Problemas de optimización combinatoria: una propuesta que combina algoritmos genéticos y metaheurísticas

Lucas Benjamín Cicerchia[1 2], Leonardo Martin Esnaola[1], Juan Pablo Tessore[1 2], Hugo Dionisio Ramón[1 3], Claudia Cecilia Russo[1 3], Cristian Alejandro Martínez[4]

[1]Instituto de Investigación y Transferencia en Tecnología (ITT), Escuela de Tecnología, Universidad Nacional del Noroeste de la Provincia de Buenos Aires (UNNOBA), Centro Asociado a la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC)

[2] Becario de la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC)

[3] Investigador Asociado de la Comisión de Investigaciones Científicas de la Provincia de Buenos Aires (CIC)

[4] Departamento de Informática, Facultad de Ciencias Exactas, Universidad Nacional de Salta, Argentina

{lucas.cicerchia, leonardo.esnaola, juanpablo.tessore, hugo.ramon, claudia.russo}@itt.unnoba.edu.ar, cmartinez@unsa.edu.ar

***Abstract*— Timetabling se refiere al conjunto de problemas de optimización combinatoria que intentan asignar recursos, sean aulas, docentes o intervalos de tiempo, para distintas necesidades de estudiantes, cursos y exámenes. El presente trabajo se ocupa de una de las variantes de este problema, que busca agendar exámenes a distintos intervalos de tiempo, cumpliendo con las restricciones de que ningún alumno debe asistir a más de un examen en el mismo momento y que, en la medida de lo posible, tenga el mayor tiempo libre entre las evaluaciones. Los intervalos de tiempo no tienen restricciones en cuanto a la cantidad de exámenes que puedan asignárseles. Como estrategia de resolución se utiliza un algoritmo genético, que combina diversas heurísticas para la construcción de soluciones factibles que conforman la población inicial con la que trabaja el algoritmo. Dichas heurísticas fueron seleccionadas priorizando la calidad de la solución construida. También se definieron operadores de cruzamiento y mutación particulares, con el objetivo de mejorar la calidad de la solución resultante del proceso genético o, al menos, evitar la generación de soluciones no factibles. Mediante el algoritmo propuesto se alcanzaron soluciones relativamente buenas con pocas evaluaciones de la función objetivo y en un tiempo de ejecución razonable.**

***Index Terms*— Timetabling, Metaheurísticas, Algoritmos Genéticos.**

I. Introducción

La asignación de exámenes y cursos, también conocida *timetabling*, es de gran interés para la comunidad educativa. Normalmente esta tarea requiere de mucho tiempo si es realizada de manera manual y es muy común que los resultados obtenidos se encuentren lejos del óptimo.

Debido a que, para instituciones de un tamaño moderado, la cantidad de variables a tener en cuenta es considerable, la obtención del óptimo no es una alternativa, ya que el tiempo que demandaría encontrarlo la hace inviable. Como consecuencia, se han desarrollado un conjunto de métodos que permiten encontrar una solución lo suficientemente buena en un tiempo razonable.

Dentro de estos métodos pueden mencionarse:

- Métodos basados en programación matemática, como por ejemplo programación lineal [1]. Sin embargo, la posibilidad de utilizarlos está limitada por el tamaño de la solución.
- Métodos basados en coloreo de grafos [2], en este enfoque y para el caso de *examination timetabling*, los exámenes se representan con nodos y los conflictos con vértices entre los mismos. Existen diversas técnicas dentro de este enfoque que pueden ser utilizadas por sí solas o combinadas.
- Métodos de *clustering* o agrupamiento [3], estos intentan

generar grupos libres de conflictos y luego intercambiarlos entre los *timeslots* disponibles para intentar cumplir mejor con algunas de las restricciones del problema.

• Métodos metaheurísticos, estos han ganado popularidad en las últimas décadas. Entre ellos se pueden destacar métodos de búsqueda local como *Hill Climbing* [4], *Tabu Search* [5], *Simulated Annealing* [6], Algoritmos Genéticos [7] o Algoritmos de Colonia de Abejas [8]. Estos, en general, comienzan con una o más soluciones iniciales y emplean diferentes técnicas de búsqueda para no quedarse estancados en óptimos locales.

El presente trabajo propone la utilización de un algoritmo genético, que construye su población inicial utilizando una combinación de diferentes heurísticas mencionadas en [9].

El resto del documento se estructura de la siguiente manera, en la sección II se describe el problema, en la sección III se describe el método constructivo de soluciones iniciales que emplea diferentes heurísticas, en la sección IV se presenta el algoritmo genético utilizado, en la sección V los resultados obtenidos y en la sección VI las conclusiones.

II. El problema

En este trabajo se analiza una instancia particular del problema de *timetabling,* que según [10] consiste en asignar un conjunto de exámenes, cada uno con un grupo específico de estudiantes, a un conjunto dado de intervalos de tiempo. En dicha instancia, denominada "*uncapacitated university examination timetabling problem*", no existen limitaciones en cuanto a capacidad de las aulas ni disponibilidad de los docentes.

En general, para evaluar la solución obtenida existen dos tipos de restricciones [11]: restricciones duras, que deben cumplirse obligatoriamente, y restricciones blandas, que tienen que cumplirse en la medida de lo posible. Una solución con mayor grado de cumplimiento de estas últimas será de mayor calidad.

