Un Framework para Big Data Optimization Basado en jMetal y Spark

Cristóbal Barba-González[1], Antonio J. Nebro[1], José García-Nieto[1], José A. Cordero[2], Juan J. Durillo[3], Ismael Navas-Delgado[1], and José F. Aldana-Montes[1]

[1] Grupo de investigación Khaos
Edificio de investigación Ada Byron
Departamento de Lenguajes y Ciencias de la Computación
Universidad de Málaga, España.
[2] Organización Europea para la Investigación Nuclear (CERN), Suiza
[3] Distributed and Parallel Systems Group
Universidad de Innsbruck, Austria

Resumen Las metaheurísticas multi-objetivo se han convertido en técnicas muy utilizadas para la resolución problemas complejos de optimización compuestos de varias funciones objetivo en conflicto entre sí. Nos encontramos en la actualidad inmersos en la era del Big Data, por lo que los problemas multi-objetivo que surjan en este contexto cumplirán algunas de las cinco V's que caracterizan a las aplicaciones Big Data (volumen, velocidad, variedad, veracidad, valor). Como consecuencia, las metaheurísticas deberán ser capaces de resolver problemas dinámicos, que pueden cambiar en el tiempo debido al procesamiento y análisis de diferentes fuentes de datos, que típicamente serán en *streaming*. En este trabajo presentamos el software jMetalSP, que combina el *framework* jMetal con Apache Spark. De esta forma, las metaheurísticas disponibles en jMetal se pueden adaptar fácilmente para resolver problemas dinámicos que se alimenten de distintas fuentes de datos en *streaming*, y que son gestionadas por Spark. Se describe la arquitectura de jMetalSP y se valida mediante un caso de uso realista basado en TSP bi-objetivo con datos abiertos reales de tráfico de la ciudad de Nueva York.

Key words: Big Data, Optimización Multi-objetivo, Problemas Dinámicos, Metaheurísticas, Framework Software, jMetal, Apache Spark

1. Introducción

Estamos inmersos en la era del Big Data, donde las aplicaciones deben gestionar y analizar ingentes cantidades de datos que no pueden ser procesados mediante el uso de tecnologías tradicionales de bases de datos. El volumen de datos no es la única característica de las aplicaciones Big Data, ya que tienen que procesar además fuentes de datos heterogéneas (la mayoría de ellas en *streaming*), que generalmente generan datos a gran velocidad. Estos datos también deben ser validados y analizados para producir un valor significativo para el

usuario final [1]. Estas características constituyen las denominadas V's del Big Data: volumen, velocidad, variedad, variabilidad, veracidad y valor [2].

En este contexto, los algoritmos evolutivos y metaheurísticas en general van a tener un papel importante en la resolución de lo que denominan problemas de *Big Data Optimization*. Si nos centramos en los problemas multi-objetivo, es decir, los que tienen dos o más funciones contradictorias entre sí, éstos se encuentran en muchas disciplinas, tales como el transporte, la economía, la medicina, la bioinformática, etc., por lo que se prevé que aparezcan variantes Big Data de estos problemas en un futuro inmediato. En este sentido, va a constituir un reto el adaptar metaheurísticas actuales para hacer frente a estos nuevos tipos de problemas [3]. Un aspecto importante será disponer de herramientas potentes que permitan aprovechar la gran cantidad de investigación en optimización multi-objetivo realizada en los últimos 15 años, con el fin de poder utilizar algoritmos clásicos y modernos en combinación con algunas plataformas de gestión de grandes volúmenes de datos.

Otra cuestión importante es que muchos problemas multi-objetivo del mundo real tienen objetivos, restricciones y parámetros que pueden cambiar a lo largo del tiempo. Estos problemas se conocen como problemas de optimización multi-objetivo dinámicos [4]. De hecho, estos se relacionan de manera natural con el Big Data ya que los cambios en los problemas se pueden deber a datos recibidos en *streaming* a partir de diferentes fuentes de datos.

