Uso de CMSA para resolver el problema de selección de requisitos

Miguel Ángel Domínguez-Ríos
Universidad de Málaga
miguel.angel.dominguez.rios@uma.es

Francisco Chicano
Universidad de Málaga
chicano@lcc.uma.es

Enrique Alba
Universidad de Málaga
eat@lcc.uma.es

***Resumen*—El problema de selección de requisitos ha sido abordado en multitud de ocasiones en la literatura, tanto en su versión monoobjetivo, como biobjetivo. En su formulación usual, consiste en seleccionar un subconjunto de requisitos que se van a desarrollar en la siguiente versión de una aplicación software. Cada requisito tiene un coste y está asociado a uno o varios clientes. Para satisfacer a un cliente es necesario implementar todos los requisitos que desea. Cada cliente tiene asociado un valor para la empresa. Al resolver el problema se pretende maximizar la suma del valor de los clientes satisfechos sin superar el presupuesto de desarrollo de la siguiente versión. Existen restricciones adicionales asociadas a los requisitos que hacen el problema más complejo. Para un número elevado de variables y restricciones, los algoritmos exactos pueden resultar inviables y ser necesario recurrir a métodos heurísticos. En este trabajo proponemos el uso de la heurística *Construct, Merge, Solve and Adapt* para resolver el problema de selección de requisitos, y comparamos los resultados obtenidos con tres métodos, un algoritmo exacto y dos heurísticos, para instancias con un gran número de requisitos, clientes y dependencias.**

I. Introducción

El problema de la siguiente versión (*Next Release Problem, NRP*) tiene como objetivo satisfacer las necesidades de los clientes, minimizando el esfuerzo de desarrollo de la siguiente versión de una aplicación software. Fue inicialmente propuesto por Bagnall *et al*. [2]. La idea consiste en encontrar un subconjunto de requisitos o un subconjunto de clientes que maximice una cierta propiedad, como la satisfacción de los clientes, mientras se mantiene una restricción de cota superior sobre otra propiedad, que generalmente es el coste. Hay que tener en cuenta que el NRP solo tiene en cuenta la siguiente versión software. La planificación sobre varias versiones es más compleja y se estudia dentro del *Release Planning Problem* [8], que puede considerarse como una generalización del NRP. En [13] se pueden ver aproximaciones híbridas del *Release Planning Problem*.

Recientemente, Veerapen *et al*. [14], resolvieron el problema con técnicas ILP tomando como resolutor el software de IBM, CPLEX, usando unas instancias proporcionadas por Xuan, *et al*. [16]. Éstas fueron resueltas en pocos segundos cuando años atrás era impensable, obviamente debido a las capacidades de cómputo de la actual generación de computadores y los avances en ILP. NRP es NP-duro porque el problema de la mochila se puede reducir a NRP.

Existen dos enfoques para modelar el actual NRP monoobjetivo. Por un lado, se puede exigir que todas las demandas de los clientes sean satisfechas para poder incluir un requisito en la siguiente versión software [1], o que la satisfación del cliente sea parcialmente aceptable [7], es decir, que se permitan satisfacer solo ciertas demandas de cierto cliente, y esto suponga también un aumento de la función objetivo. En este trabajo consideramos el primer caso, es decir, será necesario satisfacer todas las demandas del cliente, de acuerdo con la formulación general dada por Veerapen et. al. [14].

Blum, *et al*. [3] publicaron un trabajo genérico sobre el uso de una metaheurística híbrida que resolvía problemas de optimización combinatoria. Concretamente, su trabajo es una instanciación específica dentro de un marco conocido en la literatura como *Generate-and-Solve* [9], y se conoce como *Construct, Merge, Solve & Adapt* (CMSA). La idea que proponen para resolver el problema consiste en generar una subinstancia del problema original, donde la solución óptima sea también una solución factible del problema original. La subinstancia se genera mediante la selección de ciertos componentes a tener en cuenta. Después se obtiene, mediante un resolutor exacto, una solución óptima para el subproblema. Esta solución obtenida podría estar muy alejada del verdadero óptimo. Sin embargo, los componentes que forman parte de la solución óptima del subproblema tienen un peso mayor para formar parte de la siguiente subinstancia, al menos durante un tiempo determinado. Así, se irán resolviendo subproblemas y guardando la mejor solución obtenida hasta el momento, hasta que se alcance un tiempo límite de ejecución previamente establecido. La descripción de este proceso general tiene también una fase de adaptación donde aquellos componentes que forman parte del subproblema y no han aparecido como parte de la solución óptima dentro de un número establecido de iteraciones, son desechados, lo cual no implica que puedan ser seleccionados posteriormente para formar parte de la solución. Este método híbrido, donde una heurística se mezcla con el uso de un resolutor exacto, tiene la ventaja de obtener soluciones de alta calidad (debido al uso de algoritmos exactos) a la vez que resultan rápidas, por resolver de manera exacta solo subproblemas de un tamaño reducido. La idea de resolución de instancias de un problema reducido ha sido ya explorada en varios trabajos, como por ejemplo en [4] y [10]. El método CMSA ha sido aplicado a algunos problemas, como el *Minimum Common String Partition* (MCSP) [12] o el *Minimum Covering Arborescence* (MCA) [15], entre otros, pero, hasta donde sabemos, no ha sido aplicado a NRP.