Las restricciones duras consisten en que ningún alumno puede asistir a más de un examen en un mismo intervalo de tiempo y, además, que ningún examen puede asignarse a más de un intervalo de tiempo. Una solución que cumpla con estas restricciones será considerada factible y será considerada infactible o no factible, en caso contrario.

Las restricciones blandas, por su parte, consisten en maximizar el tiempo medio entre exámenes de los alumnos. Una solución que satisfaga mejor este tipo de restricciones, será considerada de mayor calidad.

Para definir formalmente el problema se recurre a las siguientes fórmulas:

$$\min Z = \sum_{n=1}^{N} \sum_{p=1}^{P} C_{np}\, t_{np} \tag{2.1}$$

Sujeto a:

$$\sum_{p=1}^{P} t_{np} = 1\,, donde\ n\ \in \{1, \dots, N\} \tag{2.2}$$

$$\sum_{n=1}^{N-1} \sum_{m=n+1}^{N} \sum_{p=1}^{P} t_{np} t_{mp} Y_{nm} = 0 \tag{2.3}$$

Siendo:

- N es el número de exámenes.
- P es el número de timeslots.
- t_{np} es 1 si el examen n es asignado al *timeslot* p.
- C_{np} es el costo de asignar el examen n al *timeslot* p.
- Y_{nm} = 1 si el examen n colisiona con el examen m (es decir, si los exámenes n y m comparten alumnos) y 0 en caso contrario.

A su vez la función que determina el costo de una solución y permite comparar la calidad entre distintas soluciones, está basada en la siguiente ecuación:

$$F_c = \frac{\sum_{i=1}^{N-1} \sum_{j=i+1}^{N} c_{ij}\ prox(i,j)}{M} \tag{2.4}$$

donde,

$$prox(i,j) = \begin{cases} 2^{5-|t_i - t_j|}, & si\ 1 < |t_i - t_j| < 4 \\ 0, & de\ otra\ manera. \end{cases}$$

Siendo:

- N es el número de exámenes.
- C_{ij} es la cantidad de alumnos en común entre el examen i y el examen j.
- M es el número de estudiantes.
- t_i es el número de *timeslot* al que está asignado el examen i.
- t_j es el número de *timeslot* al que está asignado el examen j.

Esto implica que, cuanto más cercanos estén los exámenes que deben rendir los alumnos (definido por la función "*prox(i, j)*"), mayor será el costo de la solución propuesta. La solución ideal es aquella cuyo costo fuera cero, lo que implica que siempre existen cinco o más intervalos de tiempo de espera entre un examen y el siguiente para todos los alumnos. Entonces, se intentan encontrar aquellas soluciones cuyo costo tienda a cero, y cuanto más cerca de este valor estén tales soluciones serán de mejor calidad.

III. Solución propuesta

Como solución propuesta, se plantea la utilización de algoritmos genéticos [12] para el mejoramiento de la calidad de las soluciones factibles generadas como población inicial, mediante el método constructivo presentado en la siguiente subsección.

A. Método constructivo de soluciones

Para la representación de la solución propuesta se utiliza un vector de tamaño N, donde N es la cantidad de exámenes para la instancia a resolver. Cada una de las posiciones de dicho vector tiene un valor entre 0 y P-1, donde P es la cantidad de *timeslots* disponibles en el problema. Este vector solución, presentado en la Figura 1, será el cromosoma utilizado para el algoritmo genético.

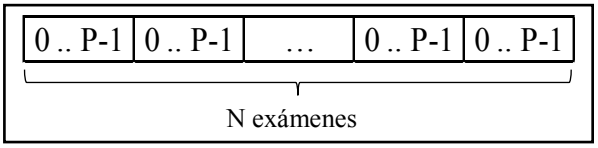

Figura 1. Formato del cromosoma

Con esta representación, el número de combinaciones posibles es: P^N, dado que son P posibles valores (los timeslots) y N posiciones donde asignar esos valores (los exámenes). Así, por ejemplo, para la instancia "car-f-92" que tiene 542 exámenes y 32 posibles timeslots, las combinaciones posibles son $32^{542} = 6.184 \times 10^{815}$.

1) Análisis de metaheurísticas en la construcción de la solución inicial

Para la construcción de la solución inicial se utilizaron distintos enfoques con resultados variados. En las primeras pruebas se intentó construir soluciones asignando los exámenes a *timeslots* en orden ascendente. Mediante este enfoque, y luego de corridas de más de 2 horas, no se lograron construir soluciones factibles, por lo cual se decidió modificar el algoritmo para que asignara los exámenes en orden aleatorio. Sin embargo, los resultados tampoco fueron satisfactorios.

Según algunas teorías de asignación secuencial basadas en grafos [13], existen diversos criterios para determinar el orden en el cual se asignan los exámenes a los *timeslots*. Estos ordenamientos permiten construir soluciones iniciales factibles y de mejor calidad. Los mismos se detallan a continuación:

- *Largest Enrollment* (LE): Los exámenes se ordenan de manera decreciente según la cantidad de alumnos.
- *Largest Weighted Degree* (LWD): Se asignan primero los exámenes con mayor cantidad de estudiantes que tienen conflictos con otras evaluaciones.
- *Largest Degree* (LD): Los exámenes se ordenan de manera decreciente según los que más conflictos tengan con otros exámenes.
- *Saturation Degree* (SD): Los exámenes se ordenan de manera ascendente según la cantidad de timeslots disponibles, es decir, los timeslots que no producen colisiones.
- *Highest cost* (HC): Se asignan primero los exámenes con mayor costo de asignación.
- *Lowest cost* (LC): Se asignan primero los exámenes con menor costo de asignación.