Nuestra motivación en este artículo es la de presentar jMetalSP, una herramienta software para problemas multi-objetivo dinámicos de optimización en Big Data, que combina el *framework* jMetal para optimización multi-objetivo con metaheurísticas [5], con el sistema de *cluster computing* para Big Data Apache Spark [6]. A la hora de diseñar el *framework* jMetalSP se han tenido en cuenta los siguientes requisitos:

- La herramienta debe permitir definir y resolver problemas dinámicos de *Big Data Optimization* sobre clusters Hadoop/Spark.
- Los algoritmos disponibles en jMetal deben ser fácilmente adaptables para resolver problemas multi-objetivo dinámicos.
- Las aplicaciones desarrolladas con jMetalSP deben poder incorporar diferentes fuentes de datos, usando para ello los servicios que ofrece Spark.
- El *framework* debe ser fácil de usar, intentando ocultar en lo posible detalles de bajo de nivel y simplificando la incorporación de las fuentes de datos.
- jMetalSP será un proyecto *Open Source*, de forma que esté libremente disponible para cualquier investigador interesado[4]. Los comentarios de los usuarios permitirán mejorarlo y hacer que evolucione.

Para validar el funcionamiento de jMetalSP hemos definido una versión dinámica y bi-objetivo del problema del viajante de comercio (TSP) [7], que incluye una mezcla de datos reales de tráfico de la ciudad de Nueva York, con datos simulados tomados mediante la API de Twitter [5] y Apache Kafka [6].

[4] Estará disponible en GitHub en esta URL: XX
[5] Available from URL `https://dev.twitter.com/overview/api`
[6] Available at URL `http://kafka.apache.org/`

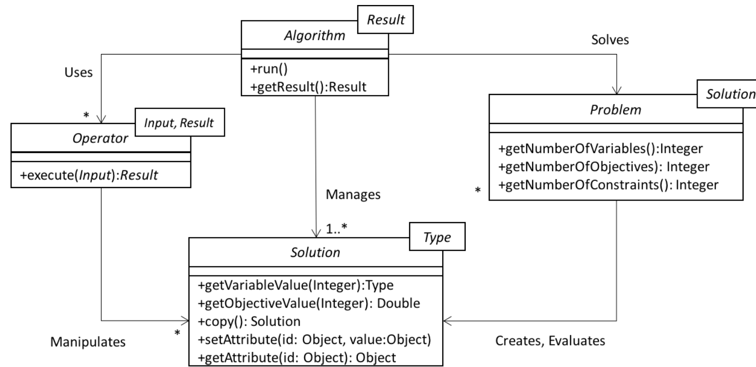

Figura 1. Architectura de jMetal 5 (diagrama UML de classes).

El resto del artículo está organizado de la siguiente manera. En la sección 2 se describen jMetal y Spark, que son los dos componentes de jMetalSP. La arquitectura del *framework* se presenta en la sección 3. La sección 4 describe el caso de estudio. Finalmente, la Sección 5 expone las principales conclusiones y las líneas de trabajo futuro.

2. Componentes software

jMetalSP es una herramienta software que incluye dos componentes. Por un lado, el *framework* de optimización multi-objetivo jMetal, que proporcionará la infraestructura de optimización para implementar tanto los problemas de Big Data Optmization, como los algoritmos dinámicos para resolverlos; por otro lado, el sistema de computación distribuida Spark, que permitirá gestionar las fuentes de datos en *streaming* y sacar partido a la potencia de cálculo del *cluster* Hadoop utilizado. Además permitirá almacenar y recuperar datos desde HDFS.

2.1. jMetal

jMetal es un *framework Open Source* basado en Java para la optimización multi-objetivo con metaheurísticas [5]. La arquitectura de jMetal 5, la versión en la que se basa jMetalSP, se muestra en la Figura 1. Podemos observar que hay cuatro interfaces principales, que modelan la idea de que un algoritmo (una metaheurística) resuelve un problema mediante una serie de soluciones que se manipulan con un conjunto de operadores, y los problemas son las entidades responsables de crear y evaluar soluciones.

