View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Repositorio Institucional Universidad de Málaga

Dos estrategias de búsqueda *anytime* basadas en programación lineal entera para resolver el problema de selección de requisitos

Francisco Chicano[1], Miguel Ángel Domínguez[1], Isabel del Águila[2], José del Sagrado[2] y Enrique Alba[1]

[1] Universidad de Málaga, Andalucía Tech, España
chicano@lcc.uma.es, miguel.angel.dominguez.rios@uma.es, eat@lcc.uma.es
[2] Universidad de Almería, España
{imaguila,jsagrado}@ual.es

Resumen El problema de selección de requisitos (o *Next Release Problem*, NRP) consiste en seleccionar el subconjunto de requisitos que se va a desarrollar en la siguiente versión de una aplicación software. Esta selección se debe hacer de tal forma que maximice la satisfacción de las partes interesadas a la vez que se minimiza el esfuerzo empleado en el desarrollo y se cumple un conjunto de restricciones. Trabajos recientes han abordado la formulación bi-objetivo de este problema usando técnicas exactas basadas en resolutores SAT y resolutores de programación lineal entera. Ambos se enfrentan a dificultades cuando las instancias tienen un gran tamaño. En la práctica, no es necesario calcular todas las soluciones del frente de Pareto (que pueden llegar a ser muchas) y basta con obtener un buen número de soluciones no dominadas bien distribuidas en el espacio objetivo. Las estrategias de búsqueda basadas en ILP que se han utilizado en el pasado para encontrar un frente bien distribuido en cualquier instante de tiempo solo buscan soluciones que pueden obtenerse minimizando una suma ponderada de los objetivos (*soluciones soportadas*). En este trabajo proponemos dos estrategias basadas en ILP que son capaces de encontrar el frente completo con suficiente tiempo y que, además, tienen la propiedad de aportar un conjunto de soluciones bien distribuido en el frente objetivo en cualquier momento de la búsqueda.

1. Introducción

La Ingeniería de Requisitos define el proceso, o conjunto de tareas, para descubrir el propósito de cualquier sistema, identificando las personas involucradas y sus necesidades [10]. Este proceso es esencial en el desarrollo de sistemas software, ya que debido a la naturaleza lógica del software, la principal medida de éxito del sistema desarrollado vendrá dada por el grado de consecución o cumplimiento de los requisitos. Los procesos relacionados con los requisitos no son fáciles de llevar a cabo porque residen en el espacio del problema y no en el de la solución, además son procesos difusos en el ciclo de vida de desarrollo

de software, en el que ideas informales deben ser traducidas a ideas formales; clientes y organizaciones deben colaborar para alcanzar un acuerdo preciso y sin ambigüedades de lo que debe ser desarrollado.

La planificación de versiones, es decir, la definición de cómo va a ir evolucionando un producto software a lo largo de su vida útil, es una tarea fundamental ligada al campo de la Ingeniería de Requisitos. En general, los clientes proponen numerosas nuevas características o requisitos que en la mayoría de los casos no todas pueden completarse dentro de las limitaciones impuestas por el tiempo y los recursos y que, por tanto, deben acotarse de alguna manera [2]. Estas decisiones suelen tener que considerar varios objetivos diferentes e incluso conflictivos, como las interacciones o dependencias entre las características candidatas, las preferencias de los clientes o las limitaciones en los recursos. En otras palabras, la planificación de versiones, en general, implica la optimización basada en múltiples criterios [7]. Los clientes, que buscan su propio interés, demandan las mejoras que consideran importantes, pero no todas ellas pueden ser satisfechas. Por un lado, cada requisito significa un coste en términos de esfuerzo que la empresa tiene que asumir, y por otro lado, ni todos los clientes son igualmente importantes para la empresa, ni todas las características son igualmente importantes para los clientes. Los factores de mercado también pueden influir en este proceso de selección: la empresa puede estar interesada en satisfacer las necesidades de los clientes más nuevos o en garantizar que cada cliente ve cumplido al menos uno de sus requisitos propuestos.

Este problema de planificación multi-objetivo se ha resuelto en el pasado usando tanto metaheurísticas [11], que no aseguran la calidad de las soluciones obtenidas pero son capaces de obtener soluciones de calidad aceptable en tiempos cortos, como técnicas exactas que calculan el frente óptimo de Pareto [1,13]. En este trabajo nos acercamos a este segundo enfoque. En particular, hemos observado que en el trabajo de Veerapen et al. [13] el algoritmo empleado para encontrar el frente completo requiere mucho tiempo para las instancias grandes, mientras que el algoritmo que utilizan para encontrar soluciones no dominadas bien distribuidas en el espacio objetivo sólo es capaz de encontrar soluciones soportadas[3]. En este trabajo proponemos dos estrategias (descritas en las secciones 4.4 y 4.5) que tratan de encontrar soluciones del frente que estén bien distribuidas, pero sin renunciar a la completitud, es decir, con suficiente tiempo, los algoritmos pueden calcular el frente de Pareto completo. Además, comparamos estas estrategias con otras tres más basadas en programación lineal entera y con técnicas metaheurísticas. Además de la propuesta de las dos estrategias de búsqueda, respondemos las siguientes preguntas de investigación:

- **RQ1**: ¿Cuál es la calidad de la parte del frente de Pareto encontrada por los distintos algoritmos comparados cuando limitamos el tiempo de ejecución de los mismos?
- **RQ2**: ¿Cuándo conviene utilizar metaheurísticas para resolver el problema y cuándo es mejor usar algoritmos basados en programación lineal entera?

[3] Se denominan *soluciones soportadas* a aquellas que pueden obtenerse minimizando una suma ponderada de los objetivos.

Para responder a ellas hemos realizado un estudio experimental con 19 instancias del problema y un total de ocho algoritmos diferentes.

