DYNA
ISSN: 0012-7353
Universidad Nacional de Colombia

Vidal, Pablo Javier; Olivera, Ana Carolina
Ensamblado de fragmentos de ADN utilizando un novedoso algoritmo de luciérnaga en GPU
DYNA, vol. 85, núm. 204, 2018, Enero-Marzo, pp. 108-116
Universidad Nacional de Colombia

DOI: https://doi.org/10.15446/dyna.v85n204.60078

Disponible en: https://www.redalyc.org/articulo.oa?id=49655628013

UNIVERSIDAD NACIONAL DE COLOMBIA
SEDE MEDELLÍN
FACULTAD DE MINAS

DYNA
http://dyna.medellin.unal.edu.co/

DNA fragment assembling using a novel GPU firefly algorithm

Pablo Javier Vidal & Ana Carolina Olivera

Consejo Nacional de Investigaciones Científicas y Técnicas, Centro de Investigaciones y Transferencia Golfo San Jorge & Universidad Nacional de la Patagonia Austral, Caleta Olivia, Argentina. pjvidal@conicet.gov.ar, acolivera@conicet.gov.ar

Received: September 15th, de 2016. Received in revised form: August 15th, 2017. Accepted: October 5th, 2017

Abstract
The Deoxyribonucleic Acid Fragment Assembly Problem (DNA-FAP) consists in reconstruct a DNA chain from a set of fragments taken randomly. Several authors solved the DNA-FAP using different approaches. In general, although it was obtaining good results; the computational time associated is high. The Firefly Algorithm (FA) is a bioinspired model based on the behaviour of fireflies. Considering that FA is a population bioinspired algorithm is possible design a parallel model of itself on Graphics Processing. In this work, a FA especially development for its execution on GPU is presented in order to accelerate the computational process to solve the DNA-FAP. Through several experiments the efficiency of the algorithm and the quality of the results were demonstrated.

Keywords: fragment assembly problem; firefly algorithm; graphics processing units; optimization; parallelism.

Ensamblado de fragmentos de ADN utilizando un novedoso algoritmo de luciérnaga en GPU

Resumen
El problema de ensamblado de fragmentos de cadenas de ácido desoxirribonucleico (Deoxyribonucleic Acid Fragment Assembly Problem, DNA-FAP) consiste en la reconstrucción de cadenas de ADN desde un conjunto de fragmentos tomados aleatoriamente. El DNA-FAP ha sido resuelto por diferentes autores utilizando distintos enfoques. Aunque se obtienen buenos resultados, el tiempo computacional asociado es alto. El algoritmo de luciérnaga (Firefly Algorithm, FA) es un modelo bioinspirado basado en el comportamiento de las luciérnagas. Al ser un algoritmo bioinspirado poblacional es posible generar un modelo paralelo del mismo sobre Unidades de Procesamiento Gráfico (Graphics Processing Units, GPU). En este trabajo un algoritmo de luciérnaga es diseñado especialmente para ser ejecutado sobre una arquitectura GPU de manera tal de acelerar el proceso computacional buscando resolver el DNA-FAP. A través de diferentes experimentos se demuestra la eficiencia computacional y la calidad de los resultados obtenidos.

Palabras clave: ensamblado de fragmentos de ADN; algoritmo de luciérnaga; unidades de procesamiento gráfico; optimización; paralelismo.

1. Introducción

La necesidad de conocer y predecir mutaciones somáticas y realizar distintos estudios relacionados con el desarrollo de los seres vivos es un tópico de relevancia por su impacto en la medicina para detección y tratamiento de patologías, la investigación forense y el desarrollo de medicamentos. Entre los problemas relacionados con el genoma podemos encontrar el estudio del ensamblado de cadenas de ADN (Deoxyribonucleic Acid Fragment Assembly Problem, DNA-FAP), donde dado un conjunto de cientos o miles de fragmentos de ADN, que pueden contener errores, debemos encontrar la secuencia de ADN original a partir de las permutaciones de los fragmentos que mejor representen a dicha secuencia [17].

Existen herramientas que automatizan el secuenciamiento de ADN, entre ellas podemos nombrar PHRAP [13], TIGR assembler [38] y EULER [32] entre muchas otras. Todas estas herramientas están enfocadas a diferentes problemas encontrados durante el ensamblado de fragmentos.

En los últimos años, varios autores han abordado el DNA-FAP con metaheurísticas y algoritmos bioinspirados [26,33].

How to cite: Vidal, P.J. and Olivera, A.C., Ensamblado de fragmentos de ADN utilizando un novedoso algoritmo de luciérnaga en GPU. DYNA, 85(204), pp. 108-116, March, 2018.

Las metaheurísticas son procedimientos robustos que encuentran buenos resultados sin tener un conocimiento específico del espacio de búsqueda. Sin embargo, obtener resultados más precisos y en tiempos razonables es aún un tema abierto de investigación. Una de las limitaciones de las técnicas actuales está relacionada con el problema de evaluar grandes secuencias de organismos. De esta manera, si consideramos instancias grandes del DNA-FAP, la evaluación de las soluciones puede requerir varias centenas de minutos. Teniendo en cuenta que además se deben realizar varios cientos de miles de evaluaciones, el tiempo computacional se puede acercar a varios cientos de días. En este contexto, las Unidades de Procesamiento Gráfico (GPU, Graphics Processing Unit) aparecen como una plataforma que provee el poder computacional de cientos de computadoras, lo que permite ejecutar en un tiempo razonable algoritmos que de otra manera serían considerados inviables.

