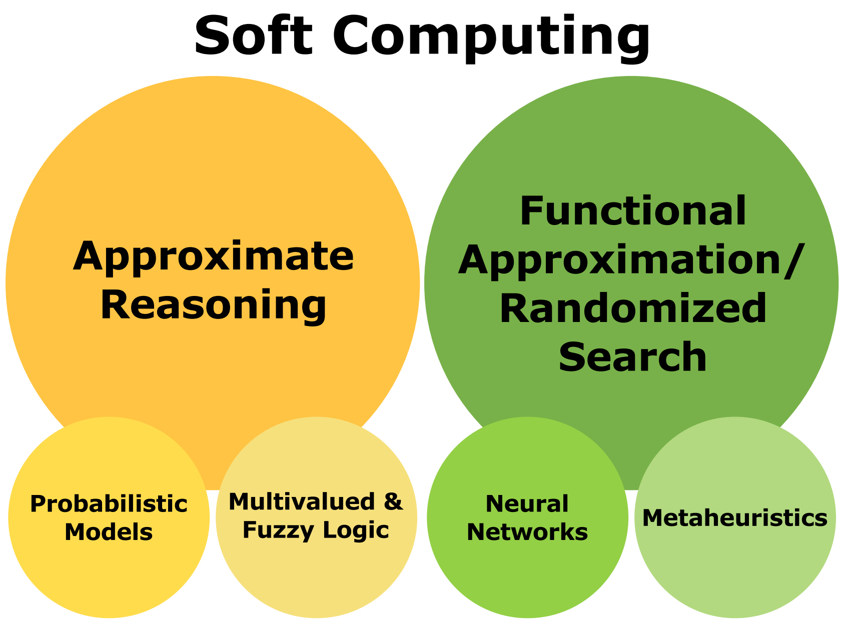


Figure 1.1 – Main components of Soft Computing, as defined in [161].

to [161].

The term "metaheuristic" appeared for the first time in a work by Glover in 1986 [65], and results from the addition of the prefix *meta* (meaning "beyond" or "of a higher level") to the word *heuristic* (from the Greek word "heuriskein" — "to discover" or "to find out" [161]). Heuristics are search-based optimization methods capable of finding high-quality solutions at a reasonable computational cost, eventhough their optimality or feasibility are not guaranteed. Typically, heuristics are considered to be opposed to *exact methods*, since these latter are required to provide optimality and feasibility. In this sense, metaheuristics arose as an attempt of being "above the heuristics", with the idea of extracting the best parts of different successful heuristics to create generic methods that could be applied to a wider variety of problems and concepts, while maintaining the "high-quality solutions", "reasonable computational cost" properties.

Although there is no widely accepted formal definition of the term metaheuristic, we can get a representation of its general notion from the following two proposals:

- Osman and Laporte [122] :"An iterative generation process which guides a subordinate heuristic by combining intelligently different concepts for exploring and exploiting the search space"
- Voss et al. [162] : "An iterative master process that guides and modifies

> the operations of subordinate heuristics to efficiently produce high quality solutions. It may manipulate a complete (or incomplete) single solution or a collection of solutions at each iteration. The subordinate heuristics may be high (or low) level procedures, or a simple local search, or just a construction method"

There is an enormous number of metaheuristics with different degrees of complexity and optimization capabilities: Local Search, Multi-Start Local Search, Simulated Annealing, Tabu Search, Scatter Search, Greedy Randomize Adaptive Search Procedure (GRASP), Variable Neighbourhood Search (VNS), Evolutionary Algorithms (EA), Ant Colony Optimization (ACO), Particle Swarm Optimization (PSO), etc. Each of these metaheuristics has given birth to multiple variations, and hybrid metaheuristics are commonly created by combining two or more of them, in a master-subordinate fashion inherent to its definition.

All these metaheuristics are based on different paradigms and possess unique features that distinguish them from the others. However, in general, all of them are ultimately based (with the possible exception of the simple Local Search) on carefully callibrated combinations of *diversification* and *intensification* processes.

The combination of these two factors make them specially well-suited for solving global optimization problems. However, and this is the specially relevant part for this thesis, these very same characteristics confer them an extraordinary capability for adaptation, while maintaining at the same time a high search potential. For all these reasons, metaheuristics can be considered as ideal candidates for solving optimization problems with *dynamism* and *uncertainty.*

Addressing the challenges posed by this type of problems and providing methods capable of dealing with them is the central issue of the research project *P07-TIC-02970* of the Andalusian Government, entitled *"Uncertain and Dynamic Optimization Models in Intelligent Systems: A Study on Solving Strategies and Applications"*, which is the framework of this doctoral thesis.

The core of the project is structured around the possible scenarios that may arise when combining certain components of the problems — basically, we are talking about the objective function and constraints —, with dynamism and uncertainty. We can see a graphical representation of these combinations in Fig. 1.2, where each cell is interpreted as an scenario associated with a kind of problem. In these cells, the complexity, expressed as a grayscale, increases from top to bottom, and from left to right.

Within the framework of this research project, **this thesis focuses on one of the scenarios studied: optimization problems with dynamism in the objective function** (Fig. 1.3).

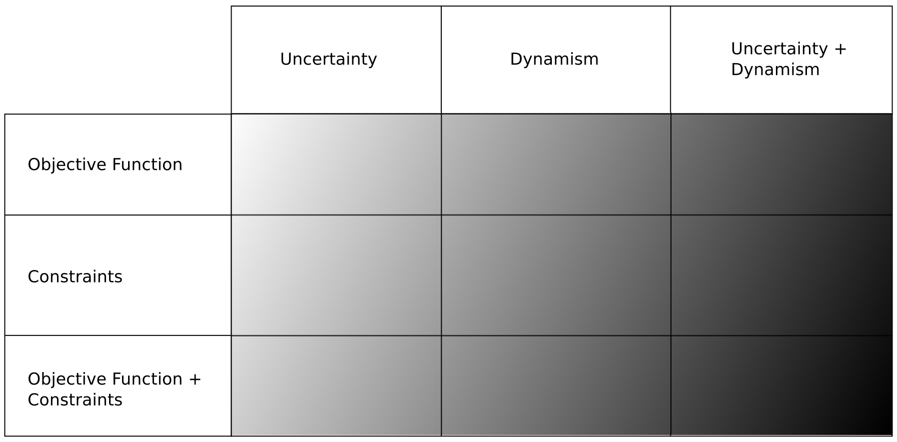


Figure 1.2 – Possible combinations of dynamism and uncertainty with the elements of an optimization problem, i.e., objective function and constraints

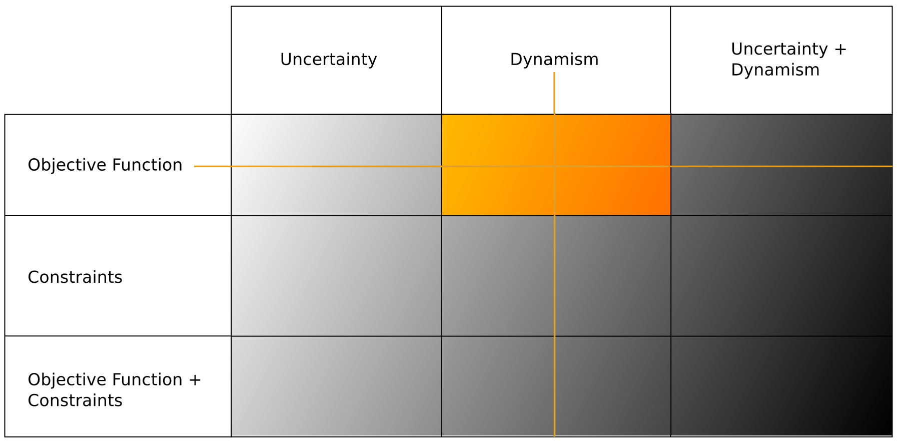


Figure 1.3 – The main working scenario of this thesis is optimization problems with dynamism in the objective function

1.2. Objectives

Considering the importance of DOPs, the necessity of solving them, and the expected suitability of Soft Computing techniques to do so — specially Metaheuristics —, **the main objective of this thesis is to study, design and implement Soft Computing based methods to solve DOPs.** In order to accomplish this objective, we performed a set of tasks aimed at answering the following research questions:

1. **Is it possible to improve the existent algorithms, and, if possible, what techniques can be used for that purpose?** In order to answer this question it will be necessary to study the state-of-the-art algorithms that are capable of operating in the presence of dynamism, and investigate how they work and their constituent elements. We will put a special interest in studying what role does cooperation among these elements play at improving the effectiveness of an algorithm. As a secondary objective when answering this question, we will search for improving techniques that are not exclusive of certain algorithms, but that can be applied to most of them in a generic way.

2. **What methodology should be applied in order to compare the performance of different algorithms on a DOP?** The presence of dynamism in the objective function makes it necessary to redefine how we evaluate the performance of an algorithm. It is not enough to find a solution to the problem once: since the problem is continuously changing, the solver must periodically report a solution for the current environment. Intuitively, an algorithm that returns reasonably good solutions in a sustainable fashion is better than one that finds the optimal solution once, but obtains poor solutions the rest of the time. With this way of evaluating the performance, comparing results turns into a complex task, and in order to answer the question of this item, it is necessary to search for novel ways of presenting these results so that the researcher can fully understand their meaning and implications.

3. **What difficulties can we find when implementing these DOPs, algorithms and performance measures, and how can we face them?** This question, more technical and application-oriented than the others, is based on the experience gathered over the research, where we have learnt that going from theory to practice is not always easy, nor direct. The implementation of algorithms, problems and performance measures present some issues by its own. The answer to this question pretends to be a compendium of such experience and the conclusions obtained.

The thesis is structured as follows:

- Chapter 2 presents the research background for this thesis, with a brief review of the literature on DOPs, focusing on the problems, algorithms and performance measures most relevant for the objectives of the thesis.
- Chapter 3 contains the most significant algorithmic contributions of this thesis, with either improvements to existent algorithms of the literature, or directly new algorithms that were developed during the research. This chapter is mainly focused on looking for answers to the first question presented in the objectives.
- In Chapter 4 we introduce a novel technique for comparing large amounts of data in a comprehensive and meaningful manner. This technique, named SRCS, has allowed to extract very valuable information regarding general trends of algorithms, as well as recommendations on which algorithm to use in which situation. This chapter is mainly focused on looking for answers to the second question presented in the objectives.
- Chapter 5 explores some of the difficulties and problems we faced during the implementation of the experiments, and explains how we solved them. This chapter is mostly concerned with software issues and the experience gathered during the research process, therefore focused on the third question presented in the objectives.
- Finally, Chapter 6 presents the conclusions obtained from this thesis, the most relevant knowledge acquired during the experimentation, and answers the questions proposed in the objectives. This chapter also includes a summary of the most relevant publications obtained during the realization of this thesis.

Chapter 2

Background

With the purpose of facing the objectives of this thesis with an appropriate perspective, in this chapter we will present a review of the different problems, performance measures and algorithms most commonly used in the literature on Dynamic Optimization.

As far as possible, we will try to offer a wide enough vision in order to be able of understanding what is the current state of the art, the most used problems and performance measures in the literature benchmarks, and the algorithms with highest popularity. We will also introduce in more detail those problems, performance measures and algorithmic families which are more relevant for this thesis.

However, this chapter does not pretend to be an exhaustive review of the literature. The topic of dynamic optimization has been already reviewed in the past, and the interested reader is referred to the books by Branke [23], Weicker [166], Morrison [115], Yang [182], Jin and Branke [84], and the PhD thesis of Nguyen [118]. These works were mainly related to Evolutionary Optimization. Some special issues were devoted to the dynamic optimization topic considering other kind of methods, like [24, 25, 181]. More recently, Cruz, Gonzalez and Pelta published a review [34] especially attuned with the objectives of this thesis.

2.1. Problems

Although there is a great variety of works on DOPs, most of them are focused on synthetic problems, obtained by means of mathematical objective-functions with artificially added dynamism. Among the smaller set of real-world applications of Dynamic Optimization, some of the most interesting examples include: solving aerospace design problems [101], path planning for ships [109], financial optimization [150], applied Dynamic Vehicle Routing Problems [74, 123], evolutionary online data mining [36], predicting the position and orientation of moving objects [139], optimization of salting truck routes [73], optimal design of an elastic structure [117], control energy consumption and quality of service aspects on Wireless Sensor Networks [129], tackle dynamic shortest path routing on mobile ad hoc networks [179], etc.

However, these works, apart from being fewer than the synthetic ones, are usually very specific, so the problem formulation and the applied methods are, typically, non-generalizable. The objectives of this thesis include the design of wide-range improving strategies and comparison techniques, and therefore, in order to test them appropriately, it is necessary to use a controlled and reproducible environment. Thus, **the experiments performed in this thesis have focused on synthetic problems**.

Many of the synthetic problems used in the literature can be qualified as DOPs *generators*. These generators start from a base fitness function, and then they apply transformations to the scenario according to some predefined parameters of the problem. These transformations include shifting the origin of coordinates through the search space, changing the "height" of the fitness functions used, dimensionality variations, XOR'ing the target solution with a bit-mask, etc. Additionally, many features related to dynamism itself can also be controlled. Depending on the corresponding parameter settings, it is usually possible to produce instances of the scenario with a certain severity (i.e., magnitude of the change), change frequency (i.e., how often changes are produced), etc.

A very usual way of producing this type of dynamic scenarios is by reusing classic *static* optimization functions and adding them dynamism by means of periodic transformations of the environment. Among this group we can distinguish between *continuous* and *discrete* DOPs. We will now review some of the most important ones in each category.

2.1.1. Continuous DOPs

2.1.1.1. Moving Peaks Benchmark (MPB)

The MPB is a test benchmark for DOP's originally proposed in [22]. It is a *maximization* problem consisting in the superposition of **m** peaks, each one characterized by its own height (**h**), width (**w**), and location of its centre (**p**). The fitness function of the MPB is defined as follows:

$$\mathbf{MPB}(\mathbf{x}) = \max_j \left(h^j - w^j \sqrt{\sum_{i=1}^{n} (x_i - p_i^j)^2} \right), \ j = 1, .., m \tag{2.1}$$

where n is the dimensionality of the problem. The highest point of each peak corresponds to its centre, and therefore, the global optimum is the centre of the peak with the highest parameter **h**.

Dynamism is introduced in the MPB by periodically changing the parameters of each peak j after a certain number of function evaluations (ω):

$$\mathbf{h}_j(t+1) = \mathbf{h}_j(t) + \mathbf{h_s} \cdot \mathbf{N}(0,1) \tag{2.2}$$

$$\mathbf{w}_j(t+1) = \mathbf{w}_j(t) + \mathbf{w_s} \cdot \mathbf{N}(0,1) \tag{2.3}$$

$$\mathbf{p}_j(t+1) = \mathbf{p}_j(t) + \mathbf{v}_j(t+1) \tag{2.4}$$

Changes to both width and height parameters depend on a given severity for each of them ($\mathbf{w_s}$ and $\mathbf{h_s}$). Changes to the centre position depend on a shift vector $\mathbf{v}_j(t+1)$, which is a linear combination of a random vector $\mathbf{r}$ and the previous shift vector $\mathbf{v}_j(t)$ for the peak, normalized to length $\mathbf{s}$ (position severity, shift distance, or simply *severity*):

$$\mathbf{v}_j(t+1) = \frac{\mathbf{s}}{|\mathbf{r} + \mathbf{v}_j(t)|}((1-\lambda)\mathbf{r} + \lambda \mathbf{v}_j(t)) \tag{2.5}$$

The random vector $\mathbf{r}$ is created by drawing random numbers for each dimension and normalizing its length to $\mathbf{s}$. Finally, parameter λ indicates the linear correlation with respect to the previous shift, where a value of 1 indicates "total correlation" and a value of 0 "pure randomness".

One of the most used configurations of the MPB is the Scenario 2, described in the web site of the MPB [1], and consisting on the set of parameters indicated in Table 2.1.

The MPB is one of the most versatile problems in the continuous DOP literature, since with it, it is possible to control in very precise way many aspects of the problem (e.g., the number of local optima, the position of the peaks, their height, etc.), as well as the dynamism (changes in the position of the peaks: linear, random, mixed; linearly increasing/decreasing changes in the height and width, change frequency, etc.). Thus, the MPB is among the most used benchmarks (see, e.g., [12, 14, 16, 26, 28, 52, 83, 88, 124, 138, 157, 193]; for a more detailed list, please check Table 2.2 at the end of this chapter).

Additionally, Morrison developed in an independent way a generator for continuous DOPs named **DF1** [113, 115], with a virtually equivalent functionality to that of the MPB. The DF1 is also based in the composition of *cone*-like functions that can independently vary their position, height and width. This generator has also been used in many works that we include in this section given the similarity between the DF1 and the MPB (see [51, 54, 55, 115, 128, 140]).

[1] `http://people.aifb.kit.edu/jbr/MovPeaks/`

Parameter	**Value**
Number of peaks (m)	$\in [10, 200]$
Number of dimensions (d)	5
Peaks heights (h_i)	$\in [30, 70]$
Peaks widths (w_i)	$\in [1, 12]$
Change frequency (ω)	5000
Height severity (h_s)	7.0
Width severity (w_s)	1.0
Shift distance (s)	$\in [0.0, 3.0]$
Correlation coefficient (λ)	$\in [0.0, 1.0]$

Table 2.1 – Standard settings for the Scenario 2 of the Moving Peaks Benchmark

2.1.1.2. The Ackley Function

The *Ackley* function is a standard static-optimization function, defined by the following equation:

$$\mathbf{Ackley}(\mathbf{x}) = -20\exp\left(-0.2\sqrt{\frac{1}{n}\sum_{i=1}^{n} x_i^2}\right) - \exp\left(\frac{1}{n}\sum_{i=1}^{n}\cos(2\pi x_i)\right) + 20 + e \tag{2.6}$$

$$x \in [-32, 32], \qquad x^* = 0^n$$

where n is the number of dimensions. This function is highly multi-modal, with its optimum at the 0^n point (it's a *minimization* problem).

When this function is used for producing dynamic environments, the dynamism is usually achieved by means of shifting the origin of coordinates. The shifts can be:

- *linear*: the origin of coordinates $\mathbf{o}$ is added a certain shift s.

$$\mathbf{o}(t+1) = \mathbf{o}(t) + s \tag{2.7}$$

- *random*: the origin of coordinates $\mathbf{o}$ is added a shift value randomly chosen in the interval $[-s, s]$.

$$\mathbf{o}(t+1) = \mathbf{o}(t) + s \cdot N(-1, 1) \tag{2.8}$$

- *semi-random*: the origin of coordinates $\mathbf{o}$ is added a shift that goes from totally random to totally linear (e.g., like in the MPB); parameter values have the same meaning as the MPB (i.e., r is a random vector normalize to length s, $\lambda \in [0,1]$ indicates randomness, where 1 means totally random and 0 means totally linear).

$$\mathbf{o}(t+1) = \frac{s}{|r + \mathbf{o}(t)|}((1-\lambda)r + \lambda\mathbf{o}(t)) \tag{2.9}$$

- *other* (e.g., circular, chaotic, etc.)

Given the shifted origin of coordinates $\mathbf{o}(t)$, the dynamic version of the Ackley function is constructed as:

$$\mathbf{ackley}_{dyn}(x,t) = \mathbf{ackley}(x - \mathbf{o}(t)) \tag{2.10}$$

This method of introducing dynamism into a static optimization function is simple, efficient, and particularly convenient. The rest of the functions presented here for continuous DOPs use this mechanism.

The Ackley function has been used in several works in the DOP literature, like, e.g., [141,142].

2.1.1.3. The Griewank Function

The *Griewank* function is a standard static-optimization function, defined by the following equation:

$$\begin{aligned} &\mathbf{Griewank}(\mathbf{x}) = \sum_{i=1}^{n} \frac{x_i^2}{4000} - \prod_{i=1}^{n} \cos\left(\frac{x_i}{\sqrt{i}}\right) + 1 \\ &x \in [-50, 50], \qquad x^* = 0^n \end{aligned} \tag{2.11}$$

where n is the number of dimensions. This function is highly multi-modal, with its optimum at the 0^n point (it's a *minimization* problem).

Dynamism is usually introduced into this function by means of the procedure explained for the Ackley function, using equations 2.7, 2.8, 2.9 and 2.10.

Works using the Griewank function include [121,143,160,169].

2.1.1.4. The Rastrigin Function

The *Rastrigin* function is a standard static-optimization function, defined by the following equation:

$$\mathbf{Rastrigin}(\mathbf{x}) = \sum_{i=1}^{n} \left(x_i^2 - 10\cos(2\pi x_i) + 10\right) \tag{2.12}$$
$$x \in [-5, 5], \qquad x^* = 0^n$$

where n is the number of dimensions. This function is highly multi-modal, with its optimum at the 0^n point (it's a *minimization* problem).

Dynamism is usually introduced into this function by means of the procedure explained for the Ackley function, using equations 2.7, 2.8, 2.9 and 2.10.

The Rastrigin function is rather popular in the DOP literature, with examples including [45, 121, 141–143, 154, 160].

2.1.1.5. The Rosenbrock Function

The *Rosenbrock* function is a standard static-optimization function, defined by the following equation:

$$\mathbf{Rosenbrock}(\mathbf{x}) = \sum_{i=1}^{n-1} \left(100(x_i^2 - x_{i+1})^2 + (x_i - 1)^2\right) \tag{2.13}$$
$$x \in [-10, 10], \qquad x^* = 0^n$$

where n is the number of dimensions. The optimum of the function is located at the 0^n point (it's a *minimization* problem).

Dynamism is usually introduced into this function by means of the procedure explained for the Ackley function, using equations 2.7, 2.8, 2.9 and 2.10.

The Rosenbrock function has been used in works like [82, 121, 143, 160].

2.1.1.6. The Sphere Function

The *Sphere* function (also referred to as the Moving Parabola problem in the DOP literature) is a standard static-optimization function, defined by the following equation:

$$\mathbf{Sphere}(\mathbf{x}) = \sum_{i=1}^{n-1} x_i^2 \tag{2.14}$$
$$x \in [-100, 100], \qquad x^* = 0^n$$

where n is the number of dimensions. The optimum of the function is located at the 0^n point (it's a *minimization* problem).

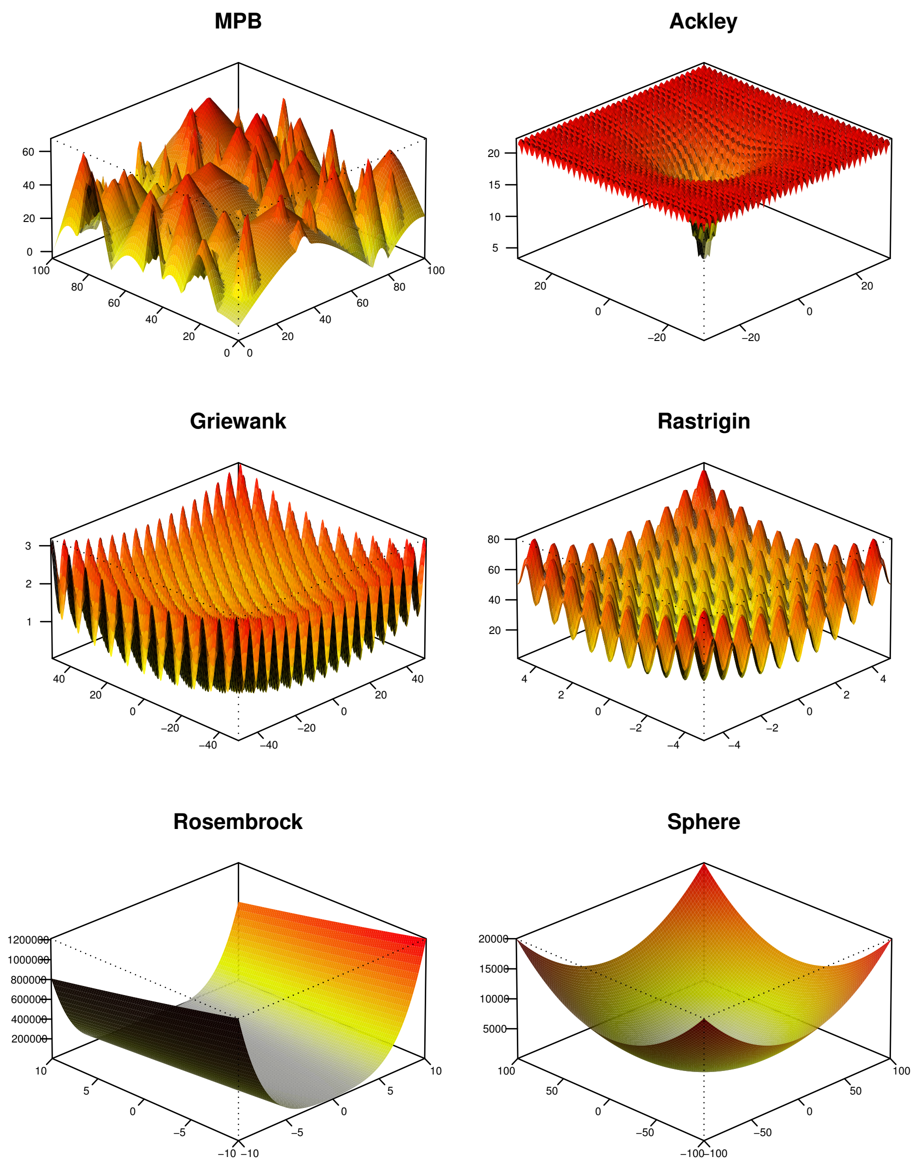


Figure 2.1 – A 3D static snapshot of the MPB, Ackley, Griewank, Rastrigin, Rosenbrock and Sphere functions, used in *continuous* DOPs.

Dynamism is usually introduced into this function by means of the procedure explained for the Ackley function, using equations 2.7, 2.8, 2.9 and 2.10.

Examples of the use of the Sphere function include [3,4,21,121,141,142]. Also, we can also find works that use this same function under the name of Moving Parabola, like [1,46,82].

2.1.2. Discrete DOPs

2.1.2.1. The Dynamic Knapsack Problem

The Knapsack Problem is a well-known problem in static combinatorial optimization. In this problem, the objective is to fill a knapsack with objects, each with a different profit value, so that the accumulated profit is maximized.

In the dynamic version, there are several ways of introducing dynamism:

- periodically changing the profit of the objects (i.e., dynamism in the objective function):

$$\mathbf{Knapsack}(\mathbf{x}, \mathbf{t}) = \sum_{i=1}^{n} x_i \mathbf{v_i}(\mathbf{t}) \quad \text{subject to} \sum_{i=1}^{n} x_i w_i < W \tag{2.15}$$

- periodically changing the weights of the objects (i.e., dynamism in the constraints):

$$\mathbf{Knapsack}(\mathbf{x}, \mathbf{t}) = \sum_{i=1}^{n} x_i v_i \quad \text{subject to} \sum_{i=1}^{n} x_i \mathbf{w_i}(\mathbf{t}) < W \tag{2.16}$$

- periodically changing the maximum capacity of the knapsack (i.e., dynamism in the constraints):

$$\mathbf{Knapsack}(\mathbf{x}, \mathbf{t}) = \sum_{i=1}^{n} x_i v_i \quad \text{subject to} \sum_{i=1}^{n} x_i w_i < \mathbf{W}(\mathbf{t}) \tag{2.17}$$

- any of the above:

$$\mathbf{Knapsack}(\mathbf{x}, \mathbf{t}) = \sum_{i=1}^{n} x_i \mathbf{v_i}(\mathbf{t}) \quad \text{subject to} \sum_{i=1}^{n} x_i \mathbf{w_i}(\mathbf{t}) < \mathbf{W}(\mathbf{t}) \tag{2.18}$$

Works based on the Dynamic Knapsack problem include [27,84,86,137,144,164,178,183,184].

2.1.2.2. The Dynamic Vehicle Routing Problem (DVRP)

The Vehicle Routing Problem (VRP) consists in designing the optimal set of routes for a fleet of vehicles in order to serve a given set of customers.

One of the most studied versions of VRP is Capacitated VRP (CVRP), where all vehicles have the same capacity. The fitness of a solution is defined as the sum of the costs of all its routes:

$$\mathbf{VRP}(\mathbf{x}) = \sum_{i=1}^{m} Cost(R_i) \tag{2.19}$$

$$Cost(R_i) = \sum_{j=1}^{n} c^i_{j,j+1} + \sum_{j=1}^{n} \delta^i_{j,j+1} \tag{2.20}$$

where R_i is the i-th route, $c^i_{j,j+1}$ is the cost (distance) between node j and node $j+1$ of route i, and $\delta^i_{j,j+1}$ is the service time needed to unload the vehicle.

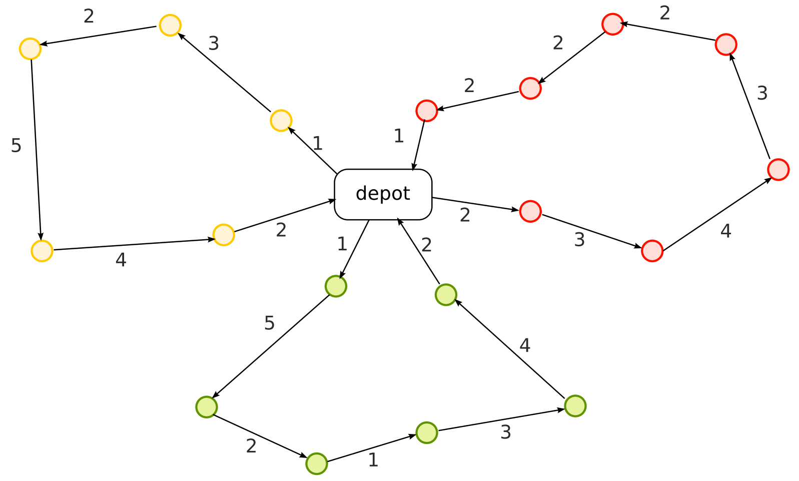


Figure 2.2 – A representation of a solution (set of routes) for the Vehicle Routing Problem (VRP).

The VRP is subject to the constraints that each vehicle cannot exceed a certain maximum load L, and the distance covered by each route cannot exceed a certain distance limit D:

$$\sum_{v_j \in R_i} d_j \leq L \tag{2.21}$$

$$Cost(R_i) \leq D \tag{2.22}$$

Dynamic VRP assumes that some kind of changes can affect the problem while the algorithm is trying to solve it. These changes may involve incidences like a car breakdown and traffic jams, as well as new order placements, unexpected withdrawal of orders, etc. Typically, this is reflected as dynamic changes in the constraints, where the maximum vehicle's load and route's distance may vary with time:

$$\sum_{v_j \in R_i} d_j \leq L(t) \tag{2.23}$$

$$Cost(R_i) \leq D(t) \tag{2.24}$$

Works with the DOP literature using the DVRP include [74, 110, 123].

2.1.2.3. XOR-based Dynamic Problems

XOR-based Dynamic Problems are problems where the objective is to match a target bit string that is periodically changed by means of exclusive-or (XOR) operations. The fitness function used for evaluating the matching is usually based on static functions, like One-Max, Plateau, Royal Road or Deceptive functions (see Fig. 2.3 for a visualization of the fitness assigned by each of these functions in a 4-bit matching case).

The way the dynamism is included into a stationary problem is well exemplified in Yang [173, 174]. The idea is to depart from a binary-encoded stationary function $f(\vec{x})$ ($\vec{x} \in \{0,1\}^l$) and use a bitwise XOR operator in the fitness calculation. Assuming that the environment changes every τ generations, for each environmental period k the XOR-ing mask $\vec{M}(k)$ is incrementally generated as follows:

$$\vec{M}(k) = \vec{M}(k-1) \oplus \vec{T}(k)$$

where $\oplus$ is the bitwise exclusive-or (XOR) operator (i.e. $1 \oplus 1 = 0$, $1 \oplus 0 = 1$, $0 \oplus 0 = 0$) and $\vec{T}(k)$ is an intermediary binary template randomly created with $\rho \times l$ ones for environmental period k (cyclic or cyclic with noise changes were also considered). For the first period $k = 1$, $\vec{M}(1)$ is set to a zero vector. Then, the population at generation t is evaluated with the formula:

$$f(\vec{x}, t) = f(\vec{x} \oplus \vec{M}(k))$$

where $k = \lceil t/\rho \rceil$ is the environmental period index. With the XOR generator defined in this way, the parameter τ controls the speed of change while $\rho \in (0.0, 1.0)$ controls the severity of environmental changes. Bigger values of ρ mean more severe environmental changes. The above procedure allows to change the fitness

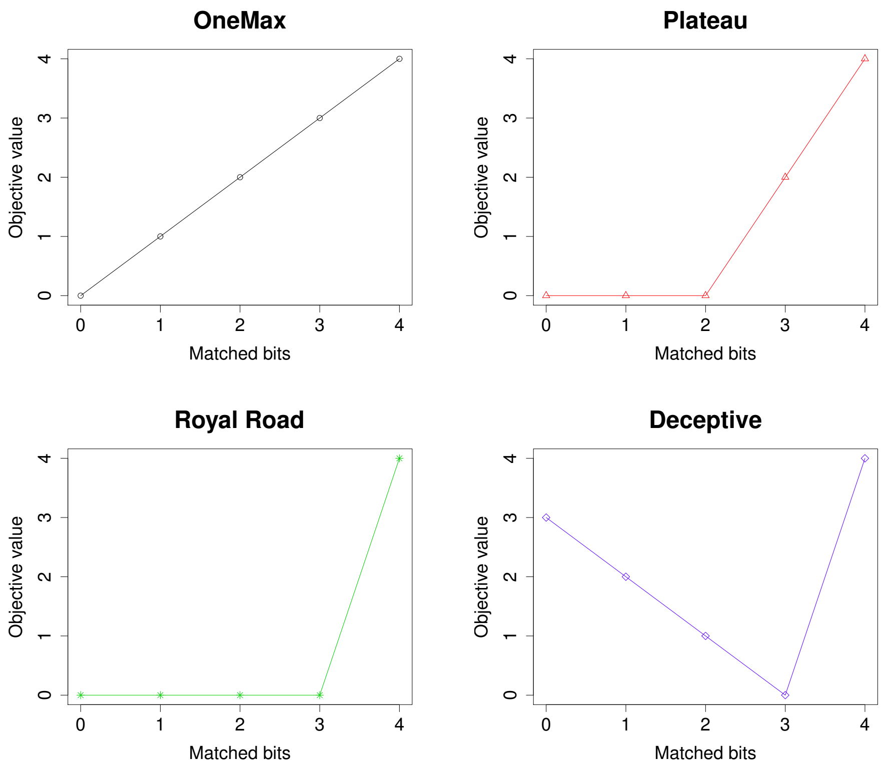


Figure 2.3 – A representation of the OneMax, Plateau, Royal Road and Deceptive functions for a 4-bit-matching problem, used in *discrete* DOPs.

landscape while keeping certain properties of the original landscape, like the total number of optima and their values despite their locations are shifted.

Works using bit-string matching problems may use dynamic versions of the One-Max function [44,58,163,173,174,177,178,183,185], the Plateau function [163, 175,178], the Royal Road function [58,59,151,163,164,173,177,183,184], Deceptive functions [151,163,164,173–175,177,183–185], the dynamic XOR-based generator described before [136,173,174,176], or other bit matching problems [84,111,147].

2.1.3. Problems chosen in the experiments of the thesis

The experiments carried out in this thesis have been performed both in discrete and continuous DOPs, although the main focus has been put in these last ones. In the case of *continuous* DOPs, most of the works that we will present have used the Moving Peaks Benchmark (MPB), and the dynamic versions of the Ackley, Griewank or Rastrigin problems (Fig. 2.1). The reasons for choosing the MPB are obvious since this is one of the most used benchmarks of the literature. The Ackley, Griewank and Rastrigin were added for having a broader test benchmark with a reasonable difficulty (these functions are widely used and highly multimodal). In the case of *discrete* DOPs, our experiments have used the OneMax, Plateau, Royal Road and Deceptive problems (Fig. 2.3), based on the XOR-generator technique explained in [173, 174].

2.2. Performance measures

The difference between DOPs and static problems is that in the formers there are changes in the environment during the optimization process. This changes may include variations in the position of the optima and their objective value, which can affect the performance measure.

Most of DOP performance measures use an approach based on averaging the performance of an algorithm over the static period that takes place between two consecutive changes, and then averaging again all the static measures obtained this way during the execution process. Fig. 2.4 shows a representation of this approach using different versions of the *error* as performance measure.

In some problems, the objective function's value of the optimum is known, and relative performance measures can be used, like, e.g., *Error* (the difference between the algorithm's best value so far and the optimum value), *Accuracy* (a sort of normalized error), etc. In other cases, either the optimum's value is unknown or the researchers have decided not to use it, so absolute measures, such as *Performance* (just the algorithm's best value) are needed.

Finally, all these measures relate to a single execution of the algorithm. In many cases, the methods used to solve DOPs are based in non-deterministic algorithms (such as, e.g., metaheuristics), that include a stochastic component in their execution. In order to guarantee that the obtained results are not produced by chance, it is customary to perform several executions of the same algorithm with different random seeds. This implies that when we measure the performance of an algorithm like this, we will not have one single value of the Offline Error or Offline Performance, but a sample of values. In order to compare performance measures between algorithms, it will be necessary to use statistical tests that assess whether the difference in the results is significant or not.

2.2.1. Offline Error

The Offline Error [28] measures the average of the error of the best solution found by the algorithm since the last change, for every function evaluation, and for all changes:

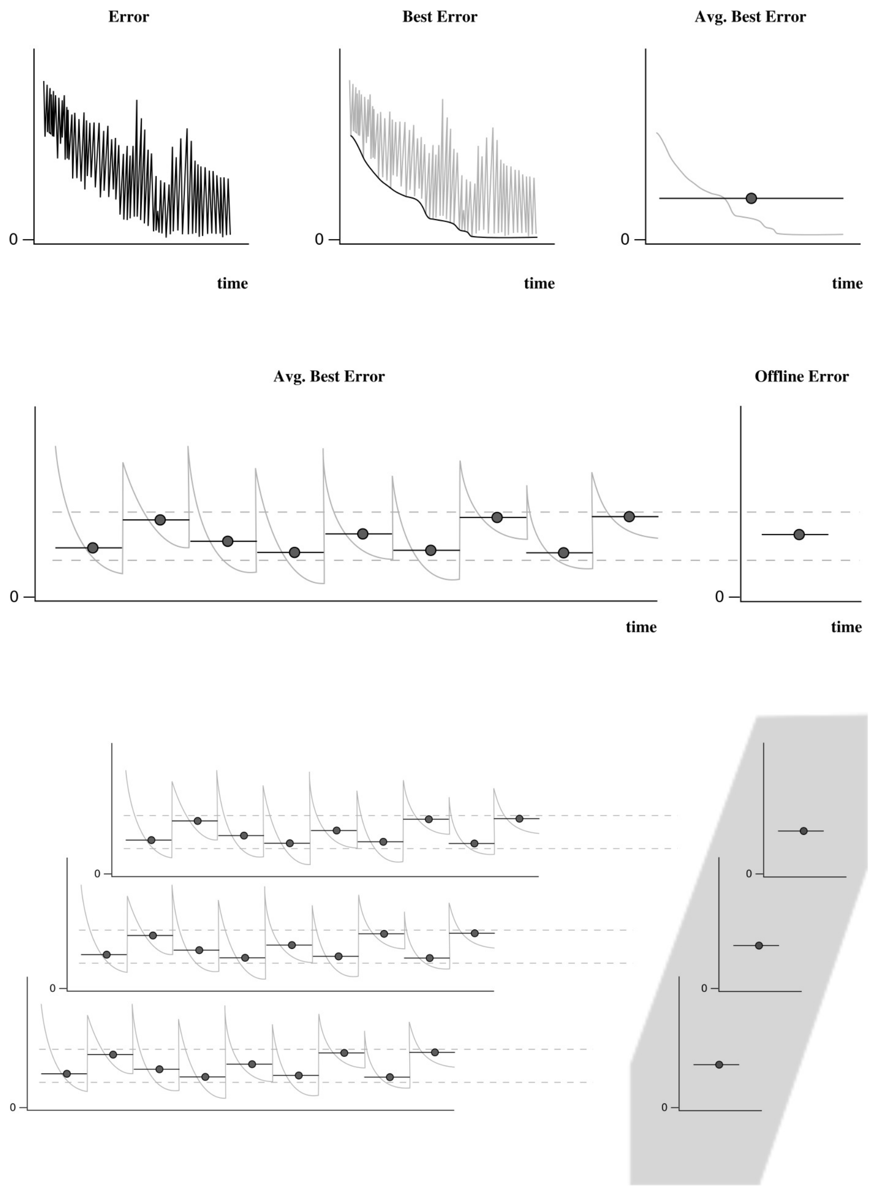


Figure 2.4 – Illustration of different performance measures of an algorithm in a DOP. The upper part shows, from left to right, the performance while the environment remains static, using different versions of *error*. The middle part shows an example of the algorithm performance over several consecutive changes in the environment using the *avg. best error* (the *best error* is displayed in the background). The sudden raisings of the *best error* values indicate a change in the environment, and the *offline error* is displayed at the right. Finally, in the lower part, several offline error measures are used, since most algorithms for DOPs have an stochastic component and it is needed to repeat their execution with different random seeds.

$$e_{off} = \frac{1}{N_c} \sum_{i=1}^{N_c} \frac{1}{N_e(i)} \sum_{j=1}^{N_e(i)} (f_i^* - f_{ij}) \tag{2.25}$$

where N_c is the total number of changes in the environment, $N_e(i)$ is the total number of evaluations allowed in the i-th change, f_i^* is the optimum value of the i-th change, and f_{ij} is the best value found by the algorithm since the beginning of the i-th change up to the j-th evaluation. If changes in the environment are produced at a fixed rate (i.e., $N_e(i) = N_e, \forall i$), the equation can be simplified as follows:

$$e_{off} = \frac{1}{N_c N_e} \sum_{i=1}^{N_c} \sum_{j=1}^{N_e} (f_i^* - f_{ij}) \tag{2.26}$$

This performance measure is one of the most used. Examples of works using this measure include [6, 13, 28, 45, 94, 96, 99, 100, 107, 116, 121, 125, 126, 155, 192].

2.2.2. Mean Fitness Error (MFE)

The Mean Fitness Error (MFE) was defined by Richter and Yang [134] as:

$$MFE = \left[\frac{1}{T} \sum_{t=1}^{T} \left(f(x_s(k), k) - \underset{x_j(t) \in P(t)}{max} f(x_j(t), k) \right) \right]_{k=\lfloor \gamma^{-1}(t) \rfloor} \tag{2.27}$$

where $\underset{x_j(t) \in P(t)}{max} f(x_j(t), \lfloor \gamma^{-1} t \rfloor)$ is the fitness value of the best-in-generation individual $x_j(t) \in P(t)$ at generation t, $f(x_s(\lfloor \gamma^{-1}(t) \rfloor), \lfloor \gamma^{-1}(t) \rfloor)$ is the maximum fitness value at generation t, and T is the total number of generations used in each run.

Works using the MFE include [55, 133, 134, 139].

2.2.3. Offline Performance

The off-line performance was defined by Branke [28] as follows:

$$\text{off-line performance}(T) = \frac{1}{T} \sum_{t=1}^{T} e_t^{'} \tag{2.28}$$

$$\text{with} \quad e_t^{'} = max \left\{ e_\tau, e_{\tau+1}, \ldots, e_t \right\} \tag{2.29}$$

where e_t is the fitness of the solution evaluated by the algorithm at time t, and T is the total number of time instants considered. τ represents the first time instant right after the last change on the environment (on the problem) occurred. This measure represents the average of the best values obtained on each time instant up to the time T. If we further average the off-line performance for all the evaluations and all the runs of an algorithm, we obtain the overall off-line performance, that gives a good idea of the average performance of the algorithm through all the optimization process.

This performance measure has been used in [55, 133, 134, 139].

2.2.4. Weicker measures

A different approach from simply averaging performance values was taken by Weicker [165]. He presented an in-depth analysis on this topic, mainly focusing on EAs for dynamic environments, and he considered the evaluation of three characteristics in a dynamic optimization process:

- **Accuracy**: The optimization accuracy at time t is defined as:

  $$accuracy^t = \frac{best^t - Min^t}{Max^t - Min^t}$$

  where $best^t$ is the fitness of the best candidate solution in the population at time t, $Max^t \in \mathbb{R}$ is the best fitness value in the search space and $Min^t \in \mathbb{R}$ is the worst fitness value in the search space.

- **Stability**: In the context of dynamic optimization, an adaptive algorithm is called stable if changes in the environment do not affect the optimization accuracy severely. Even in the case of drastic changes an algorithm should be able to limit the respective fitness drop. The stability of an optimization algorithm at time t is defined as:

  $$stability^t = max\left\{0, accuracy^t - accuracy^{(t-1)}\right\}$$

  The stability can not be relied as the only criterion to compare two algorithms since it makes no statement on the accuracy level. It can be seen as a consistency index of how reliably the algorithm keeps getting good results through the whole optimization process.

- **Reactivity**: An additional aspect that can be considered as a goal in a dynamic optimization process is the ability of an adaptive algorithm to react quickly to changes. The ε-reactivity of an algorithm at time t is:

$$\begin{aligned} react_{\varepsilon}^{(t)} = \ & min \left\{ (t' - t) \mid \tfrac{accuracy^{t'}}{accuracy^{t}} \geq (1 - \varepsilon) \right\} \\ & \bigcup \{maxgen - t\} \end{aligned}$$

where $t, t' \in \mathbb{N}$ and $t < t' \leq maxgen$, with $maxgen$ referring to the number of generations in generation-based algorithms such as evolutionary algorithms. Lower values for the reactivity mean a better and faster reaction to changes.

While these proposals were mainly oriented to evaluate EAs, they are usually applicable to any other type of algorithm or they are easily modifiable to be adapted to other algorithms. In this way, the best of the population could be a single algorithm solution if the algorithm does not use populations or the generations can be substituted by other progression steps like the number of problem changes or fitness function evaluations.

2.2.5. Performance measures chosen in the experiments of the thesis

In the experiments carried out in this thesis we have mainly used the **Offline Error** and the **Offline Performance**. When it has been necessary to use statistical tests, we have chosen non-parametric tests, mostly a combination of the **Kruskal-Wallis** test and the **Mann-Whitney-Wilcoxon** test, using **Holm's adjustment** for multiple comparisons when required.

2.3. Algorithms

Many algorithms used for dynamic optimization come from methods that have been already used in static optimization, with some modifications to manage dynamism. Some of the main strategies for doing this are listed bellow.

- *Do not do anything.* This technique, as simple as it seems, may be useful sometimes. Certain algorithms are able of maintaining a sustainable diversity through all their search process, so that when a change is produced, the algorithm adapts to it automatically. Almost all metaheuristic methods exhibit this characteristic to some extent. However, the main drawback of this approach is that if the change is produced when the algorithm is in an advanced convergence stage, the adaptation to the new environment is usually quite slow. This late reaction may result in the algorithm not finding a good enough solution before the next change occurs.

- *Detect changes.* This technique assumes that the algorithm is not informed of changes, and that changes are detectable, usually by means of reevaluating a set of solutions and comparing their value with the previous one. If any of these "sentinel" solutions change their value, the environment is assumed to have varied, and the mechanism for managing changes is activated. This mechanism may consist in reevaluating the rest of solutions, randomly restarting the algorithm, temporarily increasing the diversity, etc. This change detection based on reevaluation has the disadvantage of consuming a non-dismissible amount of time/evaluations that could be used for searching the optimum.

- *Increase diversity.* The objective of this technique is clear: if the optimum of an environment changes, the higher the diversity of the algorithm, the higher its probability of finding the new optimum quickly. This can be achieved by different means: using multiple populations of solutions spread through all the search space; evaluating solutions at random, independently of the most promising areas; actively searching for solutions as much different as possible from the current ones; etc. The problem with this technique is widely known in optimization: the higher the effort invested in diversifying, the lower the effort that can be dedicated to intensifying, and therefore, the lower the quality of the solutions obtained.

- *Reuse previous information.* Sometimes it happens that the environment changes following a pattern. For example, a problem may alternate between two states, or the optimum may move in a trajectory. In these cases, the use of memory and learning techniques may help the algorithm to predict certain valuable aspects of the problem — when will the next change take place, in which position will the new optimum be, etc. — so that the algorithm is able of responding more efficiently. The main disadvantages are, firstly, the higher requirements of the algorithm for storing and processing this information, and secondly, that a bad prediction may seriously penalize the search.

Normally, these strategies are not implemented in an isolated way, but are usually combined in order to obtain the maximum effectiveness. As it can be seen, all of them have advantages and disadvantages, and their use depends on the algorithm and the researcher using them. However, it is generally accepted that in a DOP it is better to *follow* the optimum reasonably close through all the changes, rather than finding it very precisely once and loosing it completely when the environment varies. Thus, increasing the diversity of the algorithm is a key aspect when designing or adapting a method to DOPs.

Considering all of this, it is understandable that most of the algorithms in the literature of DOPs are population-based metaheuristics, since they naturally contribute to increase diversity, and many of the previously mentioned strategies are easy to implement on them.

2.3.1. Evolutionary Algorithms

Evolutionary Algorithms (EAs) [78] are an example of population-based metaheuristics, and are also the most used algorithmic family in Dynamic Optimization. Since the first known reference to EAs applied to dynamic optimization by Fogel et al. [60], and the paper by Goldberg and Smith [67] almost twenty years later, EAs have been the most common approach used to solve this kind of problems.

These algorithms are inspired by the evolution mechanisms that appear in nature, such as natural selection and genetic recombination and mutation processes. Evolutionary Algorithms evolve a population of solutions (also called *individuals*) applying some selection, crossover and mutation operators to create the population of the next generation. The process is summarized in Algorithm 2.1.

Algorithm 2.1: Evolutionary Algorithm

```
1  $P \leftarrow$ population ;
2  $P_{new} \leftarrow \emptyset$ ;
3  $initializePopulation(P)$;
4  $evaluatePopulation(P)$;
5  while $stopping\ condition\ is\ not\ met$ do
6      while $size(P_{new}) < size(P)$ do
7          $I_1, I_2 \leftarrow selectIndividualsForMating(P)$;
8          $I_3, I_4 \leftarrow crossover(I_1, I_2)$;
9          $I_3^{'}, I_4^{'} \leftarrow mutation(I_3, I_4)$;
10         $P_{new} \leftarrow P_{new} \cup \{I_3^{'}, I_4^{'}\}$
11     end
12     $evaluatePopulation(P_{new})$;
13     $P \leftarrow P_{new}$;
14     $P_{new} \leftarrow \emptyset$ ;
15 end
```

Successful approaches of EAs to DOPs include using hypermutation [33] as an adaptive operator in GAs. This technique maintains the whole population after a change in the environment, but increases population diversity by drastically raising the mutation rate for some number of generations. And Abbass et al. [1] introduced the Extended Compact Genetic Algorithm (ECGA) to solve problems in dynamic environments. Their approach is based on random restarts of the population at each change so that diversity in the population can be increased at the beginning of each new environment.

Random Immigrants GA (RIGA) and variations [71, 151, 178] deserve an especial mention due to its good results. RIGA is a GA specifically focused at maintaining diversity. In every generation, RIGA replaces part of the population by randomly generated individuals. This introduces new genetic material in every time step and avoids the convergence of the whole population to a narrow region of the search space.

The use of memory schemes has also provided good results to GAs, either alone as abstract mechanisms [134, 176] or combined with other techniques such as hyper-mutation [86] and random immigrants [179].

Multi-population GAs have been an extremely successful approach, like in the case of the Self Organizing Scouts (SOS) introduced in [26], or in [30], where Bui, Branke and Abbass apply multiobjective GAs for solving single-objective dynamic functions.

2.3.2. Particle Swarm Optimization

Apart from EAs, the Particle Swarm Optimization (PSO) algorithm [87] is also quite popular for continuous DOPs. The PSO is a population-based metaheuristic where the individuals (particles) live within a population (swarm). In this algorithm, the optimization process is not performed using an evolution metaphor like EAs; instead, particles move around the solution space, visiting solutions according to some movement equations.

The movement of the particles is performed in an "inertial" way, such that the position of a given particle varies according to a velocity vector (2.30), and this vector itself varies according to an acceleration vector (2.31). This acceleration has 3 main components: an attraction vector to the best historical position visited by the particle (2.32), an attraction vector to the current best particle of the swarm (2.33), and the previous velocity vector of the particle (inertia) (2.34):

$$\mathbf{x}(t+1) = \mathbf{x}(t) + \mathbf{v}(t+1) \tag{2.30}$$
$$\mathbf{v}(t+1) = \mathbf{v}(t) + \mathbf{a}(t+1) \tag{2.31}$$
$$\mathbf{a}(t+1) = \chi[\eta_1 c_1(\mathbf{x}_{pbest} - \mathbf{x}(t)) \tag{2.32}$$
$$+ \eta_2 c_2(\mathbf{x}_{gbest} - \mathbf{x}(t))] \tag{2.33}$$
$$- (1-\chi)\mathbf{v}(t) \tag{2.34}$$

where $\mathbf{x}$, $\mathbf{v}$ and $\mathbf{a}$ are position, velocity and acceleration vectors respectively, $\mathbf{x}_{pbest}$ is the best so far position discovered by each particle, and $\mathbf{x}_{gbest}$ is the best so far position discovered by the whole swarm. Parameters $\eta_1, \eta_2 > 2$ are spring constants, and c_1 and c_2 are random numbers in the interval $[0.0, 1.0]$. Since particle movement must progressively contract in order to converge, a constriction factor χ, $\chi < 1$ is used, as defined by Clerc and Kennedy in [32]. This factor replaces other slowing approaches in the literature, such as inertial weight and velocity clamping [47].

The PSO has specific DOP-oriented proposals [14,31,83,124]. Some of the PSO variants use charged particles for maintaining diversity (CPSO) [12]. However, one of the most successful approaches is mQSO [15], which has been used in multiple subsequent works [13, 92, 180].

A swarm in the mQSO is formed by two types of particles:

- *Trajectory* particles (also known as *classical* or *neutral*). These are the particles used by the canonical PSO algorithm, which positions are updated following the usual movement equations 2.30, 2.31, 2.32, 2.33, and 2.34.