El resto del artículo está organizado como sigue. En la sección 2 formalizamos el problema de la siguiente versión y la sección 3 presenta los programas lineales enteros usados para resolverlo. En la sección 4 presentamos los distintos algoritmos basados en ILP que utilizamos. La sección 5 presenta los resultados de un estudio experimental realizado para comparar los distintos algoritmos. Por último, la sección 6 presenta las conclusiones y el trabajo futuro.

2. El problema de la siguiente versión

Partimos de un conjunto de requisitos con dependencias $R = \{r_1, r_2, \ldots, r_n\}$ que aún no han sido desarrollados y que han sido propuestos por un conjunto de m clientes. Cada cliente i tiene un peso asociado $w_i \in \mathbb{R}$ que mide su importancia dentro del proyecto. Cada requisito $r_j \in R$ tiene un coste c_j para la empresa, es decir cada r_j consumirá c_j recursos si se desarrolla. El mismo requisito puede ser sugerido por varios clientes y su *valor* puede ser diferente para cada uno de ellos. El valor del requisito r_j para el cliente i se representa con $v_{ij} \in \mathbb{R}$.

En este punto cabe mencionar dos formas de valorar un conjunto de requisitos seleccionado. Por un lado, Xuan et al. [14] consideran que un cliente *está satisfecho* sólo cuando todos sus requisitos se implementan y, en este caso, sumamos su peso w_i al *grado de satisfacción* asociado al conjunto de requisitos. Los elementos v_{ij} toman dos posibles valores: 1 si el requisito interesa al cliente (tiene valor para él) y 0 si no está interesado en él.

Por otro lado, Del Sagrado et al. [11] definen la *satisfacción* asociada a un requisito, s_j, como la suma ponderada del valor que le dan los clientes, $s_j = \sum_{i=1}^{m} w_i * v_{ij}$. Según esta interpretación, no es necesario implementar todos los requisitos que interesan a un cliente para obtener una cierta satisfacción del mismo.

En cualquier caso, los requisitos presentan dependencias o interacciones entre ellos, imponiendo un orden de desarrollo determinado, lo que limita las alternativas para ser elegidos [3,8]. Las interacciones entre los requisitos representan restricciones al problema y se agrupan en dos tipos: *dependencias funcionales* o *estructurales* y *dependencias por recursos consumidos* [12]. En este trabajo nos centramos en las dependencias funcionales, que pueden definirse como:

- *Implicación o precedencia.* $r_i \Rightarrow r_j$. El requisito r_i no se puede seleccionar si el requisito r_j no ha sido ya implementado.
- *Combinación o acoplamiento.* $r_i \odot r_j$. Los requisitos r_i y r_j deben ser incluidos de forma conjunta en el software.
- *Exclusión.* $r_i \oplus r_j$. El requisito r_i no puede incluirse junto al requisito r_j.

El objetivo será encontrar $\hat{R} \subseteq R$ de forma que se maximice el valor a la vez que se minimiza el coste para el conjunto de requisitos seleccionados $\hat{R}$. El coste

viene dado por la función:

$$coste(\hat{R}) = \sum_{j, r_j \in \hat{R}}^{n} c_j, \tag{1}$$

mientras que el valor viene dado por las funciones:

$$valor(\hat{R}) = \sum_{i=1}^{m} w_i \prod_{j, r_j \in \hat{R}} v_{ij} \quad \text{y} \quad valor(\hat{R}) = \sum_{j, r_j \in \hat{R}}^{n} s_j, \tag{2}$$

en las interpretaciones de Xuan et al. [14] y Del Sagrado et al. [11], respectivamente. El problema es multi-objetivo y, por tanto no existe una solución única, sino un conjunto de soluciones Pareto óptimas, también llamadas *no dominadas* o *eficientes* [6].

3. Formulación del problema como ILP bi-objetivo

A partir de la descripción del problema, su formulación es bastante directa como programa lineal entero. Para ello, utilizaremos una variable binaria (puede tomar valores 0 y 1) por cada requisito seleccionable. Por simplicidad, aquí usaremos para dichas variables el mismo nombre que los requisitos asociados: r_1, r_2, etc. Para formar el programa lineal, es necesario transformar cada interacción funcional entre requisitos en una igualdad o desigualdad de expresiones lineales. Estas transformaciones se realizan de acuerdo al siguiente esquema:

- Implicación $r_i \Rightarrow r_j$: $r_i \leq r_j$.
- Combinación $r_i \odot r_j$: $r_i = r_j$.
- Exclusión $r_i \oplus r_j$: $r_i + r_j \leq 1$.

La función de coste es:

$$coste(r) = \sum_{j=1}^{n} c_j r_j. \tag{3}$$

Las funciones de valor en las interpretaciones de Xuan et al. y Del Sagrado et al. son:

$$valor(t) = \sum_{i=1}^{m} w_i t_i \quad \text{y} \quad valor(r) = \sum_{j=1}^{n} s_j r_j. \tag{4}$$

donde las variables t_i que aparecen en la función de valor de Xuan et al. indican si un cliente está o no satisfecho. La exigencia de que un cliente esté satisfecho sólo cuando todos sus requisitos están implementados en la interpretación de Xuan et al., implica la incorporación de restricciones de la forma $t_i \leq r_j$ si y sólo si $v_{ij} = 1$.