El Algoritmo de Luciérnaga (FA, Firefly Algorithm) fue desarrollado recientemente por Yang [41-45] y desde su aparición se ha utilizado en una variedad de problemas de optimización [3,8,25,40,44]. El FA está basado en el comportamiento de las luciérnagas las cuales están caracterizadas por su capacidad de emitir luz (bioluminiscencia). El FA tiene múltiples ventajas sobre otros algoritmos como los Algoritmos Genéticos (GA, Genetic Algorithms) [10] y los basados en Cúmulos de Partículas (PSO, Particle Swarm Optimizer). Entre ellas podemos mencionar la capacidad de resolver problemas multimodales [43]. Las luciérnagas pueden aleatoriamente subdividirse en sub-grupos y cada grupo puede potencialmente acumularse alrededor de un óptimo local. Todos los óptimos locales, incluido el global pueden ser obtenidos simultáneamente [41-45]. Debido a la reciente aparición de este algoritmo existen pocos trabajos publicados sobre su implementación en GPU [6,15,40].

En particular, en este trabajo extendemos la investigación desarrollada por Vidal y Olivera (2014) [40] y proponemos un Algoritmo de Luciérnaga íntegramente diseñado para GPU con el objetivo de resolver el problema DNA-FAP considerando instancias de distinta complejidad. Para el análisis utilizamos casos de prueba existentes en la literatura [23] con longitudes de hasta 156305 (tamaño de la secuencia de pares base) (1049 fragmentos a recombinar) y comparamos nuestra propuesta con los mejores métodos existentes en la literatura.

En la Sección 2 describimos el Problema del Ensamblado de Fragmentos de ADN. La Sección 3 introduce el Algoritmo de Luciérnaga, una breve explicación sobre la GPU y los detalles de nuestra propuesta. Luego, se describen los parámetros experimentales y una explicación de las instancias a abordar en la Sección 4. La Sección 5 muestra un análisis completo de los resultados obtenidos. Finalmente, en la Sección 6 las conclusiones del trabajo son presentadas.

2. Ensamblado de fragmentos de AND

El ADN está compuesto por larguísimas sucesiones de moléculas: Adenine (A), Thynine(T), Guanine (G) y Cytosine (C) llamadas comúnmente bases. Dado lo inmenso

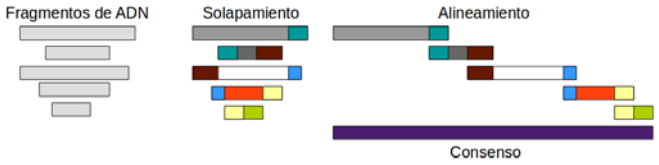

Figura 1. Pasos en el ensamblado de fragmentos de ADN.
Fuente: Los autores.

de estas cadenas estudiarlas de forma global es casi imposible. Por lo general, las cadenas son copiadas y fragmentadas para ser analizadas, lo que se conoce comúnmente como *shotgun.* Ésta técnica no mantiene el orden en que fueron fragmentadas ni ninguna otra información. Una vez que estos fragmentos fueron estudiados es necesario volver a la cadena original rearmando de manera inteligente todas las subcadenas. Debido al proceso de copia y corte de las cadenas de ADN los fragmentos pueden contener errores o faltar piezas. Al proceso de rearmado se lo conoce como ensamblado de fragmentos de ADN [17]. Pequeñas secuencias deben ser nuevamente ensambladas en orden solapando porciones de las mismas. La mayoría de los algoritmos de ensamblado realizan los siguientes tres pasos:

Solapamiento (*Overlap*). Consiste en encontrar el mejor solapado entre los sufijos y los prefijos de todos los fragmentos. La práctica común consiste en filtrar pares de fragmentos que no compartan subcadenas significativas.

Alineamiento (*Layout*). Es encontrar el orden en que estaban los fragmentos en la secuencia original. Constituye la parte más costosa del proceso de ensamblado dada la dificultad de decidir si dos fragmentos están solapados (sus diferencias están causadas por errores de copia) o en realidad son dos copias distintas de una repetición. Las subcadenas repetidas son el mayor desafío para ensamblar cadenas de genoma.

Consenso (*Consensus*). Se refiere a derivar la secuencia de ADN a partir de la disposición establecida en el paso anterior.

Para medir la calidad del consenso se observa el llamado **cubrimiento** (***Coverage***). El cubrimiento de una posición base está definido como el número de fragmentos que comparten esa posición. Esta es una medida de redundancia de un fragmento de dato y denota el número de fragmentos, en promedio, donde un nucleótido se espera que aparezca en el ADN. Esto se calcula como el número de bases leídas por fragmento sobre el tamaño del ADN obtenido. En la Fig. 1 se pueden observar los pasos usuales de las técnicas de ensamblado.

3. Algoritmo de luciérnaga

El Algoritmo de Luciérnaga es una metaheurística desarrollada por Yang [41]. Está inspirada en la imitación de las emisiones de luz utilizada por las luciérnagas para atraerse. Aunque el algoritmo está basado en este fenómeno, en el esquema general propuesto por Yang [41-45] las luciérnagas tienen características particulares:

- Todas las luciérnagas son del mismo sexo, por esto una luciérnaga puede ser atraída por cualquier otra.