Los primeros tres ordenamientos pueden establecerse de antemano, mientras que en los últimos tres el orden de asignación debe calcularse paso a paso, debido a que cada asignación realizada en un paso previo afectará las asignaciones que pueden realizarse posteriormente.

Estos métodos permiten obtener soluciones factibles, sin embargo [13] su función de costo muestra que el mismo está bastante alejado del correspondiente a las mejores soluciones reportadas a la fecha. En general deben aplicarse criterios de mayor complejidad para obtener una buena solución inicial. La combinación de los métodos mencionados, llamados heurísticas de bajo nivel, tienden a producir mejores resultados que si se las aplica de manera individual.

A partir de lo anterior, se probaron tres estrategias distintas para la construcción de la solución inicial, las cuales son presentadas en las siguientes subsecciones.

a) Combinación de heurísticas de bajo nivel

A continuación se describe la variante seleccionada para construir las soluciones iniciales. La misma fue elegida debido a que, respecto de otras pruebas realizadas, conseguía obtener mejores resultados, esto es, que la calidad de la solución construida resultaba superior.

Esta combinación secuencial de heurísticas de bajo nivel, utiliza LD como primera estrategia de asignación y SD, como estrategia secundaria. Según la bibliografía [14], la heurística SD tiende a privilegiar la factibilidad de la solución por sobre su calidad. Por el contrario, LD pone por encima la calidad de las soluciones pero descuida la factibilidad. Teniendo en cuenta lo anterior, se realizaron pruebas combinando ambas heurísticas en la construcción de soluciones iniciales. Para ellas se estableció un parámetro "*i*", que indica el número de iteraciones en las que se utiliza LD para seleccionar el examen a asignar a partir del inicio. Asimismo, se determinó que un valor muy bajo de "*i*", tiende a producir soluciones factibles pero de baja calidad, mientras que un valor muy alto de "*i*" suele producir soluciones no factibles. Según las pruebas realizadas, con un valor de "*i*" igual al número de *timeslots* por 1.5, se obtienen en su mayoría soluciones factibles de calidad aceptable.

Se realizaron pruebas preliminares para la instancia CAR-92, las mismas coinciden con los resultados expuestos en la bibliografía [9] [15].

b) Utilización de heurísticas de bajo nivel con métodos iterativos

Otra estrategia [15], considera un método iterativo en el cual la importancia relativa de cada examen a asignar se modifica con cada iteración.

Como orden inicial se toma el generado por la heurística de bajo nivel LD. A partir del mismo se intentan asignar exámenes a *timeslots*, si alguno de ellos no puede ser agendado sin colisiones se lo saltea, y se aumenta su prioridad de asignación de la siguiente manera:

$$dificultad(e,i) = heuristica(e) + modheuristico_{e,i} \quad (3.1)$$

Una vez que se obtiene una solución inicial válida se establece el examen que aporta mayor penalidad a la solución y se intenta modificar la misma para que el costo de cada examen

sea menor al máximo. Para esto se modifican las prioridades de la siguiente manera:

$$modheuristico_{e,i+1} = modheuristico_{e,i} \text{ si el examen se puede asignar} \quad (3.2)$$

$$modheuristico_{e,i+1} = modheuristico_{e,i} + 1 \text{ si el examen no se puede asignar} \quad (3.3)$$

Donde para las ecuaciones 3.1, 3.2 y 3.3:

- e representa el examen a asignar.
- i representa la iteración actual.

En el caso que luego de 100 iteraciones no se logre bajar el costo máximo, el examen que aporta el mismo es introducido en una "lista tabú", de manera tal que no sea considerado por las siguientes 250 iteraciones. Este método finaliza si se llega a un costo mínimo preestablecido o si se alcanza el número máximo de iteraciones.

c) Combinación jerárquica de heurísticas de bajo nivel

En este caso se combinan las diferentes heurísticas de bajo nivel descriptas previamente en esta sección. De esta manera, se construyen soluciones asignando los exámenes según una heurística primaria, si la heurística determina que hay más de un examen posible, entonces se utiliza otra secundaria para desempatar, y en caso de persistir el empate, el examen se selecciona mediante una heurística terciaria o bien con un criterio aleatorio.

Al probar todas las combinaciones posibles, esto es $p\binom{6}{3}$ heurísticas, más $p\binom{6}{2}$ heurísticas para el caso del criterio aleatorio como terciario, surge que el mejor desempeño en la construcción de soluciones de calidad se consigue con la combinación SD-LWD-HC, de manera tal que esta es la que se utiliza para la posterior generación de soluciones iniciales. El proceso anterior se presenta en la Figura 2, como Algoritmo 1.

Algoritmo 1 Obtención de heuristicas para algoritmo constructivo

Entrada: Lista de heurísticas $heuristicas = \{sd, hc, ld, lwd, le, lc\}$.
Salida: Generador que obtiene mejores soluciones iniciales.

```
1:  para heuristica1 in heuristicas hacer
2:    para heuristica2 in heuristicas hacer
3:      si heuristica1 ≠ heuristica2 entonces
4:        para heuristica3 in heuristicas hacer
5:          si heuristica3 ≠ heuristica1 and heuristica3 ≠ heuristica2 entonces
5:            generador_actual = generador(heuristica1, heuristica2, heuristica3)
6:            si mejor_generador = NULL OR mejor_generador > generador_actual entonces
6:              mejor_generador = generador_actual
7:            fin si
8:          fin si
9:        fin para
10:     fin si
11:   fin para
12: fin para
13: devolver mejor_generador
```

Figura 2. Algoritmo que prueba las combinaciones de heurísticas

Se realizaron pruebas preliminares con las tres estrategias mencionadas, las mismas coinciden con los resultados expuestos en la bibliografía [9] [15].