4. Algoritmos de resolución

En esta sección presentamos los distintos algoritmos basados en programación lineal entera utilizados para calcular el frente de Pareto del problema de selección de requisitos. Para la exposición de los algoritmos se considerará, sin pérdida de generalidad, que los dos objetivos deben ser minimizados. Llamaremos X al conjunto de soluciones que cumplen con todas las restricciones. Decimos que una solución $x \in X$ es *débilmente eficiente* cuando no existe $y \in X$ tal que $f_i(y) < f_i(x)$ para $i = 1, 2$, es decir, no existe solución que mejore a x en todos los objetivos. Un solución $y \in X$ *domina* a $x \in X$ si $f_i(y) \leq f_i(x)$ para $i = 1, 2$ y existe $j \in \{1, 2\}$ tal que $f_j(y) < f_j(x)$. Una solución $x \in X$ es *eficiente* si no existe $y \in X$ que la domine. Toda solución eficiente es débilmente eficiente [6]. Una solución $x \in X$ se denomina *solución soportada* si existe un vector de valores reales $\alpha_i \in \mathbb{R}$ tal que x también es solución del problema de minimización de $\sum_{i=1}^{2} \alpha_i f_i(x)$ sujeto a $x \in X$. Las principales contribuciones de este artículo son los algoritmos descritos en las secciones 4.4 y 4.5.

4.1. Algoritmo ε-constraint con un ILP por iteración

Este algoritmo se muestra en el Algoritmo 1 y es el utilizado en [13] para encontrar el frente de Pareto completo. Al principio calcula una solución z que minimice f_2 y la introduce en el conjunto FP, de soluciones óptimas de Pareto. A continuación asigna a ε el valor $f_1(z)-1$. Al entrar en el bucle, trata de minimizar f_2 sujeto a que el valor de f_1 sea menor que ε. El valor de f_1 en la nueva solución servirá para establecer el nuevo límite para f_1. El bucle termina cuando no existen soluciones con f_1 por debajo de ϵ, garantizándose de esta forma que no existirán más soluciones eficientes. El algoritmo resuelve un único subproblema en cada iteración (línea 5) y sólo puede garantizar obtener soluciones débilmente eficientes. Esto exige eliminar soluciones dominadas al finalizar el bucle (línea 9).

Algoritmo 1 ε-constraint con un ILP por iteración

1: $z \leftarrow$ resolver $\{\text{mín } f_2(x),\ \text{sujeto a } x \in X\}$
2: FP $\leftarrow \{z\}$ // Frente de Pareto
3: $\varepsilon \leftarrow f_1(z) - 1$
4: **while** $\exists x \in X, f_1(x) \leq \varepsilon$ **do**
5: $\quad z \leftarrow$ resolver $\{\text{mín } f_2(x),\ \text{sujeto a } f_1(x) \leq \varepsilon, x \in X\}$
6: $\quad$ FP = FP $\cup \{z\}$
7: $\quad \varepsilon \leftarrow f_1(z) - 1$
8: **end while**
9: Eliminar de FP las soluciones dominadas

4.2. Algoritmo ε-constraint con dos ILPs por iteración

En el Algoritmo 2 mostramos una variante de ε-constraint que resuelve dos subproblemas por iteración en lugar de uno. El objetivo es encontrar en cada

iteración una solución eficiente, que se puede añadir a FP sin filtrar. El funcionamiento del bucle es el mismo que en el caso anterior. Este algoritmo fue propuesto para el NRP en [1], donde se usó un resolutor SAT en lugar de un resolutor ILP para resolver los problemas de optimización.

Podría parecer que el Algoritmo 1 es más rápido que el Algoritmo 2, ya que el esfuerzo computacional es menor, pero esta diferencia va disminuyendo a medida que el problema a resolver posea un conjunto mayor de soluciones débilmente eficientes. Supóngase el caso en que $|N| = k$ y $|wN| > 2k$, siendo N y wN los conjuntos de puntos eficientes (no dominados) y débilmente eficientes, respectivamente. El Algoritmo 1 tendrá que realizar en el bucle más de $2k$ iteraciones y el Algoritmo 2 sólo $2k$ iteraciones.

Algoritmo 2 ε-constraint con dos ILPs por iteración

1: $z \leftarrow$ resolver $\{\text{mín}\, f_2(x),\ \text{sujeto a}\ x \in X\}$
2: $z \leftarrow$ resolver $\{\text{mín}\, f_1(x),\ \text{sujeto a}\ f_2(x) \leq f_2(z), x \in X\}$
3: FP $\leftarrow \{z\}$ // Frente de Pareto
4: $\varepsilon \leftarrow f_1(z) - 1$
5: **while** $\exists x \in X, f_1(x) \leq \varepsilon$ **do**
6: $\quad z \leftarrow$ resolver $\{\text{mín}\, f_2(x),\ \text{sujeto a}\ f_1(x) \leq \varepsilon, x \in X\}$
7: $\quad z \leftarrow$ resolver $\{\text{mín}\, f_1(x),\ \text{sujeto a}\ f_2(x) \leq f_2(z), x \in X\}$
8: $\quad$ FP = FP $\cup\{z\}$
9: $\quad \varepsilon \leftarrow f_1(z) - 1$
10: **end while**

4.3. Algoritmo ε-constraint aumentado (Aε-con)

El Algoritmo 3, conocido como *augmented ε-constraint* o AUGMECON, elimina las deficiencias de los Algoritmos 1 y 2. Por un lado, solo se resuelve un subproblema en cada iteración, y por otro, se garantiza que el punto no dominado obtenido es eficiente. El lector interesado puede consultar [9] para más detalles. En cada iteración, el algoritmo fija un valor positivo para la constante λ, que debe ser suficientemente pequeño para evitar que el algoritmo omita algunas de las soluciones eficientes, y lo bastante grande como para evitar problemas numéricos. En general, es suficiente tomar un valor de λ en $[10^{-3}, 10^{-6}]$ (véase [9]). En nuestro caso, hemos optado por calcular el valor de λ en cada iteración teniendo en cuenta el punto eficiente obtenido anteriormente. En particular la expresión que usamos es: $\lambda = 1/(f_1(z) - u_1)$, donde u_1 es una cota inferior de $\text{mín}\, f_1(x), x \in X$, es decir, la primera componente de un punto utópico.