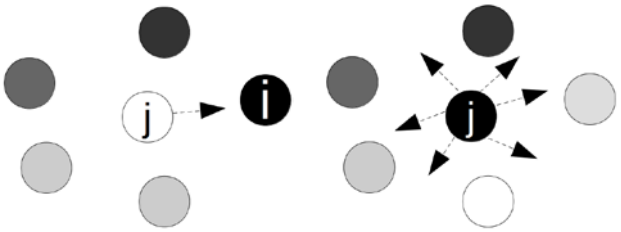

Figura 2. Movimiento de las luciérnagas según su atracción: (a) j se mueve hacia i, la luciérnaga más cercana a ella; (b) j tiene más brillo que cualquiera de las otras luciérnagas por lo que se mueve aleatoriamente. Fuente: Los autores.

- La atracción es proporcional a su brillo. Por ello para cualquier par de luciérnagas emitiendo luz, la menos brillante se moverá hacía la más brillante (ver Fig. 2.a). A medida que las luciérnagas se alejan la percepción de su luz disminuye. Si ninguna luciérnaga es particularmente más brillante que otra, las mismas se mueven aleatoriamente (ver Fig. 2.b).
- La intensidad de la luz emitida por la luciérnaga es afectada o determinada por el valor de la función de aptitud a optimizar.

El FA canónico trabaja con dos principios básicos: la variación de la intensidad de la luz *I* (brillo) y el atractivo β entre dos luciérnagas *i* y *j*. En el caso más sencillo para problemas de optimización el brillo *I* de una luciérnaga localizada en un lugar en particular coincide con su función de aptitud o *fitness*. La intensidad de la luz se reduce con la distancia a su fuente y la luz es también absorbida por el medio, por esto, el atractivo varia con respecto al grado de absorción de la luz. Para un ambiente en particular con un coeficiente de absorción γ, la intensidad de la luz *I* varia con la distancia *r* (ver eq. (1)). Como el atractivo β de una luciérnaga es proporcional a la intensidad de la luz desde el punto de vista de las demás, definimos el β de una luciérnaga a través de la eq. (2).

$$I_j(r_{ij}) = I_0 e^{-\gamma r_{ij}^2} \quad (1)$$

$$\beta_j(r_{ij}) = \beta_0 e^{-\gamma r_{ij}^2} \quad (2)$$

Donde I_0 es la intensidad de la luz y β_0 es el atractivo original de la luciérnaga cuando $r = 0$. Con respecto al coeficiente de absorción γ, si $\gamma \to 0$ el atractivo de una luciérnaga *i* coincide con su brillo (aptitud), es decir, el brillo de una luciérnaga no se ve reducido cuando es vista por otra luciérnaga. En el caso de que $\gamma \to \infty$ significa que el valor del atractivo es cercano a cero cuando es vista por otra luciérnaga por lo cual las luciérnagas no se sentirán atraídas por esta y se moverán de manera aleatoria. De esta manera, γ determina la velocidad de convergencia y el comportamiento del algoritmo y β controla el atractivo. Estudios previos indican que valores de $\beta_0 = 1$ puede ser utilizado en la mayoría de las aplicaciones [44].

La distancia entre dos luciérnagas *i* y *j* ubicadas en diferentes locaciones puede ser expresada por la distancia Euclideana. Teniendo en cuenta *r*, β e *I*, el algoritmo es capaz de definir el movimiento de una luciérnaga *i* con respecto a otra luciérnaga *j*. El pseudo-código del FA canónico puede observarse en el Algoritmo 1. Primero, la población *P* de luciérnagas es inicializada (línea 1). Se inicializa la intensidad de la luz para cada luciérnaga *i* con el valor de aptitud (línea 2). Luego, se define el valor del parámetro γ (línea 3). Mientras la condición de parada no se alcance (línea 4), para cada luciérnaga *i*, el algoritmo tratará de encontrar una luciérnaga *j* más brillante, cercana a *i* (líneas 5 a 13). El algoritmo compara entonces I_j e I_i, si $I_j > I_i$ entonces la luciérnaga *j* se moverá hacía *i* (línea 8). El movimiento de una luciérnaga *i* atraída por otra más atractiva *j* se calcula siguiendo la eq. (3).

$$i = i + \beta_0 e^{\left(-\gamma r_{ij}^2\right)}(j - i) + \alpha\epsilon_i \quad (3)$$

Donde el segundo término tiene en cuenta la atracción mientras que el tercero se calcula de forma aleatoria a partir de ϵ_i tomando una distribución Gaussiana y $\alpha \in [0,1]$ luego la atracción es actualizada (línea 10). Posteriormente, las nuevas soluciones son evaluadas y la intensidad de la luz se actualiza en la línea 11. Las luciérnagas son clasificadas y se encuentra la mejor de ellas (línea 14). Cuando el proceso termina, el Algoritmo 1 retorna los resultados de la ejecución (línea 16).

Como se puede apreciar en el Algoritmo 1 el tiempo de ejecución es $O(n^2 * t)$ donde *t* es el número de iteraciones del ciclo **while**. Además, cada luciérnaga debe ser evaluada según la eq. 2, *n* veces, para cada luciérnaga *j*. Esta complejidad computacional no es fácilmente posible de reducir.