- *Quantum* particles. These particles were newly introduced in the mQSO algorithm, and aim at reaching a higher level of diversity by moving randomly within a hypersphere of radius $\mathbf{r}$ centered on $\mathbf{x}_{gbest}$. This random movement is performed according to a probability distribution over the hypersphere, in this case, a uniform distribution:

$$\mathbf{x}(t+1) = \mathbf{rand}_{hypersphere}(\mathbf{x}_{gbest}, \mathbf{r}) \tag{2.35}$$

Beside this, the general idea of the mQSO is to use a set of multiple swarms that simultaneously explore the search space. This multi-swarm approach has the purpose of maintaining the diversity, in addition to the use of quantum particles. This is a key point for DOP's, since the optimum can change at any time, and the algorithm must be able to react and find a new optimum. In order to prevent several swarms from competing over the same area, an inter-swarm exclusion mechanism is also used, randomizing the worst swarm whenever two of them are too close.

The mQSO pseudocode is shown in Algorithm 2.2.

2.3.3. Other Algorithms

Finally, other algorithms have been used in the DOP literature, like Ant Colony Optimisation (ACO) [58,59,72,74,110], Differential Evolution (DE) [107], Cultural Algorithms (CA) [128, 140], Estimation-of-Distribution Algorithms (EDA) [19, 20, 58, 64, 177], Immune-based Algorithms (IBA) [75, 121, 144, 155, 175, 185], Extremal

Algorithm 2.2: The mQSO algorithm

```
1  Randomly initialize the particles in the search space;
2  while stopping condition is not met do
3      foreach swarm s do
4          Test for exclusion;
5          if s needs to be excluded then
6              Relocate s randomly;
7          end
8          else
9              Move particles according to equations 2.30, 2.31, 2.32, 2.33, 2.34
               and 2.35 ;
10         end
11         Evaluate each particle position;
12         Update $\mathbf{x}_{pbest}$ and $\mathbf{x}_{gbest}$;
13     end
14 end
```

Optimisation (EO) [116], and even Neural Networks (NN) [37, 153] (these ones being the only non-metaheuristic approaches for solving DOPs).

Table 2.2 has been reproduced from [34], containing cross-references between the most used problems of the DOP literature and the algorithms used for solving them. Acronyms in the columns of the table indicate Ant Colony Optimisation (ACO), Cooperative Strategies (CS), Cultural Algorithms (CA), Estimation-of-Distribution Algorithms (EDA), Evolution Strategies (ES), Evolutionary Algorithms (EA), Evolutionary Programming (EP), Genetic Algorithms (GA), Immune-based Algorithms (IBA), Memetic Algorithms (MA), Self Organising Scouts (SOS), Other Evolutionary Algorithms (OEA), Neural Networks (NN), Swarm Intelligence-based methods (PSO), and Other (O).

2.3.4. Algorithms chosen in the experiments of the thesis

The algorithms and techniques reviewed suggest that cooperation plays an important role in this type of problems. Moreover, all the metaheuristics analyzed were population-based ones. As it was already mentioned in the Objectives (Sect. 1.2), this is an especially relevant aspect to consider in this thesis. In this context, we can distinguish between those methods in which there is an *implicit* collaboration among their components, an *explicit* collaboration, or a mix of both. It can be considered, for example, that there is an implicit collaboration between the elements of a GA, since although each solution is independent of the others, the way of obtaining it by means of the crossover and mutation operators implies that some information is exchanged among the population, spreading over it. However, in the case of Collaborative Swarms, we can talk about an explicit collaboration, by their own definition, but also about an implicit one, due to the movement mechanism of a swarm's particles, interrelated through the *best*. Fig. 2.5 shows a graphical representation of this classification.

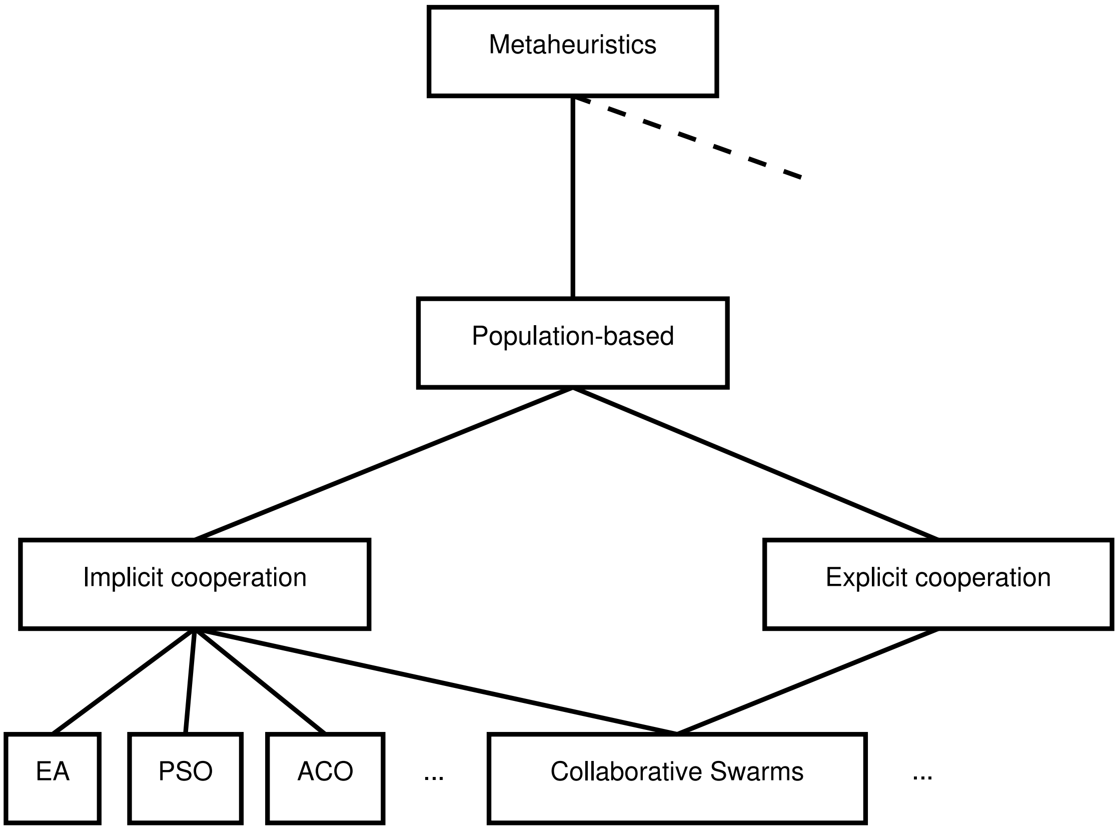


Figure 2.5 – A classification of several population-based metaheuristics depending on the type of cooperation among their constituent components.

In the research carried out by this thesis we have focused in **population-based metaheuristic algorithms**, where we have tried to cover the widest possible spectrum of types of cooperation. Therefore, we have designed new algorithms that use **implicit**, **explicit**, and **mixed cooperation**. Furthermore, we have proposed methods from different algorithmic families, including **Swarm Intelligence**, **Agents**, and **Cooperative Strategies (CS)**, comparing them in some cases with state-of-the-art *Evolutionary Algorithms (EA)*, thus covering a wide variety of methods.

Problem	Methods														
	ACO	CS	CA	EDA	ES	EA	EP	GA	IBA	MA	SOS	OEA	NN	PSO	O
Real World Applications															
Aerospace Design								[101]							
Car Distribution System															[109]
Evolutionary Robotics								[153]					[153]		
Financial Optimization Problems								[150]							
Mobile Ad Hoc Networks								[179]							
Modeling of Ship Trajectory						[145]									
Path Planning								[49]							[109]
Pollution Control															[109]
Pose Problem						[139]									
Rober Problem						[156]									
Robust Design						[8, 66, 73, 97, 135]		[56, 98, 135]	[81]	[73]		[81]			
Salting Route Optimization						[73]				[73]					
Structural Optimization						[117]									
Varied-Line-Spacing Holographic Grating								[98]							
Wireless Sensor Networks						[129]									
Synthetic Dyn. Problems															
DF1 Generator (cones)			[128, 140]			[51, 115]								[54, 55]	
Dyn. Ackley Function					[141, 142]										
Dyn. Bit Matching						[84]		[84, 111, 147]							
Dyn. Deceptive Functions				[177]		[184, 185]		[151, 163, 164, 173–175, 177, 183, 185]	[175, 185]						
Dyn. Knapsack Problem						[27, 84, 86, 184]		[84, 111, 144, 164, 178, 183]	[144]						
Dyn. Onemax Function	[58]			[58, 177]		[44, 185]		[163, 173, 174, 177, 178, 183, 185]	[185]						
Dyn. Plateau Functions								[163, 175, 178]	[175]						
Dyn. Problem Generator					[85]	[115, 158, 184]	[93]	[130, 131, 152]	[155]					[93]	
Dyn. Rastrigin Function					[141, 142]		[154]		[121]					[45, 143, 160]	
Dyn. Royal Road Function	[58, 59]			[58, 177]		[184]		[151, 163, 164, 173, 177, 183]							
Dyn. Scheduling								[105]	[75]						[5]
Dyn. Sphere					[3, 4, 21, 141, 142]				[121]						
Dyn. Vehicle Routing Problem (DVRP)	[74, 110]							[74, 123]							
Griewank Function								[169]	[121]					[143, 160]	
Moving Parabola								[1, 111]						[46, 82]	
Moving Peaks Benchmark (MPB)		[99, 100, 125, 126]			[107]	[6, 28, 52, 84, 94, 99, 100, 193]		[29, 30, 84, 88, 111, 138, 157]	[155]		[26, 28, 99, 100]	[6]		[2, 12–16, 45, 83, 94, 96, 99, 100, 107, 120, 124]	[2, 116]
Rosenbrock Function									[121]					[82, 143, 160]	
Shaky Ladder Hyperplane-Def. Funcs.								[130, 131]							
Time-Linkage Numerical Problems				[19, 20]		[19, 20]									
Trap Function based Synthetic Problems								[1, 90]							
XOR-based Synthetic Dynamic Problems						[136]		[173, 174, 176]							
Other	[59, 72]			[64]	[167]	[22, 149, 170, 172]		[111, 112]		[149]			[37]	[31]	[36, 186]

Table 2.2 – Dynamic problems and methods cross-references to the papers where they are used (table obtained from [34]).

Chapter 3

New algorithmic proposals for DOPs

In this chapter we present the new algorithms that we have proposed for dealing with DOPs. In some cases, these proposals consist in improving state of the art algorithms by adding new operators, heuristic rules, etc. In other cases, these proposals are based in the adaptation of a previously existent algorithm for static problems that was never used for DOPs, so that it can now work for them. These proposals cover very diverse algorithms, like mQSO, Cooperative Strategies or Agents, and they have been applied both to continuous and discrete DOPs, although the first ones clearly predominate over the second. The variety of algorithms and scenarios used has allowed to extract interesting conclusions about the role of cooperation among the constituent elements of a method when facing a DOP.

3.1. Control particle trajectories of a PSO

In this section we present an operator for controlling the trajectory of the particles of a multi-swarm PSO for DOPs. The operator, named Control Particle Trajectory (CPT), is able to detect particles within a swarm that are wasting function evaluations in non-promising areas, and "reset" them.

3.1.1. Motivation

In a DOP it is important to keep a balance between the time dedicated to explore promising areas of the search space, and the intensification effort in the zones where the best solutions have been found.

As we saw in Sect. 2.3.2, particles in a swarm move in an inertial way, according to equations 2.30, 2.31, 2.32, 2.33, and 2.34. This particle movement mechanism helps maintaining the diversity during the search. However, this diversity may also contribute to waste evaluations in non-promising areas. This can be verified by means of the number of consecutive *failures*. We consider that a particle produces a failure when there is a transition from a solution $x(t)$ to another solution $x(t+1)$ such that the fitness function of $x(t+1)$, $f(x(t+1))$, is worse than $f(x(t))$. One single failure may not affect the algorithm's performance, but if a "chain" of consecutive failures occurs, it would result in a waste of resources, e.g., time, evaluations, etc (see Fig. 3.1).

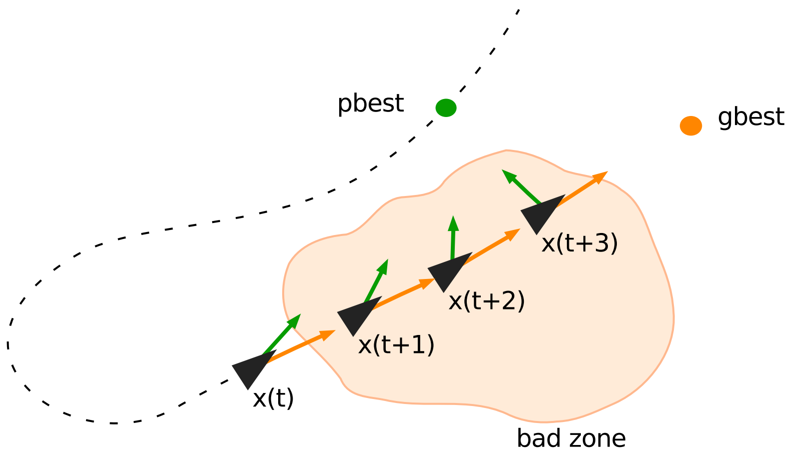


Figure 3.1 – *Control particle trajectories of a PSO.* Due to the PSO's movement mechanism, a particle may enter a "bad" zone and produce several consecutive failures, thus wasting resources.

3.1.2. Proposal

In order to deal with this situation, we have proposed a Control Particle Trajectory (CPT) operator, that can detect if a particle is in that situation and "reset" it. In order to do so, the CPT operator keeps the count of consecutive failures for each particle. If the count of consecutive failures exceeds a given threshold, the particle is repositioned at the best of its swarm, with a randomized velocity vector.

3.1.3. Validation

In order to verify the efficiency of the CPT operator it is first necessary to understand what is the typical behaviour of a particle regarding the number of consecutive failures, so that an appropriate threshold can be established.

With this objective in mind, we have performed a first experiment using a multi-swarm PSO [15] without any kind of modification, and we have count the number of consecutive failures for each particle, grouping them in a histogram. The multi-swarm PSO used is formed by 10 swarms and 10 particles per swarm, and we have used the typical configuration values for its parameters suggested by the authors in the original paper [15]. The results can be seen in Fig. 3.2.

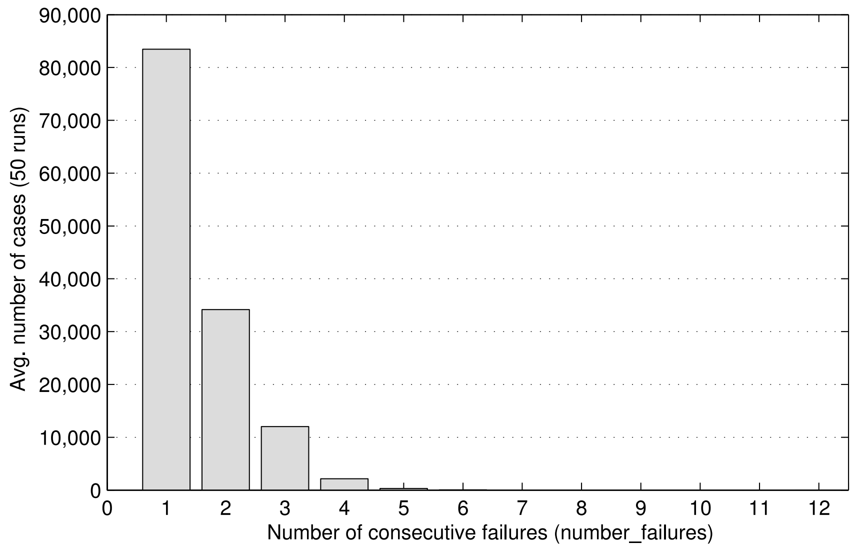


Figure 3.2 – *Control particle trajectories of a PSO.* A histogram of the number of consecutive failures of a swarm's particles, tested on the Scenario 2 of the MPB.

It is clear that having one failure is more frequent than having two. This frequency monotonically decreases as the number of failures becomes higher. The data show that it is extremely rare for a given particle to account for more than 10 failures in a row, so we can safely assume that the threshold to be used by the CPT operator should be lower than 10.

Thus, in order to determine the exact value of the threshold, we have performed a second experiment, in which we have added the CPT operator to the multi-swarm PSO previously used. We have tested multiple values for the maximum number of consecutive failures allowed (from 1 to 10), and we have logged several offline error measures for each of them. The results can be seen in Table 3.1, where we have also included the offline error of the multi-swarm PSO without the CPT operator, as a reference (corresponding to the row with *max failures* $= \infty$). A graphical representation of this experiment, for some representative values of *max failures*, is shown in Fig. 3.3.

Max Failures	Mean (Std. Dev.)	Min	Max
∞	1.52 (0.83)	0.51	5.00
1	3.06 (1.44)	0.65	7.92
2	0.40 (0.25)	0.13	1.11
3	0.52 (0.39)	0.16	2.43
4	0.85 (0.47)	0.29	2.07
5	1.18 (0.54)	0.26	2.93
6	1.44 (0.67)	0.51	3.09
7	1.45 (0.69)	0.49	3.15
8	1.50 (0.76)	0.48	4.19
9	1.48 (0.71)	0.51	3.38
10	1.52 (0.83)	0.51	5.00

Table 3.1 – *Control particle trajectories of a PSO.* Offline error values of a multi-swarm PSO with the CPT operator, for different *max failures* values.

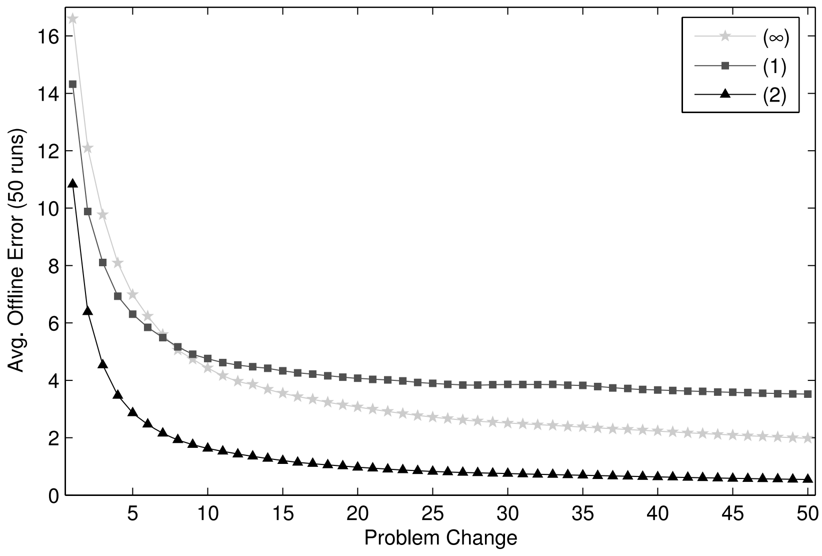


Figure 3.3 – *Control particle trajectories of a PSO.* Offline Error evolution of a PSO over time for different values of *max failures*, tested on the Scenario 2 of the MPB.

Additionally, we have performed variations of this experiment, keeping the problem conditions established in the Scenario 2 of the MPB, but using values for the change frequency of {500, 1000, 2000, 3000, 4000 and 5000} cost function evaluations. In Fig. 3.4 we have depicted a graphical representation of the average offline error versus several values of *max failures*, using the change frequency values said before.

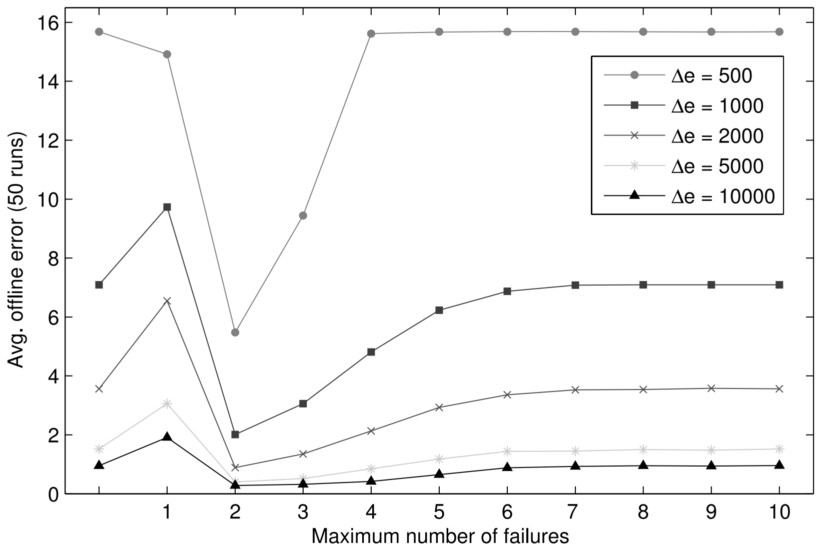


Figure 3.4 – *Control particle trajectories of a PSO.* Avg. Offline Error vs. *max failures* of a PSO, for different values of change frequency (Δe), tested on the Scenario 2 of the MPB.

Figure. 3.4 clearly shows that when *max failures* is set to 2, the offline error of the PSO+CPT (from now on, mCPT-PSO) is consistently lower, for all the change frequency values tested.

Finally, in a last experiment, we have compared the mCPT-PSO (*max failures* = 2) with two versions of the PSO introduced by Branke and Blackwell in [15]: mCPSO (multi-Charged Particle Swarm Optimization) and mQSO (multi-Quantum Swarm Optimization). For a fair comparison, we have used the same multi-swarm configurations employed by mCPSO and mQSO in the original work. In [15] these multi-swarm configurations had the form $M(N1 + N2)$, where M represents the total number of swarms, $N1$ the number of neutral particles (also known as trajectory particles) in each swarm, and $N2$ the number of charged particles (for mCPSO) or the number of quantum particles (for mQSO) of each swarm. For the mCPT-PSO, there is no distinction between particles, so this configuration will be interpreted as M swarms, each one with $N1 + N2$ particles. Table 4 shows

Configuration	mCPSO	mQSO	mCPT-PSO
5 (10+10)	3.74 (0.14)	3.71 (0.15)	2.89 (0.16)
10 (5+5)	2.05 (0.07)	1.75 (0.06)	0.40 (0.04)
14 (4+3)	2.29 (0.07)	1.93 (0.06)	0.61 (0.06)
20 (3+2)	2.89 (0.07)	2.35 (0.07)	0.83 (0.06)
25 (2+2)	3.27 (0.08)	2.69 (0.07)	1.35 (0.08)

Table 3.2 – *Control particle trajectories of a PSO.* Offline error values (average and standard deviation) for the mCPSO, mQSO and mCPT-PSO algorithms.

the average offline error and the standard deviation for the three algorithms. The results clearly show the benefits of our proposal: the average error is much lower for every configuration tested.

3.1.4. Conclusions

The proposed operator, CPT, improves the results of a multi-swarm PSO in dynamic environments, decreasing the wasted evaluations, and doing all this using a relatively simple mechanism. The results show that the optimum number of *max failures* allowed to particles should be 2, before being stop and relocated around the best particle of its swarm. This value consistently produces the best performance in different test scenarios. Also, the mCPT-PSO has obtained very good results when compared with other versions of a multi-swarm PSO.

This work was published in reference [120]:

"Controlling Particle Trajectories in a Multi-swarm Approach for Dynamic Optimization Problems", P. Novoa, D. A. Pelta, C. Cruz, and I. G. del Amo, in *Methods and Models in Artificial and Natural Computation. A Homage to Professor Mira's Scientific Legacy* (J. Mira, J. Ferrández, J. Álvarez, F. de la Paz, and F. Toledo, eds.), vol. 5601 of *Lecture Notes in Computer Science*, pp. 285–294, Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-02264-7_30`.

3.2. Using heuristic rules in a mQSO

In this section we present 3 new variants of the classical algorithm mQSO that use heuristic rules to improve its performance. The use of rules has produced very good results in other algorithms (e.g., CS, in Sect. 3.4), and allows to easily incorporate other successful approaches or variations of them, like the CPT operator of Sect. 3.1.

3.2.1. Motivation

The idea of adding rules to the mQSO was derived from a previous work in which we analyzed the properties of the two types of particles used by the mQSO [41].

The objective of such work was to investigate what proportion of *quantum* and *trajectory* particles provided more stability and performance to a mQSO in different scenarios.

In order to find out this optimal ratio, we performed several experiments on the Scenario 2 of the MPB. In these experiments we measured the average offline error of the mQSO for different configurations of the Scenario 2, that included a growing number of peaks from 5 up to 100, 2 different change frequencies (5000 and 200 function evaluations), and all possible combinations of *quantum* and *trajectory* particles, with a maximum of 10 particles per swarm. These settings are summarized in Table 3.3.

Parameter	**Value**
Number of peaks	$\in [5, 100]$
Change frequency	5000 (low freq.)
	200 (high freq.)
Number of swarms	5
Number of t-particles	$\in [0, 10]$
Number of q-particles	$10 - t$
Number of changes	100
Number of runs (repetitions)	30

Table 3.3 – *Using heuristic rules in a mQSO – Analysis of the properties of the mQSO particles.* Settings for the different experiments of the mQSO on Scenario 2 of the MPB.

The results obtained in the experiments (Tables 3.4 and 3.5) clearly show that the particle combinations that obtain better results are the ones that use more trajectory particles than quantum ones, approximately in the range between 6t4q and 9t1q (here, xtyq means x trajectory particles and y quantum particles).

To be able to explain these results we performed one additional experiment, in which we measured how many times the best particle of a swarm was a quantum one, and how many times was a trajectory one. In order to give more information,

Peaks	**0t10q**	**1t9q**	**2t8q**	**3t7q**	**4t6q**	**5t5q**	**6t4q**	**7t3q**	**8t2q**	**9t1q**	**10t0q**
5	4.39	3.73	2.91	2.59	2.27	**2.10***	**2.18**	**2.20**	**2.11**	**2.49**	**2.52**
10	7.32	6.38	5.81	5.45	4.98	4.95	4.86	**4.68***	**4.93**	**4.95**	5.11
15	8.1	7.35	6.55	6.07	6.23	**5.90**	6.17	**5.90***	**5.96**	6.31	6.48
20	8.83	7.9	7.02	7.00	6.51	**6.43**	**6.09***	**6.34**	6.90	**6.37**	6.89
25	8.45	8.08	7.55	7.21	6.68	**6.36***	**6.46**	**6.36**	**6.48**	7.14	7.06
30	8.90	8.53	7.98	7.28	7.54	6.99	**6.66***	**6.72**	**6.71**	7.28	7.16
35	9.19	8.18	8.29	7.86	7.28	**7.05**	**6.77***	**7.01**	**7.00**	**7.08**	7.53
40	8.53	8.29	7.63	7.70	7.54	**7.24**	**7.47**	**7.24**	**7.45**	**7.20***	**7.52**
45	8.59	8.52	7.51	**7.18**	7.24	**6.87***	**7.29**	**7.27**	7.53	7.61	7.79
50	8.40	7.76	7.77	6.87	7.07	7.12	**6.89**	**6.68***	7.19	7.08	8.01
55	8.79	8.53	7.77	7.20	7.60	**6.72***	7.30	7.18	**7.14**	**7.00**	7.56
60	8.70	8.52	8.05	7.23	7.67	7.38	**7.08***	**7.13**	7.34	**7.34**	7.77
65	8.58	7.77	7.70	7.51	7.46	**6.67***	7.24	7.11	7.49	7.58	8.16
70	8.51	8.17	7.98	7.23	**6.99***	**7.30**	**7.13**	**7.07**	**7.28**	**7.2**	8.08
75	8.55	7.89	7.76	**7.26**	**6.90***	**7.01**	**7.38**	**7.18**	**7.13**	**7.25**	7.7
80	8.26	8.08	7.15	**6.78**	6.94	**6.77**	**6.69***	**6.96**	**6.91**	7.1	7.20
85	8.78	8.14	7.23	7.51	7.24	7.37	**7.09**	**6.91***	**6.92**	**7.08**	7.57
90	8.55	8.05	7.08	7.04	6.88	6.77	7.20	**6.44***	**6.81**	7.09	7.58
95	8.43	7.96	7.72	7.27	**6.86**	**6.87**	**6.74***	**6.80**	**6.98**	**7.05**	7.87
100	8.44	7.57	6.89	6.96	6.67	**6.42***	**6.68**	**6.58**	**6.45**	6.99	7.63

Table 3.4 – *Using heuristic rules in a mQSO – Analysis of the properties of the mQSO particles.* Mean values of the offline error for all the t-q configurations and number of peaks (**low change-frequency**). Values marked with (*) indicate the best (lowest) mean error. Values in bold-face indicate that no statistically significant difference exists with respect to the best value in the row.

Peaks	**0t10q**	**1t9q**	**2t8q**	**3t7q**	**4t6q**	**5t5q**	**6t4q**	**7t3q**	**8t2q**	**9t1q**	**10t0q**
5	19.80	18.95	18.72	18.76	19.88	19.67	**17.84***	**18.91**	25.49	29.47	52.62
10	19.65	19.48	18.38	17.98	18.35	**17.49**	**16.57***	**17.52**	17.89	**17.78**	29.23
15	19.67	18.22	17.58	17.06	16.36	16.04	15.77	16.54	**15.36***	**15.52**	18.20
20	19.46	19.17	18.16	17.61	17.57	16.74	16.67	**16.35**	**16.29**	**15.82***	16.85
25	18.29	17.79	17.01	16.97	16.05	16.05	15.92	**15.48**	**15.33**	**15.15***	17.08
30	19.97	18.98	18.04	17.73	16.67	16.04	15.98	15.97	**15.24***	**15.35**	17.57
35	19.35	18.65	18.15	17.82	17.35	16.77	15.99	**15.17***	**15.41**	16.03	**15.68**
40	17.89	18.07	16.53	16.77	16.76	15.65	15.11	**14.48**	15.01	15.17	**14.46***
45	17.97	17.47	16.57	16.85	15.92	15.80	**14.85**	**15.11**	15.36	15.56	**14.77***
50	18.64	17.97	17.39	16.51	16.15	15.84	15.48	15.22	**14.91**	**14.71***	**14.99**
55	18.20	17.56	17.15	16.43	16.35	15.87	16.17	15.31	**14.56***	**14.98**	**14.80**
60	17.40	16.37	16.05	16.15	15.86	**14.86**	15.32	15.04	**14.42***	**14.54**	**14.74**
65	18.35	17.33	16.17	16.53	16.63	15.95	15.69	**15.25**	**14.98**	**14.86***	**14.92**
70	17.33	17.50	16.42	16.21	16.52	15.44	16.25	15.72	15.34	**15.16**	**15.10***
75	17.76	17.89	16.49	16.55	16.62	15.61	**15.41**	15.68	**15.55**	**15.08***	15.996
80	16.96	15.99	15.94	15.59	15.82	15.14	15.08	**14.51***	16.06	15.04	15.15
85	16.78	16.69	16.39	15.30	15.01	15.33	15.15	**14.37***	**14.70**	14.75	14.86
90	17.34	16.84	16.13	16.17	**15.27**	15.71	**15.04**	**14.89**	**14.99**	**14.88***	**15.32**
95	17.56	16.84	16.57	15.80	15.57	15.25	15.29	15.26	**14.48***	**14.82**	**14.82**
100	16.79	15.70	15.50	15.77	14.84	14.24	15.46	14.64	14.62	**13.75***	14.40

Table 3.5 – *Using heuristic rules in a mQSO – Analysis of the properties of the mQSO particles.* Mean values of the offline error for all the t-q configurations and number of peaks (**high change-frequency**). Values marked with (*) indicate the best (lowest) mean error. Values in bold-face indicate that no statistically significant difference exists with respect to the best value in the row.

we divided these data for each iteration of the algorithm between changes, so that we could know not only which particle contributed more to the best of the swarm, but also in which moment of the search (Fig. 3.5). For example, in the Scenario 2, with a change frequency of 5000 evaluations, the mQSO executes 100 iterations between changes; thus, iterations 1-10 are the ones produced right after a change in the environment, and iterations 90-100 are produced at the end, just before the next change.

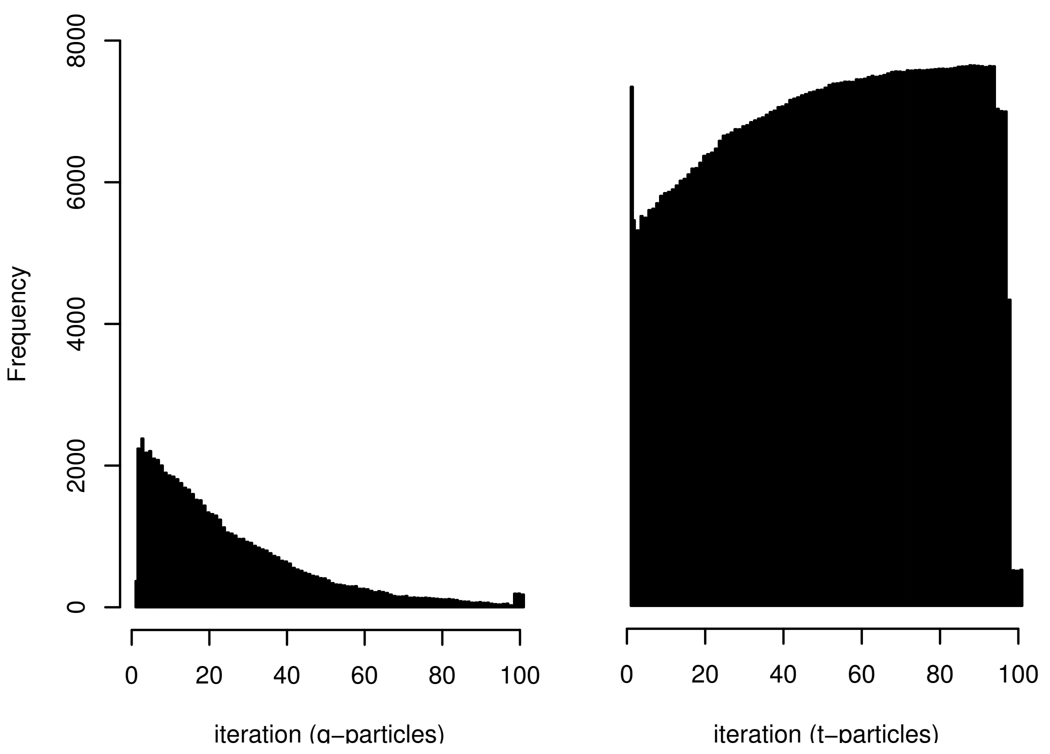


Figure 3.5 – *Using heuristic rules in a mQSO – Analysis of the properties of the mQSO particles.* Histogram representing the number of times that quantum particles and trajectory particles contributed to their swarm's best in a mQSO, distributed over the 100 iterations that the algorithm was executed between changes in the environment. Left panel shows quantum particles distribution; right panel shows trajectory particles distribution.

The results obtained (Fig. 3.5) clearly show that quantum particles are most effective right after a change in the environment, because they increase the diversity of the algorithm. However, as time goes by, their contribution to the best of the swarm fades away, while trajectory particles lead the optimization process. This suggests that quantum particles are actually wasting function evaluations most of the time except for the first moments after a change in the environment (although their contribution cannot be neither dismissed, since these particles seem to give some stability to the search).

Given the results, in that work we proposed a more efficient, fixed-ratio for the particle types, 7t3q, improving the one suggested by Branke and Blackwell (5t5q) in their original work on the mQSO [15].

However, we also believed that a better strategy would be to vary the proportion of quantum/trajectory particles in a dynamic way over the execution of the algorithm. We knew by previous works [69] that the use of rules could effectively improve the performance of algorithms for DOPs. We therefore decided to adopt this approach and incorporate heuristic rules to govern the behaviour of the mQSO. Although the initial idea was to adapt the ratio of quantum/trajectory particles, we soon realized that the use of rules had a much higher potential, and we designed new rules for new behaviors.

3.2.2. Proposal

The addition of rules into the original mQSO algorithm is rather simple, and the modifications are shown in bold-face in Algorithm 3.1, lines 5 and 15. Line 5 simply applies the rules to each swarm before moving or evaluating it, and line 15 gathers information about the mQSO performance for the last iteration.

3.2.2.1. Change Rule

The first rule created, named *Change* Rule, adjusts the proportions of each particle type according to an approximation of their expected performance, as observed in the experiments in [41]:

Change rule

IF	a change in the environment has occurred recently
THEN	temporarily increase the number of *quantum* particles of a swarm and decrease the number of *trajectory* particles

It was observed that trajectory particles are more likely to find better solutions than quantum particles through all the execution. Additionally, the experiments showed that right after a change, trajectory particles tend to contribute to the best

Algorithm 3.1: The mQSO algorithm with rules

```
1  Randomly initialize the particles in the search space;
2  while stopping condition is not met do
3      Detect changes in the environment;
4      foreach swarm s do
5          Apply rules to s;
6          Test for exclusion;
7          if s needs to be excluded then
8              Relocate s randomly;
9          end
10         else
11             Move particles;
12         end
13         Evaluate each particle position;
14         Update $\mathbf{x}_{pbest}$ and $\mathbf{x}_{gbest}$;
15         Record s's performance data;
16     end
17 end
```

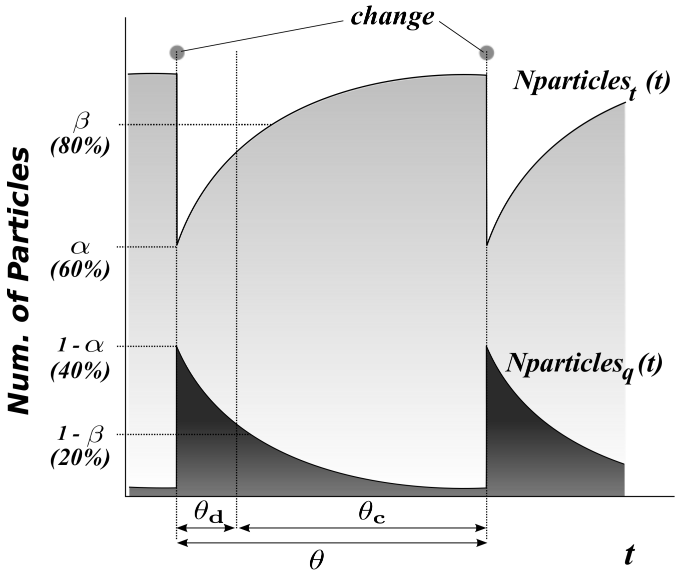


Figure 3.6 – *Using heuristic rules in a mQSO.* Contribution of quantum and trajectory particles to the *gbest* of a swarm in a mQSO. This figure illustrates the concepts used in the *Change* Rule ($\theta, \theta_d, \theta_c, \alpha$ and β), giving a graphical explanation for the chosen values.

of a swarm 60% of the times (we denote this proportion as α), while the quantum particles contribute the remaining 40% $(1 - \alpha)$. These differences are maintained for a short period of time (θ_d, diversity period), and then slowly increase until they stabilize to 80% for trajectory particles (β), and 20% for quantums $(1-\beta)$, for the rest of the execution until the next change (θ_c, convergence period). This behavior is reasonable, since quantum particles move randomly and are better suited for increasing diversity, which is more necessary, precisely, right after a change in the environment. See Fig. 3.6 for a graphical explanation of these concepts.

Finally, we experimentally estimated that, in order for the quantum particles to result beneficial after a change, a minimum of 5 iterations were needed (i.e., θ_d's minimum value should be 5, which implies that θ should be at least 34 iterations). If θ_d is determined to be less than 5 iterations, the rule is disabled, leaving the algorithm as a standard mQSO. A graphical explanation of the Change Rule is shown in Fig. 3.7.

Parameter	**Value**
θ_d	15% of total duration of θ
θ_d min. duration	5 iterations
θ_c	85% of total duration of θ
α	0.6
β	0.8

Table 3.6 – *Using heuristic rules in a mQSO.* Parameter settings for Change Rule

3.2.2.2. Rand Rule

The second rule, named *Rand* Rule, was originally designed for preventing poorly performing swarms from consuming function evaluations that lead them to no improvement, or that could be better used by other swarms. This rule is inspired in the mechanisms behind the CPT operator (Sect. 3.1) and some of the rules used by CS (Sect. 3.4). The rule can be summarized as follows:

Rand rule (I)

IF the performance of a swarm is *bad*

THEN relocate the swarm randomly or pause it if there is not enough time

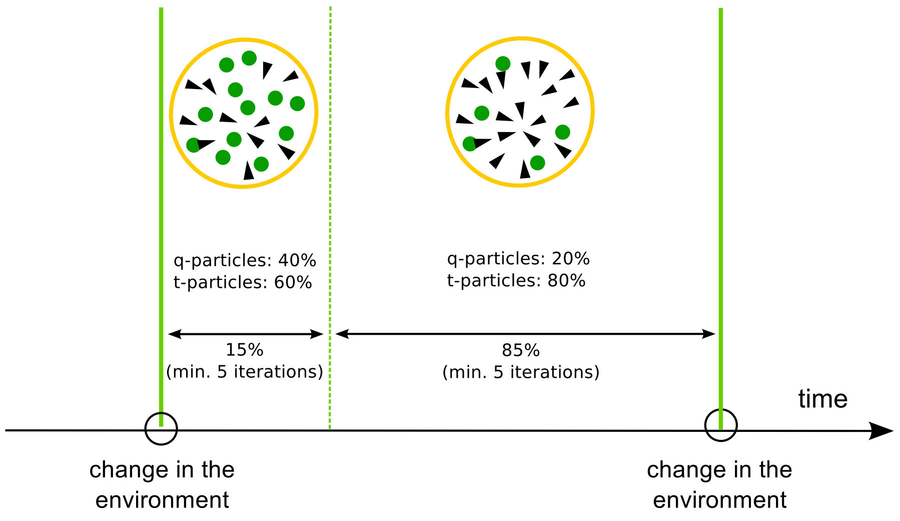


Figure 3.7 – *Using heuristic rules in a mQSO.* Change Rule explanation: the rule attempts to adjust the ratio of quantum vs. trajectory particles, in order to minimize wasted function evaluations.

The problem here is how to determine what is a *bad performance* for a swarm. A way of doing this is to compare its fitness with the rest of the swarms: if it is among the worst, then it has a bad performance. However, a swarm may be performing badly for many reasons, not all of them being necessarily punishable (fluctuations in the fitness due to the random nature of the mQSO algorithm, the swarm is still in a search process and has not yet converged, etc). Ideally, only swarms which are in *bad* zones of the search space should be relocated. In order to prevent a swarm in a possible good zone from being relocated due to temporary low fitness situations, additional conditions must be met.

First, the low fitness situation should be maintained for a certain number of consecutive iterations. If a swarm is among the worst ones only once, it may be bad luck, but if it is repeatedly among the worst ones, then there is a chance that the swarm is in a bad zone.

Second, it is necessary to check if a swarm has converged. A swarm in a search-around phase may have a low fitness, but it still has room for improvement. On the other hand, an already converged swarm with low fitness is more likely to be stuck in a bad zone. The way in which convergence is detected for each swarm in this rule is based on the swarm's improvement ratio. The improvement of a swarm is defined as the difference between the current fitness and the fitness of the previous iteration. When a swarm is searching and has not converged, it usually makes great improvements in the fitness from one iteration to another. Obviously, there may be iterations in which the improvement is low, or even nonexistent, due to the random nature of the search. However, this low improvement situation is unlikely, and usually does not last for too long. On the other hand, if a swarm has converged close to a local optimum, it will make very small improvements, if any, from one iteration to the next one. An example of this behavior is shown in Figure 3.8. Therefore, if the improvement ratio of a swarm descends below a certain percentage of its maximum improvement since the last change, it is a good indicative that it has converged.

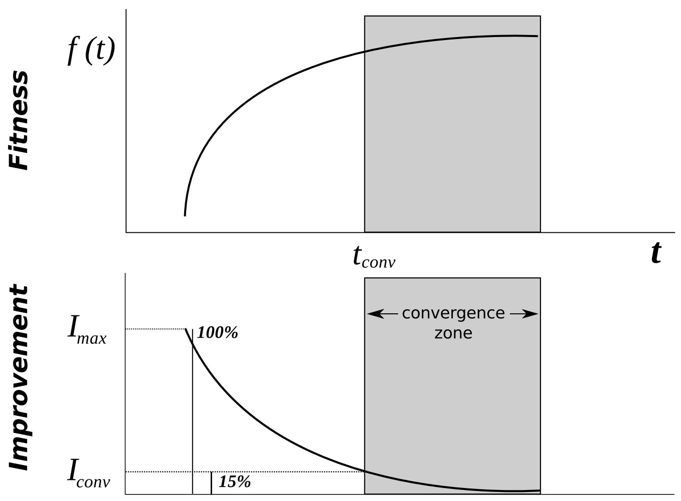


Figure 3.8 – *Using heuristic rules in a mQSO.* Typical evolution of the fitness and improvement of a swarm in a mQSO. Improvement is defined as the difference between two consecutive fitness ($I = f(t) - f(t-1)$). Improvement is bigger at the beginning of the optimization process, and become smaller as the swarm converges. If the improvement of a swarm falls bellow 15% of its historical maximum, we consider the swarm to have converged.

If we combine all these conditions (being among the worst swarms, for a certain number of consecutive iterations, with a low improvement rate that indicates that the swarm has converged), it is very likely that the swarm is indeed located in a bad zone. In this case, there is no point in wasting more function evaluations in that zone, and the swarm should be better relocated. An special case is when the algorithm is able to predict when changes in the environment will happen, and the next change is very close. In this case, relocating the swarm may be useless, since there may not be enough time for the swarm to converge. The swarm is then temporarily *paused* and the rest of the swarms have more function evaluations left available to improve their results. Therefore, the rule can be re-stated as:

Rand rule (II)

IF	a swarm's fitness is among the worst **AND** this situation lasts for several consecutive iterations **AND** the swarm has converged
THEN	relocate the swarm randomly or pause it if there is not enough time left

The parameter settings for this rule are summarized in Table 3.7, and a graphical explanation of the rule is shown in Fig. 3.9.

As a final comment, the Rand Rule requires that a swarm is considered to be among the worst for at least 5 consecutive iterations. This in practice means, as it also happened with the Change Rule, that it will be disabled for low change period scenarios: if the environment changes too fast, the swarms will not have enough time to converge and the rule will not be applied, leaving the mQSO + Rand Rule as a standard mQSO for these cases.

Parameter	**Value**
Min. improvement ratio	15% of historical maximum
Max. low improvement count	5
Percentage for *bad* performance	20%
Min. available time	20% of total duration of θ

Table 3.7 – *Using heuristic rules in a mQSO.* Parameter settings for Rand Rule

3.2.2.3. Both Rule

Finally, a third rule was used, which combines the previous ones (Change Rule and Rand Rule). This combination is very simple: just check if any of the rules can be applied, and if so, do it.

This rule uses the same settings as the Change Rule and Rand Rule combined, which can be seen in their corresponding parameter settings tables (Tables 3.6 and 3.7).

3.2.3. Validation

In order to validate the heuristic rules proposed, we have studied their performance in two different problems: the MPB and the dynamic version of the Ackley function.

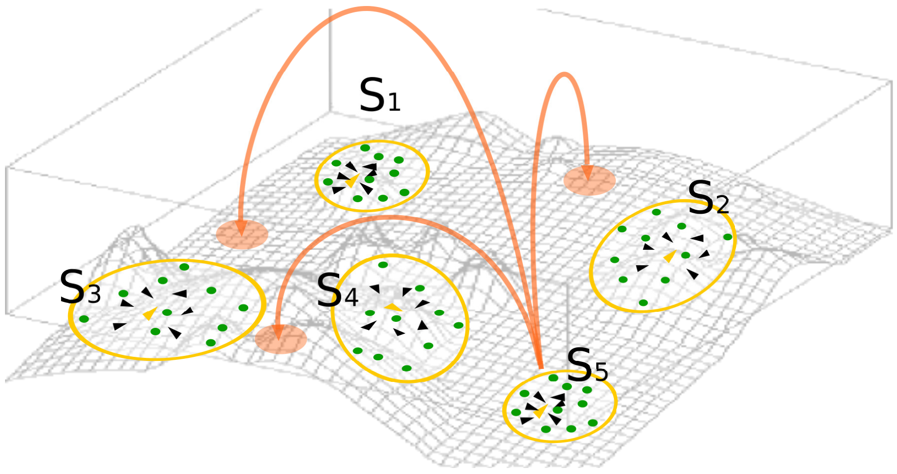


Figure 3.9 – *Using heuristic rules in a mQSO.* Rand Rule explanation: in this example, swarms S_2, S_3 and S_4 have not yet converged. Swarms S_1 and S_5 have converged, and of those, swarm S_5 is in a bad zone (low fitness). It is therefore a candidate for random relocation or pausing.

For each of these problems we have used different scenarios that include variations in the severity (from 2% up to 20%) and the frequency of the change (from 200 evaluations up to 5000), with 5 dimensions. For the case of the MPB, the rest of the parameters are taken from the standard Scenario 2. The new proposals have been compared with the mQSO base algorithm, that has also been executed in the same test scenarios.

For every experiment conducted, the conditions were always the same. The performance measure used was the offline error. Every algorithm was executed for 100 consecutive changes in the environment (we will refer to this as a *run*). In order to obtain statistically meaningful results, the algorithms were executed for 50 independent *runs*, each of them with a different random seed. The comparison of the algorithms was performed using the mean and standard deviation values for the offline error through all these 50 runs.

All mQSO variants were configured with 5 swarms ($m = 5$), each of them containing 10 particles: 8 trajectory and 2 quantum (note that for the cases of *Change Rule* and *Both Rule*, this particle type ratio is varied right after a change in the environment occurs).

The results for the change frequency variations of the MPB are shown in Table 3.8 and Fig. 3.10; the severity variations of the MPB are shown in Table 3.9 and Fig. 3.11; the change frequency variations of the Ackley function are shown in Table 3.10 and Fig. 3.12; and finally, the severity variations of the Ackley function are shown in Table 3.11 and Fig. 3.13.

Change Freq.	mQSO	mQSO Change-Rule	mQSO Rand-Rule	mQSO Both
200	**16.58 (2.09)**	**16.58 (2.09)**	**16.58 (2.09)**	**16.58 (2.09)**
500	12.08 (1.63)	12.03 (1.74)	**10.72 (1.50)**	**10.72 (1.50)**
1000	9.62 (1.07)	9.98 (1.62)	**6.35 (0.74)**	**6.37 (0.75)**
1500	8.88 (1.47)	7.69 (1.05)	**5.24 (0.52)**	6.52 (0.65)
2000	8.22 (1.21)	7.50 (0.73)	**4.60 (0.50)**	7.07 (0.86)
2500	8.12 (1.11)	6.63 (0.71)	**4.17 (0.43)**	6.05 (0.64)
3000	7.63 (1.16)	6.22 (0.66)	**3.93 (0.40)**	5.39 (0.56)
3500	6.93 (1.02)	5.82 (0.62)	**3.73 (0.41)**	4.95 (0.48)
4000	7.23 (1.28)	5.52 (0.70)	**3.55 (0.41)**	4.64 (0.49)
4500	7.06 (1.04)	5.32 (0.59)	**3.42 (0.39)**	4.48 (0.52)
5000	6.87 (0.97)	5.24 (0.58)	**3.25 (0.34)**	4.28 (0.45)

Table 3.8 – *Using heuristic rules in a mQSO.* Results of the **change frequency** variations for Scenario 2 of the **MPB** (off. error mean and std. deviation).

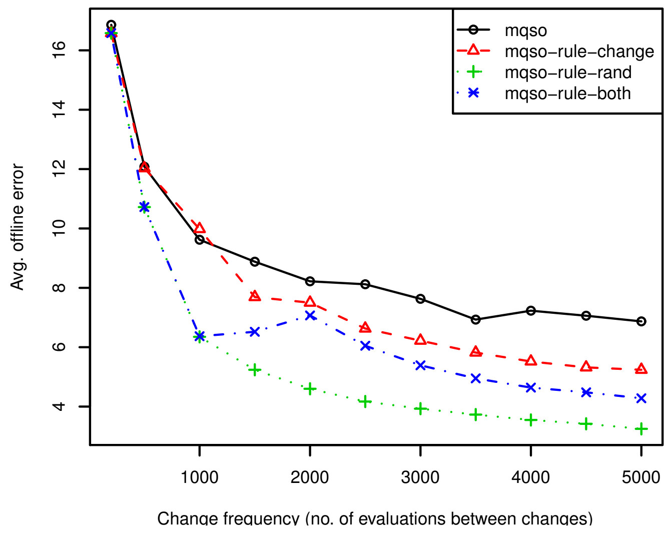


Figure 3.10 – *Using heuristic rules in a mQSO.* Evolution of the offline-error vs. **change frequency** for all the mQSO variants for the Scenario 2 of the **MPB**.

Severity	**mQSO**	**mQSO Change-Rule**	**mQSO Rand-Rule**	**mQSO Both**
2%	6.87 (0.97)	5.24 (0.58)	**3.25 (0.34)**	4.28 (0.45)
4%	6.91 (1.11)	5.96 (0.72)	**3.74 (0.39)**	4.92 (0.51)
6%	6.86 (0.82)	6.32 (0.61)	**4.07 (0.39)**	5.30 (0.54)
8%	6.61 (0.91)	6.70 (0.70)	**4.34 (0.38)**	5.67 (0.59)
10%	6.55 (0.78)	6.85 (0.71)	**4.59 (0.43)**	5.97 (0.61)
12%	6.46 (0.64)	7.06 (0.74)	**4.83 (0.43)**	6.16 (0.65)
14%	6.63 (0.75)	7.31 (0.72)	**5.04 (0.44)**	6.38 (0.55)
16%	6.73 (0.77)	7.46 (0.73)	**5.27 (0.46)**	6.46 (0.55)
18%	6.83 (0.70)	7.55 (0.63)	**5.53 (0.46)**	6.73 (0.60)
20%	6.90 (0.65)	7.69 (0.73)	**5.76 (0.50)**	6.89 (0.59)

Table 3.9 – *Using heuristic rules in a mQSO.* Results of the **severity** variations for Scenario 2 of the **MPB** (off. error mean and std. deviation).