Algoritmo 3 ε-constraint aumentado

1: $z \leftarrow$ resolver $\{\text{mín}\, f_2(x), \quad \text{sujeto a} \quad x \in X\}$
2: $z \leftarrow$ resolver $\{\text{mín}\, f_1(x), \quad \text{sujeto a} \quad f_2(x) \leq f_2(z), x \in X\}$
3: FP $\leftarrow \{z\}$ // Frente de Pareto
4: $\varepsilon \leftarrow f_1(z) - 1$
5: **while** $\exists x \in X, f_1(x) \leq \varepsilon$ **do**
6: Estimar un valor para $\lambda > 0$
7: $z \leftarrow$ resolver $\{\text{mín}\, f_2(x) - \lambda l, \quad \text{sujeto a} \quad f_1(x) + l = \varepsilon, x \in X\}$
8: FP = FP $\cup \{z\}$
9: $\varepsilon \leftarrow f_1(z) - 1$
10: **end while**

4.4. Algoritmo *anytime* basado en *augmented weighted Tchebycheff* (AAWTcheby)

Todos los algoritmos anteriores encuentran las soluciones eficientes en orden lexicográfico de sus objetivos. El principal problema de ese orden se pone de manifiesto cuando se trata de resolver una instancia tan grande que requiere mucho tiempo de cómputo. En ese caso, los algoritmos encontrarán sólo un extremo del frente. Desde un punto de vista práctico al decisor le interesa tener un conjunto de soluciones eficientes que se encuentren lo mejor distribuidas posible en el espacio objetivo. Esto puede conseguirse diseñando algoritmos que "salten" en el espacio objetivo en busca de soluciones eficientes. Estas estrategias se conocen en inglés como *anytime*. Veerapen et al. [13] utilizaron una búsqueda dicotómica para lograr este objetivo. La búsqueda dicotómica, sin embargo, adolece de un grave problema: sólo es capaz de encontrar soluciones eficientes soportadas. Como consecuencia, en frentes cóncavos, la calidad del frente que calcula puede ser muy baja. En el presente trabajo proponemos dos técnicas *aytime* que son capaces de encontrar el frente completo con suficiente tiempo. Comenzamos en esta sección describiendo la primera de ellas.

El algoritmo aumentado y ponderado de Tchebycheff permite encontrar soluciones eficientes en una zona cualquiera del frente usando una sola ejecución de resolutor ILP. La zona a explorar viene determinada por un par de puntos (generalmente eficientes) $(z^{(1)}, z^{(2)})$, que asumimos ordenados de tal forma que $z_1^{(1)} < z_1^{(2)}$. A partir de estos puntos, el algoritmo resuelve el siguiente problema lineal en cada iteración:

$$\text{mín}\, y \tag{5}$$

sujeto a

$$y \geq -(\zeta_1 - \zeta_0)(f_1(x) - \xi_1) + (\xi_1 - \xi_0)(f_2(x) - \zeta_1) \tag{6}$$
$$y \geq -(\zeta_2 - \zeta_1)(f_1(x) - \xi_1) + (\xi_2 - \xi_1)(f_2(x) - \zeta_1) \tag{7}$$
$$f_1(x) \leq \xi_2 \tag{8}$$
$$f_2(x) \leq \zeta_0 \tag{9}$$

donde los valores de ζ_i y ξ_i son: $\xi_0 = z_1^{(1)}$, $\zeta_0 = z_2^{(1)}$, $\xi_1 = z_1^{(2)} - 1/2$, $\zeta_1 = z_2^{(1)} - 1/2$, $\xi_2 = z_1^{(2)}$, $\zeta_2 = z_2^{(2)}$. El problema anterior devuelve una solución eficiente que se encuentra entre $z^{(1)}$ y $z^{(2)}$. Para más detalles consultar [4].

A partir de este programa podemos construir un algoritmo que primero calcula los dos óptimos lexicográficos (líneas 1 y 2) y, a continuación, explora el espacio entre ellos. Cada vez que encuentra un nuevo punto eficiente entre dos existentes, lo añade al frente de Pareto y divide el rectángulo explorado en dos para explorarlos más adelante (línea 10). Cuando ya no quedan más rectángulos sin explorar el algoritmo termina. En nuestra implementación los rectángulos son explorados por orden de área, el de mayor área se explora primero. Esto permite obtener un punto en cada iteración con potencial para maximizar el hipervolumen cubierto hasta ese momento.

Algoritmo 4 *Anytime augmented weighted* Tchebycheff

1: $z^{(1)} \leftarrow$ calcular óptimo lexicográfico para el orden (f_1, f_2)
2: $z^{(2)} \leftarrow$ calcular óptimo lexicográfico para el orden (f_2, f_1)
3: FP $\leftarrow \{z^{(1)}, z^{(2)}\}$ // Frente de Pareto
4: Cola $\leftarrow \{(z^{(1)}, z^{(2)})\}$
5: **while** Cola $\neq \emptyset$ **do**
6: $(z^{(1)}, z^{(2)}) \leftarrow$ extraerParDeMayorArea(Cola)
7: $z \leftarrow$ resolverTchebycheff($(z^{(1)}, z^{(2)})$)
8: **if** z no dominado en $(z^{(1)}, z^{(2)})$ **then**
9: FP = FP $\cup\{z\}$
10: Cola $\leftarrow$ Cola $\cup\{(z^{(1)}, z), (z, z^{(2)})\}$
11: **end if**
12: **end while**

4.5. Algoritmo *anytime* basado en ε-constraint aumentado (AAε-con)

El Algoritmo 5 actúa de forma similar al anterior, analizando el rectángulo en el espacio objetivo delimitado por pares de puntos. La diferencia fundamental entre ambos algoritmos es que en AAε-con se utiliza el algoritmo ε-constraint aumentado para encontrar la solución eficiente dentro de ese rectángulo.