3.1. Algoritmo de luciérnaga discreto sobre GPU

El Algoritmo de Luciérnaga Discreto (DFA) es una variación del FA canónico utilizado para problemas combinatoriales el cual ha sido exitoso en diversos campos de aplicación [16,22]. El DFA es el modelo base utilizado en el presente trabajo para la implementación sobre GPU. Esta sección presenta nuestra propuesta algorítmica la cual llamaremos GPU-DFA.

3.1.1. Unidades de procesamiento gráfico

Las GPUs se consideran como un coprocesador gráfico de la Unidad Central de Procesamiento (CPU, por sus siglas en inglés). Con este tipo de modelo se busca aliviar la carga computacional de la CPU. Los modelos actuales de GPU suelen

Algoritmo 1 FA canónico

```
1:  Inicializar la población P de luciérnagas (i = 1, 2, 3, ..., n)
2:  Inicializar la intensidad de la luz de cada i, I_i = f(i)
3:  Definir el coeficiente de absorción, γ
4:  while no se alcance la condición de parada do
5:    for i = 1 : n do
6:      for j = 1 : n, con j ≠ i do
7:        if I_j > I_i then
8:          Muevo i hacia j
9:        end if
10:       El atractivo varia con la distancia r, vía e^{-γr^2}
11:       Evaluar las nuevas soluciones y actualizar la intensidad de la luz.
12:     end for
13:   end for
14:   Ranquear las luciérnagas y encontrar la mejor de todas
15: end while
16: return Informar los resultados
```

tener una gran cantidad de procesadores, los cuales están optimizados para ejecutar una instrucción simple sobre cada elemento de un extenso conjunto de ellos.

Para poder utilizar adecuadamente la capacidad de cómputo de la GPU, NVIDIA ha desarrollado un modelo de programación llamado CUDA (Compute Unified Device Architecture) [30]. CUDA permite a los programadores implementar funciones llamadas *kernels*. Un kernel contiene la porción de código que será ejecutada en la GPU. Esta función es invocada desde el anfitrión y se despliega en la GPU. CUDA nos da la posibilidad de implementar los kernels usando lenguaje C estándar más algunas extensiones de NVIDIA. Además, nos permite organizar el paralelismo en tres niveles: rejilla, bloque e hilo. Cada vez que se invoca un kernel, se crea una rejilla de bloques, los cuales a su vez agrupan múltiples hilos. Durante la ejecución del kernel, cada hilo tiene acceso a diferentes tipos de memoria dentro de la GPU. Esta jerarquía abarca: registros, memoria local y compartida, memoria global, constante y de texturas. El lector puede consultar en el Manual del Usuario [30] para obtener información más detallada de CUDA.

Existen pocas aproximaciones que utilicen algoritmo de luciérnaga sobre GPU [6,15]. Estos modelos han sido testeados sobre dominios continuos obteniendo buenos resultados con respecto a la ganancia de tiempo.

Nuestra principal motivación al diseñar el DFA acelerado por la GPU es establecer un modelo eficiente que ejecute el procedimiento principal del DFA enteramente en GPU. Los objetivos son: (1) minimizar la transferencia entre CPU y GPU evitando cuellos de botella en la comunicación; (2) abordar problemas de optimización combinatoriales utilizando el modelo de interacción de las luciérnagas sobre una plataforma GPU. La arquitectura CUDA es empleada a fin de explotar al máximo la ejecución en paralelo y el procesamiento intensivo de operaciones aritméticas en las GPUs [30].

3.2. Esquema general de la propuesta

3.2.1. Aproximación del DFA sobre GPUs

En la Fig. 3 podemos observar un diagrama del GPU-DFA. Al inicio del GPU-DFA todos los parámetros son transferidos a la memoria principal de la GPU. Adaptar este tipo de algoritmo bioinspirado no es una tarea sencilla puesto que la gestión de la memoria en la GPU debe ser cuidadosamente manejada. La transferencia de datos entre CPU y GPU puede producir cuellos de botella y se busca minimizarla en nuestra propuesta. Utilizamos un grupo de *kernels* donde realizamos las siguientes tareas: primero, el GPU-DFA crea y evalúa cada luciérnaga de la población *P* usando un hilo (*thread*) de la GPU para generar cada solución. A continuación, hasta que no se alcance la condición de terminación, el GPU-DFA ejecuta una serie de kernels para evaluar en paralelo si cada luciérnaga *i* se moverá a otra luciérnaga *j* y posteriormente generar una población de *n* mejores luciérnagas. La división en múltiples kernels se debe a la heterogeneidad y complejidad de las tareas. Así, las operaciones costosas son ejecutadas principalmente en la GPU.

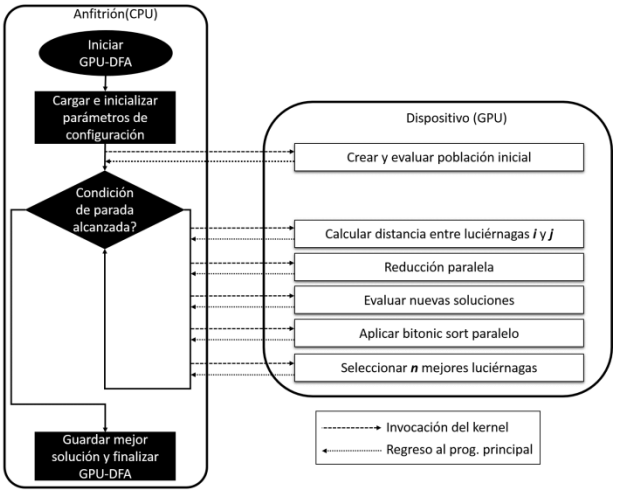

Figura 3. Modelo del GPU-DFA propuesto.
Fuente: Los autores.