II. Formulación del NRP

Siguiendo la formulación general del NRP dada por [14], estamos interesados en maximizar el beneficio de los clientes estableciendo además un límite total de coste debido a los requisitos. Supongamos que disponemos de n posibles requisitos y m clientes. Sean $x_1, x_2, \ldots, x_n$ las variables binarias que indican la inclusión o no de cada requisito en la siguiente versión. Sean $y_1, y_2, \ldots, y_m$ las variables binarias que indican si la demanda de cada cliente ha sido completamente satisfecha o no. Sean $c_1, c_2, \ldots, c_n$ los costes asociados a los requisitos, y sean $w_1, w_2, \ldots, w_m$ los valores de beneficio asociados a los clientes. Llamemos b al límite de presupuesto para el desarrollo de la siguiente versión del software. Es posible que algunos requisitos tengan dependencias. En [17] se definieron tres posibles tipos de relaciones entre requisitos, que nosotros hemos adaptado dando nombre a cada una de ellas.

- Combinación ($r_i \bigodot r_j$): los requisitos r_i y r_j deben aparecer juntos en cualquier selección.
- Exclusión ($r_i \bigoplus r_j$): los requisitos r_i y r_j no pueden aparecer juntos en una selección.
- Implicación ($r_i \Rightarrow r_j$): El requisito r_i requiere que el requisito r_j se incorpore en la misma selección.

El primer conjunto de dependencias indica que existen requisitos que deben aparecer (o no) conjuntamente. Cuando hay en el problema dependencias de este tipo se puede modelar el problema de manera que todas las variables con dependencia mutua sean substituidas por una nueva variable. De esta forma, la nueva variable formará parte de la solución si y solo si el conjunto de variables que dependen mutuamente forma parte de la solución. Por ello, no vamos a tener en cuenta esta situación, y consideraremos que cada variable representa a un único requisito, o directamente a un conjunto de requisitos que debe aparecer conjuntamente. El segundo conjunto de dependencias establece que pueden existir situaciones en las que dos requisitos determinados no pueden ser desarrollados conjuntamente en la siguiente versión. En este trabajo, de acuerdo con las instancias usadas por [17], tampoco vamos a considerar este tipo de restricciones. El tercer conjunto de dependencias implica que ciertos requisitos necesitan de la inclusión de otros prerrequisitos previos. Un requisito puede necesitar previamente de la inclusión de otro, u otros, y éstos, a su vez, de otros requisitos. En este trabajo sí vamos a considerar este tipo de dependencias.

En relación a los clientes, cada uno de ellos demanda una serie concreta de requisitos, que deben ser completamente satisfechos para poder considerar como beneficio el peso que proporciona el cliente. No se admite satisfacción parcial de un cliente en este trabajo.

Sea P el conjunto de pares (i, j) donde el requisito i es un prerrequisito para el requisito j. Sea Q el conjunto de pares (i, k) donde el requisito i es demandado por el cliente k. Con todas estas consideraciones podemos modelar el NRP como un problema ILP de la siguiente manera:

$$\text{máx} \sum_{i=1}^{m} w_i y_i \tag{1}$$

sujeto a:

$$\sum_{i=1}^{n} c_i x_i \leq b \tag{2}$$

$$x_i \geq x_j \quad \forall (i,j) \in P \tag{3}$$

$$x_i \geq y_k \quad \forall (i,k) \in Q \tag{4}$$

$$x_i, y_k \in \{0,1\} \quad \forall i = 1, \ldots, n, \forall k = 1, \ldots, m \tag{5}$$

III. Método CMSA

Christian Blum *et al.* [3] establecieron un algoritmo genérico híbrido que fue usado para resolver diversos problemas combinatorios. Exponemos en el Algoritmo 1 el pseudocódigo de CMSA, que también será la base de nuestros algoritmos heurísticos, y explicamos brevemente su significado a continuación.