Haciendo uso de esta combinación de heurísticas se crea la función que permite construir soluciones iniciales, según el Algoritmo 2, presentado en la Figura 3, donde "nueva_solucion()" crea un vector de tamaño N inicializado en "-1", siendo N el número de exámenes para la instancia.

Algoritmo 2 Obtención de solución inicial

Entrada: Lista de heurísticas $heuristicas = \{heuristica\_1, heuristica\_2, heuristica\_3\}$.
Entrada: Cantidad de exámenes $num\_examenes$.
Salida: Solución inicial factible.

```
mientras factible(solucion) = FALSE OR solucion = NULL hacer
  solucion = nueva_solucion();
  para examen in examenes hacer
    examenes_a_asignar = heuristica_1();
    si examenes_a_asignar.largo() > 0 entonces
      si examenes_a_asignar.largo() = 1 entonces
        asignar_examen(solucion, examen);
      si no
        examenes_a_asignar = heuristica_2(examenes_a_asignar);
        si examenes_a_asignar.largo() > 0 entonces
          si examenes_a_asignar.largo() = 1 entonces
            asignar_examen(solucion, examen);
          si no
            examenes_a_asignar = heuristica_3(examenes_a_asignar);
            si examenes_a_asignar.largo() > 0 entonces
              si examenes_a_asignar.largo() = 1 entonces
                asignar_examen(solucion, examen);
              si no
                asignar_examen_al_azar(solucion, examenes_a_asignar);
              fin si
            si no
              interrumpir;
            fin si
          fin si
        si no
          interrumpir;
        fin si
      fin si
    si no
      interrumpir;
    fin si
  fin para
fin mientras
devolver solucion;
```

Figura 3. Algoritmo constructivo de soluciones iniciales

B. Mejoramiento de las soluciones mediante algoritmos genéticos

La estrategia adoptada propone el uso de un algoritmo genético con las siguientes consideraciones:

- Cada posible solución que constituye la población inicial se construye mediante el uso de una combinación de heurísticas, buscando obtener desde el inicio soluciones de calidad aceptable.
- El operador de selección elegido es el operador torneo de tamaño 3.
- El operador de cruzamiento y mutación es "a medida", de manera tal que del proceso de cruzamiento y mutación se obtengan soluciones factibles.

Los algoritmos genéticos [16] se basan en los postulados de la teoría de la evolución biológica. Cada solución, llamada en este contexto individuo, se cruza y compite con otras soluciones y, eventualmente, se muta. Los algoritmos genéticos son métodos de optimización, donde el objetivo es maximizar o minimizar una función objetivo, o incluso varios componentes de la misma. Por ejemplo, se podría querer, en un determinado contexto, reducir el costo de una ruta pero también aumentar la

cantidad de kilómetros recorridos, por poner un ejemplo. Esta función objetivo, en el contexto de los algoritmos genéticos, suele denominarse el *fitness* de la solución, y a cada solución se la denomina individuo.

Se comienza con un conjunto inicial de individuos, denominado población, es importante destacar que no hay un número determinado o recomendado para utilizar como población inicial, se sugiere realizar distintas pruebas, pero hay que tener presente que se intenta cubrir el mayor espacio de búsqueda posible. Luego, estos individuos compiten entre sí para cruzarse (o reproducirse haciendo un paralelismo con la teoría de la evolución biológica), su calidad se evalúa a través de su *fitness*, y son seleccionados para cruzarse creando una nueva descendencia. En general se elegirán los mejores individuos (los más aptos), de manera tal que su descendencia también sea apta, aunque no necesariamente sea así. Cada descendiente, es decir la nueva generación, puede recibir mutaciones (como sucede en la naturaleza donde las especies van evolucionando y mutando). Por lo general, tanto la probabilidad de que se produzca una cruza, como la probabilidad de que tenga lugar una mutación, son parámetros del algoritmo genético y, nuevamente, no hay una recomendación sobre qué valores utilizar. El algoritmo tratará de obtener los mejores individuos, generación tras generación, para en última instancia quedarse con los *n* mejores, siendo *n* la cantidad de individuos con mejor *fitness* que se desea obtener.

Esta breve, y general, descripción sobre los algoritmos genéticos presenta los lineamientos básicos, la teoría detrás de la práctica. A continuación se introduce el *framework* utilizado para implementar el algoritmo genético, para luego avanzar sobre los operadores de cruzamiento (*crossover*) y mutación (*mutation*).

El *framework* utilizado para implementar el algoritmo propuesto se denomina Deap, en su versión 1.0 (estable) [17].

La documentación disponible para Deap presenta una guía paso a paso de cómo escribir un programa genético mediante este *framework*. Deap es muy flexible, define algunos tipos que pueden ser usados para representar a los individuos del problema, pero también da la posibilidad de construir los propios. Lo mismo sucede con los operadores de cruzamiento, mutación y selección. Con esta escueta presentación del *framework* se ahonda más en algunas cuestiones: la representación de los individuos, la población inicial, los operadores de selección, de cruzamiento y de mutación.