El Algoritmo 2 muestra en detalle las operaciones anteriormente mencionadas dentro del esquema del GPU-DFA. Todos los métodos presentados en el Algoritmo 2 intentan explotar al máximo el rápido acceso a los diferentes niveles de memoria de la GPU. Debido a la complejidad de algunas operaciones que son completamente diferentes unas con otras, hemos tratado de escribir kernels simples y pequeños ya que el costo de lanzarlos es insignificante con respecto a las operaciones a realizar y se utilizan menos registros de memoria.

Algoritmo 2 GPU-DFA

```
1: Definir el coeficiente de absorción γ
2: Alocar las entradas del problema en la memoria de la GPU
3: Copiar las entradas del problema a la memoria de la GPU
4: Alocar la estructura de la población de soluciones en la memoria de la GPU
5: Inicializar la población P de n luciérnagas
6: for cada luciérnaga (i) de P en parallel do
7:    InicialSolución(i);
8:    EvaluarSolución(i);
9: end for
10: while no alcanzar la condición de terminación do
11:    temp=∅
12:    for cada par i y j de P en paralelo do
13:       A =computeDistance(i, j);
14:       computeAttractiveness(A,i,j);
15:    end for
16:    Aplicar la Función de Reducción
17:    for cada par i y j de P en paralelo do
18:       if i ≠ null then
19:          k = random(2, A)
20:          temp.agregar(move_{2-opt}(j, k))
21:       else
22:          temp.agregar(move_{Random}(j))
23:       end if
24:    end for
25:    P = Tomar las mejores n luciérnagas en temp
26: end while
27: return La mejor de las n luciérnagas
```

En las siguientes secciones se explica en detalle uno a uno los procedimientos del GPU-DFA.

3.2.2. Inicialización

Durante la inicialización, se asigna un espacio de memoria en la GPU, el cual corresponde a los datos de

entrada y las soluciones candidatas. Luego, los datos de entrada iniciales y las estructuras adicionales son copiados a la GPU. Es importante observar que los datos de entrada son solamente de lectura y nunca cambian durante la ejecución. Por esta razón sus valores son copiados por una única vez y al principio durante toda la ejecución. En el GPU-DFA, la inicialización de la población y la evaluación de la aptitud de cada solución son asignadas a cada hilo de la GPU. De esta manera, se busca que cada hilo pueda acceder consecutivamente a la información de la solución buscando lograr un patrón de acceso a memoria coalescente y optimizar el tiempo de acceso a dicha memoria [18].

3.2.3. Cálculo de distancias y operación de reducción

El primer paso evolutivo para el GPU-DFA es calcula *r*, β e *I* en paralelo para cada par de luciérnagas *i* y *j*. Al evaluar cada combinación es necesario aplicar un método paralelo de reducción para conocer si la solución *j* necesita moverse de forma aleatoria o debe acercarse a una solución más brillante.

El método de reducción es utilizado para identificar posibles movimientos de cada luciérnaga con respecto a las otras. Al lanzar el kernel, se despliega una serie de bloques, cada uno de ellos asignado al conjunto de datos de una luciérnaga obtenidos en el paso anterior. El GPU-DFA utiliza una estructura auxiliar en la memoria compartida de la GPU. El resultado será la decisión sobre qué movimiento realizará la luciérnaga.

3.2.4. Modificaciones en paralelo

Definido para cada luciérnaga *j* el movimiento que va a realizar, el GPU-DFA crea y evalúa *n*m* nuevas soluciones alterando cada una con un operador específico o de forma aleatoria según sea el caso y se almacenan en *temp*.

3.2.5. Ordenamiento bitonic

El esquema del algoritmo hace necesario ordenar el conjunto *temp* según su aptitud. Para ello utilizamos el ordenamiento Bitonic paralelo [33] para luego seleccionar las *n* mejores luciérnagas y actualizar con ellas *P*. El ordenamiento Bitonic tiene una complejidad de $O(n (\log n)^2)$. Este método necesita únicamente un número de comparaciones e intercambios de $O(n \log n)$. La constante oculta en la notación asintótica es más pequeña que para otros métodos de ordenamiento en paralelo. Este ordenamiento puede ser implementado en paralelo en modernas arquitecturas como las GPUs, sin utilizar ningún acceso de escritura que pueda involucrar a la CPU.

3.2.6. Generación de números aleatorios

El desempeño de un algoritmo bioinspirado depende en gran medida de la calidad del generador de números aleatorios utilizado. Para nuestro trabajo hemos usado un generador aleatorio Mersenne Twister [36]. Al comienzo de la ejecución el GPU-DFA define una semilla global con la cual se inicializa una semilla local en cada hilo. Posteriormente, cada semilla local es invocada continuamente por cada hilo para subsecuentes generaciones de números aleatorios.

3.2.7 Codificación de la solución

La representación juega un rol importante en la eficiencia y eficacia de cualquier algoritmo. Con el objetivo de hacer más eficiente el mapeo entre cada hilo *id* y solución de *P* en la GPU se utilizó una representación vectorial. Esta codificación utiliza un alfabeto $\Sigma=\{1,\ldots,l\}$ donde *l* es el número total de fragmentos de ADN a ordenar. Entonces, una solución es una permutación de fragmentos de ADN. En su representación cada variable toma su valor del alfabeto Σ.