Algoritmo 1 *Construct, Merge, Solve and Adapt (CMSA)*

1: **input**: *instancia I, valores de los parámetros* n_a *y* age_{max}
2: S_{bsf} = `NULL`; $C' = \emptyset$
3: $age[c] = 0$ para todo $c \in C$
4: **mientras** *tiempo de CPU no excedido* **hacer**
5: **para** $i = 1$ hasta n_a **hacer**
6: *S=ProbabilisticSolutionGenerator(C)*
7: **para todo** $c \in S$ **y** $c \notin C'$ **hacer**
8: $age[c] = 0$; $C' = C' \cup c$
9: **fin para**
10: **fin para**
11: S'_{opt}=*ApplyExactSolver*(C')
12: **si** S'_{opt} es mejor que S_{bsf} **entonces** $S_{bsf} = S'_{opt}$
13: *Adapt* (C', S'_{opt}, age_{max})
14: **fin mientras**
15: **output**: S_{bsf}

El algoritmo parte de la suposición de que cada solución válida para una instancia I de un problema P dado, se puede representar mediante un subconjunto de un conjunto C de componentes. Este concepto de componente puede referirse a un conjunto de variables del problema, o representar algo distinto. Lo importante es que si tenemos a nuestra disposición todos los componentes para formar una solución y escogemos la mejor de ellas mediante un algoritmo exacto, resolvemos el problema original. Si solo disponemos de un subconjunto de C para formar soluciones, entonces estaremos resolviendo una subinstancia de la instancia original. El conjunto $S \subset C$ representa una solución, no necesariamente óptima, al problema original. Por otro lado, el subconjunto C' es un contenedor que va guardando todas las componentes seleccionadas y que serán utilizadas para resolver una subinstancia de *I*. Por tanto, también es $C' \subset C$. El bucle principal se ejecuta mientras no se alcance el máximo tiempo de ejecución permitido al algoritmo. Cada iteración del algoritmo se puede dividir en cuatro partes:

1. *Construct*: se generan probabilísticamente n_a soluciones (línea 6 del Algoritmo 1).
2. *Merge*: las componentes de dichas soluciones son añadidas al contenedor $C^{'}$ y, a cada componente c que ha sido nuevamente añadido al contenedor, se le inicializa su edad a cero.
3. *Solve*: se resuelve la subinstancia formada por las componentes del contenedor (línea 11). Si la solución encontrada es mejor que la obtenida hasta el momento, se actualiza.
4. *Adapt*: el contenedor se adapta en base a la solución obtenida al aplicar el resolutor exacto. Cada componente de la solución óptima del subproblema reinicializa su edad a cero, y el resto de componentes aumentan su edad en una unidad. Si se supera un límite máximo de edad preestablecido, age_{max} el componente es eliminado del contenedor, aunque podrá ser incluido nuevamente en una posterior iteración.

IV. CMSA para NRP

En esta sección detallamos nuestra propuesta de adaptación de CMSA para resolver el NRP. La estructura base del programa ya está expuesta en el apartado anterior y lo único que resta por explicar es el contenido de las funciones *ProbabilisticSolutionGenerator* y *ApplyExactSolver*. En primer lugar, es necesario definir el concepto de componente para resolver una instancia del NRP. De manera natural surgen dos posibilidades: que los componentes sean los requisitos o que los componentes sean los clientes. Si los componentes son los requisitos, se seleccionará aleatoriamente un subconjunto de ellos, que no viole la restricción de coste total. Si los componentes son los clientes, se seleccionan al azar también un número determinado de ellos, pero teniendo en cuenta que las demandas de todos ellos no pueden superar el coste máximo total permitido. El procedimiento *ProbabilisticSolutionGenerator* debe generar soluciones aleatorias y de gran calidad para servir de base al resolutor exacto. Para conseguir soluciones de calidad sería deseable obtener óptimos locales, para una cierta definición de vecindario. Debemos tener en cuenta que cuando se añade un requisito a la solución parcial, es necesario añadir también todos sus prerrequisitos. Explicamos a continuación como gestionar el uso de la función *ProbabilisticSolutionGenerator* para cada modalidad sobre el concepto de componente y después explicamos también la gestión del uso de prerrequisitos.