1) Representación de los individuos

El primer paso consiste en definir una representación para los individuos, en este caso se utiliza el formato propuesto en la sección III. Luego, al individuo se le asocia un *fitness*, en realidad es la función que invocará el *framework* para evaluar la calidad del mismo. Por último, es necesario especificar si se trata de un problema de minimización o maximización que, para este caso, es del primer tipo, pues el objetivo es obtener soluciones (o individuos) que tengan el costo más bajo posible, lo cual indica que satisfacen mejor las restricciones blandas.

2) Construcción de la población inicial

Los algoritmos genéticos comienzan construyendo una población inicial de individuos, que luego serán cruzados y/o mutados. Para construir esta población se le indica al *framework* que debe llamar repetidamente al método constructivo presentado previamente en la Figura 3 como Algoritmo 2, dicho método es llamado tantas veces como valor de población inicial se indique.

3) Selección del individuo para la próxima generación

El método escogido es uno de entre los métodos provistos por el *framework*, concretamente la selección por torneo de *K* individuos. Básicamente se escogen *K* (este valor fue ajustado en 3) individuos al azar y se selecciona el que tenga mejor *fitness*.

4) Evolución de los individuos

Debido a la necesidad de conocer de antemano las posibles colisiones al momento de aplicar los operadores de cruzamiento y/o mutación, se decidió construir una estructura para tal fin, denominada matriz de asignaciones.

a) Matriz de asignaciones

Es una matriz binaria de T columnas por N filas, donde T es el número de *timeslots* y N es el número de exámenes. Esta estructura se genera para cada individuo al momento de la construcción de la población inicial.

El proceso es el siguiente:

1- Se genera una matriz de N filas por T columnas con todas sus celdas inicializadas en 0.
2- Cuando se asigna un examen *n* a un timeslot *t* de la solución, también se le asigna un 1 a las celdas de la columna *t* de la matriz de asignaciones en las filas que correspondan a exámenes que entren en conflicto con n. Este proceso se realiza para todos los exámenes de la solución.

Esta matriz permite conocer rápidamente los exámenes que pueden ser asignados a un determinado *timeslot*, dado que estos corresponden a las celdas que contienen un 0.

b) Operador de cruzamiento

Se optó por definir un operador de cruzamiento o *crossover* a medida, ya que los operadores habituales, como el crossover de uno o dos puntos, arrojaba como resultado individuos no factibles, debido a que podría darse el caso de que exámenes con estudiantes en común quedasen asignados al mismo *timeslot*.

El operador definido, según se presenta en la Figura 4, como Algoritmo 3, consiste en lo siguiente: se genera la matriz de asignaciones para los dos individuos que participan de la cruza. El cruzamiento se realiza uno a uno, tomando el i-ésimo componente del individuo 1 (que es el *timeslot* asociado al i-ésimo examen del individuo 1), verificando si puede asignarse al i-ésimo examen del individuo 2, para lo cual se consulta su matriz de asignaciones. Si el cambio resulta posible se realiza. Lo mismo se repite, pero tomando ahora componente a componente del individuo 2 y viendo si el cambio es posible

sobre el individuo 1. Esta cruza de soluciones dará como resultado dos individuos hijos factibles, distintos de los individuos padres (si es que algunos de los cambios fueron posibles) caso contrario los hijos serán una copia exacta de sus padres.

Algoritmo 3 Operador de cruzamiento

Entrada: *Individuos a cruzar* = {*individuo*1, *individuo*2}.
Salida: *Resultado del cruzamiento* = {*individuo*1′, *individuo*2′}.
  *individuo*1′ = *individuo*1;
  *individuo*2′ = *individuo*2;
  *Construir matriz de asignaciones de individuo*1′;
  *Construir matriz de asignaciones de individuo*2′;
  para *examen in individuo*1 **hacer**
    *timeslot_ind*1 = *individuo*1[*examen*];
    *timeslot_ind*2 = *individuo*2[*examen*];
    si *timeslot_ind*1 ≠ *timeslot_ind*2 **entonces**
      Actualizar matriz de asignaciones de individuo2' quitando asignació
      timeslot_ind2;
      si *individuo*2′[*examen*] = *timeslot_ind*1 *es posible* **entonces**
        *individuo*2′[*examen*] = *timeslot_ind*1;
        Actualizar matriz de asignaciones de individuo2' con timeslot_inc
        para examen actual;
      si no
        Restaurar matriz de asignaciones de individuo2' con timeslot_ind2 pa
        examen actual;
      fin si
    fin si
  fin para
  para *examen in individuo*2 **hacer**
    *timeslot_ind*1 = *individuo*1[*examen*];
    *timeslot_ind*2 = *individuo*2[*examen*];
    si *timeslot_ind*1 ≠ *timeslot_ind*2 **entonces**
      Actualizar matriz de asignaciones de individuo1' quitando asignació
      timeslot_ind1;
      si *individuo*1′[*examen*] = *timeslot_ind*2 *es posible* **entonces**
        *individuo*1′[*examen*] = *timeslot_ind*2;
        Actualizar matriz de asignaciones de individuo1' con timeslot_inc
        para examen actual;
      si no
        Restaurar matriz de asignaciones de individuo1' con timeslot_ind1 pa
        examen actual;
      fin si
    fin si
  fin para
  devolver *individuo*1′, *individuo*2′;