El algoritmo ε-constraint aumentado necesita un valor de ε, que establecemos al punto medio entre $z^{(1)}$ y $z^{(2)}$ (línea 7). Si existe una solución eficiente con f_1 menor que dicho ε, el algoritmo la encontrará y la incorporará a FP, salvo que sea una solución dominada (es decir, que la haya encontrado antes). Si no encuentra ninguna solución, entonces debe añadir a la cola de exploración la mitad del rectángulo que no ha sido explorado, es decir, aquella con f_1 mayor que ε.

Algoritmo 5 *Anytime* ε-constraint aumentado

1: $z^{(1)} \leftarrow$ calcular óptimo lexicográfico para el orden (f_1, f_2)
2: $z^{(2)} \leftarrow$ calcular óptimo lexicográfico para el orden (f_2, f_1)
3: FP $\leftarrow \{z^{(1)}, z^{(2)}\}$ // Frente de Pareto
4: Cola $\leftarrow \{(z^{(1)}, z^{(2)})\}$
5: **while** Cola $\neq \emptyset$ **do**
6: $\quad (z^{(1)}, z^{(2)}) \leftarrow$ extraerParDeMayorArea(Cola)
7: $\quad \varepsilon \leftarrow (z_1^{(1)} + z_1^{(2)})/2$
8: $\quad z \leftarrow$ resolver $\{\text{mín}\, f_2(x) - \lambda l, \;\; s.a. \;\; f_1(x) + l = \varepsilon, x \in X\}$
9: $\quad$ **if** z no dominado en $(z^{(1)}, z^{(2)})$ **then**
10: $\quad\quad$ FP = FP $\cup\{z\}$
11: $\quad\quad$ Cola $\leftarrow$ Cola $\cup\{(z^{(1)}, z), (z, z^{(2)})\}$
12: $\quad$ **else**
13: $\quad\quad$ Cola $\leftarrow$ Cola $\cup\{((\varepsilon, z_2^{(1)}), z^{(2)})\}$
14: $\quad$ **end if**
15: **end while**

5. Estudio experimental

En esta sección comparamos los resultados obtenidos por los distintos algoritmos de programación lineal entera. En particular, estamos interesados en estudiar la calidad del frente de Pareto cuando limitamos el tiempo de ejecución de los algoritmos. Este escenario puede corresponderse con el habitual en la práctica, en especial, cuando se usan metodologías ágiles y debe resolverse el problema de determinar los requisitos a implementar en la siguiente iteración. Por otro lado, nos gustaría comparar los resultados de programación lineal entera con los algoritmos metaheurísticos que se han venido utilizando para resolver este problema hasta el momento. El código con la implementación de los algoritmos utilizados en el estudio está publicado en la URL `https://github.com/jfrchicanog/NextReleaseProblem`. En dicha URL también puede encontrarse un enlace a las instancias.

5.1. Instancias

Utilizaremos dos conjuntos de instancias para los experimentos. Por un lado, usaremos las 17 instancias de Xuan et al. [14], utilizadas también en el trabajo de Veerapen et al. [13]. Este conjunto a su vez contiene cinco instancias clásicas y 12 realistas. Tienen entre 140 y 4368 requisitos y la única posible relación entre requisitos es la implicación. El segundo conjunto de instancias está formado por dos instancias que han sido previamente utilizadas en el trabajo de Del Sagrado et al. [11]. Las instancias tienen 20 y 100 requisitos, respectivamente, y se caracterizan porque, además de la implicación, aparece la relación de simultaneidad. En la primer columna de la tabla 1 aparece el número de requisitos de cada instancia junto a su nombre.

5.2. Cálculo del frente con límite de tiempo

El trabajo de Veerapen et al. [13] muestra que, aunque los resolutores de programación lineal entera han avanzado hasta el punto de resultar de utilidad para el problema de la selección de requisitos, los tiempos de ejecución crecen rápidamente con el tamaño de las instancias. Así, de las 19 instancias utilizadas en dicho trabajo, sólo en tres los algoritmos terminan de calcular el frente de Pareto en menos de 30 segundos (las instancias que resultan ser más rápidas de resolver son las dos reales). Por eso, en este estudio nos planteamos la siguiente pregunta de investigación: **RQ1** ¿cuál es la calidad de la parte del frente de Pareto encontrada por cada algoritmo cuando limitamos el tiempo de ejecución de los algoritmos?

A priori, pensamos que los algoritmos *anytime* deben ser mejores que aquellos que buscan las soluciones del frente en orden lexicográfico cuando el límite de tiempo es bajo en comparación con el tiempo requerido para encontrar todo el frente. Esperamos, no obstante, que los algoritmos que siguen el orden lexicográfico aventajen a los *anytime* cuando el límite de tiempo se acerca al requerido para encontrar todas las soluciones.

Hemos limitado el tiempo que los algoritmos se ejecutan a 300 segundos y hemos realizado una sola ejecución de los algoritmos, ya que son deterministas. El tiempo de ejecución de los algoritmos puede depender de la carga del sistema operativo. No obstante, este error en el tiempo es muy inferior al límite de tiempo, por lo que no vemos necesaria la realización de más de una ejecución de los algoritmos. La comparación de los distintos algoritmos la haremos calculando el hipervolumen basado en el punto de Nadir[4] obtenido por el frente generado por cada algoritmo al finalizar el límite de tiempo. Los resultados se muestran en la tabla 1. En algunas instancias (en particular, `nrp1`, `dataset1` y `dataset2`), los algoritmos terminan en menos de 300 segundos y, como consecuencia, todos los algoritmos alcanzan el máximo hipervolumen.

Podemos observar que, con la única excepción de `nrp5`, los algoritmos que consiguen un mayor hipervolumen son los algoritmos *anytime*. Por tanto, vemos que cumplen el objetivo para el que fueron diseñados. Como habíamos adelantado, cuando el tiempo de ejecución de los algoritmos que siguen un orden lexicográfico es cercano o menor que el límite de tiempo, son estos algoritmos los que resultan más eficientes, ya que son más rápidos encontrando el frente completo. Este es el caso de `nrp5` que, según [13], es la instancia de Xuan et al. más rápida de resolver después de `nrp1`.