La arquitectura básica es lo suficientemente genérica como para permitir la flexibilidad necesaria para implementar cualquier metaheurística. Sin embargo, la mayoría de los algoritmos pertenecen a las subfamilias bien establecidas, tales como los algoritmos evolutivos (EAs), la optimización mediante enjambre de

partículas (PSO), la búsqueda dispersa (SS) y muchas otras. Estas subfamilias se caracterizan por un comportamiento común que es compartido por todos los algoritmos que pertenecen a ellas. jMetal 5 ofrece una serie de plantillas que incluyen el comportamiento las subfamilias concretas de algoritmos, por lo que el desarrollo de un algoritmo particular sólo requiere implementar algunos métodos concretos. Un ejemplo de plantilla es la clase `AbstractEvolutionryAlgorithm`, que incluye la siguiente implementación del método `run()`, que reproduce el código de un algoritmo evolutivo típico:

```
1. @Override public void run() {
2.  List<S> offspringPopulation;
3.  List<S> matingPopulation;
4.  population = createInitialPopulation();
5.  population = evaluatePopulation(population);
6.  initProgress();
7.  while (!isStoppingConditionReached()) {
8.    matingPopulation = selection(population);
9.    offspringPopulation = reproduction(matingPopulation);
10.   offspringPopulation = evaluatePopulation(offspringPopulation);
11.   population = replacement(population, offspringPopulation);
12.   updateProgress();}
13. }
```

Algunos algoritmos evolutivos multi-objetivo muy conocidos, como NSGA-II [8], SPEA2 [9] o SMS-EMOA [10] se basan en esta plantilla. De hecho, el uso de plantillas facilita la reusabilidad de código (por ejemplo, para implementar una variante un algoritmo determinado sólo hay que redefinir los métodos que difieren del algoritmo original) y es una característica que es explotada en jMetalSP.

2.2. Apache Spark

Apache Spark es un sistema de computación distribuido de propósito general [6] basado en el concepto de conjuntos de datos distribuidos *resilientes* (RDDs). Los RDDs son colecciones de elementos sobre los que se puede operar en paralelo en los nodos de un clúster, mediante el uso de dos tipos de operaciones: transformaciones (map, filter, union, etc.) y acciones (reduce, collect, count, etc.). Las características principales de Spark son: modelo de programación paralela de alto nivel, algoritmos de aprendizaje automático, procesamiento de grafos, API de programación multi-lenguaje, se ejecuta en diferentes sistemas (Hadoop, Mesos, Standalone, Cluster) y procesamiento de datos en*streaming*. Spark se está creciendo en popularidad y ya está reemplazando a MapReduce (MR) como la tecnología dominante para desarrollar aplicaciones Big Data.

Como jMetalSP está orientado a Big Data Optimization, la parte que más interesa de Spark es su capacidad de procesamiento en paralelo de diferentes fuentes de datos en *streaming*, como: Kafka, Flumme, Twitter, sockets TCP, archivos, etc.