La disposición de la población en la memoria de la GPU fue cuidadosamente diseñada. La forma de las soluciones basada en cromosoma [20] simplifica el movimiento individual de las mismas en la selección, la comparación y las fases de perturbación como así también en la comunicación anfitrión-dispositivo necesaria para la operación de reemplazo y actualización de las mejores soluciones durante el proceso evolutivo.

4. Configuración de la experimentación

En esta sección se presentan las especificaciones sobre la configuración y parámetros definidos. Primero, se introducen detalles de las instancias seleccionadas del DNA-FAP. Luego, se muestra la metodología y parametrización para la experimentación.

4.1. Instancias

Para nuestro estudio se llevaron a cabo experimentos con diferentes instancias del DNA-FAP [23]. Las mismas se pueden diferenciar en dos grupos, el primero de ellos ha sido generado mediante el uso de *GenFrag* [7] y para el segundo se ha utilizado el programa *DNAgen*. La Tabla 1 resume las instancias estudiadas, sus nombres, cubrimiento, tamaño (número de fragmentos por instancia), tamaño de la secuencia original y el óptimo para cada una de ellas. Las instancias están ordenadas por grupo y número de fragmentos.

4.2. Valores de los parámetros de experimentación

La distancia entre dos luciérnagas es definida a través de la eq. (4). El parámetro *A* es el número de arcos diferentes entre *i* y *j*. El parámetro *l* indica el tamaño del problema (número de fragmentos). El valor de *r* varía dentro del intervalo [0,10] según la eq. (4) (ver [16]).

$$r_{ij} = \frac{A}{l} * 10 \tag{4}$$

En este trabajo, el algoritmo utiliza un operador *2-opt* que consiste en remover 2 arcos y reemplazarlos con otros 2 arcos para reconectar los fragmentos resultantes de la remoción, y así obtener un secuenciamiento. El operador es aplicado *k* veces, donde *k* es un número seleccionado aleatoriamente entre 2 y el parámetro *A*. Si el movimiento de la luciérnaga es aleatorio, el operador *2-opt* es aplicado tomando *k* un valor cualquiera mayor a 0.

Tabla 1.
Información de los conjuntos de datos utilizados para realizar los experimentos.

Instancias (abrev.)	Coverage	Tamaño medio Frag.	N° de Frag.	Tamaño de la Secuencia Original	Óptimo
Instancias GenFrag					
x60189 4 (x_4)	4	395	39	3835	11478
x60189 5 (x_5)	5	386	48		14161
x60189 6 (x_6)	6	343	66		18301
x60189 7 (x_7)	7	387	68		21271
m154216 5 (m_5)	5	398	127	10089	38746
m154216 6 (m_6)	6	350	173		48052
m154216 7 (m_7)	7	383	177		48052
j02459 7 (j_7)	7	405	352	20000	116700
bx842596 4 (b_4)	4	708	442	77292	227920
bx842596 7 (b_7)	7	703	773		445422
Instancias DNAgen					
acin1 (a_1)	26	182	307	2170	47618
acin2 (a_2)	3	1002	451	147200	151553
acin3 (a_3)	3	1001	601	200741	167877
acin4 (a_5)	2	1003	751	329958	163906
acin5 (a_7)	2	1003	901	426840	180966
acin6 (a_9)	7	1003	1049	156305	344107

Fuente: [23].

Para la mayoría de los problemas se suele utilizar una población *n* entre 15 y 100. En particular para el FA el rango ideal es entre 25 a 40 [41,42]. Considerando que los hilos de la GPU están planificados en bloques de 32 es una buena práctica tomar como tamaño de población 32 o múltiplos de 32 [30]. Con el objetivo de testear el GPU-DFA se utilizaron como parámetros del algoritmo n=32 y m=16. Estos valores fueron obtenidos en un estudio previo realizado por los autores y puede ser consultado en [40].

Con la finalidad de hacer una comparación entre las dos implementaciones del DFA (CPU y GPU) se eligió como criterio de terminación alcanzar 1 millón de evaluaciones. Se realizaron 30 ejecuciones independientes para cada instancia. En la Tabla 3 hemos marcado los mejores valores obtenidos con gris oscuro y con gris claro aquellos que son los segundos mejores resultados.

Los experimentos fueron realizados sobre una CPU AMD FX(tm)-8320 Eight-Core Processor con 16GB de RAM. El sistema operativo utilizado fue Ubuntu Precise 12.04. En el caso de la GPU se utilizó una NVIDIA GeForce GTX 780 Ti con 3GB of DRAM y la versión 6.0 de CUDA.

5. Resultados

En esta sección se muestran los resultados de los experimentos realizados. Se ha cuantificado la calidad de las soluciones considerando su valor de aptitud como así también el número de contigs obtenidos para cada instancia. Además, se compararon los tiempos de ejecución para las versiones en CPU y GPU del DFA.

5.1. Análisis numérico

La Tabla 2 indica los resultados del GPU-DFA para todas las instancias mencionadas en la Sección 4. La primera columna corresponde con el nombre abreviado para cada instancia.

Tabla 2.
Resultados del GPU-DFA para las instancias del DNA-FAP.