Entre los algoritmos *anytime*, no podemos identificar un claro ganador. Vemos que AAWTcheby consigue resultados ligeramente mejores en algunas instancias mientras que AAε-con los mejora en otras, pero las diferencias son pequeñas.

[4] El hipervolumen en este caso es el área comprendida entre el punto de Nadir y las soluciones del frente obtenido.

Tabla 1. Hipervolumen de las soluciones del frente obtenidas por los distintos algoritmos en 300 segundos. Se destacan con fondo oscuro las celdas con valores máximos para cada instancia.

	Orden lexicográfico			Anytime	
Instancia (req.)	**ε-con 1-ILP**	**ε-con 2-ILPs**	**Aε-con**	**AAWTcheby**	**AAε-con**
dataset1 (20)	$5{,}23{\cdot}10^4$	$5{,}23{\cdot}10^4$	$5{,}23{\cdot}10^4$	$5{,}23{\cdot}10^4$	$5{,}23{\cdot}10^4$
dataset2 (100)	$1{,}80{\cdot}10^6$	$1{,}80{\cdot}10^6$	$1{,}80{\cdot}10^6$	$1{,}80{\cdot}10^6$	$1{,}80{\cdot}10^6$
nrp1 (140)	$1{,}32{\cdot}10^6$	$1{,}32{\cdot}10^6$	$1{,}32{\cdot}10^6$	$1{,}32{\cdot}10^6$	$1{,}32{\cdot}10^6$
nrp2 (620)	$1{,}75{\cdot}10^6$	$1{,}06{\cdot}10^6$	$1{,}57{\cdot}10^6$	$3{,}65{\cdot}10^7$	$3{,}68{\cdot}10^7$
nrp3 (1500)	$3{,}04{\cdot}10^6$	$1{,}15{\cdot}10^6$	$9{,}88{\cdot}10^5$	$5{,}84{\cdot}10^7$	$5{,}85{\cdot}10^7$
nrp4 (3250)	$1{,}21{\cdot}10^6$	$6{,}98{\cdot}10^5$	$1{,}29{\cdot}10^6$	$2{,}07{\cdot}10^8$	$2{,}16{\cdot}10^8$
nrp5 (1500)	$5{,}60{\cdot}10^7$	$1{,}19{\cdot}10^7$	$3{,}00{\cdot}10^7$	$5{,}25{\cdot}10^7$	$5{,}60{\cdot}10^7$
nrp-e1 (3502)	$3{,}14{\cdot}10^6$	$1{,}74{\cdot}10^6$	$4{,}58{\cdot}10^5$	$1{,}34{\cdot}10^8$	$1{,}34{\cdot}10^8$
nrp-e2 (4254)	$2{,}69{\cdot}10^6$	$1{,}05{\cdot}10^6$	$2{,}31{\cdot}10^6$	$1{,}52{\cdot}10^8$	$1{,}52{\cdot}10^8$
nrp-e3 (2844)	$6{,}84{\cdot}10^6$	$2{,}87{\cdot}10^6$	$5{,}42{\cdot}10^5$	$8{,}94{\cdot}10^7$	$8{,}94{\cdot}10^7$
nrp-e4 (3186)	$4{,}10{\cdot}10^6$	$2{,}08{\cdot}10^6$	$5{,}20{\cdot}10^5$	$8{,}82{\cdot}10^7$	$8{,}81{\cdot}10^7$
nrp-g1 (2690)	$1{,}26{\cdot}10^7$	$4{,}77{\cdot}10^6$	$7{,}82{\cdot}10^6$	$1{,}09{\cdot}10^8$	$1{,}09{\cdot}10^8$
nrp-g2 (2650)	$1{,}15{\cdot}10^7$	$3{,}58{\cdot}10^6$	$8{,}38{\cdot}10^6$	$7{,}56{\cdot}10^7$	$7{,}56{\cdot}10^7$
nrp-g3 (2512)	$2{,}01{\cdot}10^7$	$9{,}39{\cdot}10^6$	$1{,}79{\cdot}10^6$	$9{,}64{\cdot}10^7$	$9{,}63{\cdot}10^7$
nrp-g4 (2246)	$1{,}94{\cdot}10^7$	$5{,}13{\cdot}10^6$	$1{,}24{\cdot}10^7$	$5{,}96{\cdot}10^7$	$5{,}97{\cdot}10^7$
nrp-m1 (4060)	$1{,}86{\cdot}10^6$	$8{,}59{\cdot}10^5$	$4{,}20{\cdot}10^5$	$2{,}24{\cdot}10^8$	$2{,}24{\cdot}10^8$
nrp-m2 (4368)	$1{,}51{\cdot}10^6$	$9{,}34{\cdot}10^5$	$3{,}98{\cdot}10^5$	$1{,}96{\cdot}10^8$	$1{,}95{\cdot}10^8$
nrp-m3 (3566)	$2{,}06{\cdot}10^6$	$8{,}95{\cdot}10^5$	$3{,}72{\cdot}10^5$	$1{,}92{\cdot}10^8$	$1{,}92{\cdot}10^8$
nrp-m4 (3643)	$2{,}34{\cdot}10^6$	$1{,}20{\cdot}10^6$	$5{,}08{\cdot}10^5$	$1{,}48{\cdot}10^8$	$1{,}48{\cdot}10^8$

5.3. Comparación con metaheurísticas

El problema de selección de requisitos bi-objetivo se ha resuelto en trabajos previos usando técnicas metaheurísticas, como NSGA-II, ACO y GRASP [5,11,15]. Tanto este trabajo como [13], muestran que los algoritmos exactos basados en programación lineal entera son muy buenos encontrando un frente de alta calidad, en ocasiones en un tiempo razonable. Por tanto, nos planteamos la siguiente pregunta de investigación: **RQ2** ¿cuándo conviene utilizar metaheurísticas para resolver el problema y cuándo es mejor usar algoritmos basados en programación lineal entera?