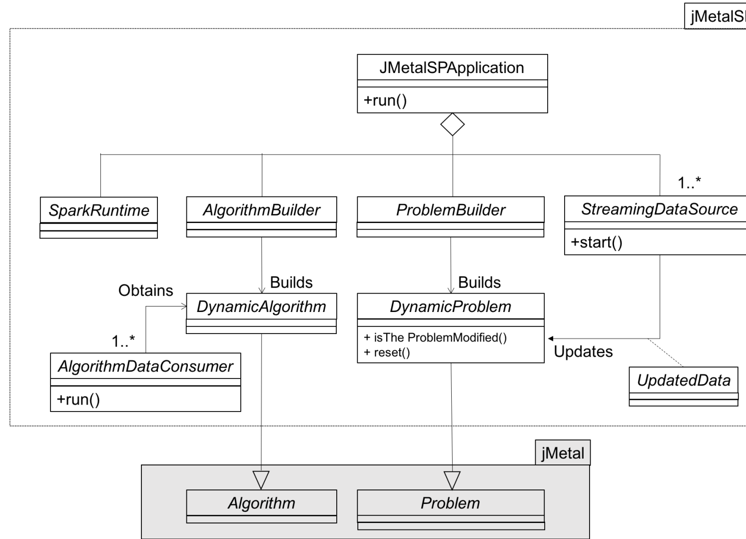

Figura 2. Arquitectura de jMetalSP

3. Arquitectura de jMetalSP

La architectura of jMetalSP se muestra en la Figura 2. Un aplicación jMetalSP tiene como fin resolver un problema dinámico de optimización utilizando un algoritmo dinámico mediante el análisis de una o más fuentes de datos en *streaming*, existiendo uno o varios consumidores de datos que irá generando el algoritmo mientras se ejecuta. Como el problema se puede inicializar de varias maneras (con valores por defecto, a partir de datos externos, etc.) se utiliza un objeto que implementa el patrón *Builder*. de forma similar, existe un *builder* para configurar y crear cada algoritmo.

La clase `SparkRuntime` se usa para configurar la componente Spark de la aplicación. El parámetro más importante es el intervalo de procesamiento (*batch interval*). Como las fuentes de datos modificarán el problema en tiempo de ejecución, se debe definir un formato común que consiste en definir una subclase de la interfaz `UpdatedData`. Siempre que se modifica un problema, si se invoca sobre él el método `isTheProblemModified()`, éste devolverá el valor verdadero hasta que se invoque el método `reset()`.

La clase `jMetalSPApplication` proporciona métodos para indicar qué *builders* del problema y algoritmo se van a usar, el Spark *runtime*, los consumidores de datos y las diferentes fuentes de datos. El método `run()` incorpora toda la lógica para crear el problema, el algoritmo, inicializar Spark, ejecutar los consumidores de datos y arrancar los componentes de procesamiento de los *streams*. Configurar y ejecutar una aplicación jMetalSP consiste en crear e inicializar un objeto de `jMetalSPApplication`, tal como se muestra en el siguiente ejemplo:

```
1. SparkSPApplicaton application = new SparkSPApplication() ;
2. application
3.   .setSparkRuntime(new SparkRuntime(2))
4.   .setProblemBuilder(new SpecificProblemBuilder())
5.   .setAlgorithmBuilder(new SpecificAlgorithmBuilder())
6.   .addAlgorithmDataConsumer(new FirstDataConsumer())
7.   .addAlgorithmDataConsumer(new SecondDataConsumer())
8.   .addStreamingDataSource(new FirstStreamingSource())
7.   .addStreamingDataSource(new SecondStreamingSource())
8.   .addStreamingDataSource(new ThirdStreamingSource())
9.   .run()
```

Cabe indicar que este código contiene algunas simplificaciones con el fin de hacerlo más fácil de entender. El código real incluye genéricos de *Java* para garantizar en tiempo de compilación que todos los componentes son compatibles.

4. Caso de Estudio: TSP dinámico con dos objetivos y NSGA-II dinámico

Para evaluar la arquitectura propuesta se ha definido un caso de estudio que mezcla un problema académico con datos reales. El problema dinámico seleccionado es el del viajante de comercio (Traveling Salesman Problem, TSP) con dos objetivos. Para resolverlo se usa una variante dinámica de NSGA-II. Los objetivos a minimizar son el *tiempo* y la *distancia* totales en recorrer todos los puntos de la instancia. Con el desarrollo de este caso de uso se busca un doble fin, primero mostrar cómo se utiliza jMetalSP para implementar las versiones dinámicas, tanto del problema TSP como de la metaheurística NSGA-II; segundo, realizar pruebas para comprobar el rendimiento de la aplicación en un clúster Hadoop/Spark.

4.1. NSGA-II dinámico

La estrategia llevada a cabo para implementar una versión dinámica de NSGA-II usando la versión estática ya proporcionada por jMetal 5 se puede dividir en dos grandes pasos. En primer lugar, se buscan los métodos que necesitan ser modificados para permitir el comportamiento dinámico. En segundo lugar, se decide la configuración del algoritmo.