Figura 4. Algoritmo de cruzamiento

Por su parte, la Figura 5 ejemplifica gráficamente el proceso de cruzamiento. La primera parte exhibe cómo se cruza el individuo 1 con el individuo 2, en este caso se verifica si el *timeslot* asignado al examen 0 del individuo 1, cuyo valor es "T1" puede ser asignado al mismo examen del individuo 2, lo cual es posible por lo que dicho componente pasa a ser parte del primer hijo. A continuación se hace lo mismo con el examen 1 del individuo 1 cuyo valor es "T5" y, como puede verse, no es posible el cambio por lo que prevalece el valor asignado previamente: "T7" del individuo 2. Este proceso se repite hasta haber analizado todos los exámenes del individuo 1. Luego, se hace lo propio para generar el segundo hijo, lo cual es mostrado en la segunda parte de la figura, pero tomando como primer individuo al individuo 2.

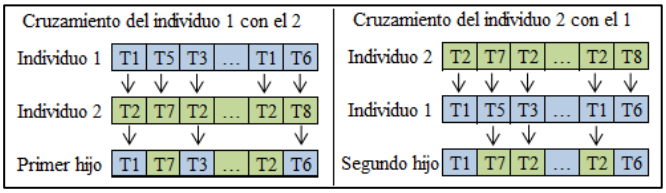

Figura 5. Funcionamiento del operador de cruzamiento

c) Operador de mutación

En primer lugar se genera la matriz de asignaciones del individuo que participa de la mutación. Para realizar el proceso de mutación se recorren uno a uno todos los exámenes del individuo, se obtiene una lista de posibles asignaciones para el examen actual (una lista de *timeslots* posibles que hagan que la solución siga siendo factible), esto calculado a partir de la matriz de asignaciones, y se escoge uno al azar. Además se tomó la política de que no sólo se realizaría el cambio si resultaba posible (es decir si la solución resultaba factible) sino que, además, dicho cambio se realizaría si mejoraba el *fitness* del individuo mutado. Caso contrario, se repetiría el proceso de escoger otro aleatoriamente, como máximo tres veces. El algoritmo de mutación descripto previamente se presenta a continuación en la Figura 6, como Algoritmo 4.

Algoritmo 4 Operador de mutación

Entrada: *Individuo a mutar y número de cambios* = {*individuo*, *cambios*}.
Salida: *Individuo mutado* = {*individuo*}.
  Construir matriz de asignaciones de individuo′;
  indices = *lista*{0..*N* − 1}; (*siendo N la cantidad de exámenes*)
  Ordenar alatoriamente indices;
  indices_elegidos = *indices*{*cambios*}; (*obtener tantos indices como cambios*)
  para *indice_aleatorio in indices_elegidos* **hacer**
    timeslot_original = individuo[indice_aleatorio];
    timeslots_posibles = listado de timeslots que pueden asignarse sin conflictos;
    si *timeslots_posibles* ≠ *vacío* **entonces**
      timeslot_posible = *opción aleatoria de timeslots_posibles*;
      costo_con_timeslot_original = *obtener costo con timeslot_original*;
      costo_con_nuevo_timeslot = *obtener costo con timeslot_posible*;
      intento = 0;
      mientras ((*intento* < 3) *AND* (*costo_con_nuevo_timeslot* ≥ *costo_con_timeslot_original*)) **hacer**
        timeslot_posible = opción aleatoria de timeslots_posibles;
        costo_con_nuevo_timeslot = obtener costo con timeslot_posible;
        intento = intento + 1;
      fin mientras
      si *costo_con_nuevo_timeslot* ≤ *costo_con_timeslot_original* **entonces**
        individuo[indice_aleatorio] = timeslot_posible;
        Actualizar matriz de asignaciones de individuo quitando asignación
        con timeslot_original;
        Actualizar matriz de asignaciones de individuo agregando asignación
        con timeslot_posible;
      fin si
    fin si
  fin para
  devolver individuo;

Figura 6. Algoritmo de mutación

Como resultado de la mutación propuesta se obtiene un individuo parecido al original, pero con algunos cambios que hacen que su *fitness* mejore. O, si ningún cambio fue posible, el individuo mutado es igual al original.

La Figura 7 ejemplifica el proceso gráficamente.

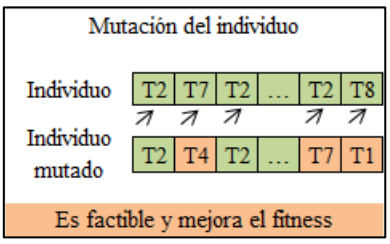

Figura 7. Funcionamiento del operador de mutación.

En este caso se verifica para el examen 0 del individuo a mutar, cuyo valor actual es "T2", si es posible cambiar su vlor por otro de manera tal de mejorar el *fitness*. Según la figura valor "T2" se mantiene por lo que, o bien no existe otro *timeslot* para asignar en esa posición o ninguno de ellos mejora el *fitness*. Luego, se hace lo mismo con el examen 1 cuyo valor actual es "T7" y puede verse que "T4" es un valor posible y mejora al fitness, por lo que el cambio se realiza. Este proceso se repite hasta finalizar con todos los exámenes.