De nuevo utilizaremos aquí las 17 instancias de Xuan et al. y las 2 de Del Sagrado. Para el caso de las 17 instancias de Xuan, Veerapen et al. ya respondieron a dicha pregunta en [13]. Allí compararon el método ε-constraint y la búsqueda dicotómica *anytime* (ADS) con NSGA-II. Su conclusión fue que para la mayoría de las instancias y, en particular, las de mayor tamaño, ADS obtenía un hipervolumen más alto que NSGA-II en aproximadamente el mismo tiempo (unos 90 segundos). Por otro lado, NSGA-II aventajaba a ADS en instancias muy pequeñas, que podían resolverse de manera completa usando ε-constraint. La única instancia para la que NSGA-II obtuvo un frente con mejor hipervolumen que una técnica basada en ILP en un tiempo razonable fue `nrp2`.

Para completar estos resultados, hemos ejecutado los dos algoritmos *anytime* durante el mismo tiempo que Veerapen et al. ejecutaron NSGA-II (el tiempo requerido para realizar 200 iteraciones del bucle principal de NSGA-II). Los resultados se observan en la tabla 2. La máquina empleada en [13] fue una Intel Core i7-3770 con cuatro núcleos a 3.4 GHz, que es aproximadamente entre 1,5 y

Tabla 2. Hipervolumen obtenido por NSGA-II y los dos algoritmos *anytime* en el tiempo que NSGA-II necesita para realizar 200 iteraciones (indicado en la segunda columna) [13]. Se destacan las celdas conteniendo valores más altos por instancia.

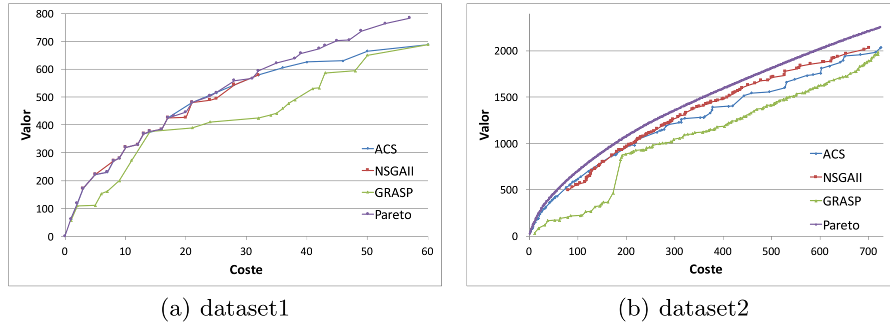
Instancia	Tiempo (s)	NSGA-II	AAWTcheby	AAε-con
nrp1	53	$1,26 \cdot 10^6$	$1,32 \cdot 10^6$	$1,32 \cdot 10^6$
nrp2	64	$2,74 \cdot 10^7$	$3,50 \cdot 10^7$	$3,62 \cdot 10^7$
nrp3	69	$4,74 \cdot 10^7$	$5,78 \cdot 10^7$	$5,82 \cdot 10^7$
nrp4	71	$1,59 \cdot 10^8$	$1,95 \cdot 10^8$	$2,13 \cdot 10^8$
nrp5	66	$3,97 \cdot 10^7$	$5,12 \cdot 10^7$	$5,60 \cdot 10^7$
nrp-e1	79	$3,97 \cdot 10^8$	$1,34 \cdot 10^8$	$1,33 \cdot 10^8$
nrp-e2	76	$1,20 \cdot 10^8$	$1,51 \cdot 10^8$	$1,52 \cdot 10^8$
nrp-e3	76	$8,07 \cdot 10^7$	$8,93 \cdot 10^7$	$8,90 \cdot 10^7$
nrp-e4	77	$7,98 \cdot 10^7$	$8,79 \cdot 10^7$	$8,74 \cdot 10^7$
nrp-g1	79	$0,98 \cdot 10^8$	$1,09 \cdot 10^8$	$1,09 \cdot 10^8$
nrp-g2	76	$6,92 \cdot 10^7$	$7,55 \cdot 10^7$	$7,55 \cdot 10^7$
nrp-g3	79	$8,78 \cdot 10^7$	$9,62 \cdot 10^7$	$9,60 \cdot 10^7$
nrp-g4	77	$5,51 \cdot 10^7$	$5,96 \cdot 10^7$	$5,96 \cdot 10^7$
nrp-m1	81	$1,96 \cdot 10^8$	$2,23 \cdot 10^8$	$2,23 \cdot 10^8$
nrp-m2	78	$1,74 \cdot 10^8$	$1,94 \cdot 10^8$	$1,94 \cdot 10^8$
nrp-m3	82	$1,67 \cdot 10^8$	$1,91 \cdot 10^8$	$1,91 \cdot 10^8$
nrp-m4	79	$1,32 \cdot 10^8$	$1,47 \cdot 10^8$	$1,47 \cdot 10^8$

2 veces más rápida que la usada en nuestros experimentos. Aún así, se observa que los algoritmos *anytime* aquí propuestos son mejores que NSGA-II.

Con respecto a las instancias de Del Sagrado et al., observamos en la sección anterior que son instancias cuya solución exacta se puede calcular en un tiempo muy corto (menos de 30 segundos), lo que significa que, en la práctica, basta con aplicar una técnica exacta para resolverlos. En cualquier caso, en la tabla 3 mostramos el hipervolumen y el tiempo de ejecución requerido por ACO, GRASP y NSGA-II para resolver estas instancias, y en la Figura 1 mostramos el frente de Pareto y las aproximaciones obtenidas por las metaheurísticas.