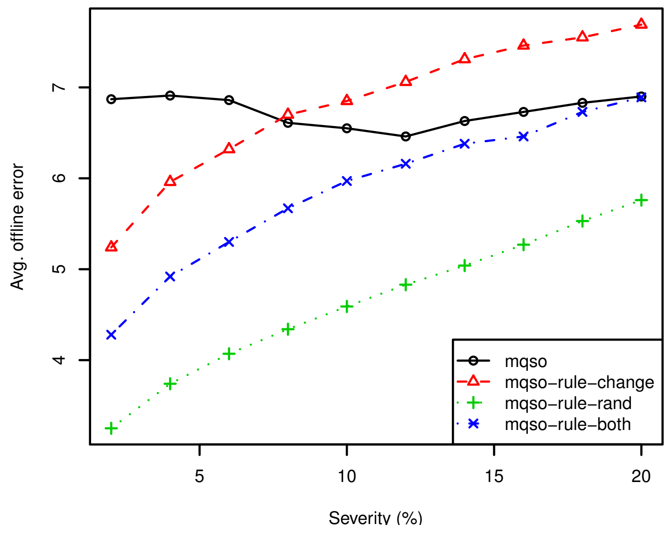


Figure 3.11 – *Using heuristic rules in a mQSO.* Evolution of the offline-error vs. **severity** of change for all the *m*QSO variants for the Scenario 2 of the **MPB**.

Change Freq.	mQSO	mQSO Change-Rule	mQSO Rand-Rule	mQSO Both
200	**5.45 (0.44)**	**5.45 (0.44)**	**5.45 (0.48)**	**5.45 (0.48)**
500	4.28 (0.27)	4.28 (0.28)	**3.86 (0.12)**	**3.89 (0.13)**
1000	3.60 (0.23)	3.53 (0.19)	**3.01 (0.14)**	**3.04 (0.17)**
1500	3.27 (0.18)	3.76 (1.02)	**2.63 (0.13)**	3.11 (0.23)
2000	2.99 (0.21)	3.91 (1.17)	**2.39 (0.14)**	3.00 (0.21)
2500	2.77 (0.15)	3.60 (1.11)	**2.20 (0.14)**	2.64 (0.19)
3000	2.67 (0.21)	3.39 (1.20)	**2.06 (0.11)**	2.39 (0.15)
3500	2.53 (0.16)	3.18 (1.16)	**1.94 (0.11)**	2.19 (0.11)
4000	2.42 (0.15)	2.99 (0.91)	**1.82 (0.11)**	2.07 (0.11)
4500	2.32 (0.18)	2.79 (1.00)	**1.71 (0.09)**	1.90 (0.08)
5000	2.24 (0.18)	2.68 (1.06)	**1.67 (0.11)**	1.78 (0.08)

Table 3.10 – *Using heuristic rules in a mQSO.* Results of the **change frequency** variations for the **Ackley** function (off. error mean and std. deviation).

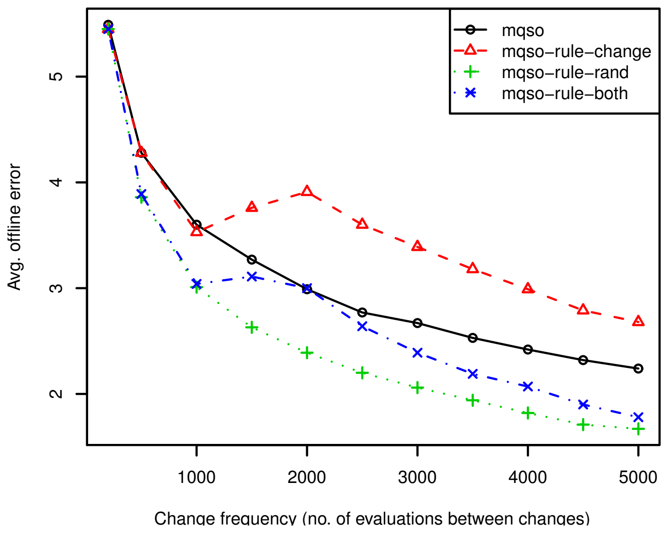


Figure 3.12 – *Using heuristic rules in a mQSO.* Evolution of the offline-error vs. **change frequency** for all the mQSO variants, for the **Ackley** function.

Severity	mQSO	mQSO Change-Rule	mQSO Rand-Rule	mQSO Both
2%	2.24 (0.18)	2.68 (1.06)	**1.67 (0.11)**	1.78 (0.08)
4%	2.78 (0.17)	2.89 (0.48)	**2.05 (0.10)**	2.16 (0.07)
6%	2.94 (0.13)	3.16 (0.44)	**2.14 (0.11)**	2.36 (0.07)
8%	3.01 (0.16)	3.48 (0.58)	**2.24 (0.08)**	2.49 (0.08)
10%	3.07 (0.16)	3.54 (0.40)	**2.30 (0.09)**	2.60 (0.09)
12%	3.15 (0.17)	3.69 (0.42)	**2.41 (0.09)**	2.71 (0.06)
14%	3.23 (0.14)	3.89 (0.57)	**2.49 (0.08)**	2.81 (0.06)
16%	3.28 (0.12)	3.94 (0.35)	**2.58 (0.08)**	2.88 (0.11)
18%	3.40 (0.14)	4.02 (0.31)	**2.65 (0.08)**	2.98 (0.06)
20%	3.45 (0.14)	4.15 (0.41)	**2.76 (0.07)**	3.04 (0.07)

Table 3.11 – *Using heuristic rules in a mQSO.* Results of the **severity** variations for the **Ackley** function (off. error mean and std. deviation).

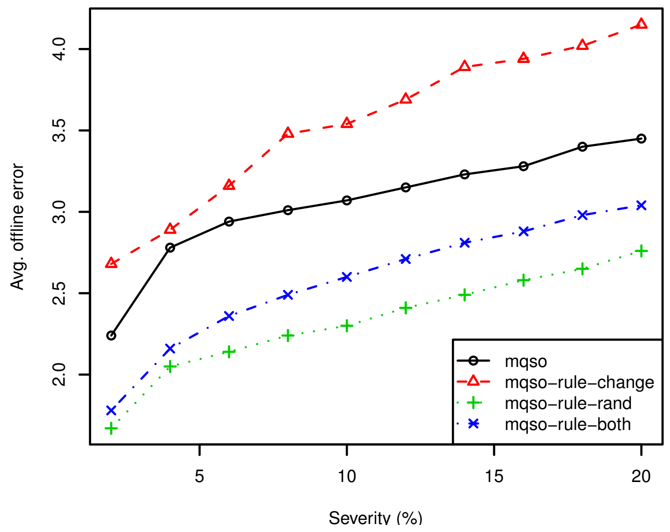


Figure 3.13 – *Using heuristic rules in a mQSO.* Evolution of the offline-error vs. **severity** of change for all the mQSO variants for the **Ackley** function.

3.2.4. Conclusions

The results allow to extract several interesting conclusions. In the first place, the use of the *Rand* rule drastically improves the performance of the mQSO, and it is, by far, the rule that obtains the best results in all the tested scenarios. Secondly, the *Change* rule was not as useful as it was expected. Although it improved the base mQSO in the scenario for which it was conceived (the Scenario 2 of the MPB), in general it obtained worse results than the rest, even worse than the original mQSO, in many experiments. In the third place, the *Both* rule performance was halfway between the first two rules, which suggests that these rules do not combine well, and their behaviours and/or mechanisms are not additive.

The results outstand the importance of testing an algorithm in multiple scenarios: what a priori seemed a good proposal — the *Change* rule —, ended up being a too specific adjustment for the Scenario 2 of the MPB. On the other hand, the good results of the *Rand* rule consolidate the mechanism behind it when facing DOPs as a very effective one: (1) actively monitoring the performance of the different elements of an algorithm; (2) evaluating them according to the average performance of the rest; and (3) correcting, or even stopping, those elements that are doing it worse.

The analysis of the particle properties was published in [41]:

"An Analysis of Particle Properties on a Multi-swarm PSO for Dynamic Optimization Problems", I. G. del Amo, D. A. Pelta, J. R. González, and P. Novoa, in *Current Topics in Artificial Intelligence* (P. Meseguer, L. Mandow, and R. Gasca, eds.), vol. 5988 of *Lecture Notes in Computer Science*, pp. 32–41, Springer Berlin / Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-14264-2_4`.

and the research on the use of heuristic rules was published in [39]:

"Using heuristic rules to enhance a multiswarm PSO for dynamic environments", I. G. del Amo, D. A. Pelta, and J. R. González, in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pp. 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586051`.

3.3. Agents for DOPs

In this section we present a DOP variant of a previous algorithm based on the decentralized cooperation of a group of agents. We will briefly describe a first implementation of this Agents algorithm for *continuous* DOPs published in [69] (we will talk about this work in Sect. 3.4), and we will go into more details about a second implementation for *discrete* DOPs, published in [68].

3.3.1. Motivation

In previous works [125, 126] we developed an algorithm named Agents that consists in a set of agents that traverse a fixed-size matrix of solutions, trying to improve each solution they arrive to in each step. Using this scheme, an agent can improve a solution that was previously modified by another agent, thus producing an implicit form of cooperation. This algorithm was originally applied to *static* optimization problems, but everything seemed to indicate that adapting it to DOPs was a feasible task.

Therefore, we proposed a first version of this method for continuous DOPs in [69], a work that we will discuss in Sect. 3.4. In that section, we will focus on the CS algorithm, and here we will give the details of the Agents implementation used for the said work.

Additionally, up until now all the works that we have presented in this chapter were focused in *continuous* DOPs. However, there are quite a high number of *discrete* — or *combinatorial* — DOPs where the techniques and algorithms that we have developed can be applied. The most used methods in the discrete DOPs area are, by far, evolutionary algorithms. Thus, we believe that the introduction of new methods can be quite beneficial. Among all the algorithms that we have developed, the one that we think that can be better adapted to this kind of discrete problems is Agents. Therefore, we hereby present an initial adaptation of Agents to combinatorial DOPs. Additionally, we wanted to investigate the possibility of the algorithm's parameters being self-adaptable, in order to avoid their configuration. For this purpose, a learning mechanism is necessary, and we therefore introduce a novel learning scheme for these problems. This scheme has been implemented separately in the Agents algorithm and in a basic Evolutionary Algorithm (EA), in order to independently assess the benefits of the learning scheme. The work of the Agents algorithm for combinatorial DOPs was published in [68].

3.3.2. Proposal

The Agents algorithm is a decentralized cooperative strategy that was originally described in [125, 126]. It consists in a set of agents that traverse a fixed-size matrix, or *grid*, of solutions, trying to improve each solution they arrive to in each step. The way in which these solutions are improved depend on the neighbors of the cell in the grid, and the structure of the grid determines the topology of the *neighborhoods*. In the implementation, a Moore neighborhood topology is used, where a solution in the cell (i, j) is neighbor of $(i-1, j-1)$, $(i-1, j)$, $(i-1, j+1)$, $(i, j-1)$, $(i, j+1)$, $(i+1, j-1)$, $(i+1, j)$ and $(i+1, j+1)$. Please note that with this definition, close solutions in terms of grid neighborhoods do not necessarily imply closeness in terms of solution-space distances. A graphical explanation of these concepts is shown in Fig. 3.14.

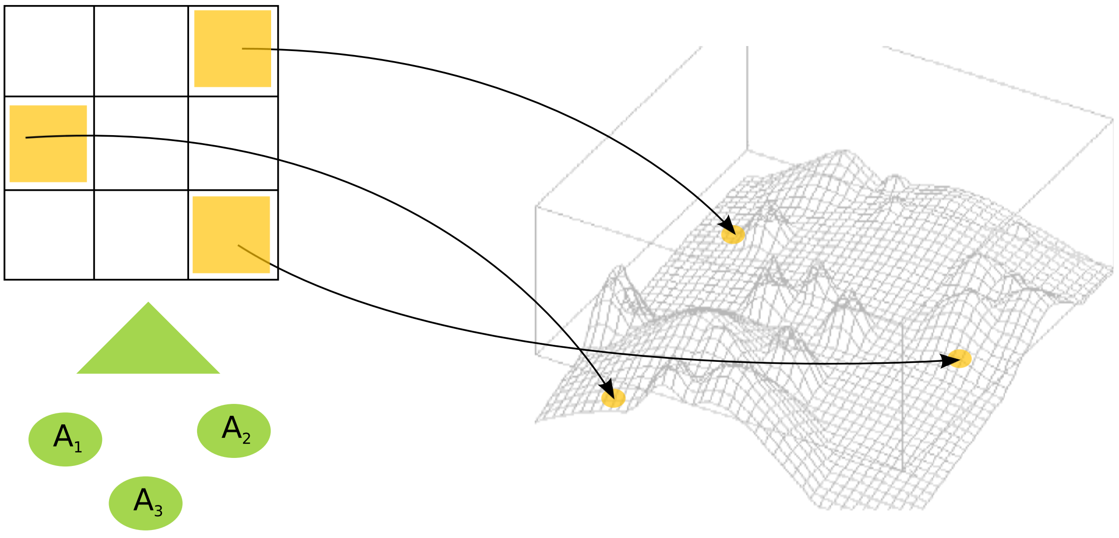


Figure 3.14 – *Agents for DOPs.* A representation of the working mechanism of the Agents algorithm: the agents move in a grid where each cell represents a solution of the problem; as agents traverse the grid, they alter and improve the solutions.

Agents can be seen as a population-based decentralized cooperative algorithm because the improvements of an agent are immediately available to the others. When an agent is moving to the best neighbor solution of the grid, it will be attracted by good grid-cell solutions that were stored by other agents. Therefore, as the agents move through the grid, they will tend to act on those cells that other agents have already improved.

Algorithm 3.2 presents the pseudocode of the main procedure of Agents. Basically, the algorithm is run until the stop condition is met, which will normally be when the resources are exhausted (such as the time or the number of evaluations/changes of the objective function). For each iteration, the function *detectChanges()* on line 4 is called. The change detection is done by recomputing the fitness value of the best solution and comparing it with the previous one to

Algorithm 3.2: Agents

```
1  initializeParameters(grid);
2  initializeParameters(agents);
3  while stopping condition is not met do
4      detectChanges() ;
5      foreach agent in grid do
6          if there is a better cell in neighborhood(agent) then
7              agent.position ← best cell in neighborhood(agent);
8          else
9              agent.position ← random cell in neighborhood(agent);
10         end
11         prevSolution ← solution at agent.position;
12         newSolution ← mutation(prevSolution) ;
13         evaluate(newSolution) ;
14         if newSolution improved prevSolution then
15             solution at agent.position ← newSolution;
16             bestSolution ← newSolution;
17         end
18     end
19 end
```

see if it has changed. If a change is detected, all the solutions of the grid are reevaluated.

After this, each agent is executed, moving it to the best neighbor solution (in terms of horizontal and vertical adjacency on the matrix) and trying to improve the solution in that cell. Since the algorithm performs as many iterations of the *foreach* loop as the number of agents, this loop is similar to a generation in other population-based algorithms (this fact is important for comparison purposes against this type of methods, specially for the work on discrete problems that will be explained later).

To improve a solution, the agent uses the *mutation()* function on line 12. This function is applied to *prevSolution* to generate *newSolution*. This *newSolution* will replace *prevSolution* on the matrix if an improvement is produced. The *mutation()* function is the place where optimization techniques can be implemented in order to increase diversification or intensification. This is also the function that contains the different strategies for dealing with continuous vs. discrete DOPs.

3.3.2.1. Agents for continuous DOPs

In the case of the Agents algorithm for continuous DOPs introduced in [69], the implementation of the *mutation()* function simply generates a new random solution inside a hypershpere of radius *perturbationRadius* around the original solution.

3.3.2.2. Agents for discrete DOPs

Regarding the Agents for discrete DOPs introduced in [68], there are several differences with respect to the continuous DOPs version. The first one is that solutions with equal fitness are also accepted as replacement solutions for the matrix, and not only better ones. The reason for this change is that on many combinatorial problems the fitness values vary on a discrete manner, so it is more common to reach solutions with the same objective value. If the solutions of the matrix were not replaced with the new ones in this case, the modification effort would be lost, and the search process could stagnate. When we allow the replacement of solutions with the same quality, we allow the search process to evolve so it is able to escape from flat zones on the search landscape.

The second difference is the implementation of the *mutation()* function itself. Generating a random solution inside a hypersphere is no longer viable for a discrete/combinatorial problem. In this case, the implementation done for *mutation()* is focused on the binary-encoded problems we are going to tackle (Sect. 3.3.3) and it allows to perform changes to a given number of consecutive bits (*numBits*) by flipping each of these bits with a given probability ($flipProb$). The starting bit for the *numBits* will be chosen randomly and if the last bit of the solution is reached, the remaining bits are taken from the initial solution bits (in a circular fashion). Each of the *numBits* will then be flipped independently with $flipProb$ probability. In this way, if $numBits = 1$ and $flipProb = 1$, the *mutation()* function will change just one random bit of the solution. If $numBits = 3$ and $flipProb = 0.8$, the *mutation()* function will choose three consecutive random bits and flip each one with an independent 80% probability. The full settings used for the mutation operator are summarized in Table 3.12.

Additionally, as we anticipated in Sect. 3.3.1, the use of fixed parameters for the configuration of an heuristic does not lead to robust results on different problems and instances. A detailed description of different approaches for adaptive evolutionary algorithms on combinatorial problems can be seen on [146]. While most of the ideas presented in that paper are feasible to be applied also to *Agents*, additional challenges appear when we are dealing with DOPs. Since DOPs change with time, it makes it more difficult to properly find out the best parameter settings for the whole search process. Moreover, it is generally not possible to test all

Variant	**Configuration**
MutOp1	$numBits = 1$ and $flipProb = 1$
MutOp2	$numBits = 2$ and $flipProb = 0.9$
MutOp3	$numBits = 3$ and $flipProb = 0.8$
MutOp4	$numBits = 4$ and $flipProb = 0.7$
Adaptive	Uses the adaptive scheme on all the previous variants

Table 3.12 – *Agents for combinatorial DOPs.* Mutation operator variants.

values for every parameter when the available time between two consecutive problem changes is short. Despite that, the use of some adaptive scheme will probably improve the robustness of the algorithms and one of the goals of the Agents for discrete DOPs work is to verify this claim.

Thus, we have implemented a credit-based adaptive scheme in a generic *Learning* library to discriminate among a set of configurations (values) of a given criterion. Each configuration of the criterion will have an associated index or value in the natural numbers. To implement this task, the library contains three main methods:

- **learn(value, credit)**. Assigns an additional credit to the configuration of the criterion represented by the value.
- **rouletteWheelSelection()**. Applies roulette wheel selection to return one of the learned values. That is, the probability of choosing each value corresponds with the quotient between its credit and the sum of credits for all the values of the criterion.
- **clearLearning()**. Deletes learned values and credits.

Since it is not realistic to learn every possible parameter of the algorithms, our approach will be to focus the learning on the configuration of the *mutation()* function. To achieve this, what we do is to consider the selection of the configuration to use for the mutation operator as the criteria and to assign a numerical value to every configuration variant that is going to be considered.

Initially, we will set the same credit for each configuration variant in order for all of them to have an equal non-zero chance of being selected. Then, each time the mutation operator is applied with a given operator configuration, the fitness change between the received solution and the mutated solution is computed. If the fitness was increased, the increment is added as an additional credit for that operator configuration (using the *learn* function). Besides that, to try to adjust the learning to the dynamic problem changes, when a change on the environment is detected, all the credits are restored to their initial values using a call to *clearLearning()* and repeating the initialisation.

3.3.3. Validation

The validation of the Agents proposal for *continuous* DOPs will be discussed in Sect. 3.4.3. Thus, we will not give the details of the said experimentation here, and we will instead focus on the Agents for *discrete* DOPs.

In order to assess the discrete version of Agents, we have tested it over a set of binary-encoded combinatorial DOPs, using the XOR generator technique [184]. These problems were explained in Sect. 2.1.2.3, and the objective is to match a set of bits, were each group of 4 bits is evaluated according to the function being used:

- **OneMax**: Each matched bit adds 1 to the fitness.
- **Plateau**: Three matched bits add 2 to the fitness while four matched bits add 4 and any other amount of bits matched leads to a 0 contribution.
- **RoyalRoad**: Each perfectly matched block adds 4 to the fitness. Partially matched blocks have 0 fitness.
- **Deceptive**: Fitness is 4 if all the 4 bits are matched. If not, the fitness for the block is 3 minus the number of matched bits.

Please see Fig. 2.3 for a graphical explanation. Remember that the XOR generator introduces dynamism every certain number of function evaluations by changing ρ bits of the target optimal solution ($\rho =$ severity of the change).

The experiments carried out over these problems had the objective of assessing the validity of both the Agents algorithm for combinatorial DOPs and the learning scheme previously mentioned. Therefore, we have structured the experimentation in two stages:

1. To test the learning scheme, we have incorporated it to the Agents algorithm, producing a version named *adaptive*. We have executed this adaptive version, as well as the versions corresponding to the different mutation operators (Table 3.12) over multiple scenarios of the OneMax, Plateau, RoyalRoad and Deceptive problems, and we have compared the results to see which one was the best. Additionally, we introduced this same learning scheme into a simple Evolutionary Algorithm (EA), and we have repeated the experimentation conducted with the Agents, in order to have an additional source of information.

2. Afterwards, we have executed the best algorithm of each of the previous (the best Agents and the best EA, which happened to be *adaptive*, the one with the learning scheme) against one of the most recent algorithms in the state of the art in these problems: the Adaptive Hill-climbing Memetic Algorithm (AHMA) [163]. The conditions of this second experimentation were the same as in the first one, but instead of comparing the results of a single algorithm against different versions of itself, we compared one single version of 3 algorithms against each other.

All the problems were configured to use a solution size of 100 bits (25 blocks of 4 bits). For each problem, the experiments performed consider different periods of change ($P \in \{1200, 6000, 12000\}$) and different severities ($\rho \in \{0.1, 0.2, 0.5, 0.9\}$). The period ($P$) is given as the number of evaluations of the objective function between two consecutive changes. The severity of change (ρ) is used when a change is produced (every P evaluations) to control how many bits of the optimal solution are flipped. If l is the length of the solutions for the problem, the new optimal solution after a change will be generated flipping $\rho * l$ bits on the previous target optimal solution.

Regarding the configuration of the algorithms, we chose several values of the number of agents and the dimension of the matrix for the *Agents* algorithm on the basis of experience and the statistical results published for the continuous DOPs case [69, 125, 126]. These values were tested again for the combinatorial DOPs and the statistical analysis showed that the best configuration overall was to use 4 agents and a matrix of size 2×2. These results are similar to the values obtained for the continuous DOPs but with a smaller matrix size, that can be explained by the nature of the problems tested, where it is probably better to do a good intensification than to have a big diverse population. Moreover, the use of just 10 individuals was proven to be statistically better in the case of the *EA*. The crossover and elitism chosen required no parameters, and since the population size was small all the individuals were used for performing the negative/positive assortative mating selection. Finally, since the problems used here correspond to the paper where *AHMA* was published, we simply set the exact same original parameter values published on [163].

To assess the performance of the algorithms, we have used the off-line performance [28] that we defined in Sect. 2.2.3.

For the first experiment, we have considered all the mutation operator variants that are shown on Table 3.12, each of which uses a different configuration of the solution mutation operator (see the description of the *mutation* function on Sect. 3.3.2.2). The values for the operators have been selected with an increasing $numBits$ and a decreasing $flipProb$. In this way, as the number of consecutive bits affected gets bigger, the configurations allow for a higher probability of leaving some of the affected bits unchanged. The last variant (*Adaptive*) is a special variant that, instead of using just a single configuration, uses all of them coupled with the adaptive scheme (Sect. 3.3.2.2). Additionally, when using the adaptive scheme, the initial credit assigned to each operator was set to the 10% of the sum of the fitness of solutions in the current population plus the 10% of the number of bits on a solution. This last addition is only included to guarantee that a non-zero initial credit is assigned to each operator configuration even when all the solutions of the population have 0 fitness. Since the time for learning is limited by the change period for the problems, we have reduced the mutation operator variants to just 4 configurations, so it becomes easier to learn the proper operator to use in a faster way. In this way, there are less values to learn, but we still have several different mutation operator configurations and it is expected that the learning will find the best performing ones for each algorithm / problem.

Finally, since the results for every possible algorithm, problem configuration and mutation operator variant would be difficult to interpret if displayed as a numerical table, we have used the SRCS technique [38] that will be explained later in Chapter 4. The idea is to compare algorithms using the offline performances for every run of each algorithm and configuration. As recommended in [9,63], first we test for differences among the results of *all* algorithms for a given configuration, using a non-parametric Kruskal-Wallis rank-sum test [89], with a significance level of 0.05. If there is enough evidence, then a second non-parametric test, the Wilcoxon's rank-sum test [168], with a significance level of 0.05 is performed for *each pair* of algorithms. This test allows to assess if there are significant differences between any two samples. If the test concludes that there are such differences, the algorithm with the highest overall offline performance adds 1 to its rank, and the other adds -1. In case of a tie, both receive a 0. The range of the rank values for n algorithms for any specific problem, period and severity will therefore be in the $[-n+1, n-1]$ interval. The higher the rank obtained, the better an algorithm can be considered in relation to the other ones. These ranks will be displayed as colors with the highest rank value $(n-1)$ being displayed as white and the lowest rank value $(-n+1)$ being painted as red. The remaining rank values will be assigned a gradient color between white and red. If we then group together the ranks of an algorithm for a given problem with every possible different period and severity we can obtain a colored matrix, where it is easy to get a good idea of how the algorithm performs for that specific problem.

The results are arranged in a table-like way, with the columns indicating the problem, and rows indicating the algorithm (or version of the algorithm). **We will therefore look for white or yellow cells, meaning that the algorithm in that row is the best or one of the best for that particular problem configuration**. **Red cells mean just the opposite, that the given algorithm**

obtains the worst results.

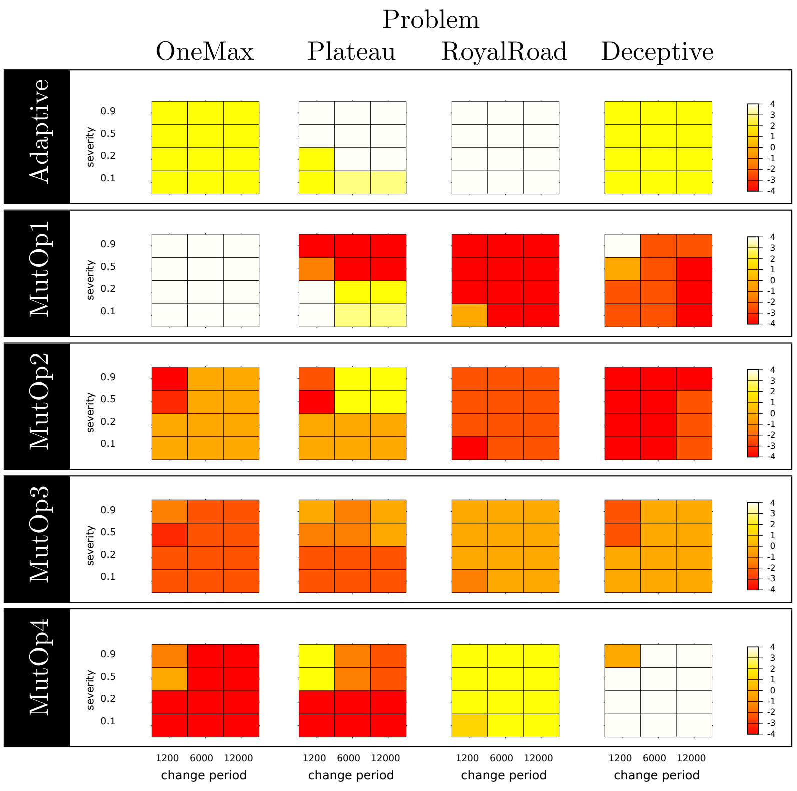


Figure 3.15 – *Agents for combinatorial DOPs.* Rank results for all the *Agents* variants.

Regarding the first experiment, the results for the comparison of the different versions of the Agents algorithm are shown in Fig. 3.15, and the results for the EA versions in Fig. 3.16.

The results for the second experiment, where the *adaptive* version of both Agents and EA was executed in the same problems along with the AHMA algorithm, are displayed in Fig. 3.17.

3.3.4. Conclusions

3.3.4.1. Agents for continuous DOPs

Regarding the Agents algorithm for continuous DOPs, as we will see in Sect. 3.4.3, this proposal did not obtain as good results as the mQSO or the CS methods. Initially, we concluded that a reason for this could be that it was the first adaptation of the Agents algorithm to DOPs, and that the implementation of the mutation

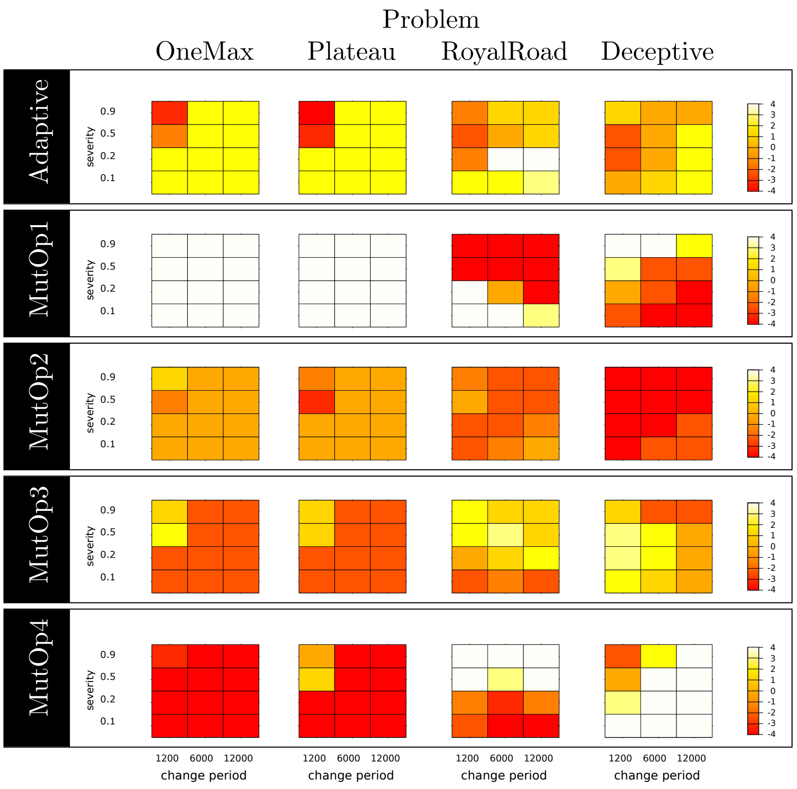


Figure 3.16 – *Agents for combinatorial DOPs.* Rank results for all the *EA* variants.

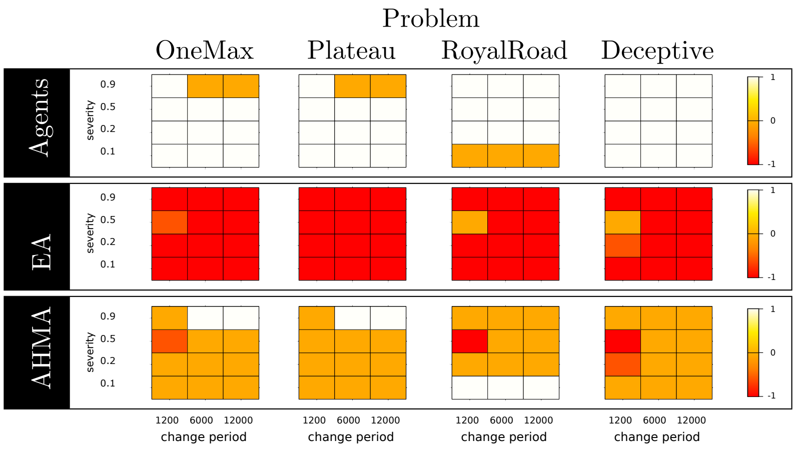


Figure 3.17 – *Agents for combinatorial DOPs.* Rank results of the Adaptive Agents and EA variants against AHMA.

function that we had chosen (a simple random perturbation of a solution within a hypersphere) was not powerful enough to outperform the aforementioned algorithms in these problems.

However, as it will be shown in Sect. 4.4, the Agents algorithm is indeed capable of obtaining very good results. It is just that in [69] we did not use the test scenarios in which this algorithm could exhibit its full potential (actually, high-dimensional scenarios for the MPB case). We will discuss these results in more detail later, in Sect. 4.4.

The results of the Agents implementation for continuous DOPs were published in [69]:

"A cooperative strategy for solving dynamic optimization problems", J. R. González, A. D. Masegosa, and I. G. del Amo, *Memetic Computing*, vol. 3, no. 1, pp. 3–14, 2010. `http://dx.doi.org/10.1007/s12293-010-0031-x`.

3.3.4.2. Agents for discrete DOPs

Regarding the Agents algorithm for discrete DOPs, we performed two experiments: a first one in which we intended to validate the learning scheme proposed, and a second one in which we compared a version of Agents and an EA using this learning scheme against the AHMA.

The results of the first experiment show that some fixed mutation operators are very well suited for certain problems (e.g., MutOp1, which flips only 1 bit, and obtains the best results for the OneMax problem), but perform quite bad in others (in this case, the MutOp1 was one of the worst for the RoyalRoad and Deceptive problems). However, the learning scheme (the *adaptive* version of the Agents and EA algorithms) is much more stable, always obtaining results among the first positions in all problems. This situation is specially noticeable in the Agents algorithm (Fig. 3.15), where the adaptive version always obtains the best or the second best results. Therefore, we can conclude that the learning scheme proposed significantly improves the results for this kind of problems.

Moreover, in the second experiment, it can be clearly seen that the Agents algorithm outperforms the others in almost all tested scenarios (Fig. 3.17). Considering that AHMA is one of the state-of-the-art algorithms for combinatorial DOPs, these results open a very promising research path.

The results of the Agents algorithm for discrete DOPs were published in [68]:

"An Adaptive Multiagent Strategy for Solving Combinatorial Dynamic Optimization Problems", J. González, C. Cruz, I. G. del Amo, and D. Pelta, in *Nature Inspired Cooperative Strategies for Optimization (NICSO 2011)* (D. Pelta, N. Krasnogor, D. Dumitrescu, C. Chira, and R. Lung, eds.), vol. 387 of *Studies in Computational Intelligence*, pp. 41–55, Springer Berlin / Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-24094-2_3`.

3.4. Cooperative Strategies for DOPs

In this section we present a novel algorithm for DOPs based in Cooperative Strategies: trajectory metaheuristics that operate in an organized way thanks to a central coordinator.

3.4.1. Motivation

In the area of DOPs, most of the algorithms used are based on populations of solutions, but trajectory-based methods have received little attention. It is also worth noting that the role of cooperation in population-based algorithms is most of the time implicit, so we cannot objectively evaluate if this cooperation is beneficial when facing a DOP.

In previous works [35, 103, 127] we developed a method for *static* optimization problems based on a set of heuristics or *solvers* that cooperate among them thanks to a central coordinator. The said coordinator keeps a list of the local optima found by the solvers, and uses a rule base to analyze the performance of each of them and correct the behaviour of those with worst results. We denominate this method Cooperative Strategies (CS).

The objective of this section is to present an adaptation of the CS algorithm to DOPs, using trajectory-based techniques as solvers, and more precisely, tabu search methods. Thus, we have been able to verify if these methods can be effectively applied to DOPs. Furthermore, since the CS uses an explicit cooperation mechanism, we could evaluate the role of such scheme in these type of problems.

Additionally, we had studied different rules that can be used by the coordinator when correcting the behaviour of solvers, and we will show which are the most effective ones in the test problems used.

3.4.2. Proposal

The CS algorithm consists in a set of solvers/threads, where each solver implements a tabu search algorithm. The coordinator processes the information received from them and produces subsequent adjustments of their behaviour by sending "orders". To achieve this exchange of data, a blackboard model is used [57]. In the implementation, two blackboards are available: one where the solvers write the reports of their performance and the coordinator reads them, and another, where the orders are written by the coordinator and read by the solvers. This working scheme is summarized in Fig. 3.18.

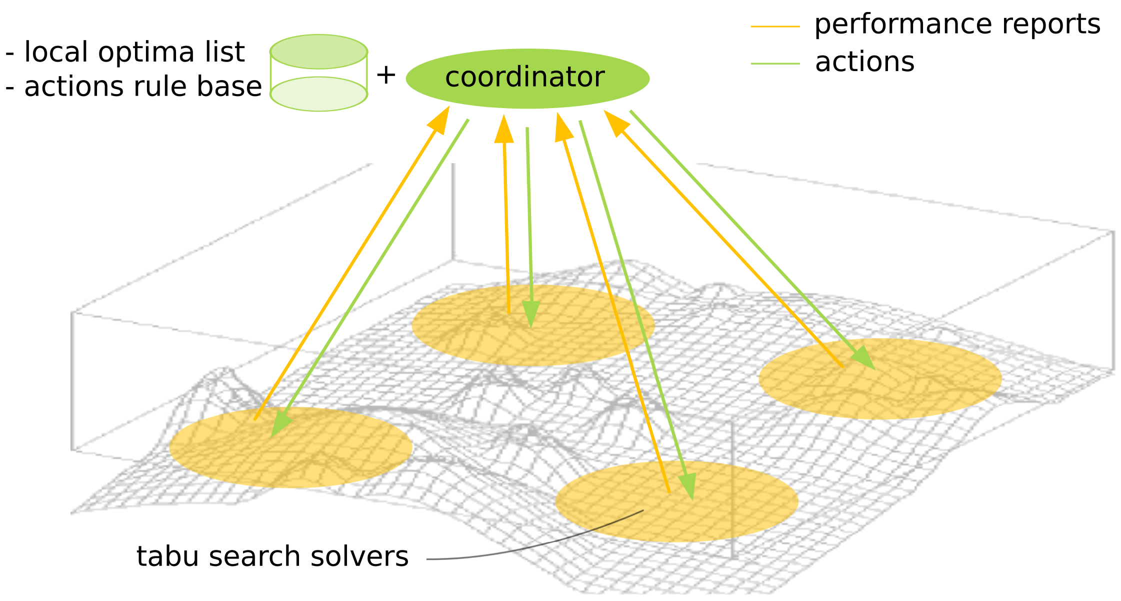


Figure 3.18 – *CS for DOPs.* A representation of the working mechanism of CS: a group of tabu search solvers work together thanks to a central coordinator that uses performance information and a rule base to modify the parameters of the solvers.

The CS starts with an initialization stage where parameters are given their starting values. After that, all solvers asynchronously execute the tabu search algorithm while sending and receiving information. The coordinator checks through the input blackboard which solver provided new information and decides whether its behaviour needs to be adapted using a rule base. If this is the case, it will calculate a new behaviour which will be sent to the solvers through the output blackboard. As for the solvers, once execution has begun, performance information is sent and adaptation orders from the coordinator are received alternatively.

Regarding the tabu search implemented by the solvers, we should point out that an important issue in continuous optimization is to determine the size of the movement when we search for better solutions around a specific point. A small step size can lead to a significant waste of objective function evaluations, whereas a big movement length makes it difficult to find solutions with enough accuracy. For this reason it is interesting to use a large step size at the beginning of the search in order to perform a better exploration of the solution space, and then reduce the step size to obtain a better accuracy. In this way, in our implementation of the tabu search, we start with a step size of length δ_{init} and every time the exploration of the neighbourhood of the current solution does not lead to a better one, this size is halved. When two consecutive steps get to improve the current solution the step size is multiplied by two. The length of the movement is delimited by the interval $[res, \delta_{init}]$, where res is a parameter which determines the precision of the tabu search solvers.

The pseudocode for the tabu search of the solvers is presented in Algorithm 3.3, assuming a *minimization* problem (for maximization problems, expressions like $f(x) < f_{best}$ should be replaced by $f(x) > f_{best}$). The method starts with an initialization stage, after which a loop is repeated until the stop condition is fulfilled. Within this loop, the procedure checks if the step size has been reduced to a value lower than res. If so, it determines that the current solution is a local minimum and checks if it is near to a previously visited local minimum (the distance between them is lower than LRR). To escape from it, the method applies a diversification method performed according to procedure 2.1 of [77], in order to restart the search in a non explored region, or to allow a certain number of non improvement movements, respectively. After this, the method explores the neighbourhood and checks if the chosen neighbour is better than the best solution found so far. In this case, the step size is increased if two consecutive improvements have taken place. Otherwise, it is halved.

The tabu search relies on a procedure for exploring the neighbourhood of a solution. Here we use three different ways to explore it:

1. **Optimistic search:** In the first place, the algorithm tries a step in the last good direction, i.e., the direction chosen in the previous neighbourhood exploration, expecting this trajectory to still be good in the next iteration.

2. **Approximate descent direction (ADD):** If the last procedure does not lead to an improvement, then the heuristic attempts to find a good descent direction using the ADD method [76].

3. **Tabu exploration:** In the case that the two former movements do not work, the algorithm carries out a more exhaustive search within the neighbourhood generating $2n$ neighbours. More precisely, for each one of the n vectors that compose the basis of the vector space defined by the problem, we take two solutions in this direction, one in the positive sense and one in the

Algorithm 3.3: Pseudocode of continuous tabu search

```
/* Initialization                                                    */
$\delta = \delta_{init}$;
$x = new\_solution$;
$x_{best} = x$;
$f_{best} = f(x_{best})$;
$consAdvances = 0$;
while !$stopCondition$ do
    if $\delta < res$ then
        $\delta = \delta_{init}$;
        if $isNearLocalMinimum(x_{best})$ then
            /* If $x_{best}$ is near a local optimum, apply the
               diversification procedure                             */
            $x = diversification(x_{best})$;
            $x_{best} = x$;
            $f_{best} = f(x_{best})$;
            $consAdvances = 0$;
        end
        else
            /* $x_{best}$ is considered as a new local minimum            */
            $acceptNoImp = maxNoImp$;
            $addLocalMinimum(x_{best})$;
            $f_{best} = \infty$;
            $consAdvances = 0$;
        end
    end
    $x = exploreNeighbourhood(x, acceptNoImp, \delta)$;
    if $f(x) < f_{best}$ then
        $x_{best} = x$;
        $consAdvances = consAdvances + 1$;
        if $consAdvances == 2$ then
            $\delta = increaseStep(\delta)$;
            $consAdvances = 0$;
        end
        else
            $\delta = reduceStep(\delta)$;
            $acceptNoImp = acceptNoImp - 1$;
        end
    end
end
```

Algorithm 3.4: $exploreNeighbourhood(x, acceptNoImp, \delta)$

```
1  $x_{new} = optimisticMovement(x, lastMovement, \delta)$ ;
2  if $f(x_{new}) < f(x)$ then
3      return $x_{new}$;
4  end
5  $x_{new} = ADD(x, \delta)$ ;
6  if $f(x_{new}) < f(x)$ then
7      return $x_{new}$;
8  end
9  $x_{new} = tabuExploration(x, \delta)$ ;
10 if $acceptNoImp == 0 \quad AND \quad f(x_{new}) > f(x)$ then
       /* If no more non-improving movements are allowed, stay in
          the same point hoping that reducing the step size will
          allow to reach a better solution                        */
11     return $x$ ;
12 end
13 else
14     return $x_{new}$ ;
15 end
```

negative. Then, the best non-tabu move (or the best tabu move, if it fulfills the aspiration level), is taken as the new current solution of the method. The tabu list is composed of the reverse of the movements previously accepted, and a movement becomes non-tabu after a given number of iterations defined by the parameter *tenure*.

The pseudocode of the neighbourhood exploration procedure is given in Algorithm 3.4.

As it has been previously said, the solvers periodically send reports to the coordinator. These reports contain the following data:

- Solver identification.
- A time stamp t.
- The current solution of the solver.
- The best solution found by the solver s_{best}.
- A list with the local optima found by the solver since the last report.

The list of local optima sent by the solver is processed by the coordinator, which keeps the history of all local optima found by the solvers in a memory denominated Visited Region List (VRL). These local optima are grouped within regions defined by a hypersphere with radius ρ and centre in the points stored in the VRL. Each entry of the VRL also maintains a register φ with the frequency of visiting that region by any search thread. These regions are used by some of the rules that will be explained in Sect. 3.4.2.1. For a graphical explanation of local optima as they are used by solvers, and the visited regions kept by the coordinator, please refer to Fig. 3.19.

To deal with DOPs, the CS algorithm must check if the fitness function has changed, and in such case, restart the memories that do not contain relevant information for the new search space. These issues are handled in both the coordinator and solver sides through a local count of the number of fitness changes detected on each thread. If the count of detected fitness changes differs between a solver and the coordinator, the higher count is communicated and the search process is adapted accordingly.

The CS is composed by 12 solvers that differ in terms of the length of the initial step size, δ_{init}, (three different values for this parameter are considered) as well as the initial solution. The parameter setting for the tabu search are displayed in Table 3.13, where σ is the diameter of the variable range, that is, the maximum of the difference between the lower and the upper limit of each variable.

Regarding the parameters of the coordinator, the radius for the visited regions ρ was set to 0.15σ and the antecedent threshold $\lambda_{reaction}$ to 1.

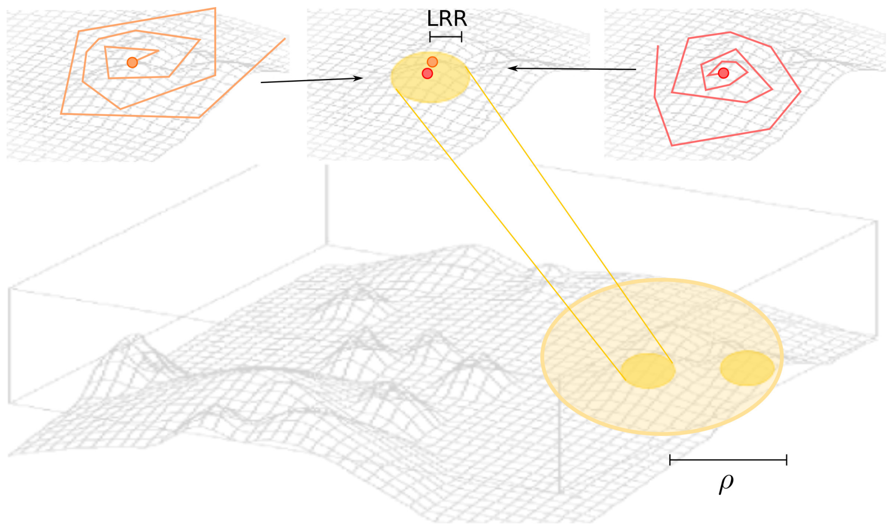


Figure 3.19 – *CS for DOPs.* A trajectory solver may arrive to different solutions close to each other after two search stages (upper-left and upper-right images). If these two solutions are within a distance of LRR units, the solver considers that it has arrived to the same local optimum (upper-center image). Two local optima are considered to be in the same region if they are within a distance of ρ units (lower image).

Parameter	**Description**	**Value**
δ_{init}	initial length of the operator step	$\sigma/2, \sigma/3, \sigma/5$
res	minimal step value	0.001
LRR	radius for the local minimum regions	0.01σ
maxNoImp	max. number of non-improving movements	3
tenure	length of the tabu list of movements	1

Table 3.13 – *CS for DOPs.* Parameter settings for the tabu search solvers. The different values set for the parameter δ_{init} of each solver are also displayed. The parameter σ is the diameter of the variable range, that is, the maximum of the difference between the lower and the upper limit of each variable.

3.4.2.1. Cooperation modes

The behaviour of the solvers is controlled by the coordinator by means of a set of cooperation rules, with the form ***if*** *condition* ***then*** *action*. These rules allow the coordinator to determine if a solver is behaving correctly as well as the action that should be performed to adjust such behaviour. All the rules used in this work share a common antecedent (condition) based on Reactive Search ideas [10]. These rules have therefore the form:

IF	φ of the last local optimum visited by $solver_i$ is bigger than $\lambda_{reaction}$
THEN	*action*

(3.1)

where φ is the frequency with which $solver_i$ visited its last local optimum, and $\lambda_{reaction}$ is a threshold that regulates the activation of the rule. The action will be used to adapt the behaviour of $solver_i$, leading to different types of cooperation depending on the specific action taken.

All the rules used by the coordinator follow the general definition given on Expression 3.1, and only differ on the consequent, which uses different actions for producing different modes of cooperation. In this case, we will consider four different actions. The first two lead to cooperation rules that induce a higher intensification of the search, although with different degrees. The remaining two actions have a higher exploration balance, also with different degrees. Their descriptions are given below:

- **Best solution (BS)**: We can consider this action as the simplest one. It consists on sending a neighbor of the best solution ever seen by the coordinator to the solver. In this way, we concentrate the threads around promising regions of the search space.

- **Approaching (AP)**: This action tries to bring solvers with bad performance near to the best thread, placing it in a intermediate point between the best solution of both solvers. This point is generated applying a crossover operator to these solutions.

- **Reactive (R)**: This consequent for the coordinator's rule was previously presented in [103]. When it is executed, the coordinator sends to the specific solver the best global solution perturbed by a degree ϕ. ϕ is given by the next formula:

$$\phi = \begin{cases} 0, & \text{if } \varphi - \lambda_{reaction} \leq 0 \\ \varphi - \lambda_{reaction}, & \text{if } \varphi - \lambda_{reaction} > 0 \text{ and} \\ & \quad \varphi - \lambda_{reaction} < \phi_{max} \\ \phi_{max}, & \text{if } \varphi - \lambda_{reaction} \geq \phi_{max} \end{cases}$$

where φ is the frequency of visiting the region of the last minimum found by the solver, $\lambda_{reaction}$ is the threshold that establishes when a higher mutation than the basic one ($\phi = 0$) is applied, and ϕ_{max} is the maximum perturbation degree. The higher the value of ϕ is, the bigger is the generated distance between the optimum and the sent solution.

- **Visited Region List (VRL)**: This action makes use of the VRL to reallocate the solver in a point outside of the previously visited regions. The procedure followed to generate these points has been taken from the work [77]. To avoid generating solutions near to the more frequently visited regions, the next function is used:

  $$\Psi(\varphi) = \gamma(1 - e^{\gamma(\varphi-1)})$$

  where $\gamma \in (0, 1]$ is a given constant and φ the frequency of visiting the corresponding region.

  The procedure that generates solutions outside the visited regions is the following:

  1. Generate a new solution randomly in the search domain of f.
  2. Compute the quantities $d_i = \frac{\|x-m_i\|}{1+\Psi(\varphi_i)}$, $i = 1, \ldots, M$. If $\min_{1 \leq i \leq M} \frac{d_i}{\rho_i} \geq 1$, then accept x. Otherwise, return to Step 1.

  being m_i the i-th centre of the VRL, M the size of the VRL and ρ the radius of the regions. We should note that a point close to more frequently visited regions is hardly accepted due to its higher value of Ψ. Therefore, the higher the value of γ is, the lower the possibility of accepting a point close to more frequently visited regions. To avoid infinitely cycling in process, we may also terminate it after a predetermined number of iterations.

The mutation operator used to modify the best global solution in the *bestSol* and the *reactive* actions consists on randomly selecting a direction according to a uniform distribution, and then take the point at a distance r from the best global solution. In the reactive action, different values for the r parameter are used depending on the ϕ degree (defined before), while for the *bestSol* action, a fixed value of $r = 0.1\sigma$ is used.

No. of functions	**Severity (%)**	**Agents**	**mQSO**	**CS**
	2	4.610(2.683)	6.702(4.889)	**3.576(2.462)***
	4	4.789(2.701)	7.011(4.882)	**4.042(2.469)***
100	6	5.040(2.819)	6.843(4.465)	**4.477(2.515)***
	8	5.270(2.821)	6.541(4.015)	**4.811(2.582)***
	10	5.445(2.911)	6.582(4.039)	**5.035(2.580)***

Table 3.14 – *CS for DOPs, first study.* MFE results for the **MPB** problem.

3.4.3. Validation

In order to validate the proposal of the CS algorithm applied to DOPs, we performed two studies to analyze its behaviour.

3.4.3.1. First study: introducing CS

The first study introduced a first implementation of CS for DOPs using only one action, *Best Sol*, and compared it against a standard implementation of the mQSO [15] and the Agents algorithm for continuous DOPs that we introduced in Sect. 3.3). In this study, we executed the 3 algorithms in 4 standard problems of the DOP literature: the MPB and the dynamic versions of the Ackley, Griewank and Rastrigin functions. Moreover, for the Ackley, Griewank and Rastrigin problems, several functions of the same type were combined to produce a more difficult scenario (1, 3, 5 and 10 functions were combined). Finally, we also tested variations of the severity of the change in the environment, in terms of % of σ (2%, 4%, 6%, 8% and 10%).

For the experiments conducted, we used the Mean Fitness Error (MFE) [134] for measuring the performance (see Sect. 2.2.2). In our case, since the algorithms proposed do not use a clear concept of generation, we considered that a generation corresponds to the period between two consecutive changes in the fitness function.

For both cases (MPB and real functions), changes to the function's optimum were performed every 5000 function evaluations, with a *run* grouping 100 consecutive changes. Each experiment consisted in 30 independent runs, each of them with its own random seed.

The results are presented in Tables 3.14, 3.15, 3.16 and 3.17. These tables show the final MFE of the 30 independent runs, with the standard deviation in parenthesis. Here, values with an asterisk ([*]) indicate that the corresponding algorithm obtained the best (lowest) MFE. For every configuration of the problem, the Kruskal-Wallis non-parametric test for multiple comparisons has been used

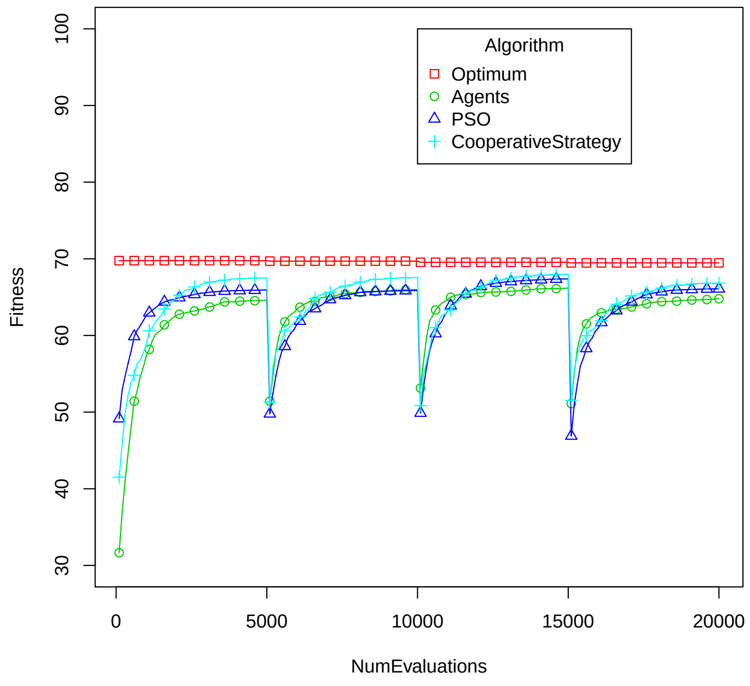


Figure 3.20 – *CS for DOPs, first study.* Single-run sample of best algorithm results vs. optimum for the **MPB** with a 10% severity.