Si consideramos que un componente está formado por un requisito, estamos interesados en generar un vector binario $(x_1, \ldots, x_n)$ que sea un máximo local. Para ello, y teniendo en cuenta de que después trataremos de maximizar el beneficio de los clientes, hemos optado por ir seleccionando requisitos que vayan saturando las demandas de los clientes, ya que una selección totalmente aleatoria de requisitos podría implicar una baja satisfacción de los clientes, y por tanto, un bajo valor objetivo. Así, iremos seleccionando aleatoriamente clientes, y añadiendo todas sus demandas mientras el coste total de los requisitos y los prerrequisitos asociados a estos requisitos no sobrepasen el límite establecido. Si la selección de un cliente no satisface la condición de coste, se descarta y se pasa a otro, y este proceso se repite hasta que todos los clientes han sido analizados. En el siguiente paso, tras analizar a todos los clientes, si todavía no se ha alcanzado la cota máxima para el coste, se seleccionan de manera aleatoria más requisitos a formar parte de la solución, siempre que sea posible.

Si consideramos que un componente está formado por un cliente, estamos interesados en generar un vector binario $(y_1, \ldots, y_m)$ que sea un máximo local. Para ello procedemos de la misma forma que en el caso anterior, pero teniendo en cuenta que vamos seleccionando clientes, y no requisitos, y que no se seleccionan los requisitos finales para tratar de saturar la restricción de coste total.

Para calcular los prerrequisitos asociados a un requisito usamos inicialmente una estructura de datos consistente en un vector *Padre* $= (p_1, \ldots, p_n)$ donde $p_i = i$ si el requisito no tiene ningún prerrequisito asociado, y $p_i = j$ si el requisito i tiene como prerrequisito a j. De esta forma, resulta sencillo gestionar de manera recursiva el cálculo de todos los prerrequisitos asociados a un requisito. Sin embargo, es posible que un requisito tenga varios prerrequisitos asociados, por lo que el vector *Padre* pasa a convertirse en un vector donde cada elemento es una lista de elementos.

Algoritmo 2 *add-components(type)*

```
1:  aux = ∅; X = (0,...,0); S = (0,...,0)
2:  mientras queden clientes por seleccionar hacer
3:    i ← elegir cliente aleatoriamente y seleccionarlo
4:    pcost = 0
5:    para cualquier requisito r no incluido previamente
6:    y demandado por i hacer
7:      X[index(r)] = 1
8:      actualizar pcost ; aux = aux ∪ {index(r)}
9:      para todo prerrequisito r' de r no seleccionado
10:     previamente hacer
11:       X[index(r')] = 1
12:       actualizar pcost ; aux = aux ∪ {index(r')}
13:     fin para
14:   fin para
15:   si pcost ≤ b entonces
16:     totalcost = pcost ; S[i] = 1
17:   si no
18:     para todo r ∈ aux hacer X[r] = 0
19:   fin si
20: fin mientras
21: si type = requisitos entonces return (X)
22: si type = clientes entonces return (S)
```

En el Algoritmo 2 se muestra como se añaden los componentes a la solución parcial dependiendo de si son requisitos o clientes, y en el Algoritmo 3 se muestra la función *ProbabilisticSolutionGenerator*. El vector X es un vector binario de tamaño n que toma el valor 1 si y solo si el requisito correspondiente ha sido seleccionado. El vector *S* es un vector binario de tamaño *m* que toma el valor 1 si y solo si el cliente correspondiente es seleccionado. La variable *pcost* almacena

Algoritmo 3 *ProbabilisticSolutionGenerator(type)*

```
1: S = add-components(type)
2: si type = requisitos entonces
3:    marcar como requisito no seleccionado cualquier r tal
4:    que S[index(r)] = 0
5:    mientras queden requisitos por seleccionar hacer
6:       r ← elegir requisito aleatoriamente y seleccionarlo
7:       r' ← seleccionar prerrequisitos de r no seleccionados
8:       previamente
9:       si el coste añadido no supera el límite entonces
10:         S[r] = 1
11:         S[r*] = 1 para todo r* ∈ r'
12:         actualizar totalcost
13:      fin si
14:   fin mientras
15: fin si
16: return (S)
```

el coste parcial que será añadido, en su caso, a la variable *totalcost* que almacena el coste total acumulado y que nunca debe ser mayor que *b*. El conjunto aux guarda los índices de los requisitos y prerrequisitos seleccionados para formar parte de la solución. Si se viola la restricción de coste máximo, gracias al conjunto aux es posible restaurar el vector *X*.

A continuación explicamos la función *ApplyExactSolver*. Suponemos que disponemos de un resolutor que resuelve problemas ILP. De una manera muy sencilla podemos adaptar el contenido de esta función sin más que añadir una restricción al modelo, aquella que no tiene en cuenta los requisitos que no están en el contenedor o los clientes que no están en el contenedor, según el caso. El pseudocódigo de esta función puede verse en el Algoritmo 4.