Con respecto al primer paso, se requiere modificar únicamente dos métodos:

- `isStoppingConditionReached()`: En el algoritmo estándar de NSGA-II, cuando se alcanza el número máximo de evaluaciones el algoritmo termina. Sin embargo, en la versión dinámica, se utilizan notificaciones para indicar que un nuevo frente de Pareto ha sido calculado. Después de esto, el algoritmo se reinicia y comienza otra ejecución.
- `updateProgress()`: Este método se utiliza para aumentar el número de soluciones evaluadas en la versión estática de NSGA-II. En la versión dinámica, se modifica para comprobar también si el problema dinámico ha cambiado y, si es así, se invoca el método `reset()` en el problema.

Figura 3. Instancia del problema real TSP dinámico: 93 nodos (localizaciones geoposicionadas de calles) de la ciudad de Nueva York.

La clase `DynamicNSGAII` resultante hereda de la clase original de NSGAII y sobreescribe estos dos métodos. Este enfoque de desarrollo se puede utilizar fácilmente con la mayoría de las metaheurísticas incluidas en jMetal 5.

4.2. TSP dinámico con dos objetivos

Un problema dinámico en jMetalSP es un problema de jMetal 5 que implementa la interfaz `DynamicProblem`, por lo que incluye el método `update()`, que será el que dará lugar a algún cambio en el problema. Dado que este método será llamado por un objeto `StreamingDataSource`, el cambio resultante puede detectarse mediante otra clase (llamando a `isTheProblemModified ()`). Esto obliga el etiquetado como *synchronized* a todos los métodos del problema (incluyendo `reset()`). La clase `DynamicMultiobjectiveTSP` incluye los métodos antes descritos. Las variables de estado del problema son las matrices de distancia/coste.

La instancia de TSP dinámico que se utiliza en este trabajo se fundamenta datos reales. Más concretamente, se han usado los datos abiertos proporcionados por el Departamento de Transporte de la ciudad de Nueva York, el cual actualiza la información del tráfico varias veces por minuto[7]. La información se proporciona mediante ficheros de texto donde cada línea indica, entre otros datos, la velocidad media en pasar entre dos puntos de un intervalo.

Después de procesar los enlaces proporcionados por el servicio de tráfico, el TSP dinámico resultante se compone de 93 ubicaciones y 315 caminos directos

[7] En la URL: `http://207.251.86.229/nyc-links-cams/LinkSpeedQuery.txt`

entre ellas, representadas en la Figura 3. Hay que señalar que los enlaces son bidireccionales, por lo que el TSP resultante es asimétrico. Los datos del problema son inicializados desde fichero por un objeto que implementa el interfaz `AlgorithmBuilder`.

4.3. Fuentes de datos en *Streaming*

Para este trabajo, se han implementado tres clases de fuentes de datos en *streaming*: `StreamingDirectoryTSP`, `StreamingKafka` y `StreamingTwitter`. Los datos de tráfico de Nueva York se leen periódicamente por una aplicación externa que escribe en un directorio un archivo cada vez que se adquieren nuevos datos. La clase `StreamingDirectoryTSP` procesa los ficheros que se están agregando a ese directorio. Esta clase debe implementar el interfaz `StreamingDataSource` (ver Figura 1), que define el método `start()`.

El aspecto más importante a tener en cuenta desde el punto de vista del rendimiento es que el tratamiento de los datos de entrada en *streaming* se realiza en paralelo, por lo que es en esta parte donde las características del clúster de Spark serán más útiles. En nuestra clase `StreamingDirectoryTSP`, este procesamiento es muy simple, ya que el proceso externo actualiza el directorio dos o tres veces por minuto, por lo que no hay beneficios del uso del paralelismo. Por este motivo, hemos incluido otras dos fuentes de datos de secuencias (en las dos clases denominadas `StreamingTwitterTSP` y `StreamingKafkaTSP`) para profundizar en este tema.