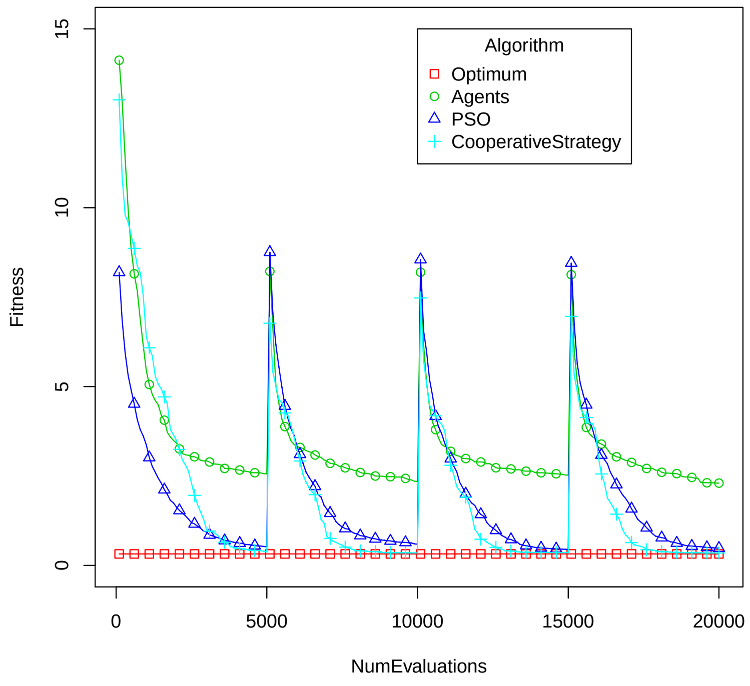


Figure 3.21 – *CS for DOPs, first study.* Single-run sample of best algorithm results vs. optimum for the **Ackley** problem with a 10% severity.

No. of functions	**Severity (%)**	**Agents**	**mQSO**	**CS**
1	2	2.227(0.891)	2.219(1.084)	**0.598(0.529)***
	4	2.340(0.868)	2.778(1.303)	**0.886(0.511)***
	6	2.462(0.962)	2.933(1.382)	**1.118(0.547)***
	8	2.552(0.857)	3.017(1.463)	**1.349(0.573)***
	10	2.659(0.775)	3.074(1.462)	**1.512(0.570)***
3	2	2.246(0.664)	1.711(0.755)	**0.563(0.431)***
	4	2.343(0.528)	2.108(0.835)	**0.889(0.459)***
	6	2.460(0.488)	2.202(0.837)	**1.137(0.471)***
	8	2.573(0.509)	2.248(0.786)	**1.347(0.481)***
	10	2.677(0.487)	2.312(0.756)	**1.524(0.508)***
5	2	2.270(0.476)	1.488(0.633)	**0.536(0.394)***
	4	2.395(0.501)	1.818(0.699)	**0.854(0.408)***
	6	2.488(0.441)	1.886(0.661)	**1.095(0.423)***
	8	2.572(0.454)	1.916(0.600)	**1.289(0.434)***
	10	2.696(0.489)	1.977(0.576)	**1.490(0.439)***
10	2	2.249(0.412)	1.299(0.623)	**0.402(0.326)***
	4	2.385(0.398)	1.491(0.646)	**0.694(0.335)***
	6	2.456(0.392)	1.598(0.595)	**0.931(0.340)***
	8	2.553(0.372)	1.685(0.545)	**1.131(0.347)***
	10	2.667(0.378)	1.693(0.487)	**1.332(0.366)***

Table 3.15 – *CS for DOPs, first study.* MFE results for the **Ackley** function.

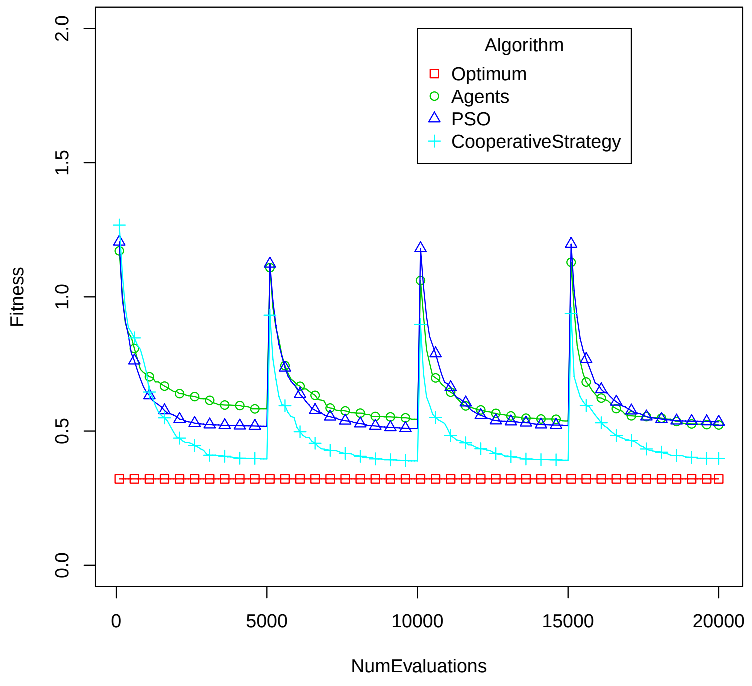


Figure 3.22 – *CS for DOPs, first study.* Single-run sample of best algorithm results vs. optimum for the **Griewank** problem with a 10% severity.

No. of functions	**Severity (%)**	**Agents**	**mQSO**	**CS**
1	2	0.132(0.074)	0.099(0.042)	**0.089(0.039)***
	4	0.180(0.067)	0.156(0.057)	**0.107(0.038)***
	6	0.220(0.077)	0.202(0.087)	**0.107(0.039)***
	8	0.251(0.084)	0.235(0.103)	**0.114(0.040)***
	10	0.260(0.088)	0.268(0.127)	**0.116(0.042)***
3	2	0.178(0.063)	**0.094(0.036)***	0.101(0.0439)
	4	0.226(0.064)	0.158(0.055)	**0.119(0.046)***
	6	0.252(0.070)	0.204(0.082)	**0.124(0.049)***
	8	0.265(0.074)	0.249(0.132)	**0.127(0.046)***
	10	0.278(0.079)	0.282(0.144)	**0.131(0.048)***
5	2	0.161(0.069)	**0.125(0.063)**	**0.119(0.049)***
	4	0.220(0.064)	0.166(0.062)	**0.134(0.048)***
	6	0.254(0.071)	0.229(0.091)	**0.135(0.051)***
	8	0.270(0.076)	0.255(0.121)	**0.142(0.052)***
	10	0.282(0.079)	0.295(0.123)	**0.147(0.053)***
10	2	0.182(0.061)	0.125(0.062)	**0.122(0.043)***
	4	0.241(0.061)	0.192(0.061)	**0.141(0.043)***
	6	0.252(0.063)	0.239(0.078)	**0.143(0.044)***
	8	0.264(0.064)	0.263(0.084)	**0.148(0.047)***
	10	0.275(0.065)	0.293(0.096)	**0.153(0.046)***

Table 3.16 – *CS for DOPs, first study.* MFE results for the **Griewank** function.

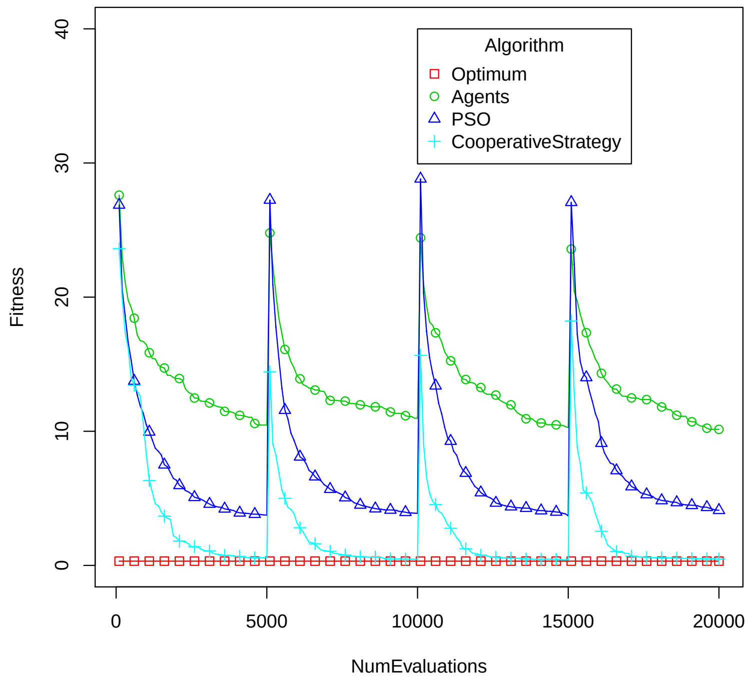


Figure 3.23 – *CS for DOPs, first study.* Single-run sample of best algorithm results vs. optimum for the **Rastrigin** problem with a 10% severity.

No. of functions	**Severity (%)**	**Agents**	**mQSO**	**CS**
1	2	10.503(3.019)	2.288(1.300)	**0.882(0.861)***
	4	13.189(3.091)	4.754(1.673)	**1.224(0.962)***
	6	13.895(3.160)	6.880(2.323)	**1.745(1.046)***
	8	14.245(3.226)	7.686(2.615)	**2.228(1.188)***
	10	14.473(3.212)	8.229(2.789)	**2.496(1.273)***
3	2	10.814(2.883)	1.849(0.695)	**1.094(0.802)***
	4	13.121(2.849)	4.568(1.411)	**1.397(0.866)***
	6	14.058(2.915)	6.903(2.089)	**1.925(0.950)***
	8	14.327(2.943)	7.693(2.363)	**2.353(1.032)***
	10	14.358(2.965)	8.000(2.435)	**2.536(1.086)***
5	2	10.525(2.759)	1.786(0.791)	**1.041(0.722)***
	4	13.076(2.763)	4.562(1.420)	**1.334(0.761)***
	6	13.969(2.764)	6.840(2.108)	**1.854(0.828)***
	8	14.078(2.807)	7.678(2.298)	**2.294(0.971)***
	10	14.161(2.817)	7.865(2.354)	**2.473(1.003)***
10	2	10.189(2.585)	1.939(0.722)	**0.826(0.601)***
	4	12.473(2.597)	4.465(1.393)	**1.133(0.672)***
	6	13.175(2.626)	6.511(2.002)	**1.667(0.802)***
	8	13.310(2.615)	7.132(2.156)	**2.054(0.872)***
	10	13.379(2.640)	7.355(2.251)	**2.194(0.913)***

Table 3.17 – *CS for DOPs, first study.* MFE results for the **Rastrigin** function.

to assess differences between the performances of the three methods. The null hypothesis could not be rejected at significance level 0.01 for all problem configurations. Apart from this, the Wilcoxon's unpaired rank sum test at a significance level 0.05 was performed between the best algorithm and each of the others, to assess if there were statistical differences. Values shown in bold-face indicate that there were no statistically significant differences between that value and the best value of the row.

Additionally, a graph of the best results of all the methods against the optimum for a sample run of each test function was generated. In order to compare the methods in the worst scenario, the instances selected were the most difficult ones where the number of functions is maximum and the biggest severity (10%) of change is used. For each selected run, the first 20000 evaluations were plotted taking values every 100 evaluations. Since the fitness function changes every 5000 evaluations, this 20000 evaluations correspond to the evolution of the fitness for the first four fitness function changes. These graphs are presented in Figures 3.20, 3.21, 3.22 and 3.23.

3.4.3.2. Second study: extending cooperation rules

In the second study, considering the results obtained in the first one, we introduced the rest of the actions for the CS algorithm, and we evaluated again their performance over several test scenarios. The different versions of CS used were:

- *independent*: uses no cooperation at all. Each thread works independently.
- *approach*: uses the Approaching (AP) action.
- *bestSol*: uses the Best Sol (BS) action.
- *VRL*: uses the VRL action.
- *reactive*: uses the Reactive (R) action.

In this case, the CS-bestSol corresponds to the CS used in the previous study, included here as a reference to compare the results. Additionally, with the objective of having a non-CS reference, we executed again the mQSO of the previous study, since this algorithm obtained better results than the Agents in most of the problems. In this study, we performed the experiments over the Ackley, Griewank and Rastrigin functions, since they were the ones in which the CS-bestSol obtained the best results, in order to verify if the new variants are able of outperforming it.

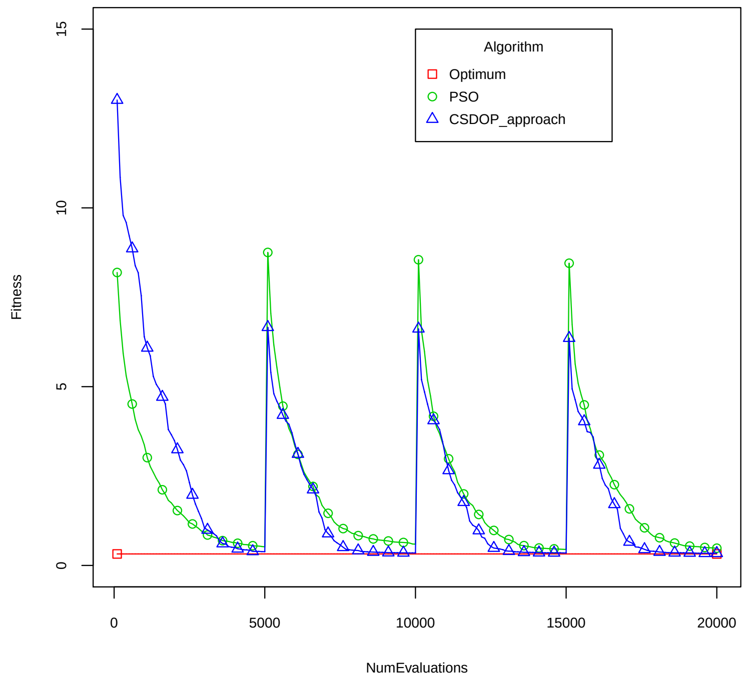


Figure 3.24 – *CS for DOPs, second study.* Single-run sample of best fitness vs. evaluations for the composition of 10 **Ackley** functions. Severity = 10%.

No. of functions	Severity	mQSO	CS independent	CS approach	CS bestSol	CS VRL	CS reactive
1	2	2.219(1.084)	**0.254(0.425)***	**0.257(0.426)**	0.598(0.529)	0.549(0.504)	0.325(0.429)
	4	2.778(1.303)	**0.527(0.401)***	0.551(0.406)	0.886(0.511)	0.823(0.483)	0.603(0.404)
	6	2.933(1.382)	**0.760(0.407)***	0.784(0.419)	1.118(0.547)	1.042(0.492)	0.828(0.411)
	8	3.017(1.463)	**0.957(0.425)***	1.000(0.435)	1.349(0.573)	1.257(0.529)	1.039(0.436)
	10	3.074(1.462)	**1.140(0.435)***	1.203(0.441)	1.512(0.570)	1.431(0.528)	1.223(0.447)
3	2	1.711(0.755)	0.304(0.374)	**0.262(0.376)***	0.563(0.431)	0.540(0.418)	0.320(0.377)
	4	2.108(0.835)	0.604(0.385)	**0.565(0.388)***	0.889(0.459)	0.835(0.436)	0.615(0.386)
	6	2.202(0.837)	0.855(0.394)	**0.797(0.404)***	1.137(0.471)	1.090(0.451)	0.850(0.404)
	8	2.248(0.786)	1.055(0.396)	**1.012(0.403)***	1.347(0.481)	1.293(0.462)	1.045(0.408)
	10	2.312(0.756)	1.237(0.421)	**1.215(0.425)***	1.524(0.508)	1.493(0.483)	1.243(0.433)
5	2	1.488(0.633)	0.332(0.351)	**0.270(0.354)***	0.536(0.394)	0.487(0.376)	0.335(0.359)
	4	1.818(0.699)	0.648(0.356)	**0.570(0.365)***	0.854(0.408)	0.807(0.380)	0.626(0.362)
	6	1.886(0.661)	0.897(0.371)	**0.820(0.372)***	1.095(0.423)	1.054(0.400)	0.858(0.376)
	8	1.916(0.600)	1.091(0.372)	**1.027(0.378)***	1.289(0.434)	1.260(0.409)	1.061(0.377)
	10	1.977(0.576)	1.278(0.386)	**1.228(0.391)***	1.490(0.439)	1.468(0.426)	1.264(0.397)
10	2	1.299(0.623)	0.324(0.316)	**0.277(0.319)***	0.402(0.326)	0.393(0.321)	0.320(0.322)
	4	1.491(0.646)	0.608(0.317)	**0.571(0.328)***	0.694(0.335)	0.676(0.328)	0.606(0.329)
	6	1.598(0.595)	0.827(0.320)	**0.799(0.334)***	0.931(0.340)	0.919(0.333)	0.838(0.335)
	8	1.685(0.545)	1.042(0.331)	**1.020(0.344)***	1.131(0.347)	1.114(0.351)	**1.031(0.345)**
	10	1.693(0.487)	1.216(0.350)	**1.199(0.368)***	1.332(0.366)	1.312(0.361)	1.220(0.369)

Table 3.18 – *CS for DOPs, second study.* MFE results for the **Ackley** function.

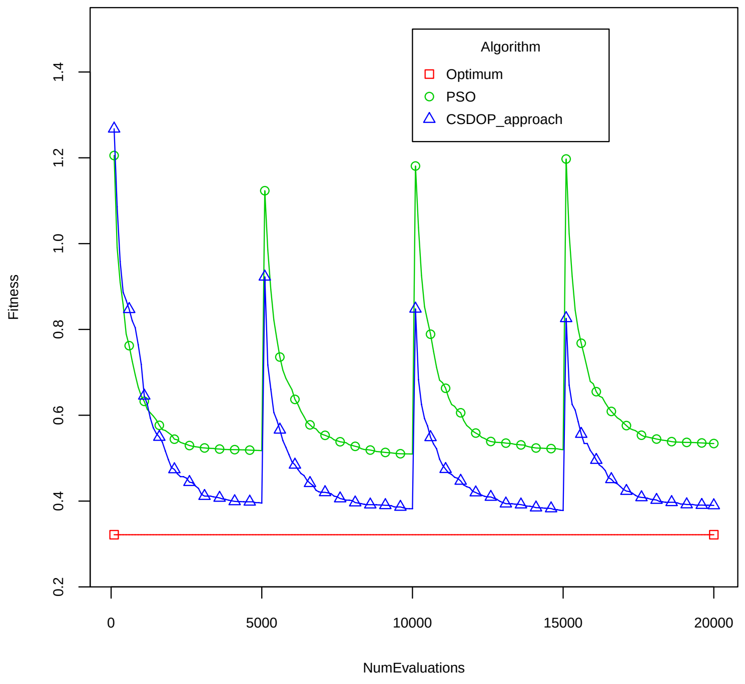


Figure 3.25 – *CS for DOPs, second study.* Single-run sample of best fitness vs. evaluations for the composition of 10 **Griewank** functions. Severity = 10%.

No. of functions	**Severity**	**mQSO**	**CS independent**	**CS approach**	**CS bestSol**	**CS VRL**	**CS reactive**
	2	0.099(0.042)	0.066(0.030)	**0.037(0.026)***	0.089(0.039)	0.093(0.039)	0.065(0.028)
	4	0.156(0.057)	0.088(0.031)	**0.066(0.030)***	0.107(0.038)	0.109(0.039)	0.084(0.030)
1	6	0.202(0.087)	0.091(0.034)	**0.074(0.033)***	0.107(0.040)	0.109(0.040)	0.089(0.033)
	8	0.235(0.103)	0.101(0.036)	**0.083(0.034)***	0.114(0.040)	0.115(0.041)	0.095(0.034)
	10	0.268(0.127)	0.104(0.037)	**0.090(0.036)***	0.116(0.042)	0.116(0.041)	0.100(0.038)
	2	0.094(0.036)	0.068(0.030)	**0.038(0.027)***	0.101(0.044)	0.104(0.046)	0.065(0.029)
	4	0.158(0.055)	0.091(0.034)	**0.067(0.032)***	0.119(0.046)	0.123(0.047)	0.086(0.033)
3	6	0.204(0.083)	0.094(0.036)	**0.076(0.035)***	0.124(0.049)	0.123(0.047)	0.090(0.034)
	8	0.249(0.132)	0.099(0.036)	**0.084(0.035)***	0.127(0.047)	0.129(0.048)	0.094(0.035)
	10	0.282(0.144)	0.107(0.039)	**0.093(0.038)***	0.131(0.048)	0.133(0.050)	0.100(0.037)
	2	0.125(0.063)	0.084(0.036)	**0.043(0.030)***	0.119(0.049)	0.120(0.050)	0.071(0.031)
	4	0.166(0.062)	0.105(0.039)	**0.071(0.034)***	0.134(0.048)	0.138(0.051)	0.089(0.033)
5	6	0.229(0.092)	0.106(0.039)	**0.081(0.037)***	0.135(0.051)	0.137(0.005)	0.094(0.035)
	8	0.255(0.121)	0.113(0.042)	**0.090(0.038)***	0.142(0.052)	0.144(0.051)	0.100(0.037)
	10	0.295(0.123)	0.119(0.043)	**0.098(0.041)***	0.147(0.053)	0.148(0.054)	0.106(0.041)
	2	0.125(0.062)	0.102(0.038)	**0.054(0.034)***	0.122(0.043)	0.126(0.044)	0.077(0.032)
	4	0.192(0.061)	0.123(0.041)	**0.088(0.040)***	0.141(0.043)	0.144(0.044)	0.100(0.037)
10	6	0.239(0.078)	0.125(0.041)	**0.098(0.041)***	0.143(0.044)	0.144(0.045)	0.103(0.038)
	8	0.263(0.084)	0.131(0.041)	**0.107(0.042)***	0.148(0.047)	0.148(0.046)	0.111(0.040)
	10	0.293(0.096)	0.137(0.042)	**0.119(0.042)**	0.153(0.046)	0.153(0.046)	**0.118(0.042)***

Table 3.19 – *CS for DOPs, second study.* MFE results for the **Griewank** function.

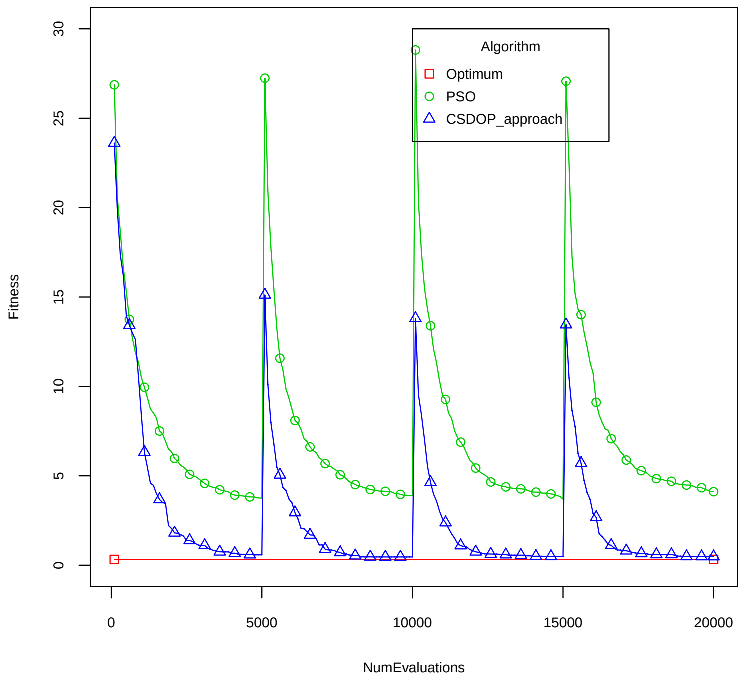


Figure 3.26 – *CS for DOPs, second study.* Single-run sample of best fitness vs. evaluations for the composition of 10 **Rastrigin** functions. Severity = 10%.

No. of functions	Severity	mQSO	CS independent	CS approach	CS bestSol	CS VRL	CS reactive
1	2	2.288(1.300)	0.322(0.578)	**0.236(0.576)***	0.882(0.861)	0.788(0.759)	0.579(0.643)
	4	4.754(1.673)	0.561(0.612)	**0.520(0.625)***	1.224(0.962)	1.101(0.835)	0.838(0.697)
	6	6.880(2.323)	1.019(0.635)	**0.926(0.633)***	1.745(1.046)	1.617(0.927)	1.259(0.717)
	8	7.686(2.615)	1.478(0.841)	**1.393(0.832)***	2.228(1.188)	2.086(1.090)	1.649(0.861)
	10	8.229(2.789)	**1.695(0.942)**	**1.680(0.957)***	2.496(1.273)	2.252(1.132)	1.813(0.916)
3	2	1.849(0.695)	0.597(0.588)	**0.297(0.561)***	1.094(0.802)	1.048(0.722)	0.609(0.612)
	4	4.568(1.411)	0.799(0.594)	**0.563(0.583)***	1.397(0.866)	1.305(0.775)	0.916(0.663)
	6	6.903(2.089)	1.394(0.664)	**1.056(0.690)***	1.925(0.950)	1.827(0.843)	1.340(0.724)
	8	7.693(2.363)	1.772(0.804)	**1.584(0.861)***	2.353(1.032)	2.233(0.945)	1.742(0.859)
	10	8.000(2.435)	2.014(0.865)	**1.860(0.986)***	2.536(1.086)	2.388(0.974)	1.909(0.907)
5	2	1.786(0.791)	0.684(0.560)	**0.311(0.512)***	1.041(0.722)	0.956(0.643)	0.686(0.602)
	4	4.562(1.420)	0.901(0.564)	**0.621(0.520)***	1.334(0.761)	1.262(0.684)	0.929(0.594)
	6	6.840(2.108)	1.403(0.622)	**1.121(0.635)***	1.854(0.828)	1.779(0.777)	1.377(0.674)
	8	7.678(2.298)	1.854(0.767)	**1.647(0.848)***	2.294(0.971)	2.190(0.883)	1.746(0.827)
	10	7.865(2.354)	1.994(0.807)	**1.909(0.916)***	2.473(1.003)	2.354(0.920)	**1.924(0.876)**
10	2	1.939(0.722)	0.534(0.474)	**0.321(0.451)***	0.826(0.601)	0.786(0.559)	0.625(0.520)
	4	4.465(1.393)	0.753(0.473)	**0.627(0.493)***	1.133(0.672)	1.066(0.594)	0.870(0.541)
	6	6.511(2.002)	1.278(0.602)	**1.182(0.647)***	1.667(0.802)	1.575(0.733)	1.343(0.673)
	8	7.132(2.156)	1.714(0.713)	**1.688(0.788)***	2.054(0.872)	1.964(0.829)	1.720(0.760)
	10	7.355(2.251)	**1.804(0.757)***	1.868(0.836)	2.194(0.913)	2.068(0.845)	1.872(0.799)

Table 3.20 – *CS for DOPs, second study.* MFE results for the **Rastrigin** function.

The performance measures and the experiments' settings were the same as the ones used in the previous study. The results obtained can be seen in Tables 3.18, 3.19 and 3.20. Additionally, as in the previous study, we have included a graphical representation of the first 20000 evaluations of a run for the most difficult configuration of all the test problems. However, due to the available space in each graphs, we did not show all the algorithms, and have only shown the optimum, the mQSO, and the best CS version — in this case, CS-*approach*. These graphs can be seen in Figures 3.24, 3.25 and 3.26.

3.4.4. Conclusions

We have presented an algorithm, CS, based on a set of solvers, implemented as trajectory-based heuristics (tabu search), that cooperate among them in an explicit fashion thanks to a central coordinator.

We have performed two studies in order to validate the proposal and identify the best cooperation scheme for the algorithm.

In the fist study, we compared an initial version of CS that only used a simple cooperation rule, *bestSol*, against a mQSO and a DOP version of the Agents algorithm. The results showed that the proposed cooperative strategies algorithm consistently outperformed the others in all the problems and scenarios tested.

In the second study, we extended the experimentation introducing more cooperation rules. Again, the CS widely overcame mQSO in all problems, and the rules that obtained the best results were the *Approaching* and *Reactive* ones. It is worth noting that the *Approaching* rule, designed to enhance intensification, produces the best results most of the time, while the *Reactive* rule, which increases diversification, starts to obtain better results as the problems get more and more difficult.

As far as our knowledge is concerned, little attention has been put on using trajectory solvers and explicit cooperation schemes for solving DOPs before these works. But CS has shown very good performance results that are also very consistent across all the configurations of the problems and all the cooperation modes tested. The success of CS suggest that trajectory methods combined with other techniques (a centralised cooperation in the scope of these works) may play an important role in the dynamic optimization field and further research should be put into this area.

The first study was published in reference [69]:

"A cooperative strategy for solving dynamic optimization problems", J. R. González, A. D. Masegosa, and I. G. del Amo, *Memetic Computing*, vol. 3, no. 1, pp. 3–14, 2010. `http://dx.doi.org/10.1007/s12293-010-0031-x`.

and the second study was published in reference [70]:

"Cooperation rules in a trajectory-based centralised cooperative strategy for Dynamic Optimisation Problems", J. R. González, A. D. Masegosa, I. G. del Amo, and D. A. Pelta, in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pp. 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586063`.

3.5. Overall conclusions

In this chapter we have reviewed a set of proposals for improving several algorithms for DOPs. The new / improved algorithms are:

- An improvement for a multi-swarm PSO based on a new operator for controlling particle trajectories (CPT) [120] for continuous DOPs.
- An enhanced version of the mQSO algorithm that uses heuristic rules to produce more efficient behaviours [39, 41]; the use of 3 rules has led to 3 variants of the algorithm: *mQSO-ChangeRule*, *mQSO-RandRule* and *mQSO-BothRule*, all of them for continuous DOPs.
- An algorithm based on the implicit cooperation of a set of agents, both for continuous DOPs [69] and discrete DOPs [68]; this last version for discrete DOPs also incorporated a learning mechanism that allows it to auto-adapt the values of its parameters (*Agents-Adaptive*).
- An algorithm based on the centralized cooperation of trajectory heuristics, by means of a coordinator, for continuous DOPs, named CS [69], as well as several cooperation schemes that have produced the variants *CS-BestSol*, *CS-VRL*, *CS-Reactive*, *CS-Approach* and *CS-Independent* [70].

These algorithms cover a wide variety of families, but all of them share some common characteristics:

1. **The use of populations of solutions**. All these methods keep a set or sets of solutions that "coexist" during the search process. These populations of solutions allow to diversify the search, something important by its own in static optimization, and absolutely essential in DOPs.

2. **The existence of some type of cooperation among its constituent elements**. In the case of CS, this is obvious, since such cooperation is explicit, and is carried out by the central coordinator while exchanging information between the different metaheuristics. In the case of the implemented variants of PSO, the particles of a swarm are connected among them through the *best* of that swarm, since it is used as a reference in the movement of the particles. Furthermore, in the case of the mQSO, which uses multiple swarms, there is an exclusion mechanism that prevents the swarms from getting too close to each other, in order to avoid focusing on the same optimum. This competition between swarms at a local level helps the algorithm to not waste too many resources in the same area, and thus, it is a form of global cooperation in a sense. Agents also implicitly cooperate, since they explore the search space indirectly through the matrix, allowing them to improve solutions that quite probably have already been modified by other agents in previous iterations.

The first characteristic is in accordance with what we said in the state of the art chapter, Sect. 2.3, regarding population-based algorithms being a general trend in DOPs. The second characteristic is quite significant in the context of this thesis because it somehow indicates a possible pathway for improving existent algorithms: all the modifications that we have performed during the research are based on promoting cooperation and increasing the exchange of information between the elements of an algorithm.

Additionally, we introduced a generic mechanism that has been incorporated into several algorithms. This mechanism consists in (1) actively monitor the performance of the different elements of an algorithm; (2) evaluate them according to the average performance of the rest; and (3) correct, or even stop, those elements that are doing it worse. This mechanism was implemented in the CS by the coordinator, in the PSO-CPT, and in the mQSO-RandRule, and the experiments confirm that these algorithms obtained very good results, always improving their performance respect to the base algorithm without this mechanism.

Chapter 4

SRCS: Statistical Ranking Color Scheme

In this chapter we will present a technique for comparing several algorithms over multiple problem configurations, named SRCS (Statistical Ranking Color Scheme). The main drawback of performing experiments over multiple combined factors is that the amount of generated data is usually so big that it becomes very difficult to fully comprehend the results. The SRCS technique compacts such data in two steps: first, it creates a ranking of the algorithms results over each scenario of the experiments using statistical tests, and then, it generates a visualization of those rankings using color schemes. Instead of focusing on small specific results, the technique allows to identify behaviours and general trends at a large scale, and it has already proven its utility in several publications.

4.1. Motivation

Let's suppose that we want to compare the performance of a set of metaheuristic algorithms over a DOP, for example, the MPB (see Sect. 2.1.1.1 for a description of this problem).

The first thing we need to do is to decide the performance measure to use for evaluating and comparing the algorithms. Since for the MPB the optimum is known, a good candidate would be the *offline accuracy* (see Sect. 2.2.4), because it is bounded in the interval $[0, 1]$ and is not affected by the variations in the optimum's value over the changes in the environment. If we average the offline accuracy over all the changes of a run, we obtain the overall or *avg. offline accuracy.*

Now that we have already defined the performance measure, let's suppose that we want to compare 4 hypothetical algorithms for a given configuration of the

	Algorithm 1	**Algorithm 2**	**Algorithm 3**	**Algorithm 4**
Avg. Offline Accuracy	0.78 ±0.05	0.84 ±0.02	0.95 ±0.01	0.89 ±0.03

Table 4.1 – Performance results of several algorithms on a single problem configuration (mean and std. deviation values of the *avg. offline accuracy*)

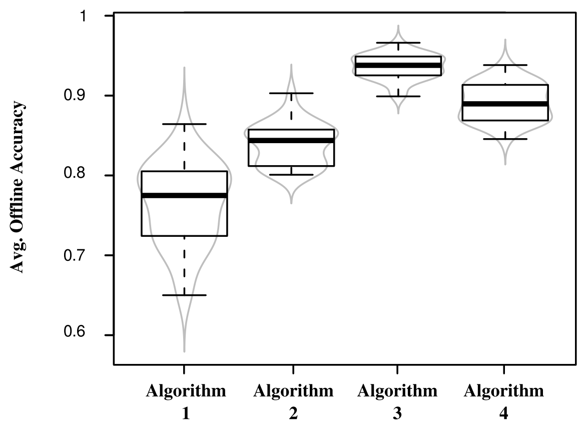


Figure 4.1 – Graphical representation of the results on Table 4.1. The distributions are displayed using a boxplot (dark lines) in combination with a kernel estimation of the distribution density (light lines)

MPB (for example, the widely used Scenario 2). As it has been mentioned in Sect. 2.2, when dealing with stochastic algorithms it is necessary to perform a series of independent repetitions of the experiments in order to obtain a representative sample of its performance. Therefore, we will execute N_r runs of the algorithm, thus obtaining N_r measurements of the *avg. offline accuracy*. In Sect. 4.3 we will analyze in more detail the influence of N_r in the results of the statistical tests, but for the moment, let's just assume that we perform a fixed amount of independent repetitions, say $N_r = 30$, for each algorithm. An example of the results that could be obtained is presented in Table 4.1 and Fig. 4.1.

In order to determine the existence of statistically significant differences in the results, we need to perform a series of hypothesis tests. Several authors have already pointed out that the results of these metaheuristic algorithms, in general, do not follow a normal distribution [63], therefore recommending the use of non-parametric tests for their analysis [79, 132]. We will use a significance level $\alpha = 0.05$, meaning that we are willing to assume a probability of mistakenly rejecting the null hypothesis of, at most, 0.05. The first issue that needs to be addressed is the fact that we are comparing multiple algorithms at the same time. Therefore,

	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
Algorithm 1	no	-	-	-
Algorithm 2	+	no	-	-
Algorithm 3	+	+	no	+
Algorithm 4	+	+	-	no

Table 4.2 – Pairwise statistical differences among the *avg. offline accuracy* distribution of the algorithms. A '+' sign indicates that there are statistically significant differences between the algorithm in the row and the algorithm in the column, and that the sign of the comparison favors the algorithm in the row (i.e., is "better"). A '-' sign indicates the opposite, that the algorithm in the row is "worse". Finally, the word 'no' indicates no statistically significant differences

we need to use a test that allows to compare more than 2 groups simultaneously. For this example, we will perform a Kruskal-Wallis (KW) test [89], among all the samples of the 4 algorithms to check if there are global differences at the 0.05 significance level. If the KW test concludes that there are statistically significant differences, we will then perform a series of pair-wise tests between each pair of algorithms, to see if we can determine which are the ones that are causing those differences. In this case, we will use the Mann-Whitney-Wilcoxon (MWW) [102, 168] test to compare each pair of algorithms. The combination of these tests is suitable, since the KW test can be considered as the natural extension of the MWW test to multiple groups. It is important to note that in order to guarantee the α-level achieved by the KW test (global), we need to adjust the α-level of each MWW test (pair-wise) to a certain value, usually much smaller than the first one. For this purpose, we will use Holm's correction [80], although other techniques are also available (for example, Hochberg's, Hommel's, etc; for a in-depth comparison on the use of these techniques, the interested reader is referred to [43,62,63]). The results of the tests are shown in Table 4.2, where individual comparisons between each pair of algorithms can be seen, along with the sign of the comparison.

Until now, the way of presenting the results (numerical data tables, boxplot graphs and statistical tests tables) has been appropriated, and the data is comprehensible. Let's assume that we now would like to extend the experimental framework. We want to know if the conclusions obtained for the algorithms follow any kind of pattern related to some characteristic of the problem (e.g., whether algorithm 3 is good only for the Scenario 2 of the MPB, or if this is a general behaviour linked to, for example, low change frequencies). In order to answer this question, we perform more experiments, keeping all problem's parameters constant, except for the change frequency, which we vary progressively.

We can see now that the number of results increases, and its presentation be-

Change Frequency	Algorithm 1	Algorithm 2	Algorithm 3	Algorithm 4
200	**0.630 ±0.03**	**0.632 ±0.03**	**0.631 ±0.03**	**0.630 ±0.03**
500	0.750 ±0.02	0.751 ±0.02	**0.783 ±0.02**	**0.781 ±0.02**
1000	0.811 ±0.02	0.798 ±0.02	**0.886 ±0.02**	**0.886 ±0.02**
1500	0.825 ±0.02	0.854 ±0.02	**0.922 ±0.02**	0.871 ±0.02
2000	0.840 ±0.02	0.859 ±0.01	**0.939 ±0.01**	0.862 ±0.02
2500	0.843 ±0.01	0.871 ±0.01	**0.943 ±0.01**	0.889 ±0.01
3000	0.852 ±0.01	0.880 ±0.01	**0.950 ±0.01**	0.913 ±0.01
3500	0.871 ±0.01	0.901 ±0.01	**0.959 ±0.01**	0.921 ±0.01
4000	0.860 ±0.01	0.906 ±0.01	**0.964 ±0.01**	0.932 ±0.01
4500	0.863 ±0.01	0.910 ±0.01	**0.968 ±0.01**	0.939 ±0.01
5000	0.869 ±0.01	0.911 ±0.01	**0.970 ±0.01**	0.941 ±0.01

Table 4.3 – Performance results of several algorithms on multiple problem configurations (mean and std. deviation values of the *avg. offline accuracy*). The different configurations are based on systematic variations of one factor, the problem's change frequency, expressed in number of evaluations. Boldface values indicate the best algorithm for the given configuration

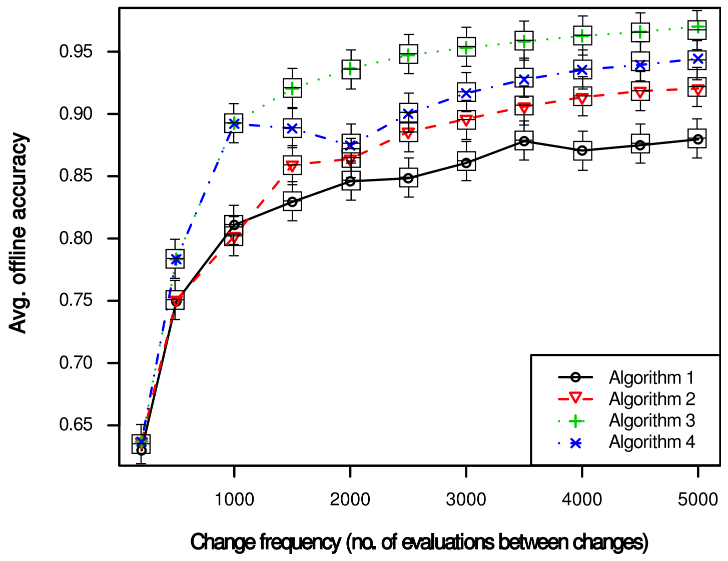


Figure 4.2 – Graphical representation of the results on Table 4.3, where each point corresponds to the results of an algorithm on a configuration of the problem. The results for each configuration are shown using a boxplot of the distribution

gins to be a problem, both at a table level, because of its extension and difficulty to comprehend the data (Table 4.3), and at a graphical level, because of its complexity (Fig. 4.2). However, it is still feasible to show the results this way, since, although data is now more difficult to grasp and manage, it is nevertheless still understandable (in Fig. 4.2 it is reasonably easy to see which algorithm is the best, and this can also be accomplished in Table 4.3 by enhancing the best algorithm's result using a boldface type). Anyway, it is worth noting that individual differences between each pair of algorithms in the statistical tests are now too lengthy to be shown, since they imply a comparison of the type *all against all* for each problem configuration, which, in general, is not practical for a publication (e.g., in this case, we are talking about 11 tables like Table 4.2).

Finally, when we consider to simultaneously analyze several factors (for example, the frequency and the severity of a change), data grows exponentially, and the presentation in the form of tables and figures becomes intractable and meaningless (see Fig. 4.3). We need alternative ways of presenting the results in order to be able of comprehending and analyzing them.

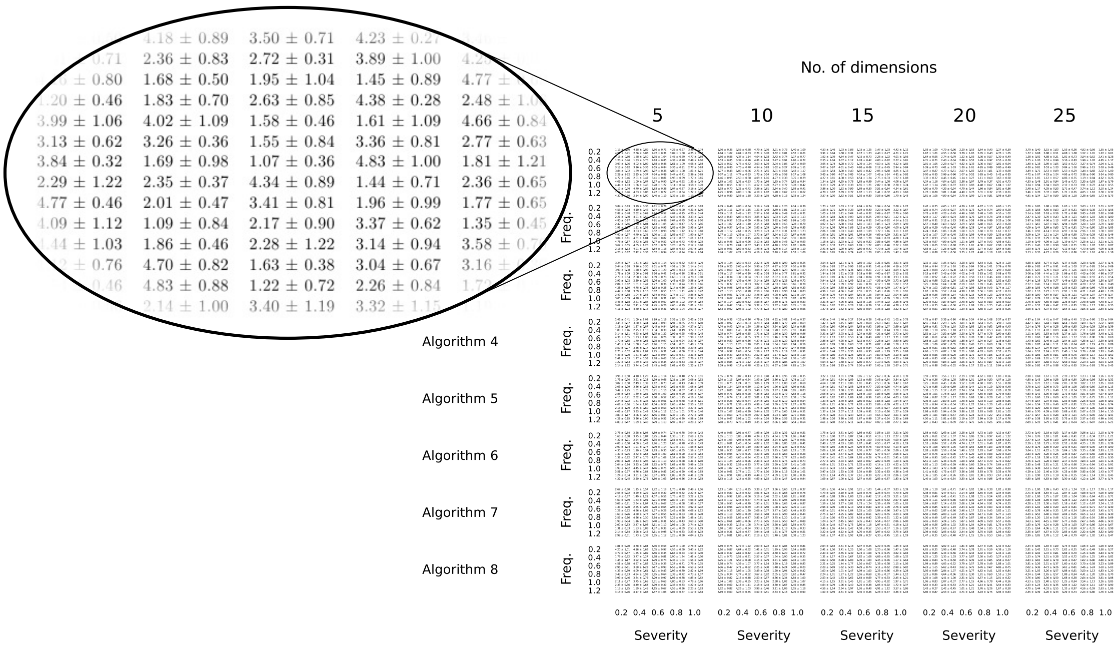


Figure 4.3 – Representation of a matrix of results for multiple algorithms, with combinations of the parameters of the problem. The huge amount of numerical data makes its comprehension almost impossible.

This situation that we have described is based in our own experience on DOPs, as it can be observed in the contributions presented in Chapter 3.

The works about the PSO-CPT (Sect. 3.1) and the mQSO with heuristic rules (Sect. 3.2) included simple variations of one parameter of the problem in the experiments, very similarly to the example data in Table 4.3 and Fig. 4.2.

Subsequent research lead to the works of the CS algorithm (Sect. 3.4), which included variations of 2 parameters. The results of these works now are more difficult to present, due to the extension of the data, as it can be seen in Tables 3.18, 3.19 and 3.20.

Finally, for the case of the work with the Agents algorithm for discrete DOPs (Sect. 3.3), the number of combinations was too high, and it made it impossible to present the results in a numerical way (there were variations of the severity, change frequency, over 4 different problems, with 5 versions of each algorithm). In that work, we applied the SRCS technique that we will explain next.

4.2. Description of the SRCS technique

With the objective of solving the previously explained problems that we may face in an experimentation with variations of multiple factors, we developed a technique named SRCS (Statistical Ranking Color Scheme). The SRCS technique compacts the data of the experiments to be presented in two steps: first, it creates a ranking of the algorithms results over each scenario of the experiments using statistical tests, and then, it generates a visualization of those rankings using color schemes.

Given a set of N algorithms, we will denote the observed performance of the said algorithms on a certain problem configuration as $P = \{p_1, p_2, \ldots, p_N\}$. Without loss of generality, we will assume that we are trying to *maximize* the performance, such that we consider that p_i is better than p_j if $p_i > p_j$.

As it has already been explained, when we work with an algorithm that contains an stochastic component, it is necessary to sample a number k of times the performance, so that for a given algorithm i, we will have that $p_i = \{p_i^1, p_i^2, \ldots, p_i^k\}$. We will denote p_i^* as the representative of p_i (usually, the mean, $\bar{p_i}$, or the median, $\tilde{p_i}$). Finally, let $MCST(P)$ be a Multiple Comparison Statistical Test, and let $PWST(p_i, p_j)$ be a Pair-Wise Statistical Test. For both functions we will assume that the tests return “YES” in case of finding statistically significant differences, and “NO” otherwise.

The ranking of an algorithm will be defined as

$$r_i = \begin{cases} 0 & \text{if } MCST(P) = NO \\ \\ \sum\limits_{\substack{j=1 \\ j \neq i}}^{N} \delta_{pw}(p_i, p_j) & \text{if } MCST(P) = YES \end{cases} \tag{4.1}$$

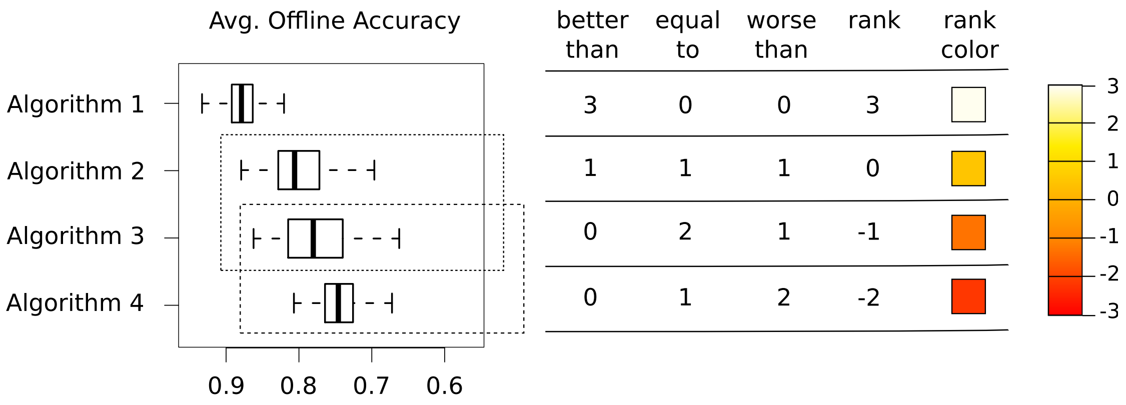
where

$$\delta_{pw}(p_i, p_j) = \begin{cases} 0 & \text{if } PWST(p_i, p_j) = NO \\ +1 & \text{if } PWST(p_i, p_j) = YES \quad \text{and} \quad p_i^* > p_j^* \\ -1 & \text{if } PWST(p_i, p_j) = YES \quad \text{and} \quad p_i^* < p_j^* \end{cases} \tag{4.2}$$

Basically, what this procedure is doing is the following: for a given DOP configuration, all the algorithms begin with an initial ranking of 0. We first compare the results of all the algorithms using a multiple comparison test (e.g., the KW test) in order to determine if there are global differences. In case there are no differences among all, that would be the end of the process, and the algorithms would finish with their initial 0 rank. If, however, significant differences were found, an adjusted pair-wise test (e.g., MWW + Holm) would be performed between each pair of algorithms, in order to assess individual differences. If the pair-wise test says there are significant differences for a given pair of algorithms, the one with the best performance value adds +1 to its ranking, and the one with the worst value, −1. If there were no differences according to the pair-wise test (a tie), neither algorithm adds anything, but maintain their previous ranking.

At the end, every algorithm will have an associated ranking value, ranging in the interval $[-(N-1), +(N-1)]$, where N is the number of algorithms being compared. A ranking value of $+r$ for a given algorithm indicates that its performance is significantly better than r algorithms, and a value of $-r$, that it is significantly worse than r algorithms.

However, until now, we have only shifted the problem, since we have a ranking, but it is still numerical, and therefore, difficult to fully understand when presented in the form of tables, if there are too many data. The solution to this comes from humans' ability to better manage images and colors than numbers. Starting off from this ranking, we associate a color (for example white) to the maximum ranking value that can be obtained, $+(N-1)$, and another very different color (a dark one preferably) to the minimum ranking value that can be obtained, $-(N-1)$. All the intermediate ranking values are associated to an interpolated color between the two previous ones. Figure 4.4 explains the calculation of the ranking and the color association of the 4 algorithms we have been using previously in the examples of Sect. 4.1, for a given problem configuration.



better than	equal to	worse than	rank	rank color
3	0	0	3	⬜
1	1	1	0	🟨
0	2	1	-1	🟧
0	1	2	-2	🟥

Figure 4.4 – Rank explanation. The boxplot shows the distribution of the performance measures of every algorithm, ordered by its median value. Dotted rectangles indicate those algorithms for which no statistical differences were found at the specified significance level (algorithms 2-3 and 3-4). The table on the right displays, for every algorithm, how many times it shows a significantly better performance (“better than”), no significant differences (“equal to”) or significantly worse performance (“worse than”) when it is compared with respect to the other 3 algorithms, and its final rank with the correspondent color key.

Color codes obtained from the ranking can now be used to represent the *relative* performance of each algorithm with respect to the others in a graphical way. This representation allows to visualize the results of many configurations at once, giving the researcher the possibility to identify behavioural patterns of the algorithms more easily.

For example, let’s suppose that we have these 4 algorithms of the examples of the previous section, and we want to extend the study of their performance in the MPB with different variations of two factors: *severity*, and *change frequency*. As it has already been justified, presenting the results of this experiments in the form of tables may not be feasible. However, using the SRCS technique, we can now arrange the rank colors of each configuration to create the images shown in Fig. 4.5. In this figure, the same color scheme as the one appearing in the explanation on Fig. 4.4 has been used, where a darker color indicates a worse performance, and a lighter one a better. Taking a quick glance at Fig. 4.5, and without having to exam any type of numerical data, we can obtain valuable overall information, like:

- In general, *algorithm 1* is the worst algorithm for almost all configurations.
- *Algorithm 3* has, for almost all configurations, a good or very good performance.