Algoritmo 4 *ApplyExactSolver* $(C^{'})$

```
1: añadir restriccion Σ_i x_i = 0 para todo x ∉ C' al
   problema original
2: S'_opt ← obtener solución de la subinstancia
3: borrar la restricción añadida previamente
4: return (S'_opt)
```

V. Resultados computacionales

En esta sección mostramos los resultados computacionales para ciertas instancias del NRP que han sido generadas aleatoriamente. Dado que las instancias dadas por Xuan *et al*. [16] ya han sido resueltas de manera exacta en pocos segundos, no tiene sentido aplicar la heurística CMSA en este caso. Para ello, hemos creado instancias aleatorias con un número muy elevado de requisitos, clientes y dependencias, para poder así comprobar los resultados en relación a lo que proporciona la heurística CMSA en sus dos variantes (siendo los componentes requisitos o clientes) y la que proporciona el propio resolutor cuando resolvemos el problema exacto durante un tiempo prefijado.

Las instancias aleatorias han sido creadas teniendo en cuenta que no se produzcan bucles en la lectura de los prerrequisitos, lo cual se ha logrado usando estructuras *Union-Find* [5, cap. 21]. Los parámetros de las instancias generadas aleatoriamente son los siguientes: Se determinan n requisitos, m clientes y k dependencias entre requisitos. Los costes de los requisitos son números aleatorios enteros entre 1 y 10. Los pesos de los clientes son números aleatorios enteros entre 1 y 100. Cada cliente demanda un número aleatorio de requisitos que varía entre 1 y 8. Se han generado seis grupos de instancias, con los prefijos desde *a* hasta *e*. Para nombrar una instancia, se indica el nombre del grupo y los valores n, m, y k, separados por guiones entre ellos. Cada grupo representa una relación distinta entre los parámetros variables n, m y k. Así, se han considerado casos en que n puede ser mayor que m (grupos a y c), mucho mayor (grupos b y d) o similar (grupos e y f), al igual que m puede ser mayor que k (grupos a, b ó e) o similar (grupos (c, d ó f). Como límite total de coste para los requisitos se suele utilizar un coeficiente reductor sobre la suma total de los costes de todos los requisitos. En nuestro caso hemos optado por usar dos valores para este coeficiente: 0.3 y 0.7. El tiempo de ejecución máximo ha sido fijado en 60 segundos. Como resolutor hemos usado CPLEX 12.6.2.

Los experimentos han sido ejecutados bajo entorno Linux (Ubuntu 16.04 LTS) en un máquina HP con Intel Core 2 Quad (Q9400), velocidad de procesador 2.7 GHz y 4 GB de RAM, usando un máximo de 2GB de RAM y un único núcleo.

Los tres métodos usados, cuyos resultados serán comparados entre ellos son:

1. Algoritmo exacto usando el problema original y estableciendo tiempo límite de ejecución (*exact*)
2. Algoritmo CMSA utilizando los requisitos como componentes (*cmsa_r*) y parámetros $n_a = 5$, $age_{max} = 2$
3. Algoritmo CMSA utilizando los clientes como componentes (*cmsa_s*) y parámetros $n_a = 5$, $age_{max} = 2$

En el caso del algoritmo exacto se ha ejecutado 10 veces cada instancia, ya que la calidad de los resultados de CPLEX pueden variar en distintas ejecuciones debido a la diferente carga de la máquina. Las variantes de CMSA se han ejecutado 30 veces para cada instancia, un número considerablemente mayor dado el componente aleatorio de la heurística. En todos los casos se utiliza el valor promedio dado por las funciones objetivo. Los resultados para los distintos coeficientes reductores pueden verse en los Cuadros I y II. Las instancias marcadas con asterisco en la primera columna de los Cuadros I y II, son aquellas en las que el algoritmo exacto obtiene el óptimo global.

El análisis de los resultados para el coeficiente 0.3 del cuadro I muestra una variedad en cuanto a los resultados. Se ha marcado en negrita el mejor resultado para cada una de las instancias, teniendo en cuenta los valores promedio. Puede observarse que para aquellas instancias en las que el algoritmo exacto es mejor, el valor objetivo proporcionado por las heurísticas no es muy distante. Sin embargo, en aquellas instancias en las que la heurística es mejor, la diferencia en relación a la solución obtenida por el algoritmo exacto es,

Cuadro I
Valores objetivo promedio para los tres métodos propuestos, usando instancias generadas aleatoriamente durante 60 segundos y con presupuesto $b = 0.3 \sum_{i=1}^{n} c_i$.