Tabla 3. Comparación del hipervolumen obtenido y tiempo requerido por NSGA-II, GRASP y ACS en las instancias de Del Sagrado et al. En el caso de las metaheurísticas el tiempo e hipervolumen son promedios de 100 ejecuciones independientes.

		dataset1		dataset2	
Algoritmo	**% esfuerzo**	**Hipervolumen**	**Tiempo (ms)**	**Hipervolumen**	**Tiempo (ms)**
NSGA-II	30	6843	1892	218138	28128
	50	15677	1981	495949	35046
	70	24409	2034	873384	38300
GRASP	30	5851	363	112419	28920
	50	14508	1208	425642	118340
	70	24474	840	769613	324604
ACS	30	7805	639	234583	616874
	50	18153	788	527685	770221
	70	29196	837	902769	881951
ε-con. 1-ILP	100	52271	290	1797324	11553



(a) dataset1 (b) dataset2

Figura 1. Frente de Pareto y aproximaciones de los algoritmos metaheurísticos.

Hemos de indicar que estos tiempos se refieren de nuevo a una máquina diferente (Pentium 4 a 3,2 GHz) y el objetivo no era encontrar el frente completo, sino sólo una parte limitada por el 70 % de la suma de esfuerzos. Aún así, se puede observar que los algoritmos requieren un mayor tiempo de ejecución y, en el caso de `dataset2`, los frentes aproximados obtenidos por las metaheurísticas se encuentran lejos del frente de Pareto.

En resumen, podemos concluir que todas las instancias utilizadas en este trabajo se pueden resolver satisfactoriamente empleando técnicas basadas en programación lineal entera como AAε-con, si la instancia es grande, o ε-constraint, si es pequeña.

6. Conclusiones y trabajo futuro

En este trabajo hemos propuesto dos algoritmos basados en programación lineal entera para encontrar un conjunto de soluciones eficientes bien distribuidas en el espacio objetivo en cualquier momento de la búsqueda. Estos algoritmos, con el tiempo suficiente son capaces de encontrar el frente completo. Hemos comparado los algoritmos con otros diseñados para encontrar el frente completo y observamos una clara ventaja en el hipervolumen al detener todos los algoritmos tras un límite de tiempo. También hemos comparado los algoritmos basados en ILP con técnicas metaheurísticas y deducimos que los primeros son más rápidos y obtienen mejores soluciones.

Como trabajo futuro se puede estudiar el impacto en la formulación y en las técnicas basadas en ILP que tendría considerar la existencia de incertidumbre en el coste de los requisitos. También sería interesante explorar con más detalle la razón por la que los distintos algoritmos basados en ILP tienen el rendimiento tan dispar que presentan. Esto puede permitir desarrollar nuevos y mejores algoritmos para este problema.

Agradecimientos

El trabajo ha sido parcialmente financiado por la Universidad de Málaga, Andalucía Tech y el Ministerio de Economía y Competitividad mediante la red TIN2015-71841-REDT y el proyecto TIN2014-57341-R.

Referencias

1. del Águila, I., del Sagrado, J., Chicano, F., Alba, E.: Resolviendo un problema multi-objetivo de selección de requisitos mediante resolutores del problema SAT. In: XX Jornadas de Ingeniería del Software y Bases de Datos. SISTEDES, Santander, España (2015)
2. Berander, P., Svahnberg, M.: Evaluating two ways of calculating priorities in requirements hierarchies–an experiment on hierarchical cumulative voting. Journal of Systems and Software 82(5), 836–850 (2009)
3. Carlshamre, P., Sandahl, K., Lindvall, M., Regnell, B., och Dag, J.N.: An industrial survey of requirements interdependencies in software product release planning. In: Proceedings of IEEE RE. pp. 84–93. IEEE Computer Society (2001)
4. Dächert, K., Gorski, J., Klamroth, K.: An augmented weighted tchebycheff method with adaptively chosen parameters for discrete bicriteria optimization problems. Computers & Operations Research 39(12), 2929 – 2943 (2012)
5. Durillo, J.J., Zhang, Y., Alba, E., Harman, M., Nebro, A.J.: A study of the bi-objective next release problem. Empirical Software Engineering 16(1), 29–60 (2011)
6. Ehrgott, M.: Multicriteria optimization. Springer (2005)
7. IIBA, A.: guide to the business analysis body of knowledge (babok guide). International Institute of Business Analysis (IIBA) (2009)
8. Karlsson, J., Olsson, S., Ryan, K.: Improving practical support for large-scale requirement prioritising. Requirement Engineering 2(1), 51–60 (1997)
9. Mavrotas, G.: Effective implementation of the ε-constraint method in multi-objective mathematical programming problems. Applied Mathematics and Computation 213(2), 455 – 465 (2009)
10. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of the Conference on the Future of Software Engineering. pp. 35–46. ACM (2000)
11. del Sagrado, J., del Águila, I.M., Orellana, F.J.: Multi-objective ant colony optimization for requirements selection. Empirical Software Engineering 20(3), 577–610 (2015)
12. del Sagrado, J., del Águila, I.M., Orellana, F.J.: Requirements interaction in the next release problem. In: Proceedings of 13th Annual Genetic and Evolutionary Computation Conference (GECCO 2011), Dublin, Ireland. pp. 241–242 (2011)
13. Veerapen, N., Ochoa, G., Harman, M., Burke, E.K.: An integer linear programming approach to the single and bi-objective next release problem. Information and Software Technology 65(0), 1 – 13 (2015)
14. Xuan, J., Jiang, H., Ren, Z., Luo, Z.: Solving the large scale next release problem with a backbone-based multilevel algorithm. Software Engineering, IEEE Transactions on 38(5), 1195–1212 (2012)
15. Zhang, Y., Harman, M., Mansouri, S.A.: The multi-objective next release problem. In: In Proceedings of GECCO. pp. 1129–1137 (2007)