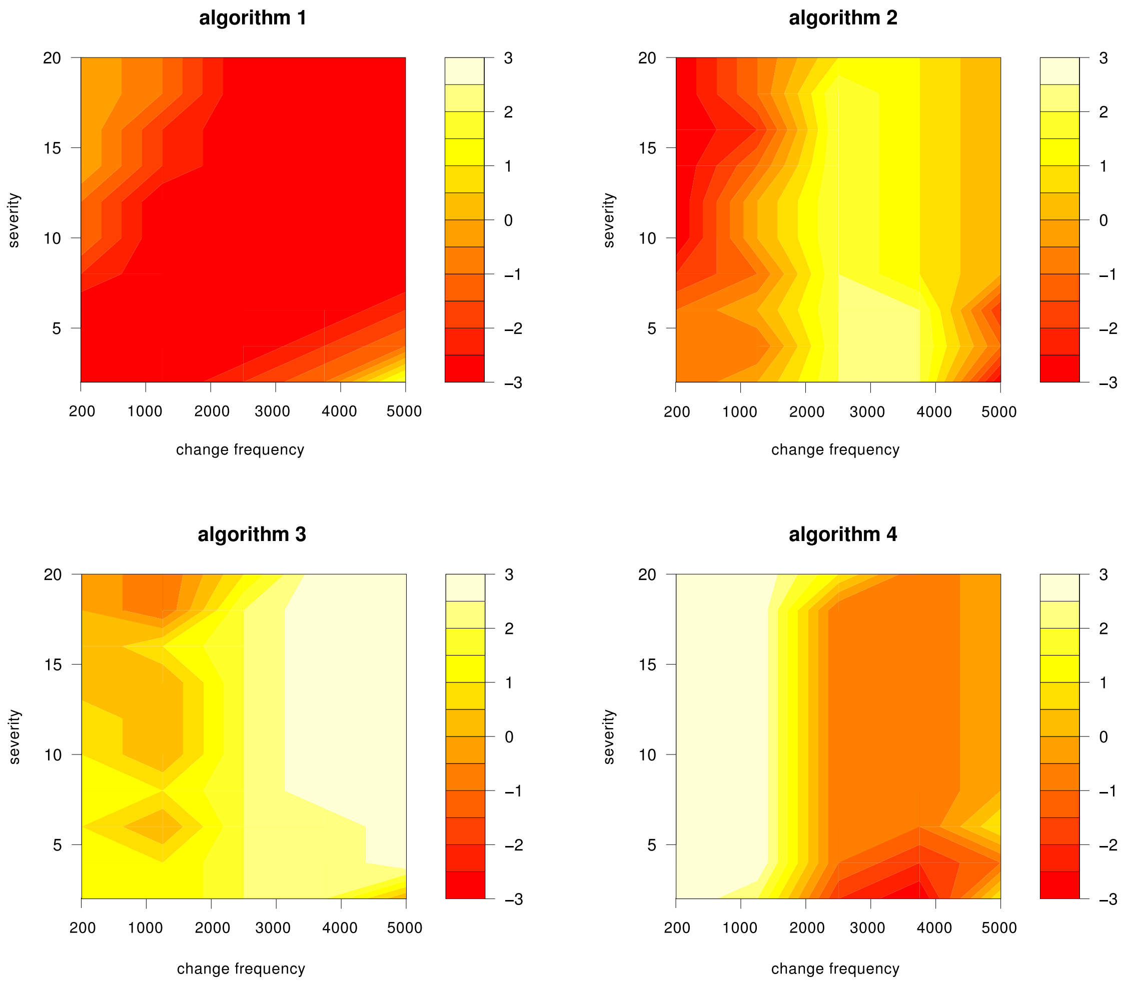


Figure 4.5 – An example of a graphical visualization of the rank-based color scheme for 4 hypothetical algorithms. The visualization shows a comparison of the results of 4 algorithms for different configurations of the factors *severity* and *change frequency* for a problem

- For high change frequencies (higher number of evaluations between changes), *algorithm 3* is the best.
- For low change frequencies, *algorithm 4* is the best
- Variations of the severity have, in general, less influence in the performance of the algorithms than variations of the change frequency.

Also, figures created using SRCS can be arranged to visualize variations of more than 2 factors, depending on the practitioner's creativity. These figures can help to further detect behavioral patterns of the algorithms, and increase our understanding of them.

Finally, although the examples in this section used the *avg. offline accuracy* as performance measure, and the KW + MWW combination as statistical tests, the SRCS technique is not restricted to these methods. Other performance measures (avg. offline error, reactivity, etc.) and statistical tests (Friedman, Iman-Davenport, etc) are also valid, as long as their usage is appropriated. In-depth examples of the use of non-parametric statistical tests for comparing optimization algorithms can be found in [43, 62, 63].

4.3. Comments on statistical issues for the SRCS technique

In this section we will comment on the number of independent repetitions for each experiment (N_r) and other issues relating statistical tests. It is generally accepted by the research community in the DOP area that the higher the number of these repetitions, the smaller the differences that a statistical test detects, which may lead to considering some differences as significant, when, in fact, they are not. This is an incomplete interpretation that may lead to erroneous statements.

When a MWW test is performed between two samples (i.e., the N_r results of two algorithms for a given configuration, in our context), the test attempts to determine if the two samples are statistically unequal, rejecting the null hypothesis H_0 of statistical equality. This is generally reported by the test in the form of a *p-value*, i.e., the probability of obtaining the actually observed differences in the two samples if the null hypothesis H_0 were true. Generally, one rejects H_0 if the p-value is smaller than a pre-fixed significance level α (e.g., 0.05). Moreover, if the two samples are concluded to be statistically unequal, the MWW test is able to estimate the magnitude of the difference, in absolute units (this is called the *effect size*).

When comparing results of two algorithms, it is very unlikely that the obtained samples are statistically equal, even if the two algorithms are very similar. There will always be differences, even if these differences are small. However, detecting those differences may require higher sample sizes at a given significance level. Quoting Vargha and Delaney [159]:

> [...] if population 1 and 2 are stochastically unequal, then at any fixed α level of significance the probability of the MWW test being significant at the α level tends to 1 as the m, n sample size values tend to infinity. By contrast, if populations 1 and 2 are stochastically equal, then the probability of the MWW test being significant at the α level will not tend to 1, however large the two samples be.

For example, suppose that the true difference between two algorithms' results is 1.5 performance units. Using 10 repetitions (sample size = 10) for each sample may not allow a MWW test to detect such differences at a significance level of $\alpha = 0.05$. In this case, the test will conclude that the null hypothesis cannot be rejected. However, using 50 repetitions (sample size = 50), the MWW may now be able to detect the differences at the same $\alpha = 0.05$ level, concluding that there are statistically significant differences.

Therefore, it is true that increasing the number of repetitions of an experiment may lead to a statistical test concluding that there are differences. However, this is not a flaw in the design of the experiment, but a sign of higher precision of the test. Think of it as if looking for differences between two physical objects using your naked eye (e.g., sample size = 10), or using a microscope (e.g., sample size = 50). It seems reasonable to think that with a microscope you are more likely to find differences between two objects than using only your naked eye.

This leads to the distinction between *statistically* significant differences and *practical* significant differences. Two lock keys (the original and a copy) may very well differ at a microscopic level, but this has little consequences, as long as both of them open their corresponding lock (which will likely require millimetric precision at most). On the other hand, if we need to copy a circuit board at a nanometric scale, we cannot use our naked eye for that purpose.

The first example (the key and the copy) illustrates a case where we have statistical significance, but no practical significance. The circuit board example, on the other hand, illustrates a case where we cannot conclude if we have practical significance because we do not have statistical significance. It should be an objective of an experiment design to gather as much data as possible to make any difference *statistically* significant. On top of it, it should be the researcher's responsibility to decide if these differences are of *practical* interest.

The concept of practical significance is closely linked to real-world problems. However, when we work with synthetic problems which value is purely academical — like, e.g., MPB, Ackley, Royal Road, etc. —, it is rather meaningless to use this concept. In these cases, our recommendation, as a rule of thumb, is that any experiment should gather as much data as possible in order to guarantee safe practical conclusions, and only be limited by practical issues like computational effort, time requirements, etc. In other words, without further information that could limit *a priori* the sample size (i.e., a minimum effect size for practical significance), **use a sample size big enough to ensure that at least the differences in the results are statistically significant**.

For more in-depth information about the use of non-parametric tests and questions on the factors that determine them, the interested reader is referred to [79, 91, 132].

4.4. Applying SRCS: algorithm comparison for continuous DOPs

The SRCS technique has been used in some published research works. The first one is [68], in which the Agents algorithm was adapted to discrete DOPs. However, since we have already discussed this work in detail in Sect. 3.3, we will not give further details here.

Instead, we are going to introduce a work that was not included in Chapter 3 because it does not contain any improvement nor new proposal of an algorithm. This work presents an extensive comparison of the performance of 8 algorithms over a wide range of continuous DOPs scenarios, using the SRCS technique for analyzing the results. This work was the proof of concept for SRCS, demonstrating that it can greatly help to obtain useful conclusions from vast amounts of experimental data.

4.4.1. Motivation

A high number of methods that deal with DOPs has already been proposed, but there are not yet clear criteria about which one is better to apply in each situation or how to face a new problem instance. Moreover, an algorithm may be well-suited for some problem configurations, but behave worse in others. Instead of developing an always best-performing method (remember the *no-free-lunch* theorem [171]), it may be more interesting to know which are the favorable conditions for each algorithm in order to choose the best available option. In this work, we introduce a first approach to this question, where we have selected a set of algorithms to compare their performance in different test benchmarks. We have attempted to select methods that are representative of diverse algorithmic families, in order to favor a wider scope of the comparison. We will now overview some of the families and algorithms that we have chosen.

Traditionally, the most commonly used algorithms in DOPs have been Genetic Algorithms (GA) and Evolutionary Algorithms (EA) [22, 23, 28, 134, 151]. An special remark is deserved to the Random Immigrants Genetic Algorithm (RIGA) [71], a GA algorithm aimed at maintaining a high level of diversity, which is a key aspect in DOPs. Some variants of RIGA are among the best-performing on this family, like SORIGA [151], which is the one that we have used in this work.

However, other approaches have been applied to DOPs, outperforming GAs and EAs in some cases. One of these approaches is Particle Swarm Optimization, PSO, [87] which is a quite competitive one, with several DOP-oriented proposals [14, 31, 83, 124]. Particularly, a PSO variant with multi-swarms, and quantum and trajectory particles, the mQSO, [15] has obtained good results and is frequently mentioned in the literature [13, 92, 180]. Some of our recent proposals include adding heuristic rules to this mQSO [39], improving its performance.

Finally, another group of algorithms that has obtained very good performance is that of Cooperative Strategies. We have conducted some experiments studying how cooperation between multiple optimization sub-components behave when applied to DOPs, with very promising results. These proposals include cooperative multi-agent systems [125, 126], or more recently, centralized schemes where a coordinator manages the information exchanged among a group of trajectory-based metaheuristic solvers (i.e., Tabu-Search solvers) [69, 70].

The objective of this paper is to compare a variety of algorithms for DOP's among the previously mentioned proposals, in order to gain knowledge on their generic properties, behavior and performance, as well as setting the bases for future comparison studies. The finally chosen algorithms are:

- an Evolutionary Algorithm, SORIGA [151], based in the RIGA approach,
- the standard mQSO [15] as well as 3 of its variations based on heuristic rules [39], that are named after the rule they use (*Change Rule*, *Rand Rule*, and *Both Rule*),
- the cooperative multi-agent algorithm proposed in [125],
- and finally, two trajectory-based Cooperative Strategies with different cooperation schemes, *independent* and *reactive* [69].

The test-case scenarios were chosen to include different representative benchmarks, namely the MPB and the dynamic versions of the Ackley, Griewank and Rastrigin functions. For each problem, a wide range of configurations which emphasize the influence of dynamism have been tested (variations of *change period*, *severity* and *dimensionality*), with a full-factorial experimental design combining all of them.

4.4.2. Algorithms used

Most of the algorithms used in this work have already been described in Chapter 3: the mQSO with heuristic rules (Sect. 3.2), the Agents algorithm (Sect. 3.3) and the CS algorithm variants (Sect. 3.4). We will only provide here a detailed description of the SORIGA algorithm, which has not been discussed before.

SORIGA (Self Organized Random Immigrants Genetic Algorithm) was proposed by Tinós and Yang in [151]. It is a variation of RIGA (Random Immigrants Genetic Algorithm) [71], a genetic algorithm where the flux of immigrants increases the diversity of the population.

The main modification introduced by SORIGA is the use of two different populations: the main population and a sub-population. Assignment of individuals to each population is performed dynamically during the execution of the algorithm, and crossover is only allowed between individuals within the same population. The pseudocode of SORIGA is given in Algorithms 4.1 and 4.2.

At the beginning, all the individuals belong to the main population, but as evolution goes on, individuals are extracted from it and assigned to the sub-population. This is done by means of the *replaceFractionPopulation()* function (Algorithm 4.1, line 8, and Algorithm 4.2). This function selects the individual with the lowest fitness and other $r_r - 1$ more around it (r_r is a user-defined parameter, the replacement rate), assign them to the sub-population, and replace them with random individuals. In case the less-fitted individual already belongs to the sub-population, all individuals are extracted from the sub-population and put back to the main population. Then, traditional mutation and crossover operators are applied, with the previously mentioned modification, i.e., that crossover is only allowed among individuals within the same population. Tinós and Yang show in their paper that the process of individuals joining the sub-population when the main population is close to a local optimum is similar to a "chain reaction", thus increasing the diversity (this feature being called *self-organized criticality*, SOC [7]).

Originally, SORIGA was designed for *discrete* DOPs. In order to adapt it to *continuous* DOPs, we have re-implemented it using continuous crossover and mutation operators. There are no previous studies on the use of SORIGA for the continuous domain that could give hints on the most appropriate operators to use. Since the objective of these work is not to search for the best design alternatives, but to gain knowledge on the generic properties, behavior and performance of the given algorithms, we decided to simply choose "reasonable" operators to adapt SORIGA. For the crossover, the well-known BLX-α operator [53] was selected. Regarding the mutation operator, we implemented a simple one that chooses one component of an individual and randomizes it uniformly over all its domain. This operator was chosen because it allows SORIGA to potentially explore the whole

Algorithm 4.1: SORIGA

```
1  $P \leftarrow$ population ;
2  $r_c \leftarrow$ crossover rate ;
3  $r_m \leftarrow$ mutation rate ;
4  $r_r \leftarrow$ replacement rate ;
5  $initializePopulation(P)$;
6  $evaluatePopulation(P)$;
7  while stopping condition is not met do
8      $P_{new} \leftarrow replaceFractionPopulation(P, r_r)$;
9      $P_{new} \leftarrow P_{new} + crossover(P, r_c)$;
10     $P_{new} \leftarrow mutation(P_{new}, r_m)$;
11     $evaluatePopulation(P_{new})$;
12     $P \leftarrow P_{new}$;
13 end
```

Algorithm 4.2: SORIGA, replaceFractionPopulation(P,r_r)

```
1  $P_{new} \leftarrow$ empty population;
2  $i \leftarrow$ index of the individual of $P$ with the lowest fitness;
3  if $P[i].replaced = false$ then
       // reset subpopulation to contain no individual
4      for each individual $j$ in $P$ do
5          $P[j].replaced \leftarrow false$;
6      end
7  end
8  for each individual $j$ in $P$ do
       // if individual $j$ is one of the $r_r$ individuals around
          individual $i$, randomize it and add it to the
          subpopulation; otherwise, just copy it
9      if $i - \lceil (r_r - 1)/2 \rceil \leq j \leq i + \lfloor (r_r - 1)/2 \rfloor$ then
10         $P_{new}[j] \leftarrow$ randomly generated individual;
11         $P_{new}[j].replaced \leftarrow true$;
12     else
13         $P_{new}[j] \leftarrow P[j]$ ;
14     end
15 end
16 return $P_{new}$ ;
```

search space.

Parameter	**Value**
Population size	50
Elite size	2
Replacement rate (r_r)	0.02
Mutation rate (r_m)	0.01
Crossover rate (r_c)	0.7
α (for the BLX-α operator)	1.0

Table 4.4 – Parameter settings for the SORIGA algorithm

The parameter settings used in the experimentation for the SORIGA algorithm are summarized in Table 4.4. In general, the parameter values were chosen to be the ones used in Tinós and Yang paper, with the exception of the BLX-α settings (not available in the original proposal since it was tested on a discrete problem) and the population size (in the original paper it was set to 120, a value that showed poor performance in some preliminary test in these problems).

4.4.3. Validation

Based on our experience, some of the parameters of a DOP that mostly affect the performance of algorithms are *dimensionality*, *change period*, and *severity* of the change.

In the experiments conducted in this work, a set of different values for each of the three previous parameters were selected, and a full factorial experimental design was used, meaning that all possible combinations of the parameter values were tested, for every problem, and every algorithm.

As we have previously mentioned, 4 problems were used to compare the performance of the algorithms: the MPB, and the dynamic versions of the Ackley, Griewank and Rastrigin functions.

The introduction of dynamism into the Ackley, Griewank and Rastrigin problems has been performed with the idea of keeping it as similar as possible to the MPB's. Therefore, we started from the MPB's dynamic base-settings, and we attempted to extend them to the rest of the functions. However, there are big differences between the MPB and the rest of the problems, and extending MPB's dynamism is not always easy to achieve, nor even possible in some cases.

The main difference among these problems is multimodality: while Ackley, Griewank and Rastrigin are inherently multimodal functions, the MPB is only multimodal by means of a composition of unimodal functions (peaks). This means that the MPB has a small number of local optima, around 100 in our experiments, while Ackley, Griewank and Rastrigin may have up to thousands. On the other hand, each local optimum of the MPB can be controlled individually (each peak has a position, height, and width), while the local optima of the other functions are pre-determined, meaning that their location depend only on the function's origin of coordinates and their height and area of influence is fixed.

Dynamism in the MPB is obtained by periodically modifying the properties of each peak: *position*, *height* and *width*. Of these 3 properties, *width* is exclusive of the MPB's peak function, and cannot be translated to any property of the Ackley, Griewank or Rastrigin functions. *Height* can be translated, but it is only meaningful in the MPB, because each peak has its own height (some peaks increase their height, while others decrease it). Varying the height in the other problems will not affect the algorithm's performance, since it is equivalent to add or subtract a constant value to the whole function's fitness, producing the same landscape with shifted values, but without changing the structure of the function nor the height of individual local optima. Therefore, the only property that can be reasonably translated is *position*. We achieve this by explicitly introducing the origin of coordinates of the function as a parameter, $\mathbf{p}$, through the following expression:

$$\begin{aligned}&f'(x) = f(x - \mathbf{p})\\&f \in \{Ackley, Griewank, Rastrigin\}\end{aligned}$$

Dynamism is added by periodically changing the origin of coordinates $\mathbf{p}$ using the MPB movement equations 2.4 and 2.5. In consequence, parameters *change period* and *severity* can also be defined for these problems.

The final result, however, is not exactly the same, since peaks in the MPB move independently of each other, while local optima of Ackley, Griewank and Rastrigin functions move all as a block. Although this dynamism may not be as elaborated as in the MPB's, this does not necessarily mean that these problems are easier to solve (it is still harder to optimize a function with thousands of local optima than one with only a hundred). However, it is possible, and even expected, that this situation may benefit some algorithms in detriment of others, for some cases. This is actually the case in the experiments, and we will discuss it in Sect. 4.4.4. Far from seeing this as a flaw, we believe this is yet another feature that increases our knowledge on what are the environmental conditions that mostly affect the performance of algorithms, which is one of the main objectives of this work.

As we have mentioned, the Scenario 2 of the MPB has been used as a baseline configuration for the experiments, in order to keep the results related with the existing literature. Therefore, the set of values for the parameters were explicitly chosen to contain a value within the range of those defined by Scenario 2.

The complete set of parameter values is listed below, where the bold-faced value indicates the one that corresponds to the Scenario 2 settings:

- dimensionality: {**5**, 10, 15, 20, 25}
- change period factors {40, 100, 200, 300, 400, 500, 600, 700, 800, 900, **1000**}
- severity {**2%**, 4%, 6%, 8%, 10%, 12%, 14%, 16%, 18%, 20%}

The change period parameter indicates how many function evaluations is the algorithm allowed to consume before a change in the environment occurs. However, it is well-known that dimensionality affects the ability of an algorithm to optimize a function: the higher the dimensionality of the problem, the lower the performance. Therefore, instead of fixing the number of function evaluations allowed for a certain configuration independently of the dimensionality, we consider the change period as a *factor*: the final change period value is obtained multiplying the change period factor by the dimensionality. For example, using a change period factor of 1000 and 5 dimensions, the number of function evaluations between changes in the environment is 5000 (which are actually the values for the Scenario 2). However, the same change period factor used in 20 dimensions means that the number of function evaluations between changes is 20000.

Also, severity depends on the range of the input variable x (all x_i dimensions have the same range in the benchmarks), but this range is different for each problem. In order to unify test configurations, instead of using absolute severity values, a percentage of x's range is used.

Parameter	**MPB**	**Ackley**	**Griewank**	**Rastrigin**
Num. dimensions (n)	$\{5, 10, 15, 20, 25\}$			
Change period (ω)	$\{40, 100, 200, 300, 400, 500, 600, 700, 800, 900, 1000\} \cdot n$			
x range	$[0, 100]^n$	$[-32, 32]^n$	$[-50, 50]^n$	$[-5, 5]^n$
Severity (s)	$\{2\%, 4\%, 6\%, 8\%, 10\%, 12\%, 14\%, 16\%, 18\%, 20\%\} \cdot x$ range			
Corr. coefficient (λ)	0.5			
Num. peaks (m)	100	-	-	-
Peak heights (h_j)	rand [30, 70]	-	-	-
Peak widths (w_j)	rand [1, 12]	-	-	-
Height severity (h_s)	7.0	-	-	-
Width severity (w_s)	1.0	-	-	-

Table 4.5 – Experiment settings for the MPB, Ackley, Griewank and Rastrigin functions

The final test configurations for all problems are summarized in Table 4.5. From this table we can calculate the number of configurations to test for every problem. If we also consider all the problems and algorithms, we obtain the total amount of experiments to perform:

$$5 \text{ dimensions} \times 11 \text{ change freq.} \times 10 \text{ severities} = 550 \text{ configurations}$$
$$550 \text{ configurations} \times 8 \text{ algorithms} \times 4 \text{ problems} = 17600 \text{ experiments}$$

Due to the high number of experiments, it is almost impossible to show all the numerical results of the algorithms in a comprehensive way using tables. In order to overcome this, we need to compact the information to be displayed, and present it in a more manageable format. Recall that we want to provide an overview of the behaviour and we are not searching for the "best" algorithm. For this purpose, we have used the SRCS technique introduced in this chapter to display the data.

Finally, regarding the performance measure used, there are several ones available in the DOP literature (see, for example, [28, 114, 165]). In order to make our results comparable with other closely related works (like, e.g., [39, 41, 68, 69], presented in Chapter 3), the performance measure selected for our experiments was the *offline error* (e_{off}) [28] (see Sect. 2.2.1).

The execution of an algorithm for N_c changes is defined as a *run*. In order to obtain statistically meaningful results, each experiment consisted of N_r independent runs, each one with a different random seed. In this work, we have chosen $N_r = 50$, and the comparison of the algorithms has been performed using those 50 e_{off} measures for each algorithm.

The results of the experiments are presented in the form of 4 figures arranged as tables (Figs. 4.6, 4.7, 4.8, and 4.9). Each one corresponds to a problem, and shows the ranking results of the algorithms for all the configurations of that problem. Each row corresponds to the results of a given algorithm, and each column to the results of a given dimensionality of the problem. Cells show the results of all change period vs. severity configurations, for that algorithm (row) and dimensionality (column).

Results for the MPB problem are shown in Fig. 4.6, for the Ackley problem in Fig. 4.7, for the Griewank problem in Fig. 4.8, and finally, for the Rastrigin problem in Fig. 4.9.

4.4.4. Conclusions

Eight algorithms have been chosen for the study: SORIGA, the Agents algorithm, the standard mQSO and 3 of its variations based on heuristic rules (mQSO + Change Rule, mQSO + Rand Rule, mQSO + Both), and 2 Cooperative Strategies based on trajectory metaheuristics, one with no cooperation between them (independent), and the other one with a cooperation scheme based on the ideas of Reactive Search.

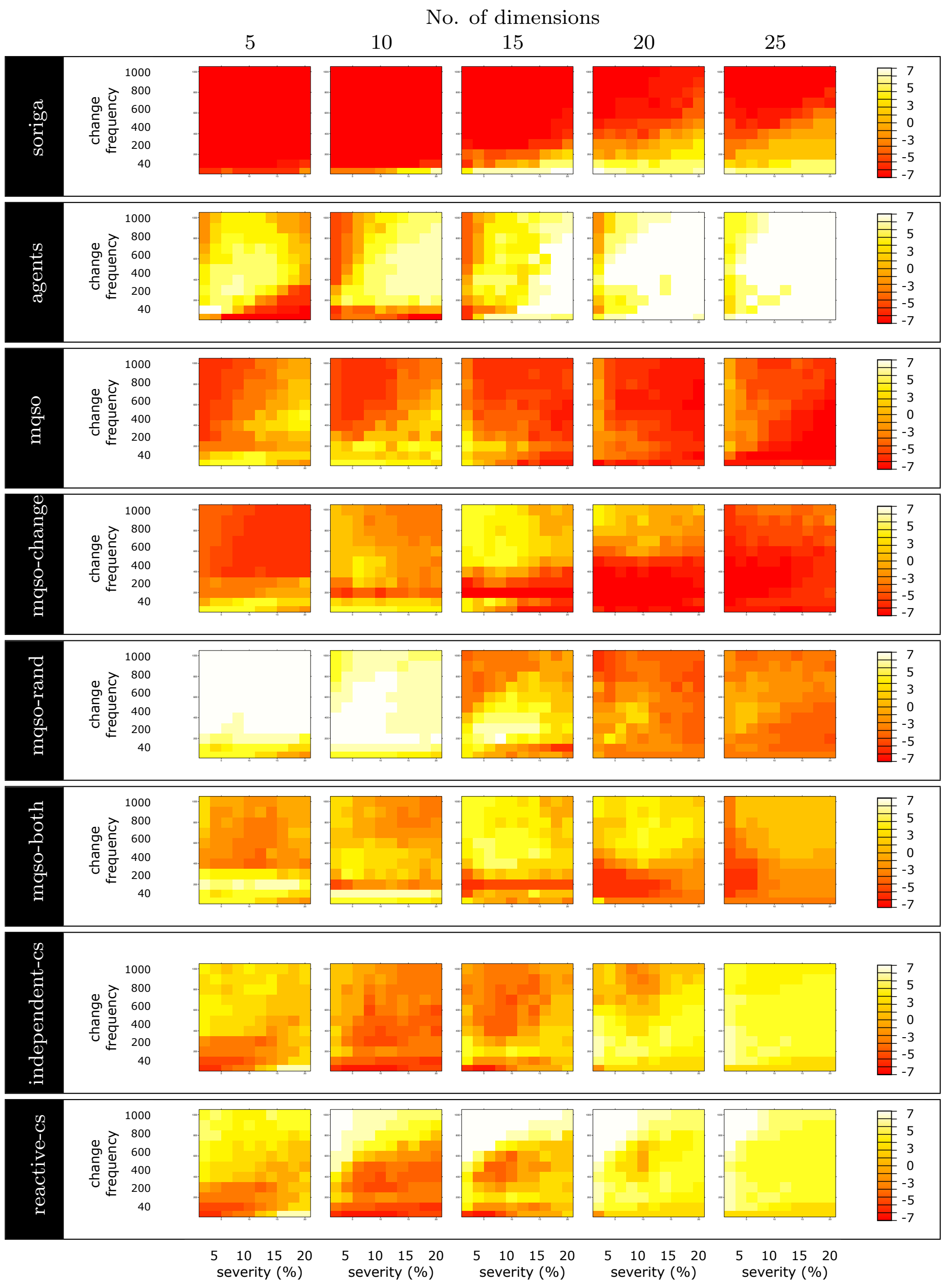


Figure 4.6 – Rank results of all the algorithms for the **MPB**.

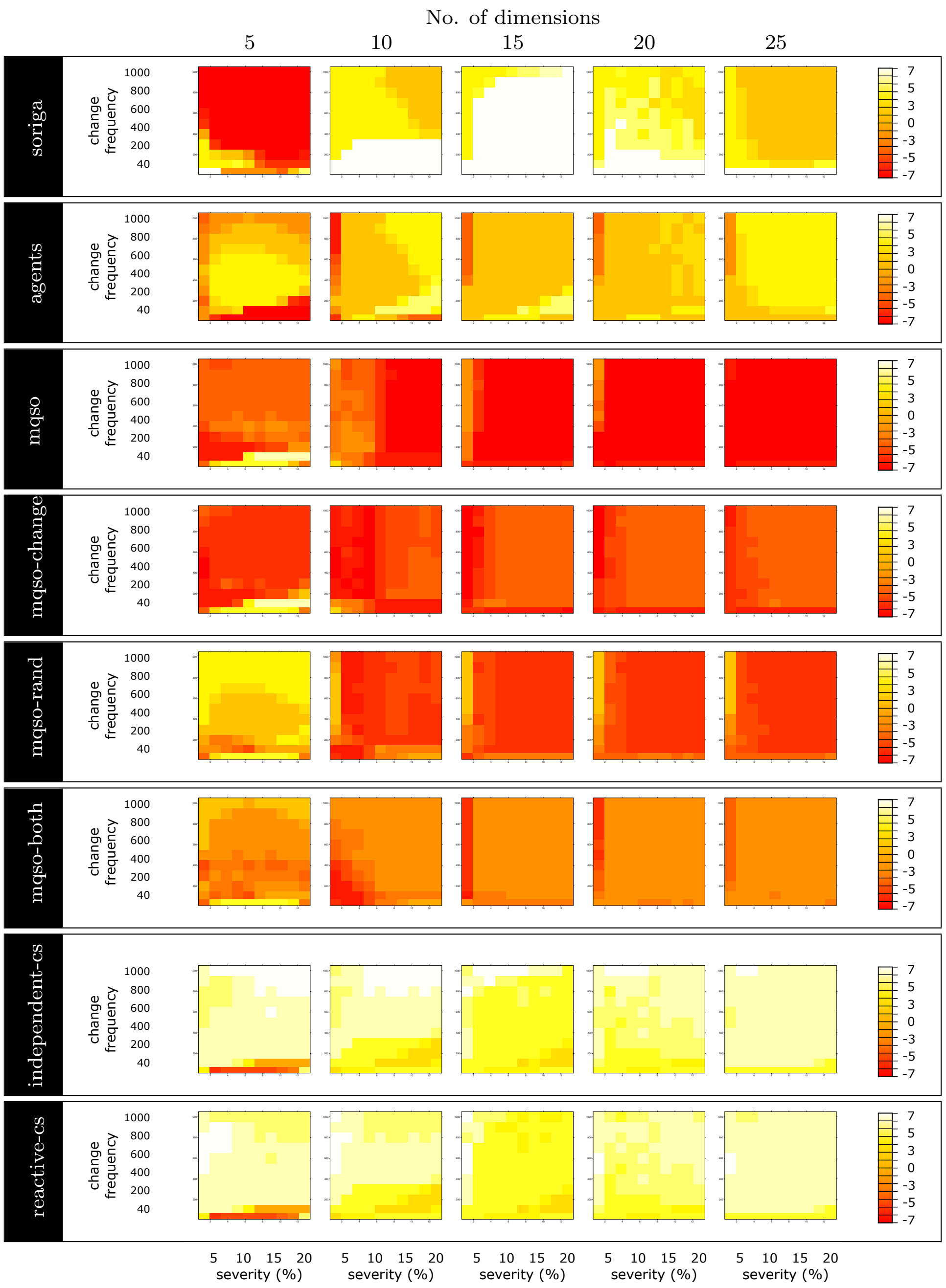


Figure 4.7 – Rank results of all the algorithms for the **Ackley** function.

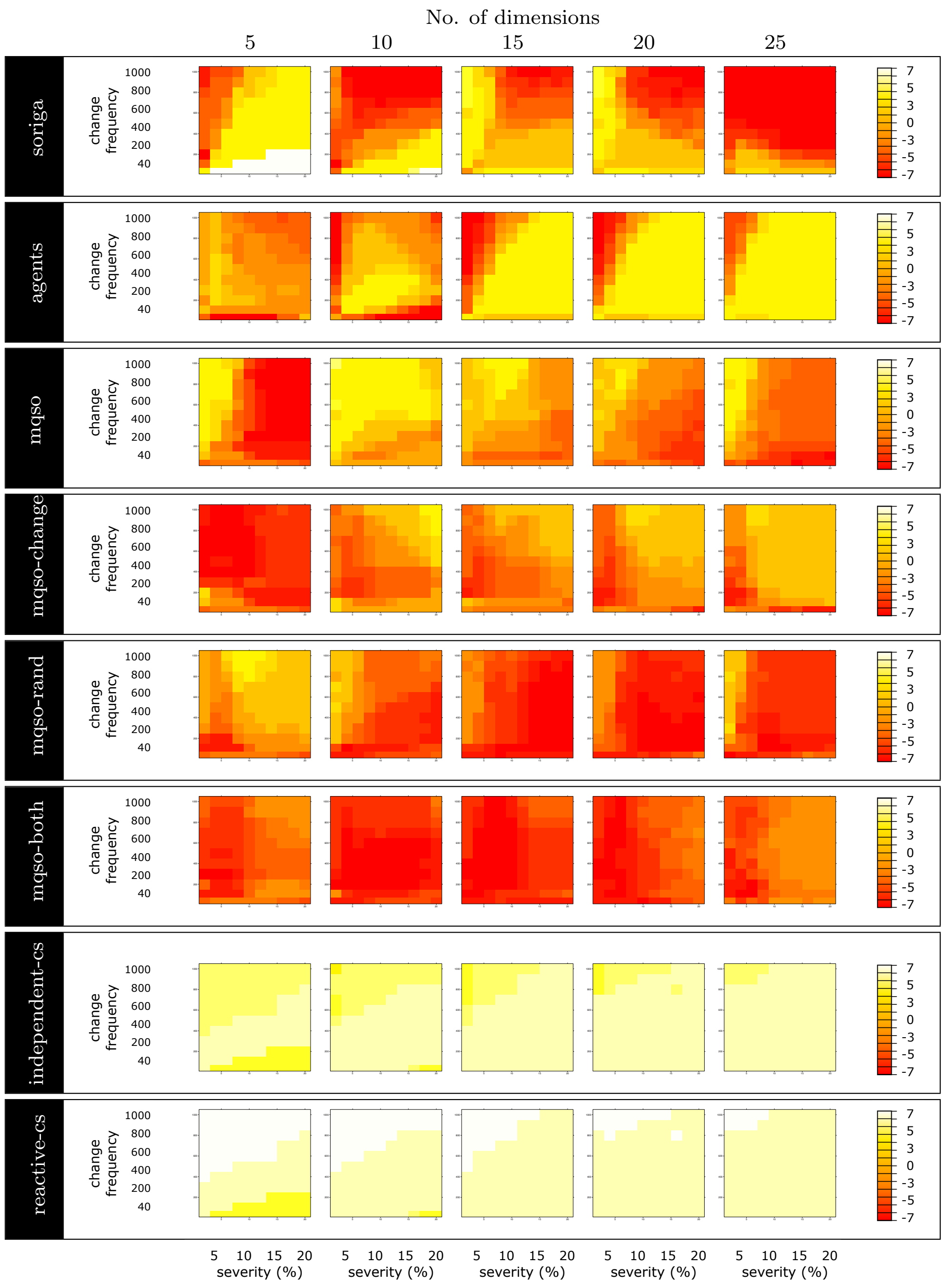


Figure 4.8 – Rank results of all the algorithms for the **Griewank** function.

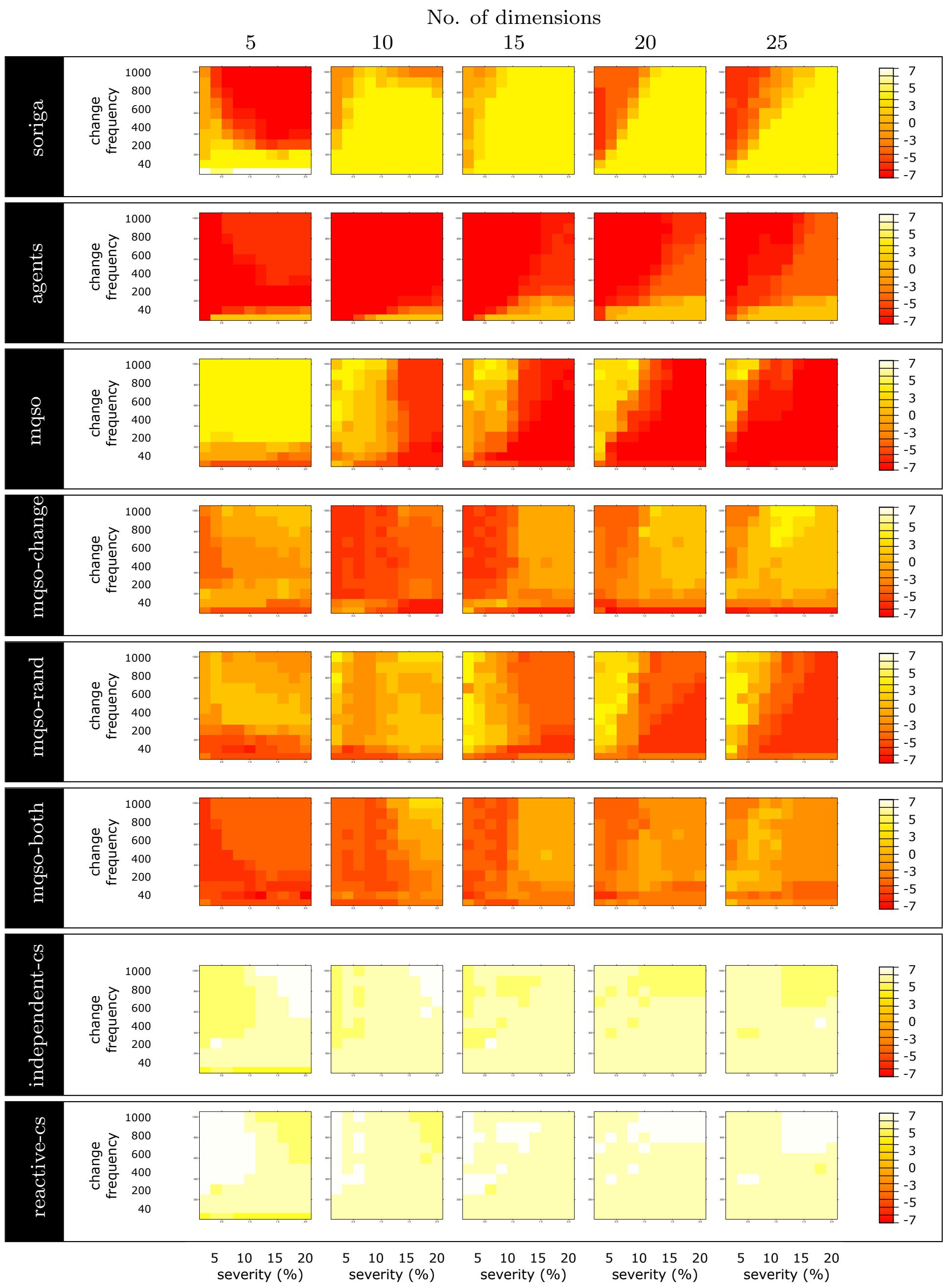


Figure 4.9 – Rank results of all the algorithms for the **Rastrigin** function.

The algorithms have been tested on the Moving Peaks Benchmark (MPB) and on dynamic versions of the well-known Ackley, Griewank and Rastrigin functions. For each problem, a wide variety of configurations have been used, including variations of the dimensionality of the problem, the frequency of the changes on the environment, and the severity of those changes.

A recent technique, SRCS, has also been used for presenting the results of a high number of experiments when comparing several algorithms. The methodology focuses on ranking the algorithms using statistical tests, and assigning color keys to each rank, in order to create a graphical matrix. These matrices are more expressive and comprehensible than their numerical counterparts.

From the results, the following conclusions can be obtained:

- For the **MPB**: there are 3 algorithms which are better than the other 5 for some configurations. These algorithms are the *agents*, the *mqso-rand*, and the *reactive-cs*.

    - the *agents* algorithm seems to be more effective than the others for higher dimensions (15-25), and is more influenced by the severity (better with higher severity) than by the change period.
    - the *mqso-rand*, on the contrary, is better suited for lower dimensions (5-10), and is more dependent on the change period (better with higher number of function evaluations), than on the severity.
    - the *reactive-cs* seems to be better for low severity, high number of function evaluations, higher dimensionality.

- For the **Ackley** function: there are, again, 3 algorithms which are clearly better than the others: *soriga*, and the two variants of the Cooperative Strategies, *independent-cs* and *reactive-cs*.

    - both *independent-cs* and *reactive-cs* have a very similar behaviour, and clearly outperform the rest of the algorithms for most of the configurations.
    - *soriga* is the only algorithm that performs better than the Cooperative Strategies for some configurations, which are mainly those with dimensionality between 10 and 20, and for very rapidly changing environments.

- For the **Griewank** and **Rastrigin** functions, the results are almost identical, with the Cooperative Strategies clearly dominating the rest of the algorithms, and *soriga* being the only one performing better for a reduced set of configurations, mainly very low dimensionality (5), very low change period (40-100), and for any severity. Between the Cooperative Strategies, *reactive-cs* seems to be slightly better than *independent-cs*.

This shows that the Cooperative Strategies are the best choice for problems with some kind of regular structure on their local optima distribution (e.g., Ackley, Griewank, Rastrigin). SORIGA is the best option for very rapidly changing environments. When the problem's local optima distribution is more random, like in the MPB, the mQSO + Rand Rule obtains the best results for low dimensionality, while the Agents behaves better for higher dimensionality and growing severity of the changes.

This comparison of algorithms has been published in [40]:

"An Algorithm Comparison for Dynamic Optimization Problems", I. G. del Amo, D. A. Pelta, J. R. González, and A. D. Masegosa, *Applied Soft Computing*, 12(10):3176–3192, 2012. `http://dx.doi.org/10.1016/j.asoc.2012.05.021`.

4.5. Conclusions

In this chapter we have presented a new technique, SRCS (Statistical Ranking Color Scheme), specifically designed to analyze the performance of multiple algorithms in DOPs over variations of several factors (e.g., change frequency, severity, dimensionality, etc). This technique is especially well-suited when we want to compare algorithms in a all-vs-all manner, for example, when we want to determine which are the best performing ones in a wide range of scenarios.

SRCS uses statistical tests to compare the performance of the algorithms for a given problem configuration, producing a ranking. Since the results of metaheuristics and non-exact algorithms do not generally follow a normal distribution, non-parametric tests are usually preferred. As a practical guideline, a multiple-comparison test must be performed first, like the Kruskal-Wallis test, in order to determine if there are global differences in the performance of the algorithms. Then, a pair-wise test is used, in order to assess individual differences between algorithm pairs, like the Mann-Whitney-Wilcoxon test. This pair-wise test must be adjusted in order to compensate for the family-wise error derived from the performance of multiple comparisons, using for example Holm's method. However, these tests are only suggestions that do not affect the way in which SRCS works, and other options can be used (Friedman's test, Iman-Davenport, etc).

The ranking produced is later used to associate color codes to each algorithm result, such that the *relative* performance of each algorithm with respect to the others can be represented in a graphical way. This representation allows to visualize the results of many algorithms on many configurations in a much more compact way by enhancing differences between the results, and giving thus the researcher the possibility of identifying behavioural patterns more easily.

Like any information compressing technique, SRCS lefts out part of the information, so its use, either isolated or as a complement to other traditional ways for displaying results (tables and plots), should be evaluated in each case. With SRCS, using rankings for stressing out the differences among algorithms implies not displaying absolute performance values.

Additionally, we have shown 2 works [40, 68] where we have used the SRCS technique. Of those, [68] was already commented on Sect. 3.3, and it has not been discussed again here. On the other hand, we have analyzed in depth the comparison of algorithms presented in [40], where we have been able of extracting valuable conclusions about what are the general behaviours of the algorithms used, their trends, and in which cases it is better to use one or another.

The SRCS technique was published in [38]:

"SRCS: a technique for comparing multiple algorithms under several factors in Dynamic Optimization Problems", I. G. del Amo and D. A. Pelta, in *Metaheuristics for Dynamic Optimization* (E. Alba, A. Nakib, and P. Siarry, eds.), volume 433 of *Studies in Computational Intelligence*, Springer Berlin/ Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-30665-5_4`

Chapter 5

Software design and implementation

In this chapter we present the most significant software contributions that we have performed through all the thesis. These contributions are grouped into two main ones: DACOS, a framework for the design and analysis of cooperative optimization systems, and the MODO Optimization Package, a set of libraries containing most of the software developed during the experiments. We will explain the circumstances that motivated these contributions, their architecture, and the conclusions extracted from their development.

5.1. DACOS

DACOS (**D**esign and **A**nalysis of **C**ooperative **O**ptimization **S**ystems) is a personalized framework for the configuration and analysis at design-time of rule-based centralized cooperative optimization systems. Software modeling tools (and more precisely, the Eclipse Modeling Framework [50, 148]) were used to build DACOS. Software modeling tools are focused on the use of models, which, among other advantages, allow for the automatic generation of code. The use of these tools speeds up the system's build up time, and what is more important, reduces the programming errors. Also, the generated code usually follows widely accepted software design patterns, which increases its robustness and flexibility.

5.1.1. Motivation

In Sect. 3.4 we talked about the CS algorithm, in which a set of metaheuristics cooperated among them by means of a central coordinator. This working mechanism is not exclusive of the CS algorithm, so we will denote any algorithm based on this architecture as Cooperative Optimization Systems (COS).

The performance of a COS is measured in terms of the quality of the solutions it can provide using certain amount of resources (e.g. time), and this is directly related with the system's definition and configuration. Configuring and analyzing a COS is somehow a generic task: it is about setting parameters, deciding which modules are going to be used and which are not, and specifying what output should be plotted. Intuitively, this task can be thought of independently of the particular problem or algorithms being used. However, when it comes to the practice, each problem and metaheuristic has its own API and parameters, and produces its output in its own format. As a consequence, the configuration and analysis processes must be personalized for each implementation, making it very difficult, and even undesirable, to develop a generic system for every problem and COS. Instead, we can help and assist the researcher as much as possible in building a specific system for his particular needs.

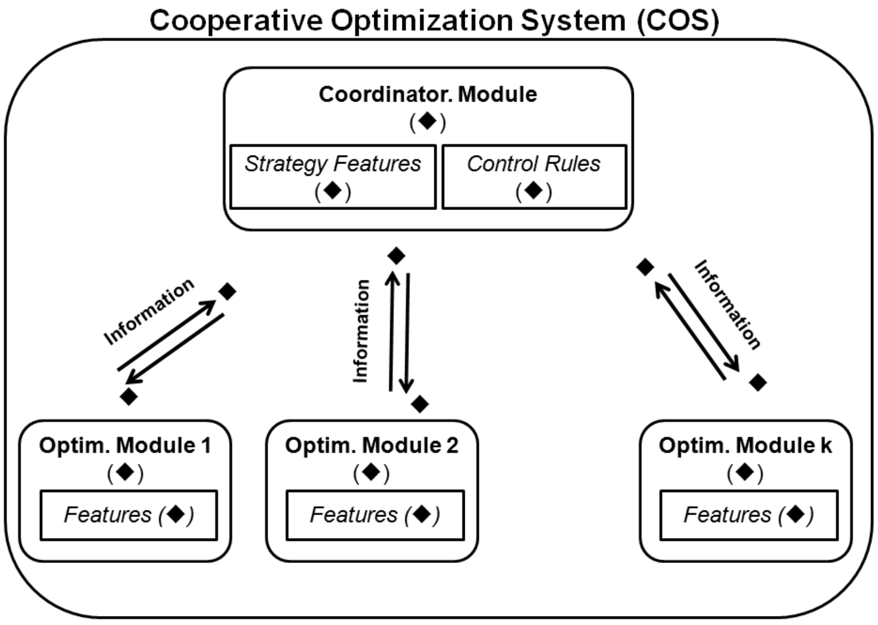


Figure 5.1 – .

The main components of a COS are depicted in Fig. 5.1. The system is composed by a set of basic optimization modules (metaheuristics) whose work is controlled in a centralized way through a coordinator module. The symbol ⧫ denotes a system's component that needs to be configured prior to the execution of the COS.

Configuring an *optimization module* implies choosing a metaheuristic and setting up all of its parameters. Different metaheuristics have different sets of parameters, and they may range from values indicating a path for a file, single real/integer/character values, or more complex parameters like operator's definitions, etc. In the context of a COS, this task must be repeated for each module.

Configuring the *coordinator* also implies setting its own parameters. Moreover, if the coordinator uses a rule base to control the optimization modules, as in [35, 69, 70, 127], it is also necessary to define the antecedent (set of conditions) and the consequent (the actions to be taken) for each of the rules.

Finally, parameters related with the *communication channels* between the optimization modules and the coordinator also need to be set. For example, the type of information exchanged (best solution, number of local optima, distance between them...), the communication frequency, etc.

On the other hand, usually this parameter-setting process is performed by repeating many times the cycle *configure-execute-analyze*. For example, we may start with three optimization modules, then perform a test, and adjust the configuration according to the results. These adjustments could mean setting some parameter values, but also duplicating some modules and having two copies of each one with different parameter's settings, or replacing one of the modules with another metaheuristic, or using a different communication frequency, etc. Besides, it is usually necessary to wait for the COS to completely finish its execution before studying its output. This way of proceeding is inefficient, since it is often possible to identify if the behaviour of the system is acceptable by simply observing partial results. Nevertheless, this visualization is not trivial, since every problem, even each metaheuristic, may require different techniques (linear plots, pie charts, boxplots, user-defined visualizations, etc).

It seems clear that defining/configuring and analyzing/visualizing a COS can be quite complex, and therefore, an integrated system for helping in these tasks would be highly desirable. In order to develop a software tool that helps us and other researchers to efficiently handle these processes, our experience suggests that the following requirements should be addressed:

1. *Ease of use*: the configuration of the system's modules must be simple and intuitive, supporting the user whenever it's possible in repetitive or automatable tasks.

2. *Flexibility*: it must be possible to adapt it to different types of modules, problems, and visualization charts in a fast and simple way.

3. *"Real-time" visualization*: it must be able to show results as they are being generated, in order to give a feedback to the researcher as soon as possible.

4. *Features integration*: it should be desirable to configure and analyze the system within the same environment, in order to facilitate an iterative application of the tasks *design* $\rightarrow$ *analysis*.

5. *Easiness of integration with other tools*: it must have a low coupling degree between components, as well as using standard data interchange formats. The tool should be preferable open source, which, along with the use of standards, would facilitate its distribution and integration.

In order to meet these requirements, we resort to software modeling techniques. A software model is an abstract and formal representation of a system, independently of its associated implementation particularities. The use of software models allows the developer to focus on the general aspects of the system, avoiding technology-related details, such as the programming language or the communication protocol between components. Since a model is also a formal representation, subject to rules, it is possible to create software tools that assist in the development of the model and automatically produce code from it. Software modeling is a mature technique nowadays, with consolidated tools that allow for a useful and productive application of its principles.

The tool presented here, DACOS, is intensively based on one of the most well-known applications in this area: the Eclipse development environment [48, 106] and its software modeling project EMF [50, 148]. These, along with the BIRT charting library [11], account for the core of DACOS.

5.1.2. DACOS Architecture

DACOS is composed by two modules, one for the design or configuration of the cooperative optimization system, and the second one for the analysis and visualization of their results.

An important feature to take into account is the level of coupling between DACOS and the particular COS we are dealing with. Both systems could be integrated within the same monolithic program, but an alternative approach is more advisable due to the differences in their requirements. The objective of a COS is to solve a hard optimization problem, which is a computationally intensive task. Its code implementation must be efficient, and once its execution has begun, it usually does not need any input from the user, so it does not require any graphical interface. On the other hand, DACOS does not participate in the optimization process, nor requires high-end hardware for its execution, but it does need to interact with the user, preferably using a GUI.

As a consequence, DACOS is implemented as an independent program. This approach not only permits to design each system in the most appropriated way, according to their requirements, but it would also allow them to be executed in different machines (e.g., DACOS in a desktop computer, and the COS in a dedicated server).

This situation makes it necessary to define a communication interface between them, in order to diminish the coupling as much as possible. This guarantees that both DACOS and COS could evolve (at a version level) without affecting each other, as long as they honour that communication interface. Also, the use of this established interface implies that a certain COS's implementation can be substituted by another one, and viceversa, the very same strategy can be configured by different implementations of DACOS, or even by external applications. Also, COS and DACOS may be run on different machines, so in order to avoid time dependencies, it is advisable that the use of an asynchronous communication interface. This can be achieved by using structured data files. Fig. 5.2 shows an example of the workflow sequence between DACOS and COS, as well as the use of data files for the communication.

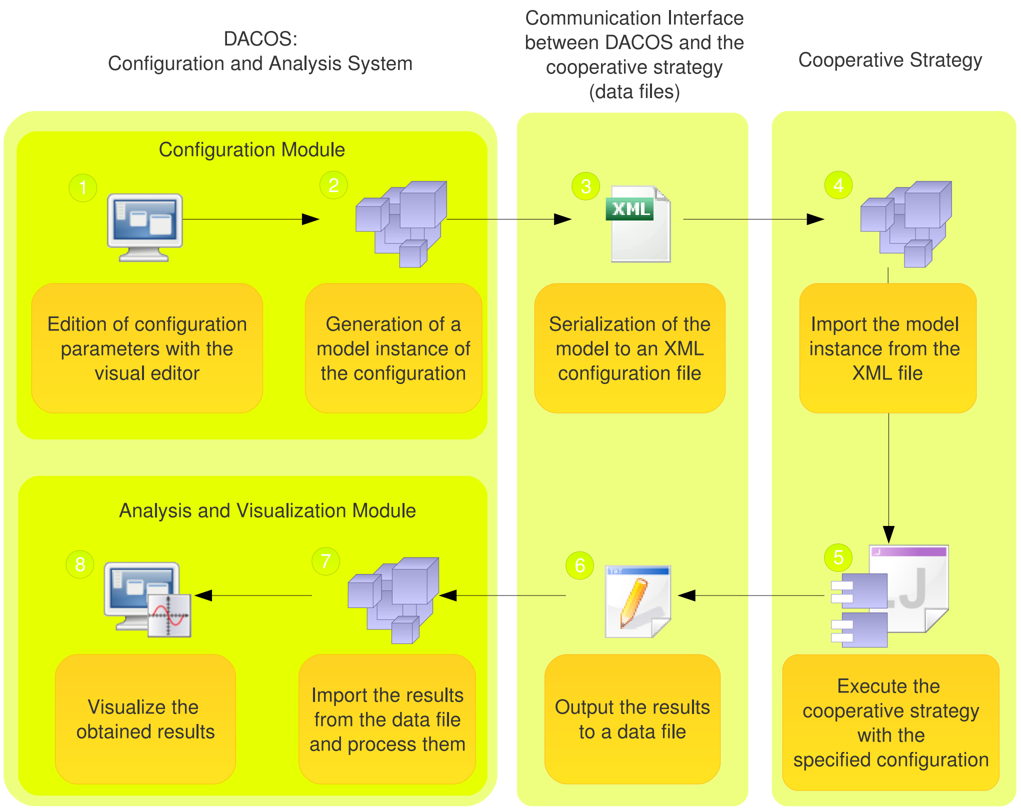

DACOS:
Configuration and Analysis System
Communication Interface between DACOS and the cooperative strategy (data files)
Cooperative Strategy
Configuration Module
1
2
3
XML
4
Edition of configuration parameters with the visual editor
Generation of a model instance of the configuration
Serialization of the model to an XML configuration file
Import the model instance from the XML file
Analysis and Visualization Module
8
7
6
5
Visualize the obtained results
Import the results from the data file and process them
Output the results to a data file
Execute the cooperative strategy with the specified configuration

Figure 5.2 – Workflow sequence between DACOS and COS.