La clase `StreamingTwitterTSP` lee tweets de Twitter con el *topic New York Traffic* y el procesamiento de cada *tweet* es simulado, es decir, el problema se actualiza con un valor aleatorio. De esta manera se combina una fuente de flujo diferente con la posibilidad de ajustar el tiempo de procesamiento.

Por último, la clase `StreamingKafkaTSP` está destinada, como la anterior, a enriquecer el caso de estudio con otra fuente de datos. Para este caso, hemos creado un productor de mensajes Kafka que genera, siguiendo distribuciones uniformes y normales, una serie de mensajes aleatorios con datos para actualizar el problema. Cada cinco segundos se producen al menos 1.000 mensajes, aunque en promedio se crean alrededor de 10.000 mensajes. `StreamingKafkaTSP` lee los mensajes creados por este productor y actualiza el problema cada segundo.

4.4. Experimentos

Para la evaluación de la arquitectura se han realizado tres experimentos. Para ello se ha usado un sistema virtualizado con VMWare con 10 máquinas esclavas, cada una con 10 cores a 2.7 Ghz, 10 GB RAM y 100 GB de disco duro. Además se dispone de una maquina física usada como maestra con 8 procesadores i7 a 3.40 GHz, 32 GB RAM y 3 TB de disco duro. La diferencia entre cada uno de los test realizados es la cantidad y forma en el que se crean los datos. Se disponen de tres fuentes de datos, mensajes Kafka, ficheros con datos reales de tráfico de Nueva York y *tweets* sobre el tráfico en Nueva York. Para aumentar el número de datos usado en el problema y comprobar la eficiencia de la lectura en *streaming*,

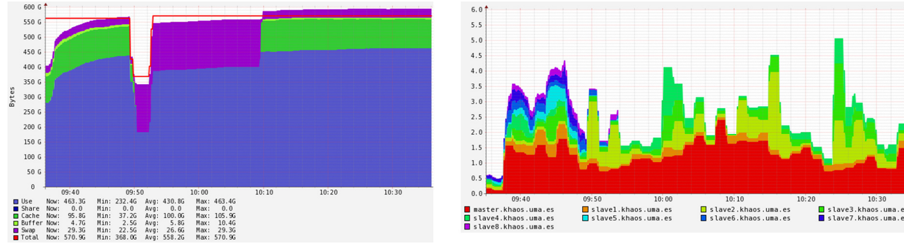

Figura 4. Experimento 3. Uso de memoria del sistema (izquierda) y número de thread/minuto creados en cada máquina (derecha)

se usa el productor de mensajes de Kafka para variar en la cantidad de datos que se deben procesar en el sistema.

Para los experimentos 1 y 2 se producen los mensajes de Kafka siguiendo una distribución aleatoria uniforme llegando a crear cada 5 segundos una media de 50.000 mensajes (140 GBs de datos leídos durante la prueba) para el experimento 1 y de 500.000 (1.37 TBs de datos leídos durante la prueba) para el experimento 2. Este segundo experimento lleva a un colapso del sistema debido a que no es capaz de leer todos los datos generados. Por otro lado, para el experimento 3 se producen también 500.000 mensajes cada 5 segundos (1.37 TBs de datos leídos durante la prueba) pero siguiendo una distribución normal, lo que favorece a una mejor distribución de la lectura de los datos en *streaming*, evitando así el colapso del sistema.

En la Figura 4 se puede observar la cantidad de memoria usada durante la ejecución del experimento 3 (1 hora) junto con el número de *threads* por minuto creados (y aceptados por CPUs) en cada una de los nodos del clúster. Como se puede observar, se realiza un uso intensivo de memoria y CPU, aunque sin sobrepasar los límites virtuales de la plataforma debido al sobrecoste en operaciones de *swap* y *buffering*.