instancia	*exact*	*cmsa_r*	*cmsa_s*
a20000-15000-4000	146018.0	**263387.1**	253287.2
a40000-30000-8000	287108.0	337520.6	**479641.2**
a80000-60000-16000	**562858.0**	506890.8	558241.5
a120000-90000-24000	**866539.0**	777432.0	852102.8
b20000-15000-15000	89272.0	103588.9	**178089.7**
b40000-30000-30000	166559.0	134943.8	**323032.7**
b80000-60000-60000	**340646.0**	276249.3	291939.1
b120000-90000-90000	360537.8	418888.4	**443369.2**
c20000-5000-4000*	**168572.0**	**168572.0**	**168572.0**
c40000-10000-8000*	**329473.0**	326309.9	325006.1
c80000-20000-16000	290985.0	301780.0	**371700.9**
c120000-30000-24000	449524.0	449683.2	**458293.3**
d20000-5000-5000*	**161697.0**	161696.7	**161697.0**
d40000-10000-10000*	**322819.0**	315514.8	313263.4
d80000-20000-20000	269804.0	281128.0	**295590.2**
d120000-30000-30000	391402.0	392500.8	**398782.4**
e20000-18000-4000	175321.0	**290948.8**	269240.6
e40000-36000-8000	355931.0	494322.9	**525250.4**
e80000-76000-16000	**770482.0**	654382.5	707105.5
e120000-108000-24000	**1078305.9**	926170.8	1004343.5
f20000-18000-18000	87259.0	107617.3	**199510.7**
f40000-36000-36000	177267.0	146402.7	**319233.9**
f80000-75000-75000	**361914.0**	295269.3	315160.4
f120000-108000-108000	-	435019.0	**459930.6**

Cuadro II
Valores objetivo promedio para los tres métodos propuestos, usando instancias generadas aleatoriamente durante 60 segundos y con presupuesto $b = 0.7 \sum_{i=1}^{n} c_i$.

instancia	*exact*	*cmsa_r*	*cmsa_s*
a20000-15000-4000	420936.0	417669.1	**438157.3**
a40000-30000-8000	856336.0	867279.3	**888971.7**
a80000-60000-16000	562858.0	557780.3	**564892.4**
a120000-90000-24000	866539.0	858235.1	**869302.4**
b20000-15000-15000	325142.0	330771.0	**343229.2**
b40000-30000-30000	650947.5	664219.9	**677113.9**
b80000-60000-60000	**1237348.0**	1164320.8	1220072.4
b120000-90000-90000	463548.6	438897.5	**733116.3**
c20000-5000-4000*	**249727.0**	**249727.0**	**249727.0**
c40000-10000-8000*	**492957.0**	**492957.0**	**492957.0**
c80000-20000-16000*	**993343.0**	**993343.0**	**993343.0**
c120000-30000-24000*	**1484724.0**	**1484724.0**	**1484724.0**
d20000-5000-5000*	**244459.0**	**244459.0**	**244459.0**
d40000-10000-10000*	**493255.0**	**493255.0**	**493255.0**
d80000-20000-20000*	**993806.0**	**993806.0**	**993806.0**
d120000-30000-30000*	**1485762.0**	**1485762.0**	**1485762.0**
e20000-18000-4000	433501.0	438125.7	**458748.1**
e40000-36000-8000	877810.0	887865.9	**924538.8**
e80000-76000-16000	770482.0	760625.6	**773035.3**
e120000-108000-24000	1077713.5	888231.8	**1082415.2**
f20000-18000-18000	87259.0	**352584.9**	319687.6
f40000-36000-36000	177267.0	704608.2	**719070.1**
f80000-75000-75000	**1383684.0**	1283686.5	1099710.2
f120000-108000-108000	-	-	**158342.4**

en ocasiones, muy elevada, como por ejemplo en la instancia *b120000-90000-90000*, donde *cmsa_s* proporciona un objetivo en torno al 67 % mejor que el algoritmo exacto. Esto justifica la utilidad del uso de estos nuevos algoritmos. También aparecen instancias en las que los resultados obtenidos por los tres métodos son iguales, como en el caso de *c20000-5000-4000*. Obsérvese también que en la instancia *f120000-108000-108000*, al tener gran cantidad de requisitos, clientes y dependencias, CPLEX es incapaz de encontrar ninguna solución durante el primer minuto de ejecución. Se conjetura que en torno a esos valores elevados de requisitos y clientes empieza a observarse un umbral a partir del cual parece ser más productivo el uso de la heurística CMSA en favor del uso de algoritmos exactos. Si comparamos *cmsa_r* con *cmsa_s*, podemos comprobar que en la mayoría de los casos *cmsa_s* proporciona unos resultados mejores, lo que nos permite deducir la importancia de elegir una buena definición de componente en el uso de la heurística CMSA, ya que los resultados para una misma instancia pueden variar considerablemente si cambiamos dicha definición.