5) Población inicial, generaciones y probabilidades de cruzamiento y mutación

Por último, en lo que respecta a los parámetros de configuración del algoritmo genético propuesto, se definen los siguientes valores de cantidad de individuos en la población inicial, número de generaciones y probabilidad de que se produzcan el cruzamiento y la mutación de individuos, los siguientes valores fueron el resultado de sucesivas pruebas, en las que se determinó que tales valores producían resultados aceptables. Nuevamente, se debe destacar que no hay recomendaciones sobre qué valor o conjunto de valores es apropiado utilizar para estos parámetros:

- Población: 150
- Generaciones: 150
- Probabilidad de cruzamiento: 80%
- Probabilidad de mutación: 60%

IV. Resultados

Los resultados presentados a continuación fueron obtenidos realizando ejecuciones simultáneas independientes, paralelizadas con GNU Parallel [18], de manera tal de dedicarle un procesador del equipo a cada ejecución. El equipo en cuestión tiene las siguientes características:

- Sistema operativo: Linux versión 4.8.0-1-amd64 (debian-kernel@lists.debian.org) (gcc version 5.4.1 20161019 (Debian 5.4.1-3)) #1 SMP Debian 4.8.7-1 (2016-11-13).
- Procesadores: 4 x Intel(R) Xeon(R) CPU E5-2609 0 @ 2.40GHz.
- Memoria: 8Gb.

El algoritmo propuesto fue escrito para python 3 (ejecutado con python 3.5), haciendo uso de bibliotecas como Numpy [19] para optimizar los cálculos numéricos.

Instancias

Los datos utilizados para probar los algoritmos implementados fueron recopilados por la Universidad de Nottingham, estos están compuestos por una colección de *datasets* de problemas de *timetabling* reales. La versión utilizada para las pruebas fue la original (I). Los *datasets* recopilados pueden verse en la Tabla 1, en la que se indica para cada una de las instancias, la cantidad de exámenes, de estudiantes, de inscripciones a exámenes, y de timeslots.

Tabla 1.

Conjunto de datasets para problemas de timetabling recopilados por la Universidad de Nottingham.

Instancia	*Exámenes*	*Estudiantes*	*Inscripciones*	*Timeslots*
CAR91	682	16925	56877	35
CAR92	543	18419	55522	32
EAR83	190	1125	8109	24
HEC92	81	2823	10632	18
KFU93	461	5349	25113	20
LSE91	381	2726	10918	18
STA83	139	611	5751	13
TRE92	261	4360	14901	23
UTE92	184	2750	11793	10
YOR83	181	941	6034	21

La Tabla 2 presenta los resultados obtenidos en las ejecuciones del algoritmo propuesto y muestra el costo de la mejor solución encontrada, los mejores resultados obtenidos de acuerdo a las publicaciones recientes disponibles [20] [21], el tiempo de ejecución en minutos y la cantidad de evaluaciones realizadas de la función objetivo, es decir, la función de *fitness*. Dichos tiempos de ejecución resultan del promedio obtenido de tres ejecuciones independientes para cada uno de los *datasets* involucrados. Los resultados de todas las ejecuciones, aunque no indicados aquí en pos de la legibilidad de los mismos, fueron consistentes mostrando una variación muy acotada entre ellos.

Tabla 2.

Comparativo de costos de las soluciones encontradas versus mejores resultados reportados, evaluaciones de la función objetivo y tiempo de ejecución en minutos.

Instancia dataset utilizada	*Mejor fitness bibliografía*	*Fitness algoritmo genético*	*Evaluaciones función objetivo*	*Tiempo de ejecución en minutos*
car-f-92	3,67	4,44	20872	382
car-s-91	4,32	5,03	20829	595
ear-f-83	29,3	36,76	20823	79
hec-s-92	9,2	12,26	20961	24
kfu-s-93	12,8	14,22	20881	296
lse-f-91	9,6	11,4	20793	225
sta-f-83	156,9	160,31	20903	50
tre-s-92	7,64	8,53	20830	122
ute-s-92	24,3	27,94	20877	72
yor-f-83	34,71	40,56	20888	72

Por su parte, la Figura 8, presenta gráficamente una comparativa entre el costo de la mejor solución conocida versus el costo de la mejor solución obtenida con el algoritmo propuesto.

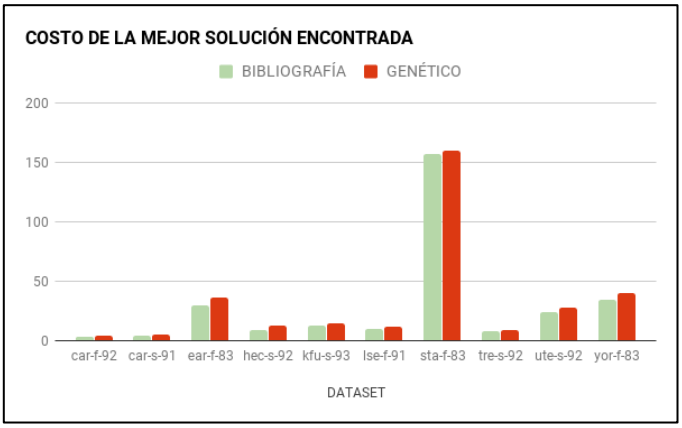

Figura 8. Costo de la mejor solución encontrada.