As will be explained later on, software models have been used for the creation of both the design and the analysis modules. For the case of the design module, it has been necessary to model the data structures that are used for specifying initial parameters for the COS. The analysis module, however, is slightly more complex, and both data structures and the objects that use them were needed to be modeled, in order to go from the raw data representation of the results to its proper visualization as an integrated chart. Eclipse and its modeling project EMF, have been used for these tasks. Eclipse is structured as a set of core functionalities with a plugin mechanism. This allows the addition of new features to the environment without interfering with the previous ones. For the case of DACOS, this meant that the configuration and analysis modules could be packed as a plugin for Eclipse, avoiding the complexity of dealing with windows or event managing, buttons, scrollbars, etc. Moreover, with the help of plotting libraries, the development of visualization tools to help understanding the behavior of the COS at hand can be effectively implemented.

5.1.2.1. Design Module

The aim of the design module is to allow the user the management of the design alternatives for a COS in an intuitive and friendly way. As was previously explained, this is not a trivial task, as it must be able to deal with a great variety of elements that depend on the problem, the metaheuristics used, the coordinator's rules, etc. In Fig. 5.3 it can be seen a screenshot of this module at work.

At the beginning of this section, it has already been justified the use of data files as a communication interface between DACOS and COS. Focusing on the configuration module, one of the most popular file formats for data interchange has been used: XML (`http://www.w3.org/TR/REC-xml/`). The use of XML as a format for the definition of input data for optimisation programs is supported by other authors which have successfully used it [119]. The main problem now consists on defining the structure of this XML file, i.e., it is necessary to specify a model for the configuration data it contains. In this work, we consider a COS where the coordinator module uses IF-THEN rules, so the following aspects have been considered:

- Regardless of the type of the optimization module, it is always necessary to specify certain parameters related to the problem at hand, like number of variables, constraints, files location, etc. Thus, in the configuration, there will always be an element labeled *Problem*. However, there are many potential different problems to fix the structure of this element. Therefore, the element *Problem* is defined to contain a list of subelements named *Property*, holding pairs (name,value). In this way, each problem will contain the attributes that the researcher requires. This type of structure will repeat in

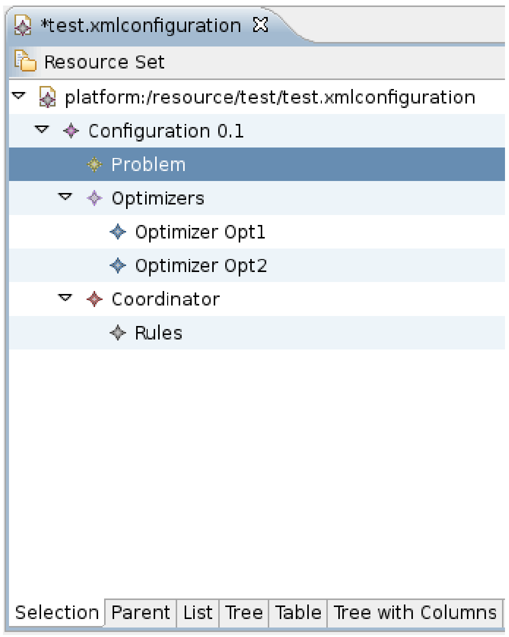


Figure 5.3 – Screenshot of the design module.

almost every element of the configuration, since it represents a compromise between a predetermined structure that can be validated, and a flexible one that can be easily adapted to the peculiarities of the problem.

- Like in the previous case, it will always be necessary to specify which metaheuristics are going to be part of the COS, including their parameters. Therefore, a list of *Optimizer* elements are used as a fixed structure. However, as it happens with the *Problem* element, the great variety of metaheuristics and configuration parameters makes it almost impossible to determine an *a priori* fixed structure for this element that would be functional. Thus, each *Optimizer* element contains again a list of *Property* elements that indicate its attributes.

- Central coordinator's properties are set via the *Coordinator* element, and its set of *Property* subelements.

- Finally, the rules are specified by a list of *Rule* elements, each of them containing an *Antecedent* element and a *Consequent* element, with their corresponding *Property* elements to indicate its values.

Fixed elements (such as *Problem*, *Optimizer*, etc) allow for an automated validation of configuration values when using software models, but force other researchers to use that elements in their configuration files; on the other hand, wildcard elements (such as *Property*) provide the researchers with greater flexibility, but does not allow to automatically check their values.

In order to provide a reasonable balance between automated validation and flexibility, we have defined fixed elements for top-level concepts, such as *Problem*, *Metaheuristic*, *Rule*, etc. These elements are generic enough to ensure that they can be used in most situations. On the other hand, we have not forced to use any fixed element for low-level concepts (say for example, some parameter α of a certain metaheuristic), and instead, we have used wildcard *Properties*, that cannot be validated by the software in run-time, but allow the researchers to used them freely according to their needs.

5.1.2.2. Analysis and Visualization Module

The analysis and visualization module is the responsible for showing the results of the COS in a meaningful way. Although there are programs that are able to perform this task, the objective here is to have a high degree of control over the visualization process, so the use of libraries is preferable to external applications.

The first problem that needs to be solved is deciding what would be the format of the COS's output data in order to be processed and visualized by this module. The first choice could be using an XML format, as it happened with the configuration module, e.g.:

```
<results>
<data>data 1</data>
<data>data 2</data>
<data>data 3</data>
...
<data>data N</data>
</results>
```

where each element `<data>` corresponds to a partial result generated as the cooperative strategy explores the solution space.

However, this format has a drawback. The element `<results>` is not closed until all the `<data>` elements are written, and thus, the XML file will not be well-formed in the meanwhile. This implies that it can not be parsed correctly, and it can not be used to visualize the results in real time while they are being generated.

The alternative chosen here is to use a different format, a variant of the CSV (comma-separated values) that uses the character ";" as separator. This format has the advantage that it does not need to be well-formed, since it doesn't contain tags that must be opened and closed, all the data is arranged in columns. Thus, it is enough to guarantee that all the data of a partial result is written in columns, and that each partial result is dumped to a different line. Another consideration is that the analysis module must not read a line before COS has finished writing it, but this is usually a problem that can be addressed and resolved at the operating system level. Additionally, many external visualization programs accept input data in CSV format, so they can be used as an additional verification method of the results or even as alternative tools.

The next question that needs to be addressed is how to move from the data in the CSV file to the chart in the analysis module. It has to be noted that a chart usually shows one or more variables against another (e.g., global error vs. time, cost of the best solution found by each metaheuristic vs. iteration, etc). However, the file contains in each line all the data generated in a partial result (time, iteration, error, fitness of each solution, etc.). Therefore, at some point in the process, it is necessary to select the variables to present. This can be better understood with the diagram presented in Fig. 5.4.

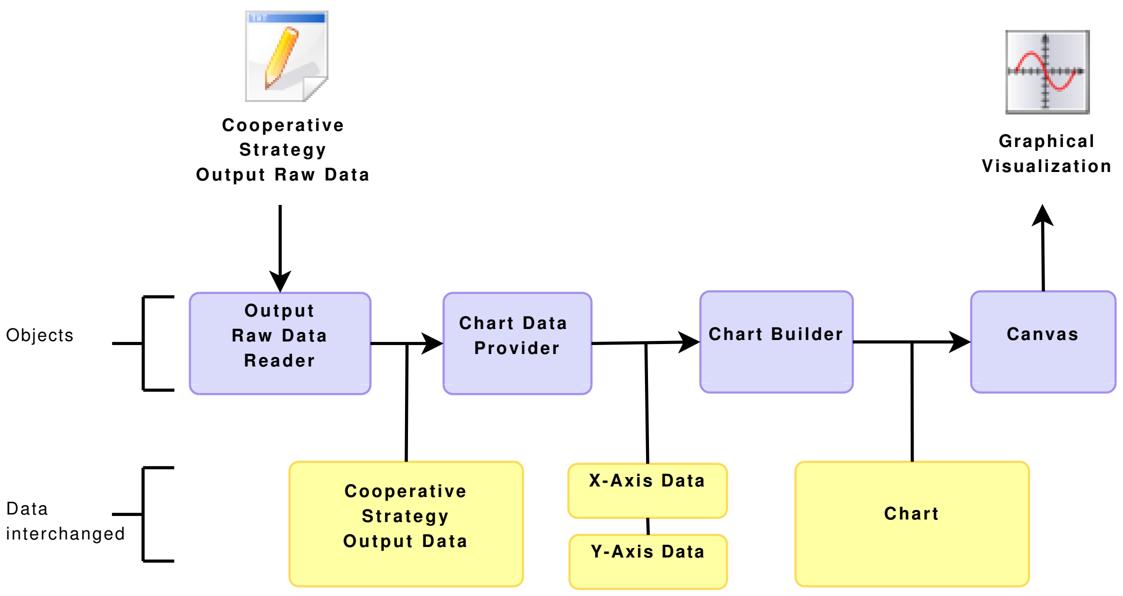


Figure 5.4 – Visualization model description.

The sequence would be the following:

1. COS writes a new line to the data file with a new partial result.
2. The `OutputRawDataReader` object is continuously monitoring the data file for any change on it. When this does occurred, it reads the raw data and parses it, exposing the read data to the rest of objects in the form

of a `CooperativeStrategyOutputData` object. This object is simply an in-memory representation of the raw data.

3. The `ChartDataProvider` object observes those data, and when it is notified that a change in the data has occur, it selects the new data to provide to a chart (x-axis data, in the form of the `XAxisData` object, and y-axis data, with a `YAxisData` object).

4. The `ChartBuilder` object is notified that a changed has happened in the `XAxisData` and `YAxisData` objects, so it regenerates the correspondent chart (`Chart` data) with the new values.

5. Finally, the `Canvas` object is repainted with the new chart and results are presented to the user.

It should be noted that this scheme is very similar to the well-known software pattern Model-View-Controller [61]. This structure of observable data and objects that watch for changes on them is elegant, and what is more important, useful. This pattern allows, e.g., that a single `CooperativeStrategyOutputData` object may be observed by multiple `ChartDataProvider` objects, which implies that a single representation of the results in the CSV file may be used to provide the necessary data to produce several charts or views.

Regarding the charts creation, because DACOS was going to be integrated in the Eclipse environment, a charting library with no graphical conflicts with that application was needed to dynamically create the plots. The way in which this was solved was to use an existent charting plugin for Eclipse, named BIRT (`http://www.eclipse.org/birt/`). BIRT calls for Business Intelligence and Reporting Tools, and is a reporting system integrated with the Eclipse platform, which, among other things, is capable of producing compelling charts and reports. Fig. 5.5 shows a screenshot of the analysis and visualization module using BIRT for creating several chart views of real data from a specific COS.

5.1.3. Conclusions

In this work, DACOS, an integrated tool for the configuration and analysis of centralized cooperative optimization systems was presented. The problems that appear when configuring and visualising these systems have been discussed, and a guideline for solving them has been proposed.

DACOS has been designed using software models, from which most of the code has been automatically generated. This methodology guarantees that the generated code follows well-known software patterns, as well as reduces the number of programming errors. In order to do this, the Eclipse platform and its software

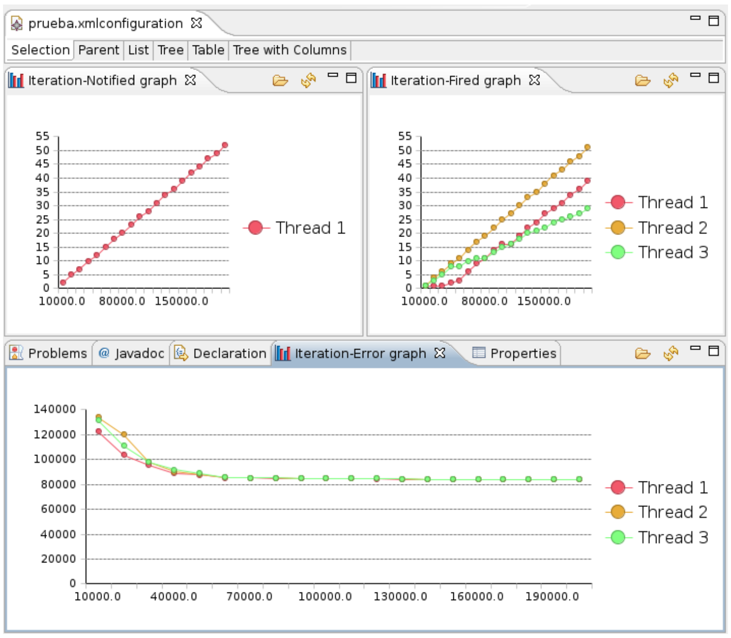


Figure 5.5 – Screenshot of DACOS's analysis and visualization module.

modeling project EMF have been used. This has allowed to avoid certain tasks unrelated with the scope of this work, such as management of events and windows, integration with the desktop system of the machine, etc.

DACOS is divided in two separated components: the configuration module and the analysis and visualization module. The system has been tested in a real use case with the USApHMP problem [104].

DACOS was published in [42]:

"A software modeling approach for the design and analysis of cooperative optimization systems", I. G. del Amo, D. A. Pelta, A. D. Masegosa, and J. L. Verdegay, *Software: Practice and Experience*, vol. 40, pp. 811–823, Aug. 2010. `http://dx.doi.org/10.1002/spe.984`.

5.2. MODO Optimization Package

The MODO Optimization Package is a software framework developed as a result of all the experiments on DOPs performed in this thesis. This framework brings together the developed software, which has also evolved in time as it was being adapted to the new experiments. In this section we show the main components of the framework and the acquired experience that motivated them.

5.2.1. Motivation

Through all this thesis, the experiments that we have performed have given us not only the opportunity of increasing our knowledge on DOPs, but they have also allowed us to gather experience for better designing and performing those very same experiments in the future.

At the beginning of the research, the experiments that we performed were focused on proving the validity of some algorithmic proposals on certain, very specific scenarios. This is the case, for example, of the PSO+CPT [120] that we talked about in Sect. 3.1, where the experimentation was performed exclusively over the Scenario 2 of the MPB. Even though, in those early researches, some small variations of parameters were already tested in order to observe their influence in the algorithm. In the case of the PSO+CPT, several values for the change frequency were used.

However, as the investigation advanced, it soon became evident that in order to acquire a deeper knowledge about the algorithms it was necessary to test them over a wider range of scenarios. In the work about the properties of the particles of a mQSO [41] (Sect. 3.2), we only used the MPB, but we analyzed a high number of configurations, in which we varied the number of peaks, the ratio of the different types of particles, and the change frequency of the environment.

Finally, subsequent works began to also include different problems in their experiments (Ackley, Griewank or Rastrigin for continuous problems, and OneMax, Plateau, RoyalRoad or Deceptive for discrete), each of them with their own multiple variations of scenarios. This is the case for example of the works of the mQSO with heuristic rules [39] (Sect. 3.2), the works about the CS algorithm [69, 70] (Sect. 3.4), the Agents algorithm [68] (Sect. 3.3), or the comparison of algorithms [40] (Sect. 4.4).

One of the consequences of this increase in the complexity of the test scenarios was the necessity of developing new techniques for comparing the results and evaluating the performance of the algorithms, in order to better exploit the information provided by the experiments. As a result of this, we created the SRCS technique [38], introduced in Chapter 4.

However, another consequence that has not been addressed yet was the necessity of developing a whole software framework that would allow us to perform this kind of experimentation. Considering the characteristics of the experiments, it was needed that this software framework could:

- **Separate algorithms from problems as much as possible**. This implies being able of executing multiple algorithms over the same problem scenario, and viceversa, executing the same algorithm over multiple instances of scenarios and problems. This should be done without changing the code, or at least, changing it the minimum possible.

- **Separate performance measures from problems and algorithms as much as possible**. The objective of this requirement is to be able of using the same performance measure in multiple problems and scenarios, and with multiple algorithms, and even being able of combining several performance measures at the same time without interfering with each other.

- **Separate mechanisms to introduce dynamism from the problems and scenarios they are applied to**. The most common mechanisms to introduce dynamism can be decomposed in two parts: (1) a condition that must be met in order for the change to take place, and (2) the change itself. This causes the change mechanism to be quite similar to the *antecedent* $\rightarrow$ *consequent* structure of a rule. The idea is, as usual, to be able of reusing these components in different problems without needing to change the code. For example, when the condition for a change is that a certain number of evaluations of the fitness function is reached, it is obvious that this does not depend on the problem at hand being, e.g., the MPB. Therefore, it should be possible to use this *max-evaluations* condition in other problems, and the framework should make it as easier as possible. The same happens to the change mechanism (the "consequent" of the rule). For example, when the condition for a change is met in the MPB, one of the changes that take place is a translation of the position of the peaks following a trajectory. This very same kind of movement can be applied to translate the origin of coordinates of other functions, such as Ackley, Griewank, Rastrigin, etc.

- **Ensure the reproducibility of the experiments**. Since we are mainly dealing with synthetic problems for performing the test, we must try to guarantee as much as possible that the experiments can be replicated. The main inconvenient for this objective is the use of pseudo-random number generators, which can be overcame using random seeds. By initializing a random number generator with the same seed we can be sure that the generated number sequence is always the same. Moreover, the problem and the algorithm must both have independent random number generators. This way, a whole experiment can be reproduced (by providing the same previously used seeds to the problem and to the algorithm), but also, the evolution of a problem can be reproduced independently of the algorithm (by providing the seed for the problem), thus fairly testing algorithms in the same scenario.

As it can be seen in the previous requirements, there is an underlying motivation in all of them: performing the maximum number of experiments with the highest effectiveness and efficiency possible. Effectiveness refers to the fact of obtaining rigorous results that can be independently verified and reproduced. Efficiency, on the other hand, focuses on optimizing the workload needed to perform experiments, mainly by recycling previously existent code. This not only reduces the probability of obtaining wrong results by reusing already tested software, but also decreases the total amount of time needed to perform an experiment, since it is not necessary to implement the whole code for it again.

When a single experiment is performed, it is generally more useful to implement the code as the requirements arise. This *agile* development approach prevents loosing time implementing complex functionality and features that eventually may even be unneeded. This approach is most beneficial when few, unrelated experiments need to be performed. However, when the number and correlation of experiments is high, this previous approach is not optimal, since the code that is usually produced this way tends to be quite entangled and highly dependent, making it harder to reuse in other experiments. In our particular case, it was needed to perform a lot of experiments with high similarity between them, so it was convenient to invest some time in separating these components in order to improve their portability.

In a previous work [191] we had already introduced, using Object Oriented Programming (OOP), a class hierarchy to deal with static optimization problems using metaheuristics. Figures 5.6 and 5.7 show, as a reference, the main class diagrams of that work. Based on that knowledge, we started to create the MODO Optimization Package. It must be said that not all the experience from the previous work could be reused: this new framework included dynamism in the problem, and our experience and new knowledge made us rethink some of the approaches used in [191] (just to cite an example, in the previous work solutions were merely data containers and the responsibility of everything related with them laid on the problem; in the MODO Optimization Package, part of this responsibility has been translated to the solution itself, for example when comparing a solution with another to decide which one is better). However, many of the main ideas introduced in [191] were still valid, and they were incorporated into the new framework, like the separation between problem and algorithm, independent classes for stop criteria, or an experiment architecture where the algorithm is in charge of executing the main optimization process.

Considering this, the main components that we initially identified for the MODO Optimization Package were: *algorithms*, *problems*, *dynamic features* and *performance measures*. It should be noted that dynamic features and performance measures are actually *subcomponents* of a problem. However, given their impor-

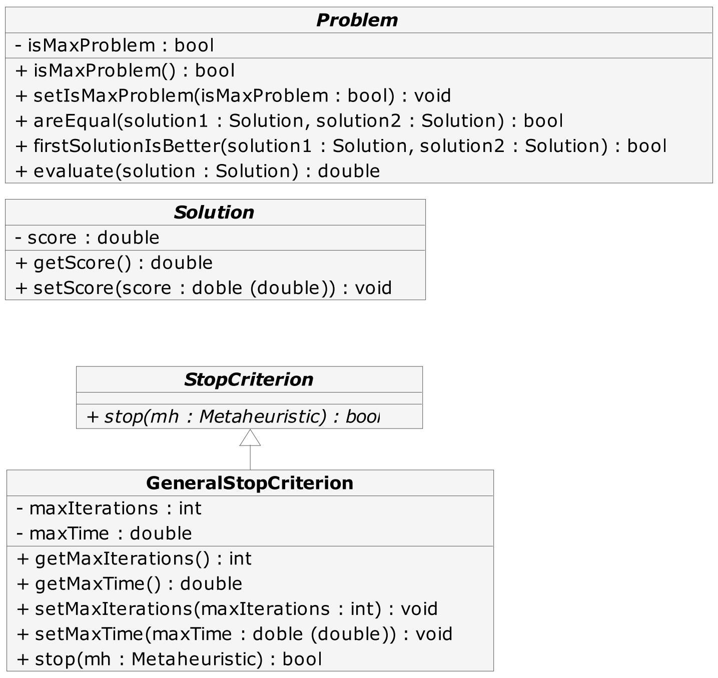

Problem
- isMaxProblem : bool
+ isMaxProblem() : bool
+ setIsMaxProblem(isMaxProblem : bool) : void
+ areEqual(solution1 : Solution, solution2 : Solution) : bool
+ firstSolutionIsBetter(solution1 : Solution, solution2 : Solution) : bool
+ evaluate(solution : Solution) : double
Solution
- score : double
+ getScore() : double
+ setScore(score : doble (double)) : void
StopCriterion
+ stop(mh : Metaheuristic) : bool
GeneralStopCriterion
- maxIterations : int
- maxTime : double
+ getMaxIterations() : int
+ getMaxTime() : double
+ setMaxIterations(maxIterations : int) : void
+ setMaxTime(maxTime : doble (double)) : void
+ stop(mh : Metaheuristic) : bool

Figure 5.6 – *MODO Optimization Package.* Main problem-related classes introduced in [191]. These classes are shown here as reference, since most of them are also part of the MODO Optimization Package, although their methods have evolved significantly.

tance and the amount of design and development effort put on them, we will analyze them separately. Therefore, the objective of the framework was to generate an architecture that could mostly favor a high modularity and separation among these components (see Fig. 5.8).

The process by which the algorithm performs the main optimization is explained in Fig. 5.9. The algorithm is continuously performing optimization tasks within a loop, and it calls the problem whenever it needs to evaluate a solution. When the problem begins its execution, before evaluating the solution it must check if the conditions for a change in the environment are met. If this is the case, the problem performs the changes and continues with the normal evaluation of the solution. In this sense, the functionality associated with each of the previously identified components *algorithm*, *problem* and *dynamic features* is clearly separated and well-defined.

The fourth component initially identified was the *performance measures.* This component is special because it needs to access information both from the problem and from the algorithm during its execution. We decided to integrate the

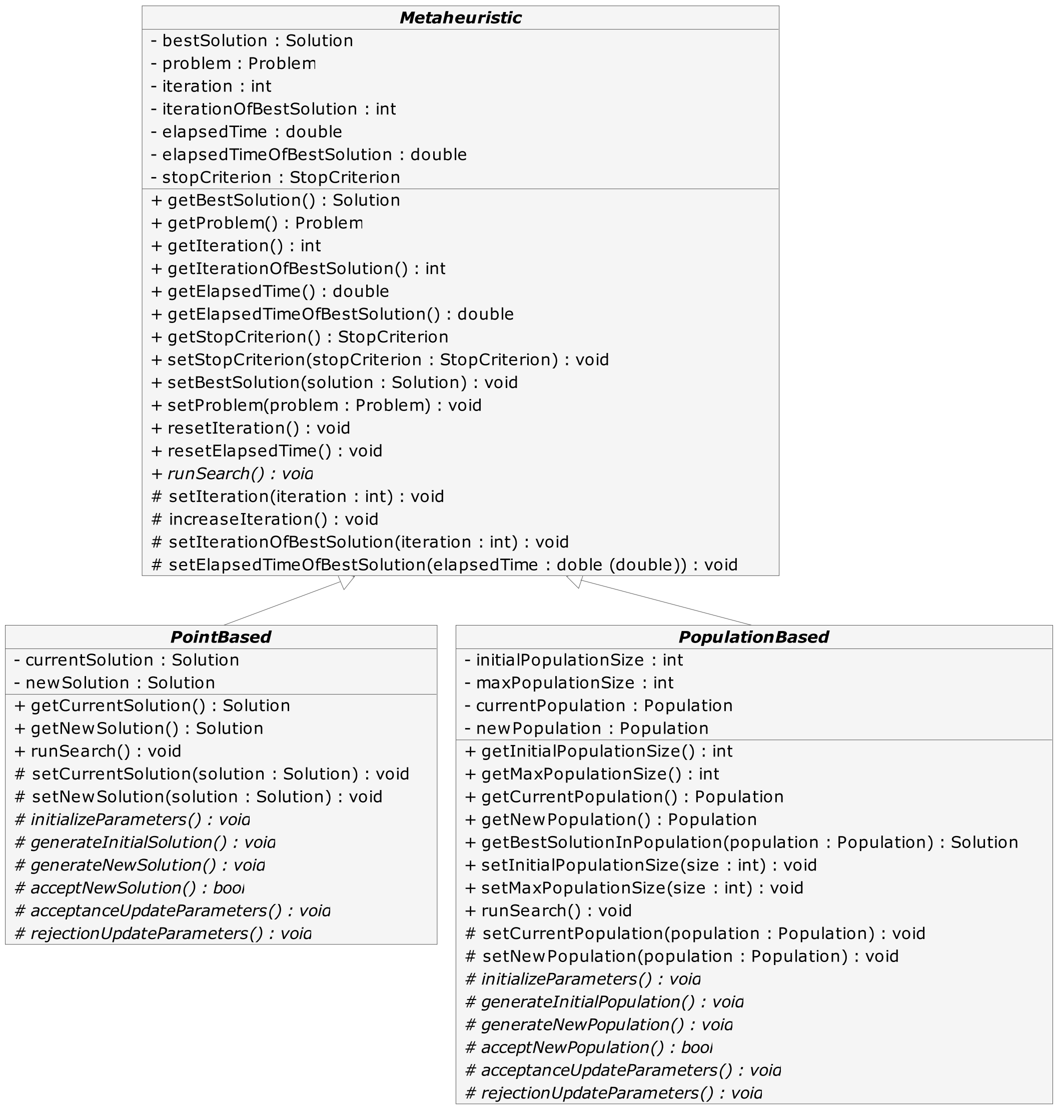

Metaheuristic
- bestSolution : Solution
- problem : Problem
- iteration : int
- iterationOfBestSolution : int
- elapsedTime : double
- elapsedTimeOfBestSolution : double
- stopCriterion : StopCriterion
+ getBestSolution() : Solution
+ getProblem() : Problem
+ getIteration() : int
+ getIterationOfBestSolution() : int
+ getElapsedTime() : double
+ getElapsedTimeOfBestSolution() : double
+ getStopCriterion() : StopCriterion
+ setStopCriterion(stopCriterion : StopCriterion) : void
+ setBestSolution(solution : Solution) : void
+ setProblem(problem : Problem) : void
+ resetIteration() : void
+ resetElapsedTime() : void
+ runSearch() : void
# setIteration(iteration : int) : void
# increaseIteration() : void
# setIterationOfBestSolution(iteration : int) : void
# setElapsedTimeOfBestSolution(elapsedTime : doble (double)) : void
PointBased
- currentSolution : Solution
- newSolution : Solution
+ getCurrentSolution() : Solution
+ getNewSolution() : Solution
+ runSearch() : void
# setCurrentSolution(solution : Solution) : void
# setNewSolution(solution : Solution) : void
# initializeParameters() : void
# generateInitialSolution() : void
# generateNewSolution() : void
# acceptNewSolution() : bool
# acceptanceUpdateParameters() : void
# rejectionUpdateParameters() : void
PopulationBased
- initialPopulationSize : int
- maxPopulationSize : int
- currentPopulation : Population
- newPopulation : Population
+ getInitialPopulationSize() : int
+ getMaxPopulationSize() : int
+ getCurrentPopulation() : Population
+ getNewPopulation() : Population
+ getBestSolutionInPopulation(population : Population) : Solution
+ setInitialPopulationSize(size : int) : void
+ setMaxPopulationSize(size : int) : void
+ runSearch() : void
# setCurrentPopulation(population : Population) : void
# setNewPopulation(population : Population) : void
# initializeParameters() : void
# generateInitialPopulation() : void
# generateNewPopulation() : void
# acceptNewPopulation() : bool
# acceptanceUpdateParameters() : void
# rejectionUpdateParameters() : void

Figure 5.7 – *MODO Optimization Package.* Main metaheuristic-related classes introduced in [191]. In the MODO Optimization Package, these classes have been replaced by "Algorithm", a more generic one, but with similar features. Some of the ideas introduced in [191] regarding metaheuristics have been adapted to subclasses of Algorithm, like, e.g., a specific class for population-based algorithms, with methods inspired in the PopulationBased class shown here.

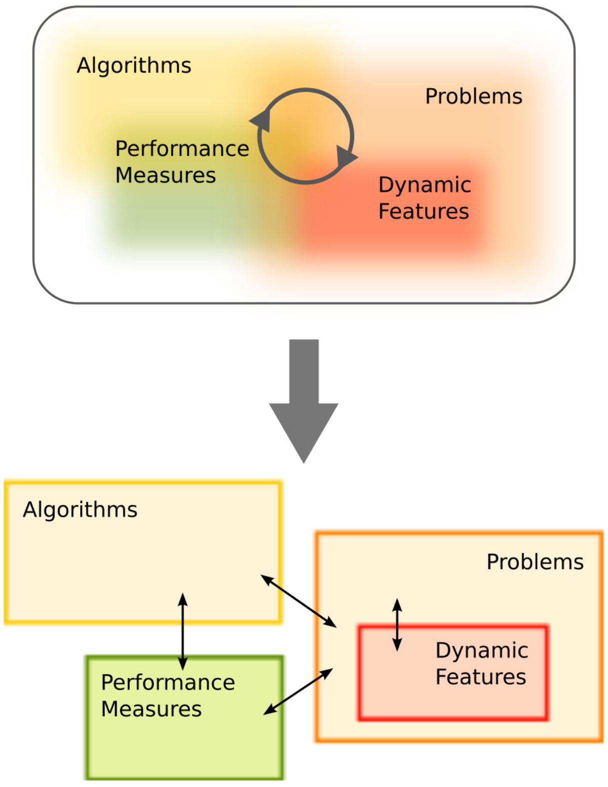


Figure 5.8 – *MODO Optimization Package.* Usually, experiments are coded on-demand, and problems, dynamic features, algorithms and performance measures tend to blend with each other. This produces software components with fuzzy boundaries and interdependencies that can hardly be translated to other experiments. When multiple experiments need to be performed, this situation is highly inefficient. In order to increase productivity, it is necessary to make these modules more portable, separating them into clearly defined components that interact between them.

performance measures within the problem component, since this approach allowed a much finer control of when they were executed (e.g., just before a change in the environment, right at the end of the evaluation of a solution, etc.). However, as we performed more experiments, we realized that performance measures could have quite diverse functionalities and requirements. This hindered the creation of a unified component with a precise and closed interface. Eventually, we concluded that these performance measures were actually a particular case of a more generic "extra functionality" component of a problem. Therefore, we created a specific component for this extra functionality, named *plugin*, instead of the more restricting one *performance measures*.

Plugins are modules that can be inserted at some pre-defined points in the

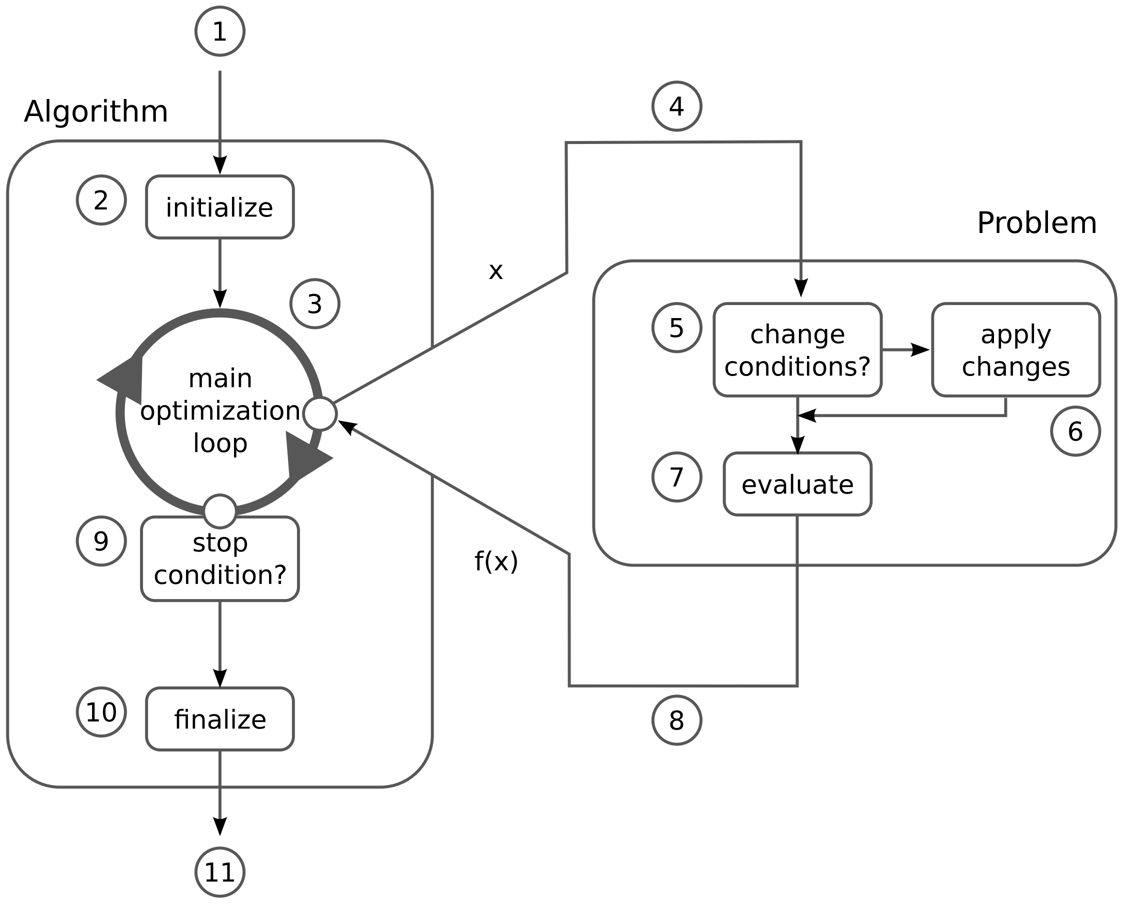


Figure 5.9 – *MODO Optimization Package.* Typical execution flow in an experiment. The module in charge of the execution is the algorithm, which continuously optimizes within a loop until a stop criterion is met. In each iteration, the algorithm asks the problem to evaluate a solution. During this call, the problem checks if the conditions for a change in the environment are met, and in such case, performs the corresponding changes.

middle of the evaluation of a solution, that extend the normal capabilities of the problem in order to perform a specific task. Plugins can be used to perform almost any action ranging from measuring the performance of an algorithm, to reporting that data to the user (maybe using standard output, files, GUIs, etc.). Or even for **adding dynamism to the problem**. This last conclusion meant a very important event in the development of the framework, because we realized that dynamic features could also be implemented using plugins. Therefore, we merged the *dynamic features* component into the *plugin* component, reducing the main modules of the framework to 3: *algorithms*, *problems* and *plugins*.

As we have previously said, in an experiment, the main optimization process is carried out by the algorithm. However, there are other tasks that need to be performed at the beginning of the execution, before running the algorithm. First, it is necessary to load the configuration of the experiment, possibly from files. This includes reading and parsing the parameters for the problem, the algorithm, the plugins (including those related with dynamism, performance measures, graphical user interfaces, etc.), as well as other additional settings, such as the number of runs to execute, or the master seed for the random number generators. Once the configuration has been loaded, it is necessary to instantiate the appropriate modules for the problem, the algorithm, and the plugins. The order in which all of this is done is important in this case: the first one is the problem, since it does not have any dependencies on the others for being created; next is the algorithm, because it may have components that could need the problem beforehand (e.g., problem-dependent heuristics); and finally, plugins are instantiated in the last place, since they may depend both on the problem and the algorithm. With the components created, the algorithm is executed, entering the optimization loop until the stop condition is fulfilled. However, it is worth reminding that in order to obtain statistically meaningful results, it may be necessary to independently execute the algorithm a certain number N_r of times. Strictly speaking, this would imply to repeat the previous load-configure-execute process N_r times. However, with a proper handling of the configuration settings and the use of $reset()$ methods, the loading and configuration phases can be performed only once, at the beginning. This initialization sequence of an experiment is summarized in Fig. 5.10.

In the following sections we will explain in more detail the *algorithms*, *problems* and *plugins* components.

5.2.2. Algorithms

The algorithm component groups the strategies that we have implemented for optimizing a DOP. In general terms, an algorithm must take care of guiding the whole process of searching for the best solution, adapting to the changes in the environment when these occur, possibly using one or more of the techniques

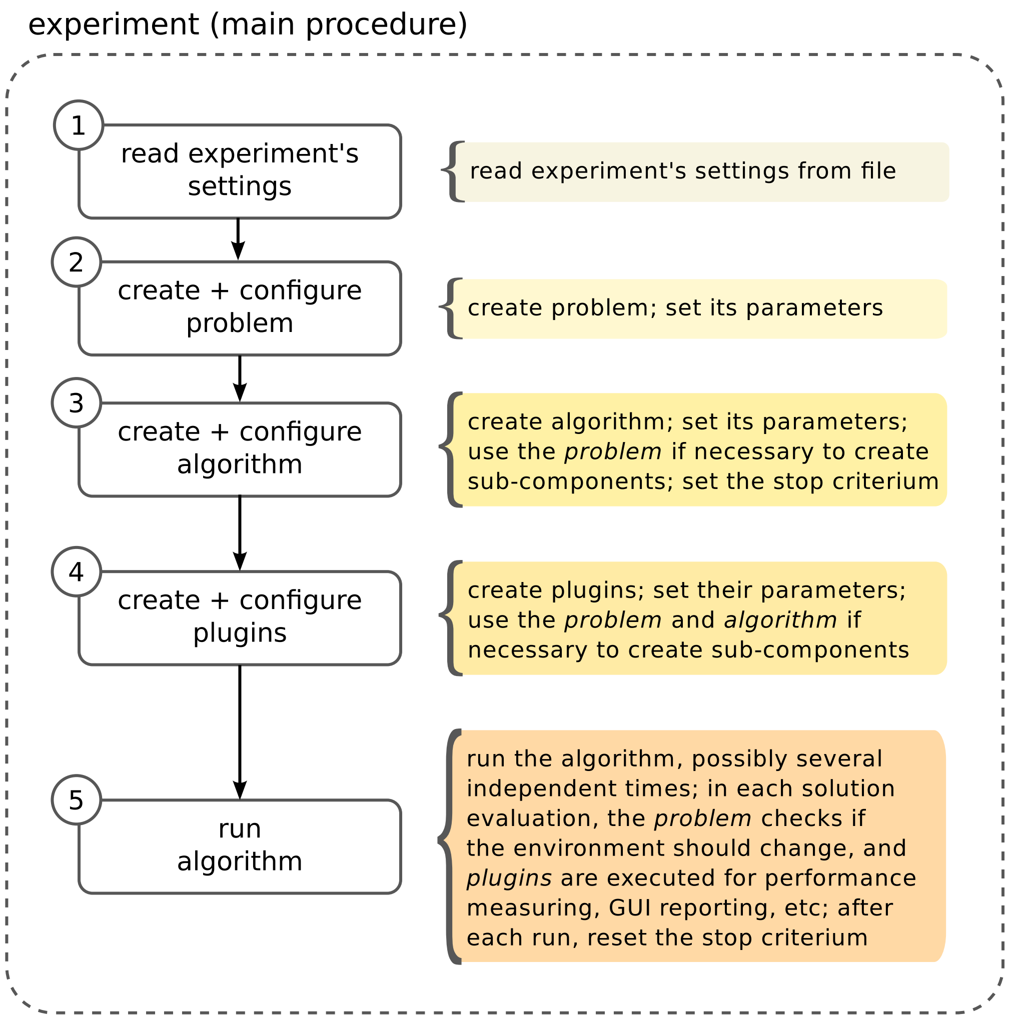

experiment (main procedure)
1
read experiment's settings
read experiment's settings from file
2
create + configure problem
create problem; set its parameters
3
create + configure algorithm
create algorithm; set its parameters; use the *problem* if necessary to create sub-components; set the stop criterium
4
create + configure plugins
create plugins; set their parameters; use the *problem* and *algorithm* if necessary to create sub-components
5
run algorithm
run the algorithm, possibly several independent times; in each solution evaluation, the *problem* checks if the environment should change, and *plugins* are executed for performance measuring, GUI reporting, etc; after each run, reset the stop criterium

Figure 5.10 – *MODO Optimization Package.* Experiment initialization sequence.

explained in Sect. 2.3.

Even though we counted with previous experience on implementing metaheuristics from [191], the diversity of methods and algorithmic families that we have used in this research made the hierarchy shown in Fig. 5.7 too specific. In our particular case, an algorithm is implemented as a class that just contains:

- A reference to the problem to solve, usually provided at creation time.
- A *StopCondition* object, that indicates the algorithm if it must stop or if it can continue optimizing (see Fig. 5.9). This object does not need to be provided at creation time, it only needs to be available during the execution of the algorithm. It can be even altered or replaced while the algorithm is running (this is actually used by, e.g., the framework's GUIs to pause/resume the execution at the user's request).
- A random seed, provided at creation time. This seed is used by the algorithm to initialize the random number generators that it may use (in case of any). As we have discussed in Sect. 5.2.1, the algorithms and the problems use independent random number generators.
- A method for executing the algorithm (this method is the one called by the experiment).

These are the only constraints imposed on an algorithm, with the objective of creating a common interface that can be called from the main procedure of the experiment without limiting the possibilities or the resources that an algorithm can use.

All the algorithms used in the works presented in this thesis have been implemented as subclasses of this Algorithm class. However, the great variety of methods used (mQSO and its heuristic-rules-based variants, evolutionary algorithms, cooperative strategies, agents, etc.) make it a too extensive task to describe here the implementation details and the class hierarchies associated to them. Nevertheless, we can mention, e.g., that there are classes for population-based algorithms as well as for trajectory-based algorithms, as it was proposed in [191], generic base classes for evolutionary algorithms, etc. On the other hand, following the principles of modularity and component reuse, in the framework there are also classes for operators and commonly used techniques, such as, e.g., learning mechanisms, diversification methods, procedures for randomizing solutions within an area of the search space, intensification, etc.

5.2.3. Problems

The problem component is the one in charge of managing everything related with the environment being optimized. The main functionality of this component is evaluating solutions, returning the fitness value associated to them for the current environment. Additionally, dynamic changes in the environment take place during the execution of this component (if the conditions are met), as well as other extended functionality such as the calculation of performance measures, visualization of the results, etc. We will talk about this extended functionality in Sect. 5.2.4.

The evaluation of a solution is a complex task, but contrary to what happened with the algorithms, it can be decomposed in a sequence of clearly defined steps that are always performed in the same order. Thanks to this modularity, it is possible to design a single class for evaluating a solution. This class would contain subcomponents that perform the previously referred steps, and since these subcomponents are simpler, they are also easier to implement and are more likely to be reused. Moreover, by simply substituting one of these subcomponents for another compatible one, a new different problem is obtained, so this class is effectively acting as a *problem generator*. In our case, this problem generator, that we named *UDOPEngine*, is capable of simulating *dynamic* and *uncertain* optimization problems. We will now discuss in detail the solution evaluation process.

The evaluation of a solution can be divided into intermediate phases in our implementation, mainly conditioned by the presence of constraints. As it is shown in Fig. 5.11, the process begins with the fitness evaluation itself, at the same time that it checks if the solution satisfies all the constraints. If they are satisfied, the fitness remains unchanged, but if not, there are multiple ways of handling this situation [108, 187]. The most common one usually consists in modifying the fitness according to some type of penalty. Among the most frequent penalty types we can highlight the following:

- *death penalty*, consists in applying the maximum penalty possible to every solution that does not satisfy the constraints, independently of the degree of violation. The way of accomplishing this is usually by assigning the fitness a value of $\pm\infty$, depending on whether it is a maximization or a minimization problem. This is the most simple penalty, although it prevents the algorithm from extracting any kind of information from unfeasible solutions that could guide the search to feasible zones.

- *static penalty*, that consists in a static penalty factor that is added to the fitness, usually dependent on the degree of violation of the constraints.

- *dynamic penalty*, that consists in a penalty factor that grows with time (number of evaluations), usually being also dependent on the degree of violation of the constraints.

- *adaptive penalty*, where information gathered from the search process is used to control the amount of penalty added to infeasible individuals.

Once the fitness has been adjusted according to the unsatisfied constraints, the process returns both the fitness and the constraints violation degree.

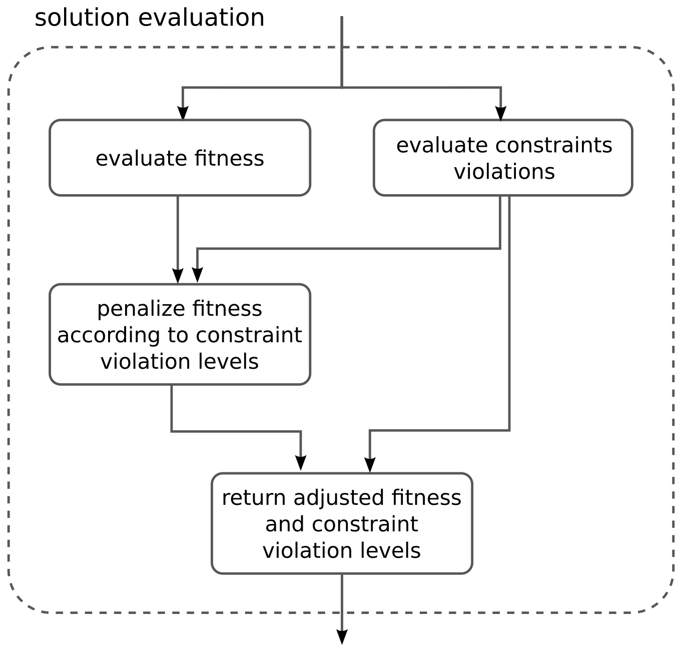

solution evaluation
evaluate fitness
evaluate constraints violations
penalize fitness according to constraint violation levels
return adjusted fitness and constraint violation levels

Figure 5.11 – *MODO Optimization Package.* Basic scheme of the solution evaluation process.

In the experiments performed in our research we frequently found out that the fitness functions used in the problems consisted in a composition of simpler functions. For example, in the MPB, the final resulting scenario is obtained by merging multiple *cone*-like functions using the *maximum*. It is not unusual to find other benchmarks that use function composition to generate more complex environments (e.g., the ones used in the CEC-2009 competition on Dynamic Optimization [95]). This led us to include a functionality for composing simple functions in the UDOPEngine, in such a way that they could be combined using the maximum, the minimum, the average, etc.

The context of the research project that encompasses this thesis contemplates the investigation of dynamic environments with uncertainty, although such uncertainty has not been used in the works presented here. Nevertheless, we decided to include a mechanism for generating a first approach to these uncertain environments in the MODO Optimization Package, foreseeing its application in future research works.

In our preliminary studies we concluded that the types of uncertainty that were easier to deal with using the UDOPEngine architecture were those that concerned the communication channels between the algorithm and the problem during the evaluation of a solution. This uncertainty in the communication channels represents possible mismatches that can happen between:

1. The information sent by the algorithm to the problem (the solution) and what the problem really receives.

2. The information returned by the problem to the algorithm (the fitness and the constraint violation degrees) and what really arrives to the algorithm.

We have called the first one *input uncertainty*, and the second one *output uncertainty* (see Fig. 5.12).

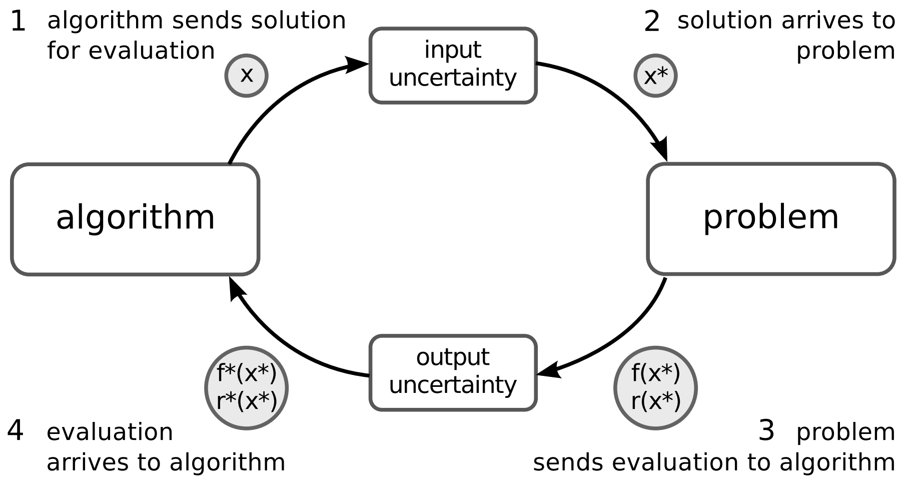

1 algorithm sends solution for evaluation
x
input uncertainty
2 solution arrives to problem
x*
algorithm
problem
f*(x*)
r*(x*)
output uncertainty
f(x*)
r(x*)
4 evaluation arrives to algorithm
3 problem sends evaluation to algorithm

Figure 5.12 – *MODO Optimization Package.* Including uncertainty in the communication channels between the algorithm and the problem. First, the algorithm sends a solution x for its evaluation. The *input uncertainty* module may alter this solution, producing a modified version, x^*, which is the one that the problem receives. The problem evaluates x^*, returning its fitness value, $f(x^*)$ and the set of constraint violation degrees, $r(x^*)$. The *output uncertainty* module may alter these values, producing $f^*(x^*)$ and $r^*(x^*)$, which are the ones that finally arrive to the algorithm.

This uncertainty in the communication channels can represent real aspects associated to the physical implementation of a solution for a given problem or to the measurement of variables. With this scheme it is possible to simulate, among others:

- Precision or rounding problems when “manufacturing” a solution or performing a measurement.
- Presence of noise.
- Errors due to deviations caused by a miscalibration or a faulty sensor.

This uncertainty modules can alter the value of the data they receive by, e.g., adding a fixed deviation (simulating a faulty sensor or bias errors), adding a random quantity (simulating gaussian noise), removing decimals or rounding digits (simulating imprecision), flipping a bit (simulating transmission errors), etc. It is also important to note that these modules do not need to be both present: it is possible to have only input uncertainty, only output uncertainty, or no uncertainty at all — which is the base case that we part from.

Considering these new factor, i.e., the input and output uncertainty, and the combination of multiple fitness functions, we obtain the extended version of the solution evaluation process. This process is shown in Fig. 5.13, and it constitutes a quite close representation to the actual implementation of the UDOPEngine.

Finally, one last consideration: the whole solution evaluation process has been performed considering only mono-objective optimization problems; however, the UDOPEngine allows to simulate *multi-objective* problems, where the process described in Fig. 5.13 is repeated for each of the objectives of the problem. In the end, the solution evaluation that is returned to the algorithm consists in a structure containing all the fitness values and constraint violation degrees for each of the objectives of the problems.

5.2.4. Plugins

A *plugin* is a widely extended concept in computation, generally referring to software components that are inserted in certain predefined points (*extension points*) of an application to extend its functionality. Some examples of this architecture can be seen in current web browsers, integrated development environments, etc.

In our UDOPEngine, plugins emerged from the necessity of adding certain extra functionality that was not related with the evaluation of a solution. This functionality mainly consisted in:

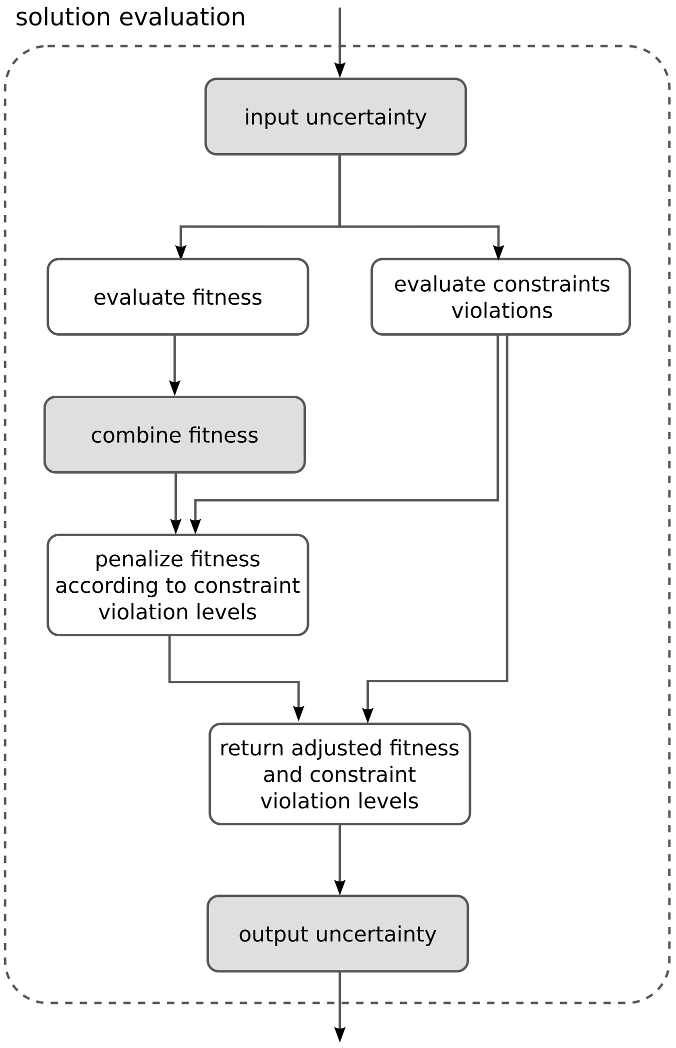

solution evaluation
input uncertainty
evaluate fitness
evaluate constraints violations
combine fitness
penalize fitness according to constraint violation levels
return adjusted fitness and constraint violation levels
output uncertainty

Figure 5.13 – *MODO Optimization Package.* Extended scheme of the solution evaluation process.