Analizando ahora el Cuadro II para las mismas instancias de antes, pero aumentado el coeficiente reductor a 0.7, realizando el mismo número de ejecuciones que en el caso anterior, y calculando promedios, se observa como los resultados varían considerablemente. Para los grupos de instancias *c* y *d* los objetivos obtenidos son iguales para los tres casos (la solución óptima), por lo que no se detecta ninguna diferencia en el uso de los tres algoritmos. Sin embargo, para la mayoría de las restantes instancias, es claramente *cmsa_s* el método con el que se obtienen mejores resultados. Si suprimimos los grupos en el que los resultados son iguales, el método exacto solo es mejor en dos casos, mientras que *cmsa_r* solo lo es en un caso. Además, para la instancia *f120000-108000-108000*, ni el algoritmo exacto, ni *cmsa_r* consiguen encontrar una solución factible dentro del primer minuto de ejecución. Este hecho resalta nuevamente la importancia de una buena elección de componentes, que en este caso parece ser favorable a *cmsa_s*.

VI. Conclusiones y líneas futuras de investigación

Tras los resultados obtenidos con el uso de CMSA para resolver el NRP en instancias aleatorias con gran número de requisitos, clientes y dependencias, se puede conjeturar que el uso de la heurística *cmsa_s* es la más adecuada para resolver instancias grandes de NRP. Todos los experimentos se han realizado fijando los parámetros $n_a = 5$ y $age_{max} = 2$, aunque podrían haberse obtenido resultados diferentes (posiblemente mejores) para otra configuración de estos parámetros. Tras realizar el test de Wilcoxon con datos apareados a cada pareja de algoritmos y en cada una de las modalidades, los resultados de los cuadros III y IV muestran que existen diferencias significativas en cuanto al comportamiento del algoritmo *cmsa_s* respecto a los otros dos (al nivel de significancia del 5 %). Realizando nuevamente los tests pero estableciendo como hipótesis alternativa que *cmsa_s* es mejor, se obtienen los resultados que se muestran en el cuadro V. Acorde a [6]

el número de algoritmos usados para la comparación, que es tan solo de tres, no recomienda el uso del test de Friedman. En conclusión, podemos garantizar estadísticamente que, con estas instancias, el algoritmo *cmsa_s* es el mejor de los tres algoritmos analizados.

Una línea futura de investigación sobre este trabajo podría ser aplicar el paquete *iRace* [11], que permitiría obtener una configuración óptima de dichos parámetros dentro de un rango preestablecido. También se podría estudiar más a fondo el problema, con un rango más amplio de requisitos, clientes y dependencias, y estimar a partir de qué valores de estos parámetros se obtienen mejores resultados para un método u otro. Esto se podría conseguir también usando el paquete *iRace* y se matizaría en un algoritmo mediante combinación de los anteriores que sirviese como método híbrido que determina cuál es el mejor método a utilizar en función de los parámetros de entrada.

Cuadro III
TEST DE WILCOXON (DATOS APAREADOS) PARA CONTRASTAR SI HAY DIFERENCIA ENTRE LOS ALGORITMOS PROPUESTOS, $b = 0{,}3\sum_{i=1}^{n} c_i$

	exact	*cmsa_r*
cmsa_r	0.89110	-
cmsa_s	0.01629	0.00040

Cuadro IV
TEST DE WILCOXON (DATOS APAREADOS) PARA CONTRASTAR SI HAY DIFERENCIA ENTRE LOS ALGORITMOS PROPUESTOS, $b = 0{,}7\sum_{i=1}^{n} c_i$

	exact	*cmsa_r*
cmsa_r	0.97730	-
cmsa_s	0.01620	0.02449

Cuadro V
CONTRASTES UNILATERALES CON HIPÓTESIS ALTERNATIVA $H_1 : cmsa\_s > col$, SIENDO $col \in \{exact, cmsa\_r\}$

	exact	*cmsa_r*
$b = 0{,}3\sum_{i=1}^{n} c_i$	0.00814	0.00020
$b = 0{,}7\sum_{i=1}^{n} c_i$	0.00810	0.01225

También es posible generalizar el método CMSA para el caso multiobjetivo y aplicarlo inicialmente al NRP biobjetivo. De hecho, hemos trabajado en esta posibilidad, pero no hemos obtenido unos resultados iniciales satisfactorios, debido principalmente a dos problemas. El primero es que la elección de componentes como aquellos entes más válidos a formar parte de una solución, no viene a ser una tarea fácil para un problema biobjetivo, donde el espacio de soluciones es bidimensional y el comportamiento de las variables de decisión puede variar considerablemente para distintas zonas del espacio objetivo. El segundo problema encontrado es la forma de aplicar la función *ExactSolver* para el caso biobjetivo, ya que un barrido completo por el espacio objetivo en cada iteración, no solo consume demasiado tiempo de ejecución, sino que además las soluciones obtenidas suelen estar muy lejos de los valores óptimos, por lo que el prototipo de CMSA biobjetivo resulta poco eficaz. Hemos pensado en la posibilidad de subdividir el espacio objetivo en varias zonas y aplicar CMSA a cada una de ellas. Seguimos trabajando en este sentido.