V. Conclusiones

Como aspectos positivos de este trabajo se puede resaltar la aplicación de diferentes heurísticas, tanto para obtener la solución inicial, como para refinar su calidad en sucesivas iteraciones, lo que permitió obtener soluciones de una calidad aceptable en un tiempo de ejecución relativamente corto y con pocas evaluaciones de la función objetivo. También se han probado distintas heurísticas para la construcción de la solución inicial, como las basadas en programación matemática, coloreo de grafos, métodos de *clustering* o agrupamiento y métodos metaheurísticos. Sumado a esto también se exploraron algoritmos genéticos. Estas pruebas permitieron sumar valiosas herramientas para resolver el problema en cuestión.

Entre las mejoras a implementar se pueden incluir: variar el tiempo de ejecución de los métodos de construcción de solución inicial, por ejemplo incrementando la cantidad de iteraciones que se utilizan para la búsqueda de una solución con menor costo; modificar los operadores de mutación y/o cruzamiento, con el fin de poder explorar mejor el espacio de búsqueda de soluciones posibles. Otro aspecto que podría ser tenido en cuenta, es el de tener un número variable de generaciones y un algoritmo constructivo particular según el tamaño y características del *dataset* sobre el cual se está aplicando el método, esto es, cantidad de exámenes, alumnos e inscripciones, con el fin de determinar si dicha cantidad influye de algún modo en el desempeño de los métodos analizados.

Referencias

[1] M. W. Carter, “A Lagrangian Relaxation Approach To The Classroom Assignment Problem,” *INFOR Inf. Syst. Oper. Res.*, vol. 27, no. 2, pp. 230–246, May 1989.

[2] E. Burke, Y. Bykov, J. Newall, and S. Petrovic, “A time-predefined local search approach to exam timetabling problems,” *IIE Trans.*, vol. 36, pp. 509–528, 2004.

[3] Y. Bykov, “Time-Predefined and Trajectory-Based Search: Single and Multiobjective Approaches to Exam Timetabling,” University of Nottingham, 2003.

[4] E. K. Burke and Y. Bykov, “The late acceptance Hill-Climbing heuristic,” *Eur. J. Oper. Res.*, vol. 258, no. 1, pp. 70–78, Apr. 2017.

[5] L. Di Gaspero and A. Schaerf, “Tabu Search Techniques for Examination Timetabling,” 2001, pp. 104–117.

[6] J. M. Thompson and K. A. Dowsland, “A robust simulated annealing based examination timetabling system,” *Comput. Oper. Res.*, vol. 25, no. 7–8, pp. 637–648, Jul. 1998.

[7] S. Innet, “A noval approach of genetic algorithm for solving examination timetabling problems: A case study of Thai Universities,” in *2013 13th International Symposium on Communications and Information Technologies (ISCIT)*, 2013, pp. 233–237.

[8] M. Alzaqebah and S. Abdullah, “An adaptive artificial bee colony and late-acceptance hill-climbing algorithm for examination timetabling,” *J. Sched.*, vol. 17, no. 3, pp. 249–262, Jun. 2014.

[9] N. Pillay and W. Banzhaf, “A study of heuristic combinations for hyper-heuristic systems for the uncapacitated examination timetabling problem,” *Eur. J. Oper. Res.*, vol. 197, no. 2, pp. 482–491, 2009.

[10] H. Asmuni, “Fuzzy methodologies for automated University timetabling solution construction and evaluation,” *Techniques*, no. April, p. 295, 2010.

[11] E. K. Burke, B. McCollum, A. Meisels, S. Petrovic, and R. Qu, “A graph-based hyper-heuristic for educational timetabling problems,” *Eur. J. Oper. Res.*, vol. 176, no. 1, pp. 177–192, 2007.

[12] E. Burke, D. Elliman, and R. Weare, “A Genetic Algorithm Based University Timetabling System,” *EAST-WEST Conf. Comput. Technol. Educ. CRIMEA, Ukr.*, vol. 1, pp. 35--40, 1994.

[13] D. J. Welsh and M. B. Powell, “An Upper Bound for the Chromatic Number of a Graph and its Application to Timetabling Problem,” *Comput. J. 10*, pp. 85–86, 1967.

[14] E. K. Burke, R. Qu, and A. Soghier, “Adaptive Selection of Heuristics within a GRASP for Exam Timetabling Problems,” *MISTA*, 2009.

[15] E. K. Burke and J. P. Newall, “A New Adaptive

Heuristic Framework for Examination Timetabling Problems," 2002.

[16] J. H. Holland, "Outline for a Logical Theory of Adaptive Systems," *J. ACM*, vol. 9, no. 3, pp. 297–314, Jul. 1962.

[17] "Deap v1.0," 2016. [Online]. Available: https://github.com/DEAP/deap. [Accessed: 04-Nov-2016].

[18] O. Tange, "GNU Parallel - The Command-Line Power Tool," *login: The USENIX Magazine*, 2011. [Online]. Available: http://www.gnu.org/s/parallel. [Accessed: 09-Dec-2016].

[19] "Numpy." [Online]. Available: http://www.scipy.org/scipylib/download. [Accessed: 15-Oct-2016].

[20] E. K. Burke, Y. Bykov, and C. C. Burke, "Solving Exam Timetabling Problems with the Flex-Deluge Algorithm," *Proc. 6th Int. Conf. Pract. Theory Autom. Timetabling VI (PATAT 2006), Brno, Czech Republic, 30 August–1 Sept. 2006; pp. 370–372. 62.*, pp. 9–11, 2006.

[21] E. K. Burke and Y. Bykov, "An Adaptive Flex-Deluge Approach to University Exam Timetabling," *INFORMS J. Comput.*, vol. 28, no. 4, pp. 781–794, Nov. 2016.