- Performing some kind of measurement, for example, the performance of an algorithm. For this we can simply register the fitness value of the current solution being evaluated, or we can compare the fitness with that of another solution, generally the optimum of the problem. We can also measure time, data for statistical usage, etc.

- Reporting some type of information, for example, performance measures or statistics. These reports can be generated in all the evaluations, or every certain number of them, or just before a change in the environment, etc. Furthermore, the report of this information can be performed using the application's standard output, or writing to files, or even visualizing it in specific graphical interfaces.

Although the objectives of this extended functionality could be classified in the two big previous groups, it is clear that the functionality itself was quite diverse and extensive (performance measures, graphical interfaces, statistics, etc.). Therefore, there was no point on imposing any kind of software hierarchy nor predefined method signature. Plugins should be simply modules that are called at certain moments within the evaluation of a solution, with enough freedom to perform the actions they consider appropriate.

However, one of the most important aspects of these plugins is the moment in the evaluation of a solution in which they are called. A plugin that displays a solution in a Graphical User Interface (GUI) only needs that said solution, and thus can be executed at any time, for example, at the beginning of the evaluation. On the other hand, a plugin that calculates the performance needs the solution's fitness to be available, so it can only be executed once the penalized fitness has been calculated, at the end of the process.

Therefore, we defined certain fixed points within the solution evaluation process, named Extension Points (EP), where the plugins can be inserted. Multiple plugins can be inserted at the same EP; the UDOPEngine handles this automatically, so when the execution of the evaluation process reaches that EP, it sequentially calls all the plugins registered on it, according to the order in which they were inserted. Additionally, the plugin call sequence can be altered using some special plugins called Conditions. These plugins, included by default in the framework, contain a boolean condition and two separated output flows where plugins can be attached. Depending on whether the condition is met or not, the plugin executes the *true* output flow or the *false* output flow, effectively bifurcating the execution sequence of the EP.

Since the plugins are executed *during* the evaluation process, they can access very specific and volatile information that is not available outside of this process. Depending on the EP where the plugins are inserted, they can access the data produced by the immediately previous module in the execution flow. This data is passed to the plugin as a parameter when they are called, in a *read-only* mode. The existent EPs and the associated data that the plugins can access on each of them are listed bellow:

- EP_PRE_INPUT_UNCERTAINTY: located just before executing the Input Uncertainty module. Plugins inserted at this EP receive as a parameter the solution as it was sent by the algorithm.
- EP_POST_INPUT_UNCERTAINTY: located just after executing the Input Uncertainty module. Plugins inserted at this EP receive as a parameter the modified solution outputted by the Input Uncertainty module.
- EP_INDIVIDUAL_FITNESS: located just after executing one of the Fitness Function modules. In order to insert a plugin in this EP it is necessary to also specify the corresponding Fitness Function. Plugins inserted at this EP receive as a parameter the fitness of the solution calculated by this Fitness Function.
- EP_COMBINED_FITNESS: located just after combining all the fitness values obtained for the solution. Plugins inserted at this EP receive as a parameter the combined fitness of the solution.
- EP_CONSTRAINT: located just after executing a Constraint module. In order to insert a plugin in this EP it is necessary to also specify the corresponding Constraint. Plugins inserted at this EP receive as a parameter the CVD (Constraint Violation Degree) of the solution for this Constraint.
- EP_PENALIZED_FITNESS: located just after penalizing the fitness according to the constraints. Plugins inserted at this EP receive as a parameter the penalized fitness of the solution.
- EP_PRE_OUTPUT_UNCERTAINTY: located just before executing the Output Uncertainty module. Plugins inserted at this EP receive as a parameter the solution evaluation (penalized fitness + CVDs).
- EP_POST_OUTPUT_UNCERTAINTY: located just after executing the Output Uncertainty module. Plugins inserted at this EP receive as a parameter the solution evaluation outputted by the Output Uncertainty module.

In the first experiments that we performed using the MODO Optimization Package, the problem's dynamic features wer considered as an independent mechanism (not a plugin). The point where these dynamic features were executed was fixed, just before performing any kind of evaluation to the solution (i.e., in the plugins terminology, at the EP_PRE_INPUT_UNCERTAINTY point). However, as the experimentation advanced, we realized that this design was too restrictive. Why should changes in the environment needed to be executed before the evaluation of a solution? Why not after? Why not letting the user decide when they should be executed? Eventually, we realized that by adding this type of flexibility we were actually getting very close to the execution mechanism of the plugins. This led us to think that dynamic feature could be considered as a type of extended functionality, and could therefore be implemented as a set of plugins, allowing the user to decide when to execute them using the previously defined EPs.

However, in this case there was an additional problem: dynamic features could *modify* internal values of the UDOPEngine, like, e.g., the origin of coordinates of the environment, the perturbation degree of an uncertainty module, etc. The amount and type of data that could be modified were too big to be passed as a parameter to the plugins during their execution while maintaining their generic interface. The way in which we solved this was by allowing the components of the UDOPEngine to “expose” internal variables so that they could be modified externally (e.g., the origin of coordinates for a Fitness Function, the amount of perturbation of an Input Uncertainty module, etc.). These variables are registered in a central storage structure of the UDOPEngine that plugins can access. This way, there is no necessity of passing any extra parameter to the plugins: if a plugin wants to modify a variable, it simply accesses it at the catalog and requests the variable for a writing operation.

Figure 5.14 shows the design diagram of an Objective of the UDOPEngine (it should be reminded that the UDOPEngine is capable of handling multi-objective problems), based on the solution evaluation process (Fig. 5.13). The diagram also shows the different data types expected and produced by each module, as well as the EPs and an example of the sequence of plugins that can be created in one of those EPs.

With this plugin-based architecture we provide a great flexibility to the users for adapting the UDOPEngine problem generator to their particular necessities. Not only there is a wide variety of components for the standard modules of the UDOPEngine, but also, extra functionality that was not initially thought of can be added in a transparent way.

5.2.5. Outstanding features

Up until now, we have analyzed the different components of the MODO Optimization Package at separate. However, when all these components act together in a coordinated way, they are capable of performing highly specialized and complex tasks. We will now illustrate some of the most relevant features of the framework:

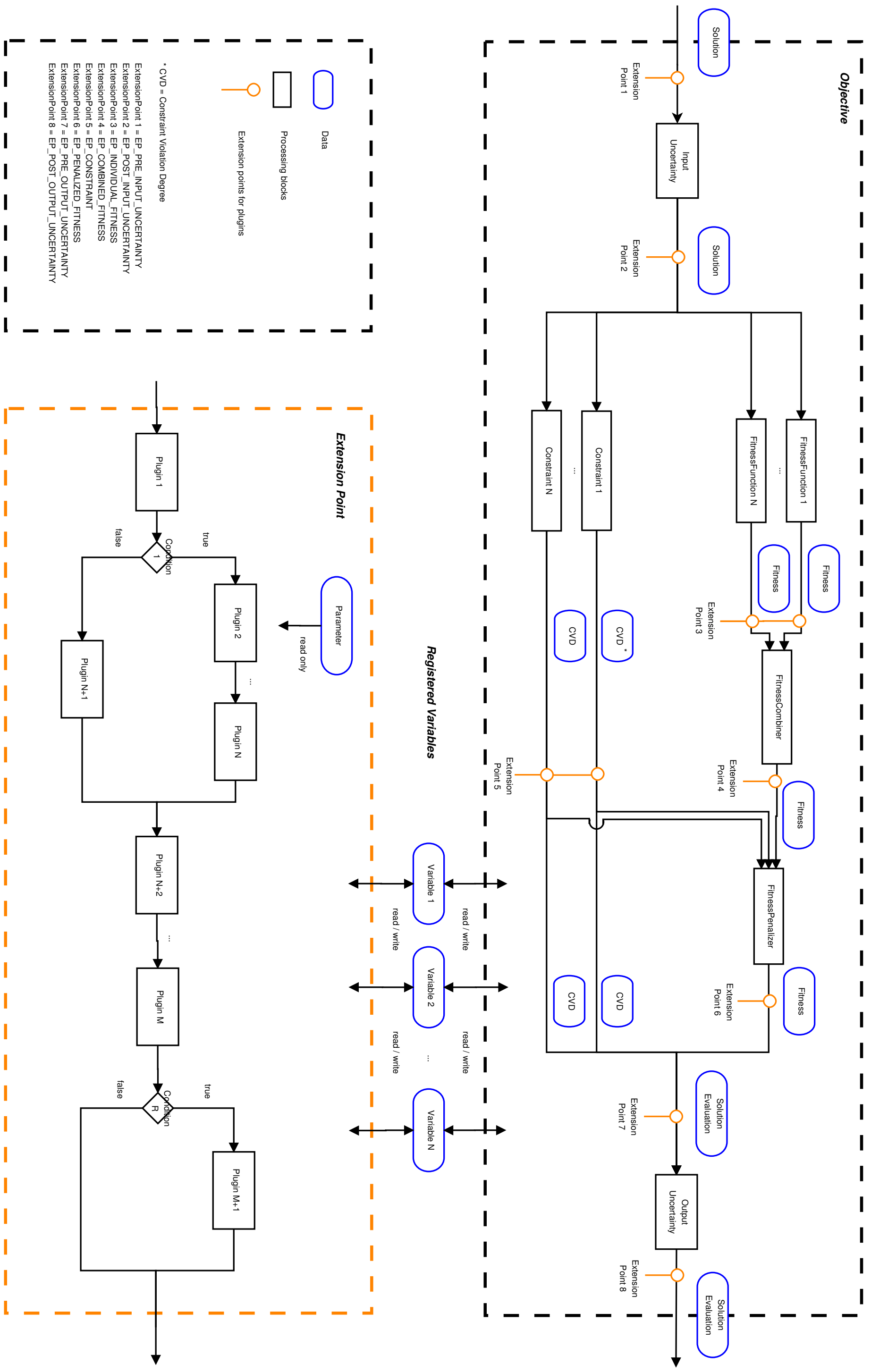


Figure 5.14 – *MODO Optimization Package.* Design-diagram of an Objective in the UDOPEngine.

- **Ability to work both with continuous and discrete problems**. The UDOPEngine has been designed from the beginning to handle abstract data types, such as *Solution*, *Fitness* or *Constraint Violation Degree (CVD)*. By encapsulating the main data types we make the framework independent of particular implementations of such data. Thus, the UDOPEngine can manage: a) continuous problems where the components of the solutions are of type *double*, *float*, etc; b) discrete problems where the components of the solutions are of type *integer*, *long*, etc; c) mixed problems where solutions have both continuous and discrete components; d) problems where the fitness is expressed as a real number, an integer number, or even a fuzzy number. However, most of the experiments of this thesis have been performed on continuous problems with fitness values in $\mathbb{R}$, so many of the main features that we will talk about next have been designed for this type of problems.

- **Graphical User Interfaces (GUIs) for visualizing the problem at hand in real time**. Due to the experience obtained during the development of the DACOS tool (Sect. 5.1), one of the priorities in the design of the framework was that it could graphically visualize both the problem that was being optimized and the evolution of the algorithm solving it. As we already saw in Sect. 5.1, this is specially useful in the initial stages of the experimentation in order to obtain a first idea of the behavior of the algorithm, verifying that there are no programming errors, and even exploring in a rough way the different values for the configuration parameters. Thus, we have designed several plugins that create a GUI, such that when they are executed, they use that GUI to display the problems using several techniques. The most interesting of these plugins, and the one used for all the figures shown here, performs an exhaustive sampling of the environment, thus allowing to display the fitness landscape in a very precise way. This obviously introduces a delay in the execution, although it has the advantage that it can be executed only right after a change in the environment. GUI plugins add very useful functionality when evaluating the problem or the performance of the algorithms. Some of this functionality includes the possibility of pausing/resuming the execution, alternating between a visualization for a maximization problem or for a minimization one (see Fig. 5.15), changing the luminosity of the image in order to enhance solutions closer to the optimum, using the mouse to inspect the value and the fitness of a point in the image (see Fig. 5.19), etc. It is worth noting that these GUI plugins only allow to display 2 dimensions, so they have a limited utility for higher dimensionality problems. Moreover, as we have previously mentioned, the graphical visualization usually introduces a significant delay in the execution of the experiment, specially in the case of the exhaustive sampling

GUI plugin. Therefore, the use of these components is only recommended for the initial stages of the tests, and it is convenient to disable them in the final ones, when the execution is more intensive and the main objective is to gather data for a later analysis.

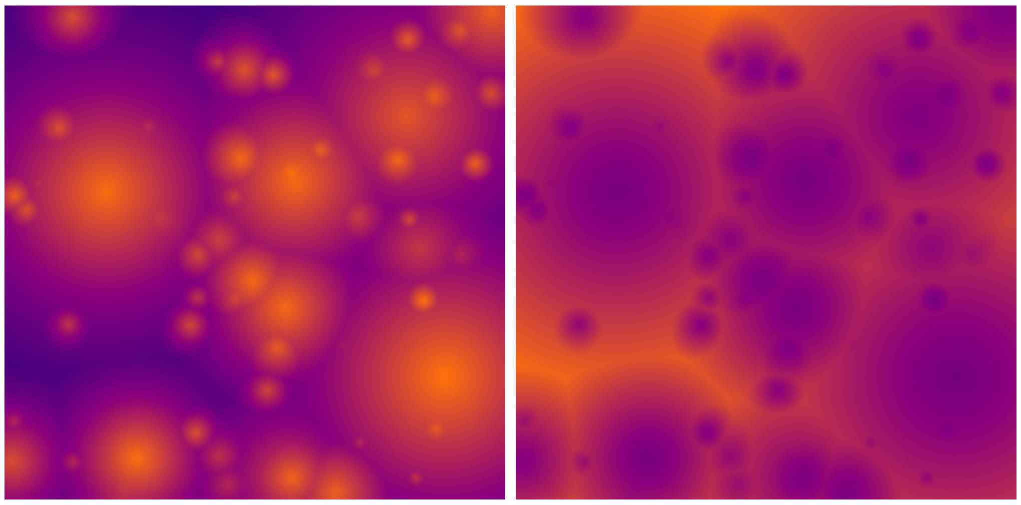
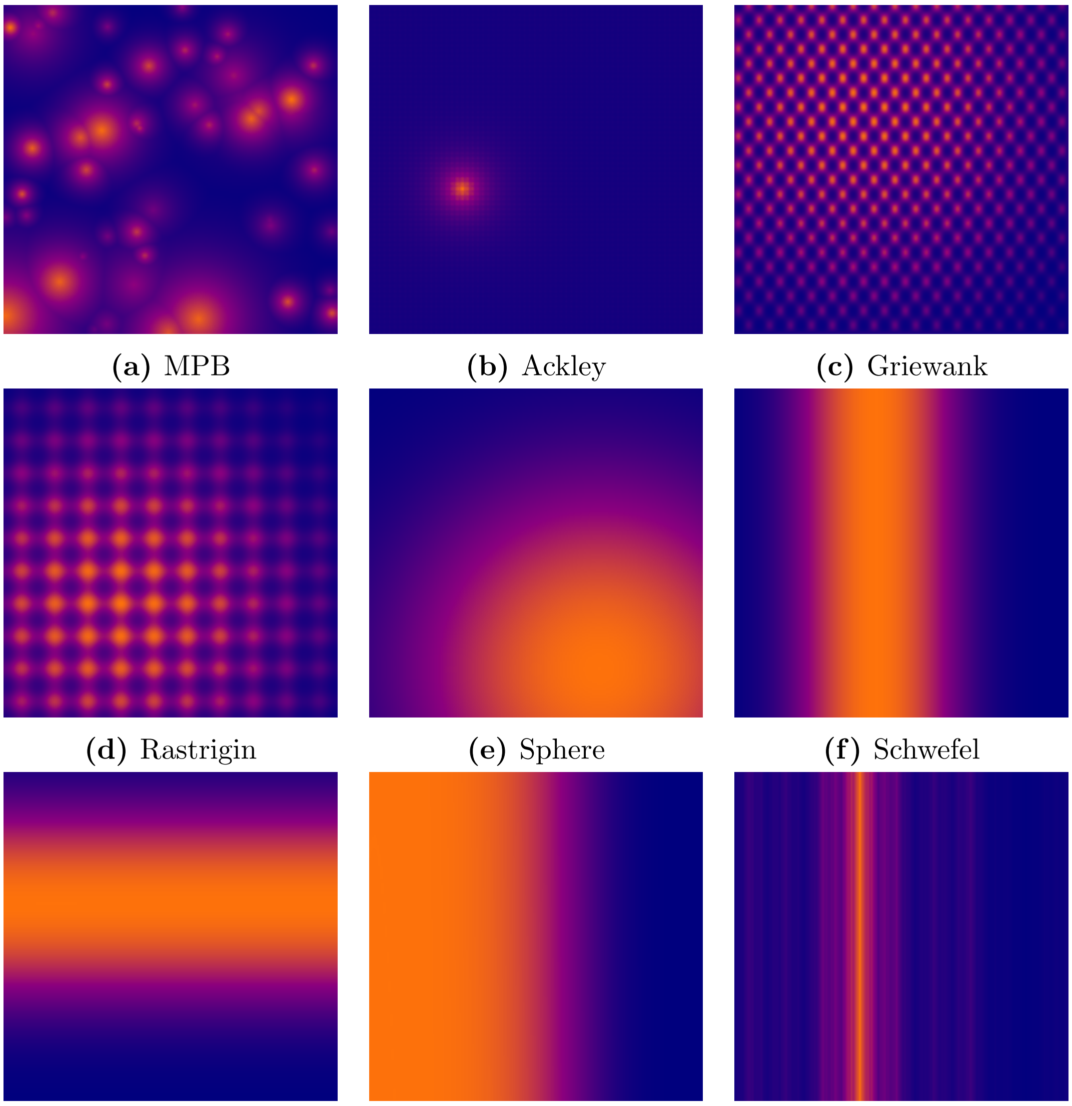


Figure 5.15 – *MODO Optimization Package.* In the exhaustive sampling GUI plugin, yellowish colours are used for displaying near-to-optimum points, while blueish colours are used for the most far-appart-from-optimum points. However, this plugin allows the user to display the environment as a *maximization* problem (**left image**) or as a *minimization* problem (**right image**).

- **Compatibility with a high number of problems and benchmarks from the literature**. The framework includes by default a set of problems and benchmarks widely used in Dynamic Optimization. Among them, we can highlight, in continuous optimization, the MPB or the functions used in the CEC 2009 Competition on Dynamic Optimization [95], such as Ackley, Griewank, Rastrigin, Sphere, Schwefel, High Conditioned Elliptic, etc. (Fig. 5.16 shows visualizations of some of these functions). For discrete optimization, we can mention the OneMax, Plateau, RoyalRoad or Deceptive problems (see Fig. 2.3).

- **Constraints**. The architecture of the UDOPEngine allows not only to incorporate constraints to the evaluation of a solution, but also to penalize the fitness according to the violation degree of the said constraints. Among other benefits, this penalizing mechanism allows to directly visualize the constraints in the GUI plugins. An example of this is shown in Fig. 5.17, representing a scenario of the MPB with some unfeasible regions added. These



(a) MPB **(b)** Ackley **(c)** Griewank

(d) Rastrigin **(e)** Sphere **(f)** Schwefel

(g) High-Cond. Elliptic **(h)** Rosenbrock **(i)** Weierstrass

Figure 5.16 – *MODO Optimization Package.* Some of the continuous fitness functions available in the framework. All of the functions shown here were also used in the CEC-2009 competition on Dynamic Optimization [95].

regions are displayed in dark blue, indicating values very far away from the optimum (the “frame” around the solution space and the vertical fringe on the left). In this case, we have used a *death penalty*, where any violation of a constraint is penalized with the worst possible fitness for the current environment (in this case, although the fitness in the unfeasible regions is $-\infty$, in the image is shown with a value of -50, in order to prevent the colors from saturating too much). However, as it has already been addressed, other penalty functions can be used that would give more information to the algorithm. For example, a fixed penalty that worsens the fitness in a constant way (e.g., by subtracting 100 to the fitness value) would keep the gradient information, helping the algorithm to escape those areas.

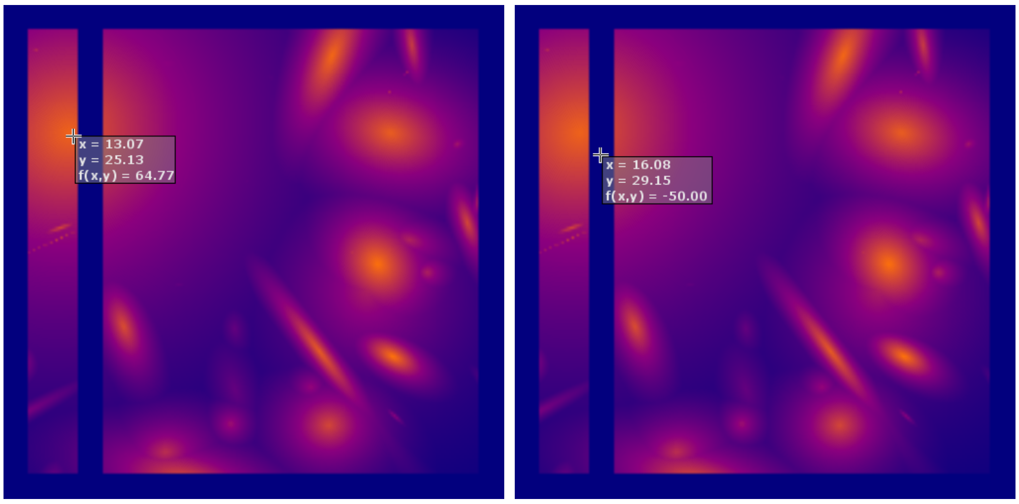


Figure 5.17 – *MODO Optimization Package.* The GUI plugin also allows to visualize constraints. These images show an environment, the MPB, where linear constraints were added, represented in dark blue (the “frame” around the environment boundaries and the vertical line on the left). In these unallowed regions, solutions are assigned a fitness value of $-\infty$, since the MPB is a *maximization* problem. However, in the GUI these values are visualized using a value of -50, in order to prevent image colors from saturating too much. Also notice another feature of the GUI plugin that allows to “inspect” the solution’s value and fitness under the mouse pointer.

- **Composable affine transformations in continuous fitness functions**. The UDOPEngine contains by default a class for implementing continuous fitness functions with an origin of coordinates, where affine transformations can be applied to the solution space defined by such fitness functions. These type of transformations include translating, rotating and scaling, all of which conveniently coded using matrices that can be combined within them to create complex transformations. These transformations can be applied to each fitness function individually, and furthermore, they can also be used to produce changes in the environment. Figure 5.18 shows a sequence of consecutive changes in the environment. The images depict a scenario of the MPB where some of these transformations have been applied to modify the shape of the cone functions. Thus, some of these peaks have a squashed shape, some of them are rotated, and some of them are shifted. Furthermore, since these transformations can be applied in every change, we can see in the images how some of these peaks move through the solution space and even rotate (each peak with its own linear and angular velocity). All these transformations are available to all the continuous fitness functions implemented, so this can also be applied to problems such as Ackley, Griewank, Rastrigin, etc.

- **Uncertainty**. The UDOPEngine allows to include uncertainty in the experiments, although this feature has not been used in any of the papers presented in this thesis. Figure 5.19 shows some visualizations of a MPB scenario with different types of uncertainty. In this case, we have used gaussian noise to introduce both input and output uncertainty, changing the solution values or the fitness values respectively. It is worth noting that although the image is a static snapshot of the environment, each corresponding uncertainty is applied on *each* evaluation of a solution. A visualization closer to what really happens during the optimization process would depict an environment where the image would be continuously changing, quite similarly to an analog TV with a badly tuned channel.

5.2.6. Conclusions

We have presented the MODO Optimization Package, a software framework that contains all the code developed through the experimentation performed in this thesis. This framework reflects not only the code produced to fulfill the necessities of the experiments, but also the experience that we have acquired as we were performing them. This experience has significantly influenced the design and architecture of the components of the framework.

The framework contains 3 main components:

- *Algorithms.* This component groups all the different optimization methods that we have implemented for the experiments, including mQSO and variants, Agents, CS and variants, evolutionary algorithms, etc., as well as learning mechanisms and other techniques of general utility. The algorithms

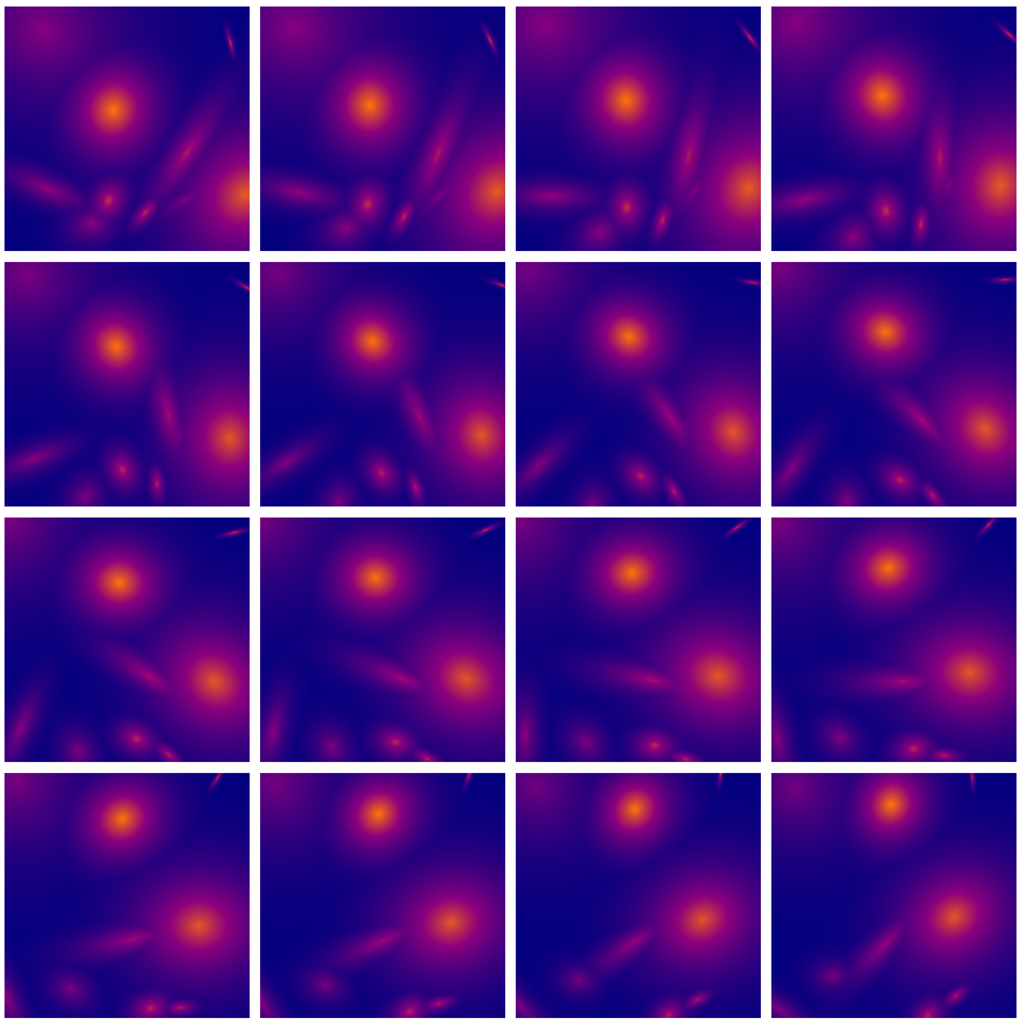


Figure 5.18 – *MODO Optimization Package.* From left to right, and from top to bottom, a sequence of different consecutive changes in a dynamic environment (the MPB). Each peak has its own set of properties (position, height, width, scale factor for each dimension, rotation angle for each pair of dimensions, etc). This sequence shows different dynamic changes applied to some of these properties, such as changes to rotation angles and peak positions.

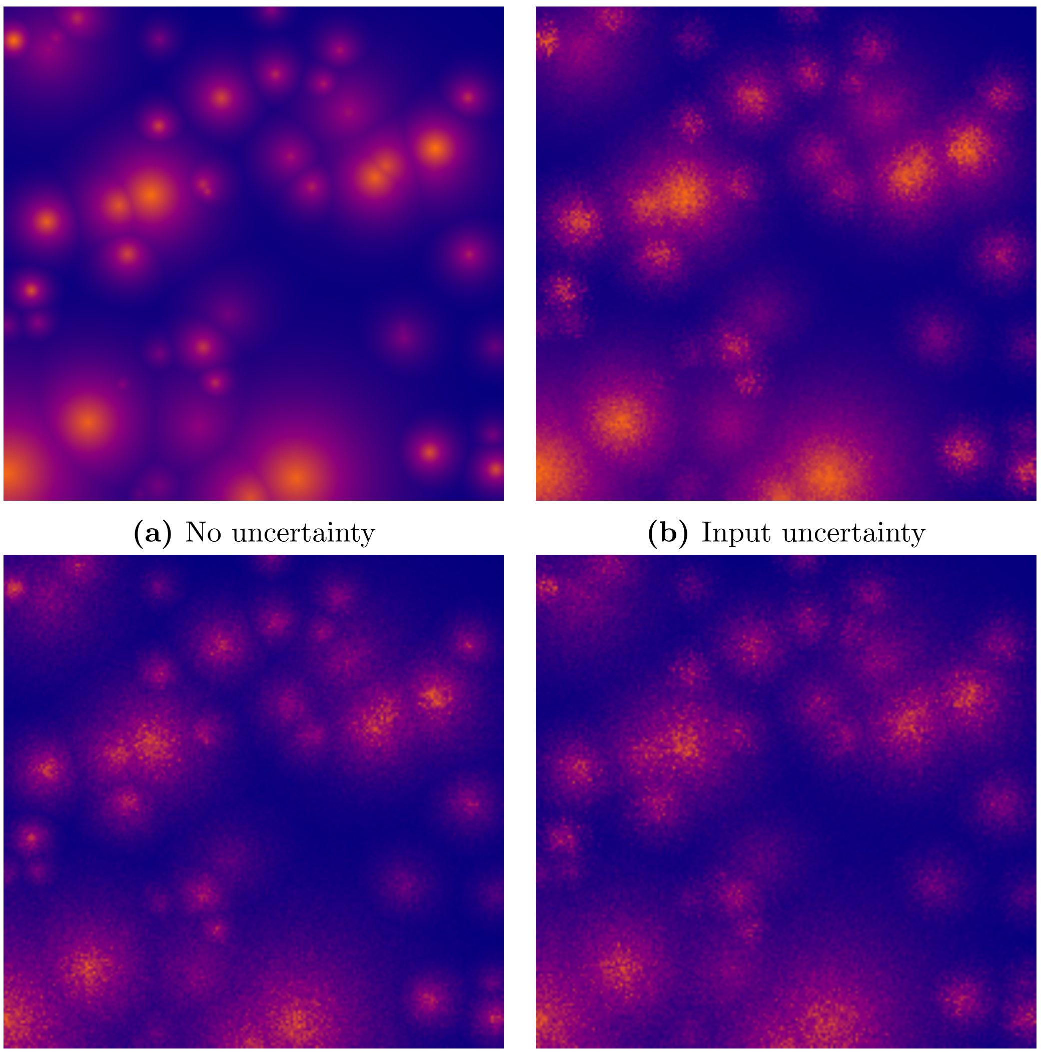

(a) No uncertainty **(b)** Input uncertainty **(c)** Output uncertainty **(d)** Both

Figure 5.19 – *MODO Optimization Package.* Different uncertainty configurations for the same MPB scenario. Uncertainty is added in the form of a 10% gaussian noise. For the input uncertainty this means that for each solution x, the real value x' that is evaluated is $x' = x \pm rand(0.1 * range(x))$. For the output uncertainty, this means that for each fitness $f(x)$, the real value $f'(x)$ that is returned is $f'(x) = f(x) \pm rand(0.1 * range(f(x)))$

developed are based in a class hierarchy presented in [191] that has been adapted to reflect the requirements of the project.

- *Problems.* This component is in charge of everything related with the environment to be optimized. Although the fitness functions used in DOPs are quite diverse, the process of evaluating a solution is very uniform. Therefore, we have designed a *metaclass*, a problem generator called UDOPEngine, that allows to simulate a huge amount of environments. This UDOPEngine is extremely modular, with specific components for dealing with fitness functions, constraints, ways of combining results, and even uncertainty. Moreover, its architecture is designed to allow the addition of extra functionality by means of *plugins*.
- *Plugins.* Plugins are components that allow to extend the capabilities of the UDOPEngine. Plugins can be inserted at special locations in the middle of the evaluation process, named Extension Points, which allow for a fine-grained control of their execution. Plugins, among other things, allow to provide dynamic features to a problem, measuring performance, or reporting results to the user with, e.g., graphical user interfaces.

This framework has been used for the development of all the experiments performed, and has evolved with them. The framework has been released as open source under the Modified BSD License, and is available in the RedIRIS public forge, at the following url:

`https://forja.rediris.es/projects/modooptim/`.

Chapter 6

Conclusions

Considering the investigations carried out in this thesis and the publications that have derived from them, we will organize the conclusions according to the research questions posed in the Objectives.

6.1. Is it possible to improve the existent algorithms, and, if possible, what techniques can be used for that purpose?

The research work has produced, among other results, the following new algorithms or improvements of existent ones:

- A new algorithm for DOPs based on Cooperative Strategies (CS) [69], as well as an improvement for that method with more effective rules [70].
- An Agents-based algorithm for DOPs, with interesting results in continuous problems [40], and very promising ones in combinatorial problems [68].
- A number of improvements and studies on the properties of a PSO variant for DOPs based on the use of heuristic rules [39, 41, 120].

The algorithms that we have developed cover a wide variety of methods: PSO with heuristic rules, cooperative trajectory metaheuristics or agents. However, all these algorithms share some common characteristics:

1. **Use of solution populations**. All these methods keep a set or sets of solutions that "coexist" during the search process. These populations of solutions allow to diversify the search, something important by its own in static optimization, and absolutely essential in DOPs.

2. **Existence of some type of cooperation among its constituent elements**. In the case of CS, this is obvious, since such cooperation is explicit, and is carried out by the central coordinator while exchanging information between the different metaheuristics. In the case of the implemented variants of PSO, the particles of a swarm are connected among them through the *best* of that swarm, since it is used as a reference in the movement of the particles. Furthermore, in the case of the mQSO, which uses multiple swarms, there is an exclusion mechanism that prevents the swarms from getting too close to each other, in order to avoid focusing on the same optimum. This competition between swarms at a local level helps the algorithm to not waste too many resources in the same area, and thus, it is a form of global cooperation in a sense. Agents also implicitly cooperate, since they explore the search space indirectly through the matrix, allowing them to improve solutions that quite probably have already been modified by other agents in previous iterations.

These characteristics confirm what was proposed in Sect. 2.3 about the general trend of the most used algorithms for DOPs. The second characteristic is quite significant in the context of this thesis because it somehow indicates a possible way of improving existing algorithms. All the modifications that we have introduce through the research are based on promoting cooperation and increasing the exchange of information between the elements of an algorithm.

Specially interesting is the case of the *Rand* rule of the mQSO with heuristic rules [39]. The philosophy behind this rule is very similar to that of the PSO with the CPT operator [120] and to the internal working of the CS [69, 70, 104]: observe the behaviour of the elements, and correct those that are doing worse, either by trying to make them imitate the best ones, or by temporarily stopping them to prevent them from wasting resources in critical moments. When this cooperation scheme has been applied to an algorithm, it has improved its results in *all* cases.

Additionally, from the results obtained in [40], it is noticeable the overwhelming supremacy of the Cooperative Strategies (CS) in the Ackley, Griewank and Rastrigin problems. As it was explained in that work, it seems that the CS are specially indicated when the DOP to be solved has some kind of structure in the relative positions of the local optima. Instead, in the problems where such structure does not exist, such as the MPB, the results are much more tied, and the behaviours and tendencies of each algorithm are better appreciated. SORIGA is the best option for very rapidly changing environments, the mQSO + Rand Rule obtains the best results for low dimensionality, and the Agents algorithm behaves better for higher dimensionality and growing severity of the changes.

One interpretation that we can make about this is that the type of environment information used by CS to decide how to cooperate allows them to get the most out of the problem's structure. This information, however, is not as useful in the case of the MPB, because there is no such structure that can be profited, so CS loose their advantage. In the case of the mQSO, the cooperation induced by the *Rand* rule improves its performance, but not that much that it can overcome the intrinsic limitations of the mQSO itself: particles tend to explore the environment following a trajectory and swarms have an effective action radius that, in practice, limits the area in which they perform the search. This is not a big inconvenient in low dimensionality scenarios, but penalizes the algorithm when dimensionality increases, favoring Agents. Finally, SORIGA has an implicit and generic cooperation mechanism that does not use problem-specific knowledge. This is a disadvantage when the conditions of the problem are favorable to other algorithms. However, it gives SORIGA a high robustness, and allows it to obtain good results in scenarios with very fast changes, a situation in which other methods usually do not have enough time to converge.

In the case of discrete problems, the use of the Agents algorithm combined with a learning scheme allows it to outperform one of the state-of-the-art methods in this area, AHMA. The use of explicit cooperation in this case was complicated to perform, because the structure of the problem was unclear and rather random. However, the implicit cooperation of the Agents was appropriate for handling these scenarios, and the learning scheme was combined flawlessly with the Agents, allowing them to outperform AHMA.

A summary of the algorithmic contributions of this thesis is shown in Fig. 6.1, using as a reference the diagram already introduced in Fig. 2.5.

Additionally, from the experience that we have acquired during the development of the experiments, we have elaborated a classification of the algorithms used in this thesis according to two criteria: *optimization power* and *flexibility*. When we talk about *optimization power*, we refer to the ability of an algorithm of obtaining high-quality solutions, close to the optimum. When we talk about *flexibility*, we mean a double concept: the easiness with which an algorithm reacts to changes in the environment at *run-time*, and the easiness with which an algorithm can be adapted to other problems at *design time* (how good is the algorithm in a new problem with some standard parameter settings, how much code it is needed to modify or create in order to make it work in other problems, etc). This classification is shown in Fig. 6.2. The chart is a compendium of the results obtained in different works (specially [40], presented in Sect. 4.4) and our personal experience during their development.

In our opinion, the mQSO family is worse-fitted for DOPs than the other families we have analyzed. Only the mQSO-Rand heuristic variant obtains good results in terms of *optimization power* in some scenarios, thanks to the cooperation scheme that it uses. However, these algorithms contain a high number of

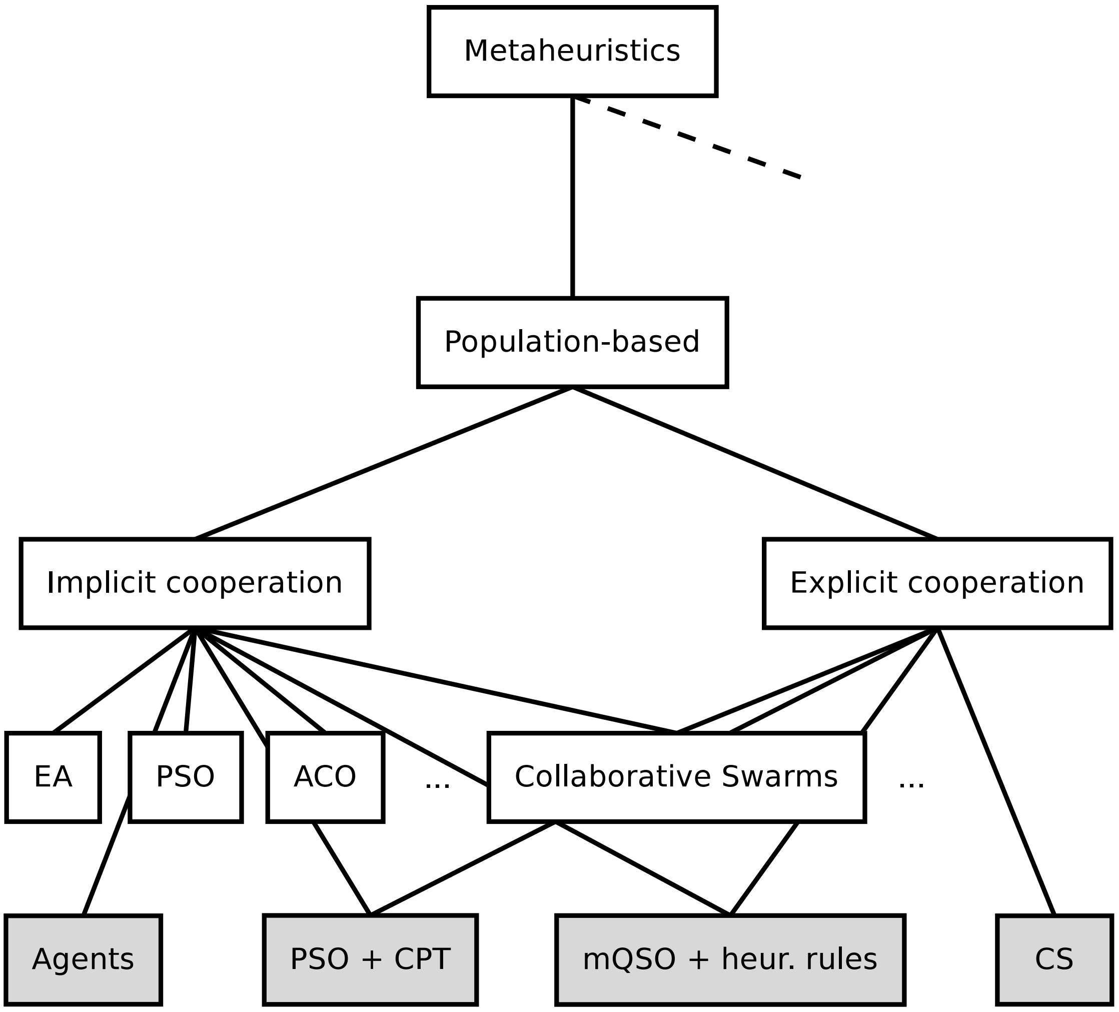


Figure 6.1 – A classification of several population-based metaheuristics depending on the type of cooperation among their constituent components. The contributed methods of this thesis are also represented.

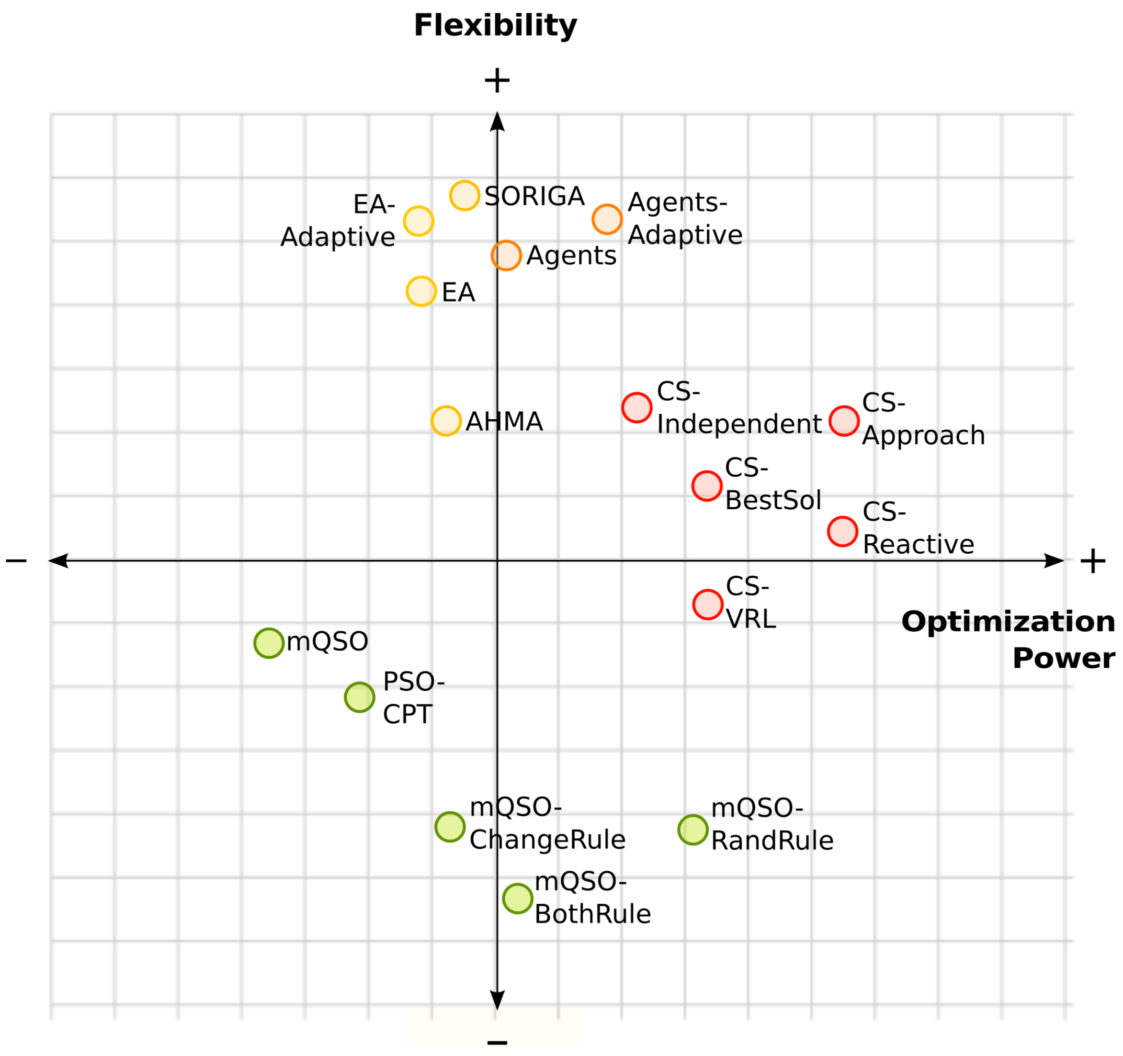


Figure 6.2 – *Conclusions.* Chart showing a classification of all algorithms reviewed in this thesis, according to two variables: *optimization power* and *flexibility*. *Optimization power* refers to the capacity of an algorithm of obtaining high-quality solutions, while *flexibility* refers to the easiness with which an algorithm can either react to changes in the environment at run time, or be ported to other conditions, environments, problems, etc. at design time.

problem-dependent parameters, such as the quantum particles radius, the minimum distance between swarms, the configuration settings of the rules, etc. As we saw in [39, 40], this can lead to settings that only work well in certain scenarios. Moreover, the mQSO is specifically designed for continuous optimization, and the formulation itself of the algorithm is based on some assumptions that can hardly be translated to discrete problems (the movement equations, the concept of "distance", etc). Therefore, our overall impression is that this family has *low flexibility*.

Regarding the CS family, it is obvious that these methods are the ones with the *highest optimization power*, based on the results obtained in [40, 69, 70]. Furthermore, they are *reasonably flexible*, since the CS algorithm has been successfully applied in a wide variety of problems, including continuous and discrete ones. However, the CS algorithm is quite complex, with several abstraction layers including solver implementation, coordinator implementation, configuration settings, communication messages, blackboards, synchronization, etc. Thus, it may be necessary to modify quite a lot of code in order adapt the CS from one problem to another. Therefore, we believe these methods have a *moderate flexibility*.

Finally, the Agents family and the Evolutionary Algorithms family are, by far, the *most flexible* of all. They use very few parameters, or have parameters with very good default values, which allows them to obtain acceptable performance values in most of the problems in the very first executions. For example, SORIGA has no problem-dependent parameters, and the 3x3 grid default setting for the Agents algorithm obtains very good results in all the experiments performed. Moreover, these algorithms perform well even under extreme circumstances, such as DOPs with a very high change frequency or severity (see [40]). On the other hand, they do not have an *optimization power* as high as the CS, but they are able of competing quite well with the mQSO. And in the case of the Agents-Adaptive, the learning scheme applied to the operator selection obtains the best results of this family in the experiments performed.

In conclusion, the results of all these works allow to assert that **not only is possible to improve existent algorithms, but we have also found cooperation-based techniques and learning schemes to do so in a generic way, with great effectiveness and flexibility. Additionally, we have also obtained results that allow us to know what are the most favorable scenarios for using each algorithm**.

6.2. What methodology should be applied in order to compare the performance of different algorithms on a DOP?

The results obtained in the different stages of the research show that an algorithm may be effective for a certain DOP scenario, and perform quite poorly in another one. We saw an example of this in [39] with the *Change* rule in scenarios other than Scenario 2 of the MPB, or with the Agents algorithm compared against the AHMA in some configurations of the Royal Road function, in [68]. This result is widely known in the literature, usually referred to as the *no-free-lunch theorem* [171], i.e., no algorithm can be the best in every possible situation.

With the objective of obtaining a more complete vision of the behaviour, strengths and weaknesses of an algorithm, it is necessary to evaluate its performance in multiple scenarios. On the other hand, as it was previously justified, this multiple evaluation may produce too many results to be fully understood by the reader if only pure numerical data are used.

For all these reasons, we have developed the SRCS technique [38]. SRCS allows to compress the information to be shown in 2 stages:

1. Transform the *absolute* performance data of the algorithms into *relative* data among the compared methods, in order to establish a ranking that determines if an algorithm is *better than* another in a given scenario.

2. Assign colors to each ranking, so that the relative performance of every scenario can be represented in a colored image.

This technique allows to visualize the results of a high number of experiments in a comprehensive manner, helping to identify tendencies and behavioural patterns of the analyzed algorithms. For all these reasons, **thanks to SRCS we can obtain very practical conclusions, not only about which algorithm is better, but — and this is the most important — about in which scenario is better to use one or another**. SRCS has been used in [68] and [40], where it has helped to extract very valuable conclusions.

6.3. What difficulties can we find when implementing these DOPs, algorithms and performance measures, and how can we face them?

The research performed in this thesis contained a strong experimental component. Implementing the software we needed to carry it out was not always an easy task nor it was exempt of difficulties, and the following lessons were extracted:

- In order to obtain useful results in DOPs it is necessary to perform a potentially high number of algorithm executions, possibly over a wide number of scenarios and variations.
- In most cases the algorithms are based in metaheuristics and contain a random factor. Therefore, it is necessary to repeat each execution a certain number of times with different random seeds, in order to obtain a statistically representative sample of the results.
- Due to the random nature of the algorithms, in the analysis of the results is necessary to use statistical tools. Software packages with such capabilities are therefore almost mandatory.
- Test scenarios usually also have a random component related to dynamism. However, contrary to what happens to algorithms, this component must be reproducible, in order to guarantee that the experimental conditions are exactly the same for all the algorithms tested, thus ensuring a fair comparison.
- Most of the algorithms have a set of parameters that must be set before the execution, with a wide range of possible values. This is particularly complex in Cooperative Optimization Strategies (COS), where each component may have its own set of parameters. Configuration mechanisms that ease this process are therefore quite recommended.
- A visualization of an algorithm's performance is extremely helpful, specially in the first stages of an experiment. Visualizations help to decide if the problem is correct and if the algorithm is evolving reasonably. If visualizations can be performed at real time, the better.
- Although there is a great variety of algorithms for DOPs, most of them are based in populations of elements, with some kind of cooperation among them, either explicit or implicit. This opens the door for creating class hierarchies and modules that allow the reutilization of components.

- Similarly, although there is also a great variety of DOP problems, each of them with its own objective function, search space, etc., the process by which a solution is evaluated and assigned a fitness is usually common to all of them. Moreover, dynamic features (such as change frequency, severity, the type of change, etc.), as well as other extended functionality (including performance measures, GUIs, statistics, etc.) can usually be interchanged between problems. As with the algorithms, this again hints that a framework with these components is possible.

Considering this experience, we can obtain several conclusions. The first of all is that **the initial configuration and adjustment of algorithms for DOPs is not an easy task**. With the objective of alleviate this situation, we have created a tool based in software models, DACOS [42], that allows to configure COS — and actually, other algorithms as well — and visualize their performance in the initial stages of the experimentation. DACOS was designed to cover a wide variety of algorithms and problems, but for those cases where specific necessities require to adapt it, the use of software models allows to do so in a semi-automatic way.

Besides, the characteristics of the algorithms used in DOPs imply that most of their constituent components, and even their inner structure itself, can be reused from one algorithm to another when implementing them. The same situation is present in the problems, where the mechanisms for creating and controlling the dynamism are usually the same, and they only differ in the objective function used. This also happens in the performance measures, where in many cases they only need some standardized information of the algorithms or the problems — e.g., the number of solutions in an algorithm's population, when are they evaluated, the value of the optimum, or the time in which the last change in the environment occurred —, no matter what particular algorithm or problem are we testing at the moment.

These circumstances have favoured the **creation of a framework for DOPs with different algorithms, problems, dynamic features and measures**, that we have been developing and improving through all the research. This framework has taken some ideas from a previous publication of the author [191], where a class hierarchy for metaheuristics in static optimization problems was designed. The framework allows to exchange different problems, use multiple algorithms, utilize several performance measures, and even visualize results during run-time. Furthermore, it also allows to introduce stochastic uncertainty in different points of the evaluation process of a solution, which, although is not the main topic of this thesis, it is relevant for the objectives of the research project in which it is enclosed, and it has already contributed to other on-going works. The framework has been released as open source in the RedIRIS public forge, and is accessible through the following url: `https://forja.rediris.es/projects/modooptim/`.

6.4. Summary of publications

This doctoral thesis has produced the following directly related works:

"An Algorithm Comparison for Dynamic Optimization Problems", I. G. del Amo, D. A. Pelta, J. R. González, and A. D. Masegosa, *Applied Soft Computing*, 12(10):3176–3192, 2012. `http://dx.doi.org/10.1016/j.asoc.2012.05.021`.

"A software modeling approach for the design and analysis of cooperative optimization systems", I. G. del Amo, D. A. Pelta, A. D. Masegosa, and J. L. Verdegay, *Software: Practice and Experience*, vol. 40, pp. 811–823, Aug. 2010. `http://dx.doi.org/10.1002/spe.984`.