VII. Agradecimientos

Este trabajo ha sido parcialmente financiado por el Ministerio de Economía y Competitividad (proyectos TIN2014-57341-R y TIN2017-88213-R) y por la Universidad de Málaga.

Referencias

[1] Philip Achimugu, Ali Selamat, Roliana Ibrahim, and Mohd Naz'ri Mahrin, *A systematic literature review of software requirements prioritization research*, Inform. and software technology **56** (2014), no. 6, 568–585.

[2] Anthony J. Bagnall, Victor J. Rayward-Smith, and Ian M Whittley, *The next release problem*, Information and software technology **43** (2001), no. 14, 883–890.

[3] Christian Blum, Pedro Pinacho, Manuel López-Ibáñez, and José A Lozano, *Construct, merge, solve & adapt a new general algorithm for combinatorial optimization*, Computers & Operations Research **68** (2016), 75–88.

[4] W. Cook and P. Seymour, *Tour merging via branch-decomposition*, INFORMS Journal on Computing **15** (2003), no. 3, 233–248.

[5] Thomas H Cormen, Charles E Leiserson, Ronald L Rivest, and Clifford Stein, *Introduction to algorithms*, MIT press, 2009.

[6] Joaquín Derrac, Salvador García, Daniel Molina, and Francisco Herrera, *A practical tutorial on the use of nonparametric statistical tests as a methodology for comparing evolutionary and swarm intelligence algorithms*, Swarm and Evolutionary Computation **1** (2011), no. 1, 3–18.

[7] J.J. Durillo, Y. Zhang, E. Alba, and Antonio J Nebro, *A study of the multi-objective next release problem*, Search Based Software Engineering, 2009 1st International Symposium on, IEEE, 2009, pp. 49–58.

[8] Des Greer and Guenther Ruhe, *Software release planning: an evolutionary and iterative approach*, Information and software technology **46** (2004), no. 4, 243–253.

[9] Janusz Kacprzyk, *Studies in computational intelligence, volume 153*, (2008).

[10] Gunnar W Klau, Ivana Ljubić, Andreas Moser, Petra Mutzel, Philipp Neuner, Ulrich Pferschy, Günther Raidl, and René Weiskircher, *Combining a memetic algorithm with integer programming to solve the prize-collecting steiner tree problem*, Genetic and Evolutionary Computation Conference, Springer, 2004, pp. 1304–1315.

[11] Manuel López-Ibáñez, Jérémie Dubois-Lacoste, Leslie Pérez Cáceres, Mauro Birattari, and Thomas Stützle, *The irace package: Iterated racing for automatic algorithm configuration*, Operations Research Perspectives **3** (2016), 43–58.

[12] Claudio N Meneses, Carlos AS Oliveira, and Panos M Pardalos, *Optimization techniques for string selection and comparison problems in genomics*, IEEE Engineering in Medicine and Biology Magazine **24** (2005), no. 3, 81–87.

[13] Günther Ruhe and Moshood Omolade Saliu, *The art and science of software release planning*, IEEE Software **22** (2005), 47–53.

[14] Nadarajen Veerapen, Gabriela Ochoa, Mark Harman, and Edmund K Burke, *An integer linear programming approach to the single and bi-objective next release problem*, Information and Software Technology **65** (2015), 1–13.

[15] V Venkata Rao, R Sridharan, et al., *The minimum weight rooted arborescence problem: weights on arcs case*, Tech. report, Indian Institute of Management Ahmedabad, Research and Publication Department, 1992.

[16] Jifeng Xuan, He Jiang, Zhilei Ren, and Zhongxuan Luo, *Solving the large scale next release problem with a backbone-based multilevel algorithm*, IEEE Transactions on Software Engineering **38** (2012), no. 5, 1195–1212.

[17] Yuanyuan Zhang, Mark Harman, and S Afshin Mansouri, *The multi-objective next release problem*, Proceedings of the 9th annual conference on Genetic and evolutionary computation, ACM, 2007, pp. 1129–1137.