Instancias	Mejor	Taza de Aciertos	Avg.	Std.
Instancias GenFrag				
x_4	11478	27	11477.67	±1.15
x_5	14027	24	13783.33	±63.09
x_6	18301	29	18291.40	±15.27
x_7	21271	28	21257.83	±14.41
m_5	38592	18	38555.80	±73.37
m_6	48048	13	47969.29	±91.79
m_7	55067	9	53748.50	±58.50
j_7	114358	3	114343.60	±59.45
b4	225858	2	225173.60	±678.54
b7	441992	0	433958.00	±7053.02
Instancias DNAgen				
a_1	46876	1	45976.60	±68.62
a_2	144634	2	144513.00	±133.93
a_3	156776	1	155751.30	±147.10
a_5	145880	0	159304.50	±1535.86
a_7	157032	0	176439.90	±1217.15
a_9	329015	0	338201.70	±5036.775

Fuente: Los autores.

La columna dos muestra el mejor valor de aptitud obtenido y la tercera columna la taza de acierto, definida como el porcentaje de ejecuciones en las que se alcanzó el óptimo conocido, en las 30 ejecuciones. Las columnas cuatro y cinco indican el valor de aptitud promedio (Avg.) con su correspondiente desviación estándar (Std.) para las 30 ejecuciones. El GPU-DFA obtiene valores de aptitud iguales al óptimo o muy cercanos a éste. En particular, el algoritmo obtiene el óptimo para 11 de las 16 instancias al menos una vez. Para el grupo de las instancias b_7, a_5, a_7 y a_9 el GPU-DFA no alcanza la solución óptima, sin embargo, las soluciones obtenidas son competitivas y cercanas al óptimo en general. Estos resultados indican que el GPU-DFA es capaz de explorar el espacio de búsqueda de manera efectiva mostrando un comportamiento promisorio.

La Tabla 3 compara los resultados del GPU-DFA en contraposición con otros algoritmos. Estos algoritmos son Optimización por Cúmulo de Partículas (Particle Swarm Optimization) y Evolución Diferencial (Differential Evolution) (PPSO+DE) [24], Algoritmo Evolutivo Reina de Abejas basado en un Algoritmo Genético (Queen Bee Evolution Based on Genetic Algorithm, QEGA) [19], Recocido Simulado (Simulated Annealing, SA) [9], Búsqueda local Consciente (Problem Aware Local Search, PALS), SAX [26] y la Optimización de Prototipos con Mejora de Pasos Evolutivos (Prototype Optimization with Evolved Improvement Steps, POEMS) [21]. Se puede observar que la inicialización de forma inteligente en las soluciones candidatas resulta en un beneficio en el proceso de búsqueda, obteniendo estos algoritmos resultados competitivos para todas las instancias.

En general, el GPU-DFA obtiene soluciones competitivas para todas las instancias. Si bien, en algunas de ellas no se llega a alcanzar el óptimo, se logra encontrar el segundo mejor valor superando a las otras metaheurísticas presentadas. Los resultados demuestran que el DFA propuesto optimiza la función de aptitud en las 16 instancias. En particular, se puede observar que en las instancias *DNAAgen* los algoritmos PPSO+DE y SA superan al GPU-

Tabla 3.
Resultados del GPU-DFA para las instancias del DNA-FAP.

	GPU-DFA	*PPSO +DE* [24]	*QEGA* [19]	*SA* [9]	*PALS* [26]	*SAX* [26]	*POEMS* [21]
Instancias GenFrag							
x_4	**11478**	**11478**	*11476*	**11478**	**11478**	**11478**	**11478**
x_5	**14131**	13642	*14027*	*14027*	14021	*14027*	–
x_6	**18301**	**18301**	18266	**18301**	**18301**	**18301**	–
x_7	**21271**	20921	21208	**21271**	21210	**21268**	21261
m_5	**38726**	*38686*	38578	38583	38526	**38726**	38610
m_6	**48048**	*47669*	47882	**48048**	**48048**	**48048**	–
m_7	54258	54891	55020	55048	55067	*55072*	**55092**
j_7	16358	114381	116222	16257	115320	115301	**116542**
b_4	223858	224797	**227252**	226538	225783	223029	*227233*
b_7	436992	429338	*443600*	436739	438215	417680	**444634**
Instancias DNAgen							
a_1	**47618**	47264	47115	46955	46876	46865	–
a_2	144634	**147429**	144133	*144705*	144634	144567	–
a_3	*161776*	**163965**	156138	156630	156776	155789	–
a_5	*151082*	**161511**	144541	146607	146594	145880	–
a_7	*163155*	**180052**	155322	157984	158004	157032	–
a_9	324525	**335522**	322768	324559	*325930*	314354	–

Fuente: Los autores, [23].

DFA. En el caso de las instancias de pocos fragmentos la mayoría de los algoritmos obtienen resultados similares.

5.2. Comparación de contigs

En esta sección se presenta una comparación entre nuestra propuesta y otros algoritmos ensambladores enfocada en la calidad de las soluciones devueltas. Este análisis se enfoca en la calidad de contigs obtenidos. El cálculo del contig asegura que la mejor solución obtenida representa una secuencia continua.