5. Conclusiones y Trabajo Futuro

En el presente trabajo se ha presentado jMetalSP, una solución software para hacer frente a problemas de optimización dinámica en entornos Big Data, que combinan el *framework* de optimización jMetal con el sistema de *cluster computing* Apache Spark.

La motivación de este trabajo ha sido impulsada por el aumento en el uso de Spark como plataforma de computación distribuida, así como la utilización del *framework* jMetal como motor de optimización multi-objetivo.

Se ha generado un caso de estudio que considera una formulación del problema TSP con dos objetivos: minimizar la distancia total y el tiempo de viaje. Se ha generado una instancia real del problema creada a partir de los datos abiertos de la ciudad de Nueva York. Dichos datos se han usado para actualizar la información del problema durante la ejecución. Además, se han tratado dos fuentes

de datos artificiales adicionales basados en Twitter y Kafka. Esta instancia a servido como base de validación experimental en un *cluster* Hadoop compuesto por 100 nodos. La arquitectura propuesta obtiene un rendimiento estable, a pesar de ser sometida a un flujo de datos intensivo. Por simplicidad, se han omitido resultados en cuanto a los frentes de soluciones obtenidos. No obstante, podemos indicar que los resultados en este sentido vienen abalados por el comportamiento de NSGA-II en la resolución de TSP, altamente justificado en la literatura.

Como trabajo futuro, tenemos la intención de definir problemas más realistas, incluyendo nuevas fuentes de datos adicionales, así como la utilización de otros algoritmos de optimización además de NSGA-II.

Agradecimientos

Este trabajo ha sido financiado por los proyectos TIN2014-58304-R (MECD), P11-TIC-7529 (Proy. Ex. Junta de Andalucía) y P12-TIC-1519 (PAIDI). Cristóbal Barba-González dispone de un beca pre-doctoral BES-2015-072209 (MECD).

Referencias

1. Marr, M.: Big Data: Using SMART Big Data, Analytics and Metrics To Make Better Decisions and Improve Performance. Wiley (2015)
2. Hashem, I.A.T., Yaqoob, I., Anuar, N.B., Mokhtar, S., Gani, A., Khan, S.U.: The rise of big data on cloud computing: Review and open research issues. Information Systems **47** (2015) 98 – 115
3. Zhou, Z.H., Chawla, N.V., Jin, Y., Williams, G.J.: Big data opportunities and challenges: Discussions from data analytics perspectives [discussion forum]. IEEE Computational Intelligence Magazine **9** (2014) 62–74
4. Farina, M., Deb, K., Amato, P.: Dynamic multiobjective optimization problems: test cases, approximations, and applications. IEEE Trans. on Evol. Comp. **8** (2004) 425–442
5. Durillo, J., Nebro, A.: jMetal: A java framework for multi-objective optimization. Advances in Engineering Software **42** (2011) 760 – 771
6. Zaharia, M., Chowdhury, M., Franklin, M., Shenker, S., Stoica, I.: Spark: Cluster computing with working sets. In: Proceedings of the 2Nd USENIX Conference on Hot Topics in Cloud Computing. HotCloud'10, Berkeley, CA, USA, USENIX Association (2010) 10–10
7. Papadimitriou, C.H.: The euclidean travelling salesman problem is NP-Complete. Theoretical Computer Science **4** (1977) 237 – 244
8. Deb, K., Pratap, A., Agarwal, S., Meyarivan, T.: A fast and elitist multiobjective genetic algorithm: NSGA-II. IEEE Trans. on Evol. Comp. **6** (2002) 182–197
9. Zitzler, E., Laumanns, M., Thiele, L.: SPEA2: Improving the strength pareto evolutionary algorithm. In Giannakoglou, K., Tsahalis, D., Periaux, J., Papailou, P., Fogarty, T., eds.: EUROGEN 2001. Evol. Methods for Design, Optimization and Control with Applications to Industrial Problems, Greece (2002) 95–100
10. Beume, N., Naujoks, B., Emmerich, M.: SMS-EMOA: Multiobjective selection based on dominated hypervolume. European Journal of Operational Research **181** (2007) 1653–1669