"SRCS: a technique for comparing multiple algorithms under several factors in Dynamic Optimization Problems", I. G. del Amo and D. A. Pelta, in *Metaheuristics for Dynamic Optimization* (E. Alba, A. Nakib, and P. Siarry, eds.), volume 433 of *Studies in Computational Intelligence*, Springer Berlin/ Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-30665-5_4`

"A cooperative strategy for solving dynamic optimization problems", J. R. González, A. D. Masegosa, and I. G. del Amo, *Memetic Computing*, vol. 3, no. 1, pp. 3–14, 2010. `http://dx.doi.org/10.1007/s12293-010-0031-x`.

“Cooperation rules in a trajectory-based centralised cooperative strategy for Dynamic Optimisation Problems”, J. R. González, A. D. Masegosa, I. G. del Amo, and D. A. Pelta, in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pp. 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586063`.

“Using heuristic rules to enhance a multiswarm PSO for dynamic environments”, I. G. del Amo, D. A. Pelta, and J. R. González, in *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pp. 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586051`.

“An Analysis of Particle Properties on a Multi-swarm PSO for Dynamic Optimization Problems”, I. G. del Amo, D. A. Pelta, J. R. González, and P. Novoa, in *Current Topics in Artificial Intelligence* (P. Meseguer, L. Mandow, and R. Gasca, eds.), vol. 5988 of *Lecture Notes in Computer Science*, pp. 32–41, Springer Berlin / Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-14264-2_4`.

“An Adaptive Multiagent Strategy for Solving Combinatorial Dynamic Optimization Problems”, J. González, C. Cruz, I. G. del Amo, and D. Pelta, in *Nature Inspired Cooperative Strategies for Optimization (NICSO 2011)* (D. Pelta, N. Krasnogor, D. Dumitrescu, C. Chira, and R. Lung, eds.), vol. 387 of *Studies in Computational Intelligence*, pp. 41–55, Springer Berlin / Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-24094-2_3`.

“Controlling Particle Trajectories in a Multi-swarm Approach for Dynamic Optimization Problems”, P. Novoa, D. A. Pelta, C. Cruz, and I. G. del Amo, in *Methods and Models in Artificial and Natural Computation. A Homage to Professor Mira’s Scientific Legacy* (J. Mira, J. Ferrández, J. Álvarez, F. de la Paz, and F. Toledo, eds.), vol. 5601 of *Lecture Notes in Computer Science*, pp. 285–294, Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-02264-7_30`.

Additionally, other works in which Ignacio G. del Amo is author or co-author are also indirectly related to this thesis. The indirect relationship is caused by the fact that these works are not explicitly devoted to DOPs, but have a clear influence in some of the results presented here:

“From Theory to Implementation: Applying Metaheuristics.”, A. Zaslavski, I. G. del Amo, F. G. López, M. G. Torres, B. M. Batista, J. A. M. Pérez, and J. M. Moreno-Vega. From Theory to Implementation: Applying Metaheuristics. In P. Pardalos, L. Liberti, and N. Maculan, editors, *Global Optimization*, volume 84 of *Nonconvex Optimization and Its Applications*, pages 311–351. Springer US, 2006. `http://dx.doi.org/10.1007/0-387-30528-9_11`.

“On the Performance of Homogeneous and Heterogeneous Cooperative Search Strategies”, A. Masegosa, D. Pelta, I. G. del Amo, and J. Verdegay. In N. Krasnogor, M. Melián-Batista, J. Pérez, J. Moreno-Vega, and D. Pelta, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2008)*, volume 236 of *Studies in Computational Intelligence*, pages 287–300. Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-03211-0_24`.

Finally, most of the software produced in the thesis has been released as open source under the MODO Optimization Package, in the RedIRIS public forge, and is accessible through the following url:

`https://forja.rediris.es/projects/modooptim/`.

Bibliography

[1] H. A. Abbass, K. Sastry, and D. E. Goldberg. Oiling the Wheels of Change: The Role of Adaptive Automatic Problem Decomposition in Non-Stationary Environments (IlliGAL Report No. 2004029). Technical report, University of Illinois at Urbana-Champaign, Illinois Genetic Algorithms Laboratory (IlliGAL), 2004.

[2] A. Ajith, G. Crina, R. Vitorino, M. De, N. Slawomir, and B. Mark. Stochastic Diffusion Search: Partial Function Evaluation In Swarm Intelligence Dynamic Optimisation. In *Stigmergic Optimization*, volume 31 of *Studies in Computational Intelligence*, pages 185–207. Springer Berlin / Heidelberg, 2006. `http://dx.doi.org/10.1007/978-3-540-34690-6_8`.

[3] D. V. Arnold and H.-G. Beyer. Random Dynamics Optimum Tracking with Evolution Strategies. In *Proceedings of the VII Int. Conference on Parallel Problem Solving from Nature (PPSN-2002)*, pages 3–12. Springer, 2002.

[4] D. V. Arnold and H.-G. Beyer. Optimum Tracking with Evolution Strategies. *Evolutionary Computation*, 14(3):291–308, Aug 2006. `http://dx.doi.org/10.1162/evco.2006.14.3.291`.

[5] M. E. Aydin and E. Öztemel. Dynamic job-shop scheduling using reinforcement learning agents. *Robotics and Autonomous Systems*, 33(2-3):169–178, 2000.

[6] D. Ayvaz, H. Topcuoglu, and F. Gurgen. A comparative study of evolutionary optimization techniques in dynamic environments. In *Proceedings of the 2006 Annual Conference on Genetic and Evolutionary Computation (GECCO-2006)*, pages 1397–1398, New York, NY, USA, 2006. ACM. `http://dx.doi.org/10.1145/1143997.1144213`.

[7] P. Bak. *How Nature Works: The Science of Self-Organised Criticality.* Copernicus Press, New York, USA, 1996.

[8] C. Barrico and C. Antunes. An Evolutionary Approach for Assessing the Degree of Robustness of Solutions to Multi-Objective Models. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 565–582. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_25`.

[9] T. Bartz-Beielstein. *Experimental Research in Evolutionary Computation: The New Experimentalism*. Natural Computing Series. Springer Berlin / Heidelberg, Germany, 2006. `http://dx.doi.org/10.1007/3-540-32027-X`.

[10] R. Battiti, M. Brunato, and F. Mascia. *Reactive Search and Intelligent Optimization*, volume 45 of *Operations Research/Computer Science Interfaces*. Springer Verlag, New York, USA, 2008. `http://dx.doi.org/10.1007/978-0-387-09624-7`.

[11] BIRT. Business Intelligence and Reporting Tools (BIRT). `http://www.eclipse.org/birt/`, 2008.

[12] T. Blackwell. Swarms in Dynamic Environments. In E. Cantú-Paz, J. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. Potter, A. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the 2003 Annual Conference on Genetic and Evolutionary Computation (GECCO-2003)*, volume 2723 of *Lecture Notes in Computer Science*, pages 1–12. Springer Berlin / Heidelberg, 2003. `http://dx.doi.org/10.1007/3-540-45105-6_1`.

[13] T. Blackwell. Particle Swarm Optimization in Dynamic Environments. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 29–49. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_2`.

[14] T. Blackwell and J. Branke. Multi-swarm optimization in dynamic environments. In *Applications of Evolutionary Computing*, volume 3005 of *Lecture Notes in Computer Science*, pages 489–500. Springer, 2004. `http://dx.doi.org/10.1007/978-3-540-24653-4_50`.

[15] T. Blackwell and J. Branke. Multiswarms, exclusion, and anti-convergence in dynamic environments. *IEEE Transactions on Evolutionary Computation*, 10(4):459–472, 2006. `http://dx.doi.org/10.1109/TEVC.2005.857074`.

[16] T. M. Blackwell. Particle swarms and population diversity. *Soft Computing*, 9(11):793–802, 2005. http://dx.doi.org/10.1007/s00500-004-0420-5.

[17] P. Bonissone, Y.-T. Chen, K. Goebel, and P. Khedkar. Hybrid soft computing systems: industrial and commercial applications. *Proceedings of the IEEE*, 87(9):1641–1667, 1999. http://dx.doi.org/10.1109/5.784245.

[18] P. P. Bonissone. Soft computing: the convergence of emerging reasoning technologies. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 1:6–18, 1997. http://dx.doi.org/10.1007/s005000050002.

[19] P. Bosman. Learning and Anticipation in Online Dynamic Optimization. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 129–152. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_6.

[20] P. A. N. Bosman. Learning, anticipation and time-deception in evolutionary online dynamic optimization. In *Proceedings of the 2005 Annual Conference on Genetic and Evolutionary Computation (GECCO-2005)*, pages 39–47, New York, NY, USA, 2005. ACM. http://doi.acm.org/10.1145/1102256.1102264.

[21] A. Boumaza. Learning environment dynamics from self-adaptation: a preliminary investigation. In *Proceedings of the 2005 Annual Conference on Genetic and Evolutionary Computation (GECCO-2005)*, pages 48–54, New York, NY, USA, 2005. ACM. http://dx.doi.org/10.1145/1102256.1102265.

[22] J. Branke. Memory enhanced evolutionary algorithms for changing optimization problems. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation (CEC-1999)*, volume 3, pages 1875–1882. IEEE, 1999. http://dx.doi.org/10.1109/CEC.1999.785502.

[23] J. Branke. *Evolutionary Optimization in Dynamic Environments*, volume 3 of *Genetic algorithms and evolutionary computation.* Kluwer Academic Publishers, Massachusetts, USA, 2001.

[24] J. Branke. Editorial: special issue on dynamic optimization problems. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 9:777–777, 2005. http://dx.doi.org/10.1007/s00500-004-0418-z.

[25] J. Branke and Y. Jin. Guest Editorial Special Issue on Evolutionary Computation in the Presence of Uncertainty. *IEEE Transactions on Evolutionary Computation*, 10(4):377–379, aug. 2006. `http://dx.doi.org/10.1109/TEVC.2005.859466`.

[26] J. Branke, T. Kaubler, C. Schmidt, and H. Schmeck. A Multi-Population Approach to Dynamic Optimization Problems. *Adaptive Computing in Design and Manufacture*, pages 299–308, 2000.

[27] J. Branke, M. Orbayı, and c. Uyar. The Role of Representations in Dynamic Knapsack Problems. In F. Rothlauf, J. Branke, S. Cagnoni, E. Costa, C. Cotta, R. Drechsler, E. Lutton, P. Machado, J. Moore, J. Romero, G. Smith, G. Squillero, and H. Takagi, editors, *Applications of Evolutionary Computing*, volume 3907 of *Lecture Notes in Computer Science*, pages 764–775. Springer Berlin / Heidelberg, 2006. `http://dx.doi.org/10.1007/11732242_74`.

[28] J. Branke and H. Schmeck. Designing Evolutionary Algorithms for Dynamic Optimization Problems. In A. Ghosh and S. Tsutsui, editors, *Advances in Evolutionary Computing*, Natural Computing Series, pages 239–262. Springer Berlin Heidelberg, 2003. `http://dx.doi.org/10.1007/978-3-642-18965-4_9`.

[29] L. Bui, H. Abbass, and J. Branke. Multiobjective optimization for dynamic environments. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, volume 3, pages 2349–2356 Vol. 3, sept. 2005. `http://dx.doi.org/10.1109/CEC.2005.1554987`.

[30] L. T. Bui, J. Branke, and H. A. Abbass. Diversity as a selection pressure in dynamic environments. In *Proceedings of the 2005 Annual Conference on Genetic and Evolutionary Computation (GECCO-2005)*, pages 1557–1558, New York, NY, USA, 2005. ACM. `http://dx.doi.org/10.1145/1068009.1068257`.

[31] A. Carlisle and G. Dozier. Adapting Particle Swarm Optimization to Dynamic Environments. In *Proceedings of the 2000 International Conference on Artificial Intelligence (ICAI-2000)*, pages 429–434, 2000.

[32] M. Clerc and J. Kennedy. The particle swarm - explosion, stability, and convergence in a multidimensional complex space. *IEEE Transactions on Evolutionary Computation*, 6(1):58–73, 2002. `http://dx.doi.org/10.1109/4235.985692`.

[33] H. G. Cobb. An Investigation into the Use of Hypermutation as an Adaptive Operator in Genetic Algorithms Having Continuouis, Time-Dependent Nonstationary Environments. Technical Report AIC-90-001, Naval Research Laboratory, 1990.

[34] C. Cruz, J. González, and D. Pelta. Optimization in dynamic environments: a survey on problems, methods and measures. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 15:1427–1448, 2011. `http://dx.doi.org/10.1007/s00500-010-0681-0`.

[35] C. Cruz and D. Pelta. Soft Computing and Cooperative Strategies for Optimization. *Applied Soft Computing*, 9(1):30–38, 2009. `http://dx.doi.org/10.1016/j.asoc.2007.12.007`.

[36] H. Dam, C. Lokan, and H. Abbass. Evolutionary Online Data Mining: An Investigation in a Dynamic Environment. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 153–178. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_7`.

[37] K. Deb and P. Nain. An Evolutionary Multi-objective Adaptive Meta-modeling Procedure Using Artificial Neural Networks. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 297–322. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_13`.

[38] I. G. del Amo and D. Pelta. SRCS: A Technique for Comparing Multiple Algorithms under Several Factors in Dynamic Optimization Problems. In E. Alba, A. Nakib, and P. Siarry, editors, *Metaheuristics for Dynamic Optimization*, volume 433 of *Studies in Computational Intelligence*, pages 61–77. Springer Berlin / Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-30665-5_4`.

[39] I. G. del Amo, D. A. Pelta, and J. R. González. Using heuristic rules to enhance a multiswarm PSO for dynamic environments. In *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pages 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586051`.

[40] I. G. del Amo, D. A. Pelta, J. R. González, and A. D. Masegosa. An Algorithm Comparison for Dynamic Optimization Problems. *Applied Soft Computing*, 12(10):3176–3192, 2012. `http://dx.doi.org/10.1016/j.asoc.2012.05.021`.

[41] I. G. del Amo, D. A. Pelta, J. R. González, and P. Novoa. An Analysis of Particle Properties on a Multi-swarm PSO for Dynamic Optimization Problems. In P. Meseguer, L. Mandow, and R. Gasca, editors, *Current Topics in Artificial Intelligence*, volume 5988 of *Lecture Notes in Computer Science*, pages 32–41. Springer Berlin / Heidelberg, 2010. `http://dx.doi.org/10.1007/978-3-642-14264-2_4`.

[42] I. G. del Amo, D. A. Pelta, A. D. Masegosa, and J. L. Verdegay. A software modeling approach for the design and analysis of cooperative optimization systems. *Software: Practice and Experience*, 40(9):811–823, Aug. 2010. `http://dx.doi.org/10.1002/spe.984`.

[43] J. Demšar. Statistical Comparisons of Classifiers over Multiple Data Sets. *Journal of Machine Learning Research*, 7(1), 2006.

[44] S. Droste. Analysis of the (1+1) EA for a Dynamically Bitwise Changing OneMax. In E. Cantú-Paz, J. Foster, K. Deb, L. Davis, R. Roy, U.-M. O'Reilly, H.-G. Beyer, R. Standish, G. Kendall, S. Wilson, M. Harman, J. Wegener, D. Dasgupta, M. Potter, A. Schultz, K. Dowsland, N. Jonoska, and J. Miller, editors, *Proceedings of the 2003 Annual Conference on Genetic and Evolutionary Computation (GECCO-2003)*, volume 2723 of *Lecture Notes in Computer Science*, pages 202–202. Springer Berlin / Heidelberg, 2003. `http://dx.doi.org/10.1007/3-540-45105-6_103`.

[45] W. Du and B. Li. Multi-strategy ensemble particle swarm optimization for dynamic optimization. *Information Sciences*, 178(15):3096 – 3109, 2008. `http://dx.doi.org/10.1016/j.ins.2008.01.020`.

[46] R. Eberhart and Y. Shi. Tracking and optimizing dynamic systems with particle swarms. In *Proceedings of the 2001 IEEE Congress on Evolutionary Computation (CEC-2001)*, volume 1, pages 94–100 vol. 1, 2001. `http://dx.doi.org/10.1109/CEC.2001.934376`.

[47] R. C. Eberhart and Y. Shi. Comparing inertia weights and constriction factors in particle swarm optimization. In *Proceedings of the 2000 IEEE Congress on Evolutionary Computation (CEC-2000)*, volume 1, pages 84–88. IEEE, 2000. `http://dx.doi.org/10.1109/CEC.2000.870279`.

[48] Eclipse. Eclipse. `http://www.eclipse.org/`, 2008.

[49] A. Elshamli, H. Abdullah, and S. Areibi. Genetic Algorithm for Dynamic Path Planning. In *Canadian Conference on Electrical and Computer Engineering*, 2004.

[50] EMF. Eclipse Modeling Framework. `http://www.eclipse.org/emf`, 2008.

[51] R. Eriksson and B. Olsson. On the Behavior of Evolutionary Global-Local Hybrids with Dynamic Fitness Functions. In J. J. M. Guervós, P. Adamidis, H.-G. Beyer, H.-P. Schwefel, and J.-L. Fernández-Villacañas, editors, *Proceedings of the VII Int. Conference on Parallel Problem Solving from Nature (PPSN-2002)*, volume 2439 of *Lecture Notes in Computer Science*, pages 13–22. Springer Berlin Heidelberg, 2002. `http://dx.doi.org/10.1007/3-540-45712-7_2`.

[52] R. Eriksson and B. Olsson. On the performance of evolutionary algorithms with life-time adaptation in dynamic fitness landscapes. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC-2004)*, 2004. `http://dx.doi.org/10.1109/CEC.2004.1331046`.

[53] L. J. Eshelman and J. D. Schaffer. Real-coded genetic algorithms and interval-schemata. In L. D. Whitley, editor, *Proceedings of the II Workshop on Foundation of Genetic Algorithms (FOGA-1992)*, pages 187–202, San Mateo, CA, USA, 1993. Morgan Kaufmann.

[54] S. Esquivel and C. Coello. Particle Swarm Optimization in Non-stationary Environments. In C. Lemaître, C. Reyes, and J. González, editors, *Advances in Artificial Intelligence – IBERAMIA 2004*, volume 3315 of *Lecture Notes in Computer Science*, pages 757–766. Springer Berlin / Heidelberg, 2004. `http://dx.doi.org/10.1007/978-3-540-30498-2_76`.

[55] S. C. Esquivel and C. A. Coello Coello. Hybrid particle swarm optimizer for a class of dynamic fitness landscape. *Engineering Optimization*, 38(8):873–888, 2006. `http://dx.doi.org/10.1080/03052150600772226`.

[56] Z. Fan, J. Wang, M. Wen, E. Goodman, and R. Rosenberg. An Evolutionary Approach For Robust Layout Synthesis of MEMS. In *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 519–542. Springer, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_23`.

[57] J. Ferber. *Multi-Agent Systems: An Introduction to Distributed Artificial Intelligence*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[58] C. M. Fernandes, C. Lima, and A. C. Rosa. UMDAs for dynamic optimization problems. In *Proceedings of the 2008 Annual Conference on Genetic and Evolutionary Computation (GECCO-2008)*, pages 399–406, New York, NY, USA, 2008. ACM. `http://dx.doi.org/10.1145/1389095.1389170`.

[59] C. M. Fernandes, A. C. Rosa, and V. Ramos. Binary ant algorithm. In *Proceedings of the 2007 Annual Conference on Genetic and Evolutionary Computation (GECCO-2007)*, pages 41–48, New York, NY, USA, 2007. ACM. http://dx.doi.org/10.1145/1276958.1276965.

[60] L. J. Fogel, A. J. Owens, and M. J. Walsh. *Artificial Intelligence through Simulated Evolution*. John Wiley, 1966.

[61] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley Professional, January 1995.

[62] S. García and F. Herrera. An Extension on "Statistical Comparisons of Classifiers over Multiple Data Sets" for all Pairwise Comparisons. *Journal of Machine Learning Research*, 9:2677–2694, Dec. 2008.

[63] S. García, D. Molina, M. Lozano, and F. Herrera. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization. *Journal of Heuristics*, 15(6):617–644, 2009. http://dx.doi.org/10.1007/s10732-008-9080-4.

[64] A. Ghosh and H. Mühlenbein. Univariate marginal distribution algorithms for non-stationary optimization problems. *International Journal of Knowledge-based and Intelligent Engineering Systems*, 8(3):129–138, 2004.

[65] F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers and Operations Research*, 13(5):533–549, May 1986. http://dx.doi.org/10.1016/0305-0548(86)90048-1.

[66] C. Goh and K. Tan. Evolving the Tradeoffs between Pareto-Optimality and Robustness in Multi-Objective Evolutionary Algorithms. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 457–478. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_20.

[67] D. E. Goldberg and R. E. Smith. Nonstationary function optimization using genetic algorithm with dominance and diploidy. In *Proceedings of the Second International Conference on Genetic Algorithms and their Application*, pages 59–68, Hillsdale, NJ, USA, 1987. L. Erlbaum Associates Inc.

[68] J. González, C. Cruz, I. G. del Amo, and D. Pelta. An Adaptive Multiagent Strategy for Solving Combinatorial Dynamic Optimization Problems. In

D. Pelta, N. Krasnogor, D. Dumitrescu, C. Chira, and R. Lung, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2011)*, volume 387 of *Studies in Computational Intelligence*, pages 41–55. Springer Berlin / Heidelberg, 2012. `http://dx.doi.org/10.1007/978-3-642-24094-2_3`.

[69] J. R. González, A. D. Masegosa, and I. G. del Amo. A cooperative strategy for solving dynamic optimization problems. *Memetic Computing*, 3(1):3–14, 2010. `http://dx.doi.org/10.1007/s12293-010-0031-x`.

[70] J. R. González, A. D. Masegosa, I. G. del Amo, and D. A. Pelta. Cooperation rules in a trajectory-based centralised cooperative strategy for Dynamic Optimisation Problems. In *Proceedings of the 2010 IEEE Congress on Evolutionary Computation (CEC-2010)*, pages 1–8, 2010. `http://dx.doi.org/10.1109/CEC.2010.5586063`.

[71] J. Grefenstette. Genetic Algorithms for Changing Environments. In R. Maenner and B. Manderick, editors, *Proceedings of the II Int. Conference on Parallel Problem Solving from Nature (PPSN-1992)*, pages 137–144. Elsevier Science Inc., 1992.

[72] M. Guntsch, M. Middendorf, and H. Schmeck. An Ant Colony Optimization Approach to Dynamic TSP. In L. Spector and et al., editors, *Proceedings of the 2001 Annual Conference on Genetic and Evolutionary Computation (GECCO-2001)*, pages 860–867. Morgan Kaufmann, 2001.

[73] H. Handa, L. Chapman, and X. Yao. Robust Salting Route Optimization Using Evolutionary Algorithms. In *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 497–517. Springer, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_22`.

[74] F. Hanshar and B. Ombuki-Berman. Dynamic vehicle routing using genetic algorithms. *Applied Intelligence*, 27:89–99, 2007. `http://dx.doi.org/10.1007/s10489-006-0033-z`.

[75] E. Hart and P. Ross. An immune system approach to scheduling in changing environments. In *Proceedings of the 1999 Annual Conference on Genetic and Evolutionary Computation (GECCO-1999)*, pages 1559–1565. Morgan Kaufmann, 1999.

[76] A. Hedar and M. Fukushima. Heuristic pattern search and its hybridization with simulated annealing for nonlinear global optimization. *Optimization Methods and Software*, 19(3–4):291–308, 2004. `http://dx.doi.org/10.1080/10556780310001645189`.

[77] A.-R. Hedar and M. Fukushima. Tabu Search directed by direct search methods for nonlinear global optimization. *European Journal of Operational Research*, 170:329–349, 2006. http://dx.doi.org/10.1016/j.ejor.2004.05.033.

[78] J. H. Holland. *Adaptation in Natural and Artificial Systems: An Introductory Analysis with Applications to Biology, Control, and Artificial Intelligence.* University of Michigan Press, 1975.

[79] M. Hollander and D. Wolfe. *Nonparametric Statistical Methods.* John Wiley & Sons, Inc., second edition, 1999.

[80] S. Holm. A Simple Sequentially Rejective Multiple Test Procedure. *Scandinavian Journal of Statistics*, 6(2):65–70, 1979.

[81] J. Hu, S. Li, and E. Goodman. Evolutionary Robust Design of Analog Filters Using Genetic Programming. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 479–496. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_21.

[82] X. Hu and R. Eberhart. Adaptive particle swarm optimization: detection and response to dynamic systems. In *Proceedings of the 2002 IEEE Congress on Evolutionary Computation (CEC-2002)*, volume 2, pages 1666–1670, 2002. http://dx.doi.org/10.1109/CEC.2002.1004492.

[83] S. Janson and M. Middendorf. A Hierarchical Particle Swarm Optimizer for Dynamic Optimization Problems. In *Applications of Evolutionary Computing*, volume 3005 of *Lecture Notes in Computer Science*, pages 513–524. Springer, 2004. http://dx.doi.org/10.1007/978-3-540-24653-4_52.

[84] Y. Jin and J. Branke. Evolutionary optimization in uncertain environments-a survey. *IEEE Transactions on Evolutionary Computation*, 9(3):303–317, june 2005. http://dx.doi.org/10.1109/TEVC.2005.846356.

[85] Y. Jin and B. Sendhoff. Constructing Dynamic Optimization Test Problems Using the Multi-objective Optimization Concept. In G. Raidl, S. Cagnoni, J. Branke, D. Corne, R. Drechsler, Y. Jin, C. Johnson, P. Machado, E. Marchiori, F. Rothlauf, G. Smith, and G. Squillero, editors, *Applications of Evolutionary Computing*, volume 3005 of *Lecture Notes in Computer Science*, pages 525–536. Springer Berlin Heidelberg, 2004. http://dx.doi.org/10.1007/978-3-540-24653-4_53.

[86] A. Karaman, c. Uyar, and G. Eryiğit. The Memory Indexing Evolutionary Algorithm for Dynamic Environments. In F. Rothlauf, J. Branke, S. Cagnoni, D. Corne, R. Drechsler, Y. Jin, P. Machado, E. Marchiori, J. Romero, G. Smith, and G. Squillero, editors, *Applications of Evolutionary Computing*, volume 3449 of *Lecture Notes in Computer Science*, pages 563–573. Springer Berlin Heidelberg, 2005. `http://dx.doi.org/10.1007/978-3-540-32003-6_59`.

[87] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of 1995 IEEE International Conference on Neural Networks (ICNN-1995)*, volume 4, pages 1942–1948. IEEE, 1995. `http://dx.doi.org/10.1109/ICNN.1995.488968`.

[88] G. Kramer and J. Gallagher. Improvements to the *CGA Enabling Online Intrinsic Evolution in Compact EH Devices. In *Proceedings of the NASA/DoD Conference on Evolvable Hardware*, pages 225–231, 2003.

[89] W. H. Kruskal and W. A. Wallis. Use of Ranks in One-Criterion Variance Analysis. *Journal of the American Statistical Association*, 47(260):pp. 583–621, 1952.

[90] J. Laredo, P. Castillo, A. Mora, J. Merelo, A. Rosa, and C. Fernandes. Evolvable Agents in Static and Dynamic Optimization Problems. In G. Rudolph, T. Jansen, S. Lucas, C. Poloni, and N. Beume, editors, *Parallel Problem Solving from Nature – PPSN X*, volume 5199 of *Lecture Notes in Computer Science*, pages 488–497. Springer Berlin / Heidelberg, 2008. `http://dx.doi.org/10.1007/978-3-540-87700-4_49`.

[91] R. V. Lenth. Some Practical Guidelines for Effective Sample Size Determination. *The American Statistician*, 55(3):187–193, August 1 2001. `http://dx.doi.org/10.1198/000313001317098149`.

[92] W.-F. Leong and G. Yen. Dynamic swarms in PSO-based multiobjective optimization. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC-2007)*, pages 3172–3179. IEEE, sept. 2007. `http://dx.doi.org/10.1109/CEC.2007.4424877`.

[93] C. Li and S. Yang. A Generalized Approach to Construct Benchmark Problems for Dynamic Optimization. In *Simulated Evolution and Learning*, volume 5361 of *Lecture Notes in Computer Science*, pages 391–400. Springer Berlin / Heidelberg, 2008. `http://dx.doi.org/10.1007/978-3-540-89694-4_40`.

[94] C. Li and S. Yang. Fast Multi-Swarm Optimization for Dynamic Optimization Problems. In *Natural Computation, 2008. ICNC '08. Fourth International Conference on*, volume 7, pages 624–628, oct. 2008. `http://dx.doi.org/10.1109/ICNC.2008.313`.

[95] C. Li, S. Yang, T. T. Nguyen, E. L. Yu, X. Yao, Y. Jin, H.-G. Beyer, and P. N. Suganthan. Benchmark Generator for CEC'2009 Competition on Dynamic Optimization. Technical report, University of Leicester, University of Birmingham, Nanyang Technological University, Sept. 2008.

[96] X. Li, J. Branke, and T. Blackwell. Particle swarm with speciation and adaptation in a dynamic environment. In *Proceedings of the 2006 Annual Conference on Genetic and Evolutionary Computation (GECCO-2006)*, pages 51–58, New York, NY, USA, 2006. ACM. `http://dx.doi.org/10.1145/1143997.1144005`.

[97] D. Lim, Y.-S. Ong, M.-H. Lim, and Y. Jin. Single/Multi-objective Inverse Robust Evolutionary Design Methodology in the Presence of Uncertainty. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 437–456. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_19`.

[98] Q. Ling, G. Wu, and Q. Wang. Deterministic Robust Optimal Design Based on Standard Crowding Genetic Algorithm. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 583–598. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_26`.

[99] R. Lung and D. Dumitrescu. Evolutionary swarm cooperative optimization in dynamic environments. *Natural Computing*, 9:83–94, 2010. `http://dx.doi.org/10.1007/s11047-009-9129-9`.

[100] R. I. Lung and D. Dumitrescu. A new collaborative evolutionary-swarm optimization technique. In *Proceedings of the 2007 Annual Conference on Genetic and Evolutionary Computation (GECCO-2007)*, pages 2817–2820, New York, NY, USA, 2007. ACM. `http://dx.doi.org/10.1145/1274000.1274043`.

[101] Y. Mack, T. Goel, W. Shyy, and R. Haftka. Surrogate Model-Based Optimization Framework: A Case Study in Aerospace Design. In S. Yang, Y.-S.

Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 323–342. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_14`.

[102] H. B. Mann and D. R. Whitney. On a Test of Whether one of Two Random Variables is Stochastically Larger than the Other. *The Annals of Mathematical Statistics*, 18(1):pp. 50–60, 1947. `http://dx.doi.org/10.1214/aoms/1177730491`.

[103] A. Masegosa, F. Mascia, D. Pelta, and M. Brunato. Cooperative Strategies and Reactive Search: A Hybrid Model Proposal. In T. Stützle, editor, *Learning and Intelligent Optimization*, volume 5851 of *Lecture Notes in Computer Science*, pages 206–220. Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-11169-3_15`.

[104] A. Masegosa, D. Pelta, I. G. del Amo, and J. Verdegay. On the Performance of Homogeneous and Heterogeneous Cooperative Search Strategies. In N. Krasnogor, M. Melián-Batista, J. Pérez, J. Moreno-Vega, and D. Pelta, editors, *Nature Inspired Cooperative Strategies for Optimization (NICSO 2008)*, volume 236 of *Studies in Computational Intelligence*, pages 287–300. Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-03211-0_24`.

[105] D. C. Mattfeld and C. Bierwirth. An efficient genetic algorithm for job shop scheduling with tardiness objectives. *European Journal of Operational Research*, 155(3):616–630, 2004. `http://dx.doi.org/10.1016/S0377-2217(03)00016-X`.

[106] J. McAffer and J.-M. Lemieux. *Eclipse Rich Client Platform : designing, coding, and packaging Java applications*. Addison-Wesley, 2006.

[107] R. Mendes and A. Mohais. DynDE: a differential evolution for dynamic optimization problems. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, volume 3, pages 2808–2815, sept. 2005. `http://dx.doi.org/10.1109/CEC.2005.1555047`.

[108] E. Mezura-Montes. *Constraint-Handling in Evolutionary Optimization*, volume 198 of *Studies in Computational Intelligence*. Springer Berlin / Heidelberg, 2009. `http://dx.doi.org/10.1007/978-3-642-00619-7`.

[109] Z. Michalewicz, M. Schmidt, M. Michalewicz, and C. Chiriac. Adaptive Business Intelligence: Three Case Studies. In S. Yang, Y.-S. Ong, and

Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 179–196. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_8.

[110] R. Montemanni, L. Gambardella, A. Rizzoli, and A. Donati. A new algorithm for a Dynamic Vehicle Routing Problem based on Ant Colony System. In *Second International Workshop on Freight Transportation and Logistics*, pages 27–30, 2003.

[111] N. Mori and H. Kita. Genetic algorithms for adaptation to dynamic environments - a survey. In *26th Annual Conference of the IEEE (IECON 2000)*, volume 4, pages 2947–2952, 2000. http://dx.doi.org/10.1109/IECON.2000.972466.

[112] N. Mori, T. Kude, and K. Matsumoto. Adaptation to a dynamic environment by means of the environment identifying genetic algorithm. In *26th Annual Confjerence of the IEEE (IECON 2000)*, volume 4, pages 2953–2958, 2000. http://dx.doi.org/10.1109/IECON.2000.972467.

[113] R. Morrison and K. De Jong. A test problem generator for non-stationary environments. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation*, volume 3, pages 2047–2053, 1999. http://dx.doi.org/10.1109/CEC.1999.785526.

[114] R. W. Morrison. Performance Measurement in Dynamic Environments. In A. Barry, editor, *Proceedings of the 2003 Annual Conference on Genetic and Evolutionary Computation (GECCO-2003)*, pages 99–102, 2003.

[115] R. W. Morrison. *Designing Evolutionary Algorithms for Dynamic Environments*. Springer-Verlag, Berlin, Heidelberg, 2004.

[116] I. Moser and T. Hendtlass. A Simple and Efficient Multi-Component Algorithm for Solving Dynamic Function Optimisation Problems. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC-2007)*, pages 252–259, 2007. http://dx.doi.org/10.1109/CEC.2007.4424479.

[117] F. Neri and R. Mäkinen. Hierarchical Evolutionary Algorithms and Noise Compensation via Adaptation. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 345–369. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_15.

[118] T. T. Nguyen. *Continuous dynamic optimisation using evolutionary algorithms*. PhD thesis, University of Birmingham, 2011. http://etheses.bham.ac.uk/1296/.

[119] J. Nieto, E. Alba, and F. Chicano. Using Metaheuristic Algorithms Remotely via ROS. In *Proceedings of the 2007 Annual Conference on Genetic and Evolutionary Computation (GECCO-2007)*, 2007. http://dx.doi.org/10.1145/1276958.1277239.

[120] P. Novoa, D. A. Pelta, C. Cruz, and I. G. del Amo. Controlling Particle Trajectories in a Multi-swarm Approach for Dynamic Optimization Problems. In J. Mira, J. Ferrández, J. Álvarez, F. de la Paz, and F. Toledo, editors, *Methods and Models in Artificial and Natural Computation. A Homage to Professor Mira's Scientific Legacy*, volume 5601 of *Lecture Notes in Computer Science*, pages 285–294. Springer Berlin / Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-02264-7_30.

[121] F. Olivetti de França, F. J. Von Zuben, and L. Nunes de Castro. An Artificial Immune Network for Multimodal Function Optimization on Dynamic Environments. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, pages 289–296. ACM, 2005. http://dx.doi.org/10.1145/1068009.1068057.

[122] I. Osman and G. Laporte. Metaheuristics: A bibliography. *Annals of Operations Research*, 63:511–623, 1996. http://dx.doi.org/10.1007/BF02125421.

[123] G. Pankratz. Dynamic vehicle routing by means of a genetic algorithm. *International Journal of Physical Distribution & Logistics Management*, 35(5):362–383, 2005. http://dx.doi.org/10.1108/09600030510607346.

[124] D. Parrott and X. Li. A particle swarm model for tracking multiple peaks in a dynamic environment using speciation. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC-2004)*, volume 1, pages 98–103. IEEE, 2004. http://dx.doi.org/10.1109/CEC.2004.1330843.

[125] D. Pelta, C. Cruz, and J. R. González. A study on diversity and cooperation in a multiagent strategy for dynamic optimization problems. *International Journal of Intelligent Systems*, 24(7):844–861, 2009. http://dx.doi.org/10.1002/int.20363.

[126] D. Pelta, C. Cruz, and J. L. Verdegay. Simple control rules in a cooperative system for dynamic optimisation problems. *International Journal of Gen-*

eral Systems, 38(7):701–717, 10/2009 2009. http://dx.doi.org/10.1080/03081070802367366.

[127] D. Pelta, A. Sancho-Royo, C. Cruz, and J. L. Verdegay. Using memory and fuzzy rules in a co-operative multi-thread strategy for optimization. *Information Sciences*, 176(13):1849–1868, 2006. http://dx.doi.org/10.1016/j.ins.2005.06.007.

[128] B. Peng and R. Reynolds. Cultural algorithms: Knowledge learning in dynamic environments. In *Proceedings of the 2004 IEEE Congress on Evolutionary Computation (CEC-2004)*, pages 1751–1758, 2004. http://dx.doi.org/10.1109/CEC.2004.1331107.

[129] F. Quintão, F. Nakamura, and G. Mateus. Evolutionary Algorithms for Combinatorial Problems in the Uncertain Environment of the Wireless Sensor Networks. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 197–222. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_9.

[130] W. Rand and R. Riolo. Shaky Ladders, Hyperplane-Defined Functions and Genetic Algorithms: Systematic Controlled Observation in Dynamic Environments. In *Applications on Evolutionary Computing*, volume 3449 of *Lecture Notes in Computer Science*, pages 600–609. Springer Berlin / Heidelberg, 2005. http://dx.doi.org/10.1007/978-3-540-32003-6_63.

[131] W. Rand and R. Riolo. The Effect of Building Block Construction on the Behavior of the GA in Dynamic Environments: A Case Study Using the Shaky Ladder Hyperplane-Defined Functions. In F. Rothlauf, J. Branke, S. Cagnoni, E. Costa, C. Cotta, R. Drechsler, E. Lutton, P. Machado, J. Moore, J. Romero, G. Smith, G. Squillero, and H. Takagi, editors, *Applications of Evolutionary Computing*, volume 3907 of *Lecture Notes in Computer Science*, pages 776–787. Springer Berlin / Heidelberg, 2006. http://dx.doi.org/10.1007/11732242_75.

[132] R. Randles and D. Wolfe. *Introduction to the Theory of Nonparametric Statistics*. John Wiley & Sons, Inc., 1979.

[133] H. Richter. A study of dynamic severity in chaotic fitness landscapes. In *Proceedings of the 2005 IEEE Congress on Evolutionary Computation (CEC-2005)*, volume 3, pages 2824–2831, 2005. http://dx.doi.org/10.1109/CEC.2005.1555049.

[134] H. Richter and S. Yang. Learning behavior in abstract memory schemes for dynamic optimization problems. *Soft Computing - A Fusion of Foundations, Methodologies and Applications*, 13(12):1163–1173, 2009. http://dx.doi.org/10.1007/s00500-009-0420-6.

[135] C. Rocco and D. Salazar. A Hybrid Approach Based on Evolutionary Strategies and Interval Arithmetic to Perform Robust Designs. In *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 543–564. Springer, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_24.

[136] P. Rohlfshagen, P. K. Lehre, and X. Yao. Dynamic evolutionary optimisation: an analysis of frequency and magnitude of change. In *Proceedings of the 2009 Annual Conference on Genetic and Evolutionary Computation (GECCO-2009)*, pages 1713–1720, New York, NY, USA, 2009. ACM. http://dx.doi.org/10.1145/1569901.1570131.

[137] P. Rohlfshagen and X. Yao. The Dynamic Knapsack Problem Revisited: A New Benchmark Problem for Dynamic Combinatorial Optimisation. In M. Giacobini, A. Brabazon, S. Cagnoni, G. Di Caro, A. Ekárt, A. Esparcia-Alcázar, M. Farooq, A. Fink, and P. Machado, editors, *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science*, pages 745–754. Springer Berlin / Heidelberg, 2009. http://dx.doi.org/10.1007/978-3-642-01129-0_84.

[138] C. Ronnewinkel and T. Martinetz. Explicit Speciation with few a priori Parameters for Dynamic Optimization Problems. In *Proceedings of the 2001 Annual Conference on Genetic and Evolutionary Computation (GECCO-2001)*, pages 31–34. Morgan Kaufmann, 2001.

[139] C. Rossi, M. Abderrahim, and J. César Díaz. Tracking Moving Optima Using Kalman-Based Predictions. *Evolutionary Computation*, 16(1):1–30, 2008. http://dx.doi.org/10.1162/evco.2008.16.1.1.

[140] S. Saleem and R. Reynolds. Cultural Algorithms in Dynamic Environments. In *Proceedings of the 2000 Congress on Evolutionary Computation (CEC-2000)*, volume 2, pages 1513–1520, 2000. http://dx.doi.org/10.1109/CEC.2000.870833.

[141] L. Schönemann. The impact of population sizes and diversity on the adaptability of evolution strategies in dynamic environments. In *Proceedings of the 2004 Congress on Evolutionary Computation (CEC-2004)*, volume 2, pages 1270–1277, 2004. http://dx.doi.org/10.1109/CEC.2004.1331043.

[142] L. Schönemann. Evolution Strategies in Dynamic Environments. In *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 51–77. Springer, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_3`.

[143] Y. Shi and R. Eberhart. Fuzzy adaptive particle swarm optimization. In *Proceedings of the 2001 IEEE Conference on Evolutionary Computation (CEC-2001)*, volume 1, pages 101–106, 2001. `http://dx.doi.org/10.1109/CEC.2001.934377`.

[144] A. Simões and E. Costa. An immune system-based genetic algorithm to deal with dynamic environments: Diversity and memory. In D. W. Pearson, N. C. Steele, and R. Albrecht, editors, *Proceedings of the Sixth international conference on neural networks and genetic algorithms (ICANNGA03)*, pages 168–174. Springer, 2003.

[145] R. Smierzchalski and Z. Michalewicz. Modeling of Ship Trajectory in Collision Situations by an Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 4(3):227–241, 2000. `http://dx.doi.org/10.1109/4235.873234`.

[146] J. Smith. Self-Adaptation in Evolutionary Algorithms for Combinatorial Optimisation. In C. Cotta, M. Sevaux, and K. Sörensen, editors, *Adaptive and Multilevel Metaheuristics*, volume 136 of *Studies in Computational Intelligence*, pages 31–57. Springer Berlin / Heidelberg, 2008. `http://dx.doi.org/10.1007/978-3-540-79438-7_2`.

[147] S. Stanhope and J. Daida. (1+1) Genetic Algorithm Fitness Dynamics in a Changing Environment. In *Proceedings of the 1999 Congress on Evolutionary Computation (CEC-1999)* , volume 3, pages 1851–1858, 1999. `http://dx.doi.org/10.1109/CEC.1999.785499`.

[148] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework*. Addison-Wesley Professional, 2008.

[149] Y. Tenne and S. Armfield. A Memetic Algorithm Using a Trust-Region Derivative-Free Optimization with Quadratic Modelling for Optimization of Expensive and Noisy Black-box Functions. In *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 389–415. Springer, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_17`.

[150] M. Tezuka, M. Munetomo, and K. Akama. Genetic Algorithm to Optimize Fitness Function with Sampling Error and its Application to Financial Optimization Problem. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 417–434. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_18`.

[151] R. Tinós and S. Yang. A self-organizing random immigrants genetic algorithm for dynamic optimization problems. *Genetic Programming and Evolvable Machines*, 8(3):255–286, 2007. `http://dx.doi.org/10.1007/s10710-007-9024-z`.

[152] R. Tinos and S. Yang. Continuous dynamic problem generators for evolutionary algorithms. In *Proceedings of the 2007 IEEE Congress on Evolutionary Computation (CEC-2007)*, pages 236–243, 2007. `http://dx.doi.org/10.1109/CEC.2007.4424477`.

[153] R. Tinós and S. Yang. Genetic Algorithms with Self-Organizing Behaviour in Dynamic Environments. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 105–127. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_5`.

[154] R. Tinos and S. Yang. Evolutionary programming with q-Gaussian mutation for dynamic optimization problems. In *Proceedings of the 2008 IEEE Congress on Evolutionary Computation (CEC-2008)*, pages 1823–1830, 2008. `http://dx.doi.org/10.1109/CEC.2008.4631036`.

[155] K. Trojanowski and S. T. Wierzchon. Immune-based algorithms for dynamic optimization. *Information Sciences*, 179(10):1495–1515, 2009. `http://dx.doi.org/10.1016/j.ins.2008.11.014`.

[156] K. Tumer and A. Agogino. Evolving Multi Rover Systems in Dynamic and Noisy Environments. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 371–387. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_16`.

[157] R. K. Ursem. Multinational GAs: Multimodal Optimization Techniques in Dynamic Environments. In *Proceedings of the 2000 Annual Conference on Genetic and Evolutionary Computation (GECCO-2000)*, pages 19–26. Morgan Kaufmann, 2000.

[158] R. K. Ursem, T. Krink, M. Jensen, and Z. Michalewicz. Analysis and modeling of control tasks in dynamic systems. *IEEE Transactions on Evolutionary Computation*, 6(4):378–389, 2002. `http://dx.doi.org/10.1109/TEVC.2002.802871`.

[159] A. Vargha and H. D. Delaney. A Critique and Improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Journal of Educational and Behavioral Statistics*, 25(2):101–132, 2000.

[160] G. Venayagamoorthy. Adaptive critics for dynamic particle swarm optimization. In *Proceedings of the 2004 IEEE International Symposium on Intelligent Control*, 2004. `http://dx.doi.org/10.1109/ISIC.2004.1387713`.

[161] J. L. Verdegay, R. R. Yager, and P. P. Bonissone. On heuristics as a fundamental constituent of soft computing. *Fuzzy Sets and Systems*, 159(7):846–855, 2008. `http://dx.doi.org/10.1016/j.fss.2007.08.014`.

[162] S. Voß. *Meta-Heuristics: Advances and Trends in Local Search Paradigms for Optimization.* Kluwer Academic Publishers, 1999.

[163] H. Wang, D. Wang, and S. Yang. A memetic algorithm with adaptive hill climbing strategy for dynamic optimization problems. *Soft Computing*, 13(8-9):763–780, 2009. `http://dx.doi.org/10.1007/s00500-008-0347-3`.

[164] H. Wang, S. Yang, W. Ip, and D. Wang. Adaptive Primal-Dual Genetic Algorithms in Dynamic Environments. *IEEE Transactions on Systems, Man, and Cybernetics, Part B: Cybernetics*, 39(6):1348–1361, 2009. `http://dx.doi.org/10.1109/TSMCB.2009.2015281`.

[165] K. Weicker. Performance Measures for Dynamic Environments. In J. Guervós, P. Adamidis, H.-G. Beyer, H.-P. Schwefel, and J.-L. Fernández-Villacañas, editors, *Proceedings of the VII International Conference on Parallel Problem Solving from Nature (PPSN-2002)*, volume 2439 of *Lecture Notes in Computer Science*, pages 64–73. Springer Berlin / Heidelberg, 2002. `http://dx.doi.org/10.1007/3-540-45712-7_7`.

[166] K. Weicker. *Evolutionary Algorithms and Dynamic Optimization Problems.* Der Andere Verlag, 2003.

[167] K. Weicker and N. Weicker. On evolution strategy optimization in dynamic environments. In *Proceedings of the 1999 IEEE Congress on Evolutionary Computation (CEC-1999)*, pages 2039–2046, 1999. `http://dx.doi.org/10.1109/CEC.1999.785525`.

[168] F. Wilcoxon. Individual Comparisons by Ranking Methods. *Biometrics Bulletin*, 1(6):pp. 80–83, 1945. `http://dx.doi.org/10.2307/3001968`.

[169] M. Wineberg and F. Oppacher. Enhancing the GA's Ability to Cope with Dynamic Environments. In *Proceedings of the 2000 Annual Conference on Genetic and Evolutionary Computation (GECCO-2000)*, pages 3–10. Morgan Kaufmann, 2000.

[170] Y. G. Woldesenbet and G. G. Yen. Dynamic Evolutionary Algorithm With Variable Relocation. *IEEE Transactions on Evolutionary Computation*, 13(3):500–513, 2009. `http://dx.doi.org/10.1109/TEVC.2008.2009031`.

[171] D. Wolpert and W. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, apr. 1997. `http://dx.doi.org/10.1109/4235.585893`.

[172] X.-S. Yan, L.-S. Kang, Z.-H. Cai, and H. Li. An approach to dynamic traveling salesman problem. In *International Conference on Machine Learning and Cybernetics*, 2004.

[173] S. Yang. Non-stationary problem optimization using the primal-dual genetic algorithm. In *Proceedings of the 2003 IEEE Congress on Evolutionary Computation (CEC-2003)*, volume 3, pages 2246–2253. IEEE Press, 2003. `http://dx.doi.org/10.1109/CEC.2003.1299951`.

[174] S. Yang. Memory-based immigrants for genetic algorithms in dynamic environments. In *Proceedings of the 2005 Annual Conference on Genetic and Evolutionary Computation (GECCO-2005)*, pages 1115–1122. ACM, 2005. `http://dx.doi.org/10.1145/1068009.1068196`.

[175] S. Yang. A comparative study of immune system based genetic algorithms in dynamic environments. In *Proceedings of the 2006 Annual Conference on Genetic and Evolutionary Computation (GECCO-2006)*, pages 1377–1384. ACM, 2006. `http://dx.doi.org/10.1145/1143997.1144209`.

[176] S. Yang. Associative Memory Scheme for Genetic Algorithms in Dynamic Environments. In *Applications of Evolutionary Computing*, volume 3907 of *Lecture Notes in Computer Science*, pages 788–799. Springer Berlin / Heidelberg, Springer Berlin / Heidelberg, 2006. `http://dx.doi.org/10.1007/11732242_76`.

[177] S. Yang. Explicit Memory Schemes for Evolutionary Algorithms in Dynamic Environments. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in*

Computational Intelligence, pages 3–28. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5_1.

[178] S. Yang. Genetic Algorithms with Memory- and Elitism-Based Immigrants in Dynamic Environments. *Evolutionary Computation*, 16(3):385–416, 2008. http://dx.doi.org/10.1162/evco.2008.16.3.385.

[179] S. Yang, H. Cheng, and F. Wang. Genetic Algorithms With Immigrants and Memory Schemes for Dynamic Shortest Path Routing Problems in Mobile Ad Hoc Networks. *IEEE Transactions on Systems, Man, and Cybernetics, Part C: Applications and Reviews*, 40(99):52–63, 2010. http://dx.doi.org/10.1109/TSMCC.2009.2023676.

[180] S. Yang and C. Li. A Clustering Particle Swarm Optimizer for Locating and Tracking Multiple Optima in Dynamic Environments. *IEEE Transactions on Evolutionary Computation*, 14(6):959–974, dec. 2010. http://dx.doi.org/10.1109/TEVC.2010.2046667.

[181] S. Yang, Y.-S. Ong, and Y. Jin. Editorial to special issue on evolutionary computation in dynamic and uncertain environments. *Genetic Programming and Evolvable Machines*, 7(4):293–294, 2006. http://dx.doi.org/10.1007/s10710-006-9016-4.

[182] S. Yang, Y.-S. Ong, and Y. Jin, editors. *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*. Springer Berlin / Heidelberg, 2007. http://dx.doi.org/10.1007/978-3-540-49774-5.

[183] S. Yang and R. Tinós. A Hybrid Immigrants Scheme for Genetic Algorithms in Dynamic Environments. *International Journal of Automation and Computing*, 4(3):243–254, 2007. http://dx.doi.org/10.1007/s11633-007-0243-9.

[184] S. Yang and X. Yao. Experimental study on population-based incremental learning algorithms for dynamic optimization problems. *Soft Computing*, 9(11):815–834, 2005. http://dx.doi.org/10.1007/s00500-004-0422-3.

[185] S. Yang and X. Yao. Population-Based Incremental Learning With Associative Memory for Dynamic Environments. *IEEE Transactions on Evolutionary Computation*, 12(5):542–561, 2008. http://dx.doi.org/10.1109/TEVC.2007.913070.

[186] G. Yen, F. Yang, T. Hickey, and M. Goldstein. Coordination of exploration and exploitation in a dynamic environment. In *Proceedings of the*

2001 International Joint Conference on Neural Networks (IJCNN-2001), volume 2, pages 1014–1018, 2001. `http://dx.doi.org/10.1109/IJCNN.2001.939499`.

[187] Ö. Yeniay. Penalty function methods for constrained optimization with genetic algorithms. *Mathematical and Computational Applications*, 10:45–56, 2005.

[188] L. Zadeh. Soft computing and fuzzy logic. *Software, IEEE*, 11(6):48–56, nov. 1994. `http://dx.doi.org/10.1109/52.329401`.

[189] L. A. Zadeh. Fuzzy Sets. *Information and Control*, 8:338–353, 1965. `http://dx.doi.org/10.1016/S0019-9958(65)90241-X`.

[190] L. A. Zadeh. Applied Soft Computing – Foreword. *Applied Soft Computing*, 1(1):1–2, 2001. `http://dx.doi.org/10.1016/S1568-4946(01)00003-5`.

[191] A. Zaslavski, I. G. del Amo, F. G. López, M. G. Torres, B. M. Batista, J. A. M. Pérez, and J. M. Moreno-Vega. From Theory to Implementation: Applying Metaheuristics. In P. Pardalos, L. Liberti, and N. Maculan, editors, *Global Optimization*, volume 84 of *Nonconvex Optimization and Its Applications*, pages 311–351. Springer US, 2006. `http://dx.doi.org/10.1007/0-387-30528-9_11`.

[192] S. Zeng, H. Shi, L. Kang, and L. Ding. Orthogonal Dynamic Hill Climbing Algorithm: ODHC. In S. Yang, Y.-S. Ong, and Y. Jin, editors, *Evolutionary Computation in Dynamic and Uncertain Environments*, volume 51 of *Studies in Computational Intelligence*, pages 79–104. Springer Berlin / Heidelberg, 2007. `http://dx.doi.org/10.1007/978-3-540-49774-5_4`.

[193] X. Zou, M. Wang, A. Zhou, and B. Mckay. Evolutionary optimization based on chaotic sequence in dynamic environments. In *Proceedings of the 2004 IEEE International Conference on Networking, Sensing and Control*, pages 1364 – 1369, 2004. `http://dx.doi.org/10.1109/ICNSC.2004.1297146`.