La Tabla 4 presenta los valores contigs de todas las instancias para cada algoritmo. La columna uno muestra el nombre abreviado de la instancia. Desde la columna dos a la columna siete se muestran los valores contigs de los siguientes algoritmos: GPU-DFA, Colonia de Abejas Artificiales (Artificial Bee Colony, [9]), QEGA, SA, PALS y GA [9].

De la Tabla 4 podemos concluir que los algoritmos que poseen algún método inteligente o de búsqueda local obtienen un mejor valor (un número de contigs pequeño). Para GPU-DFA y, en particular, para las instancias m_7, a_7 y a_9 las secuencias reportadas por el GPU-DFA no mejoran los resultados obtenidos por PALS y SA. A diferencia de las técnicas con las que comparamos nuestra propuesta al GPU-DFA no se le ha dotado de ningún operador específico sobre el problema y, aun así, el algoritmo logra resultados equiparables a los existentes en la literatura.

Tabla 4.
Resultados del GPU-DFA para las instancias del DNA-FAP.

Inst.	*GPU-DFA*	*ABC* [9]	*QEGA* [19]	*SA* [9]	*PALS* [26]	*GA* [9]
Instancias GenFrag						
x_4	1	1	1	1	1	1
x_5	1	1	1	1	1	1
x_6	1	1	1	1	1	1
x_7	1	1	1	1	1	1
m_5	1	3	1	1	1	1
m_6	1	2	1	1	1	2
m_7	2	2	1	1	1	2
j_7	1	3	1	1	1	1
b_4	1	12	8	1	1	6
b_7	1	12	3	1	1	3
Instancias DNAgen						
a_1	2	8	4	1	1	5
a_2	1	237	233	1	1	236
a_3	1	362	358	1	1	358
a_5	1	556	552	1	1	522
a_7	722	726	722	1	1	722
a_9	552	552	552	1	1	552

Fuente: Los autores.

Podemos establecer que el rendimiento observado del GPU-DFA indica que éste es capaz de generar soluciones que alcanzan el óptimo o quedan muy cercanas al óptimo. De la misma forma se observa que el GPU-DFA puede explorar el espacio de búsqueda de forma efectiva, por lo cual es capaz de identificar la región donde se encuentra localizado el óptimo.

5.3. Análisis de tiempo de ejecución

En la Tabla 5 se muestra el tiempo promedio consumido en segundos para las versiones en CPU y en GPU del DFA. En la primera columna se informa el nombre abreviado de cada instancia. La columna dos y tres indican los tiempos de ejecución promedio consumidos por el CPU-DFA y el GPU-DFA respectivamente. Finalmente, la columna cuatro muestra la ganancia de tiempo obtenida.

Con respecto a la ganancia obtenida, ésta métrica se calculó dividiendo el tiempo promedio consumido por el CPU-DFA sobre el tiempo promedio consumido por el GPU-DFA. Los valores obtenidos que se encuentran por arriba del 1.00 indican que la CPU consume más tiempo que la GPU.

De la Tabla 5 podemos concluir que aun cuando el tamaño de las instancias crezca, el tiempo del GPU-DFA no lo hace significativamente en contraste con el tiempo del CPU-DFA. Esto puede deberse a las características intrínsecas del DFA el cual posee un alto grado de paralelización que maximiza la eficiencia de cada hilo de ejecución manteniendo la simplicidad en cada kernel. La ganancia de tiempo se encuentra entre los valores 1.73 a 10.14, lo que demuestra la capacidad del GPU-DFA para acelerar procesos de ensamblado de ADN.

Tabla 5.
Tiempo medio en segundos para encontrar la solución óptima para cada implementación y cada instancia.

Inst.	*CPU-DFA*	*GPU-DFA*	*Ganancia*
Instancias GenFrag			
x_4	4.13	2.38	1.73
x_5	5.58	2.33	2.39
x_6	18.58	2.33	7.97
x_7	21.59	3.82	5.65
m_5	39.12	7.20	5.43
m_6	52.37	5.16	10.14
m_7	55.21	6.71	8.23
j_7	186.31	34.37	5.42
b_4	393.18	139.73	2.81
b_7	502.66	140.41	3.58
Instancias DNAgen			
a_1	171.98	29.54	5.82
a_2	382.93	134.58	2.84
a_3	486.36	136.95	3.55
a_5	719.58	157.91	4.55
a_7	1156.12	138.18	8.36
a_9	1418.25	143.57	9.88

Fuente: Los autores.

6. Conclusiones y trabajo futuro

En este trabajo proponemos un Algoritmo de Luciérnaga Discreto implementado sobre Unidades de Procesamiento Gráfico (GPU-DFA) para el Problema del Secuenciamiento de Cadenas de ADN (DNA-FAP). La principal ventaja del Algoritmo de la Luciérnaga frente a otros algoritmos bioinspirados es su capacidad de dividir la población de soluciones en forma automática y de enfrentarse a problemas multimodales.

Se realizaron experimentos sobre el GPU-DFA para analizar su comportamiento. Se ha evaluado la calidad de las soluciones obtenidas y el tiempo de ejecución del algoritmo. Por otro lado, se evaluó la capacidad del GPU-DFA para reensamblar de manera satisfactoria los fragmentos de distintos conjuntos de instancias.

Primero se compararon los valores de aptitud del GPU-DFA en contraposición con otros algoritmos populares en la literatura. Se observó que el GPU-DFA solo era superado por aquellos que poseen una inicialización inteligente de la población inicial para las instancias de mayor cantidad de fragmentos. Luego se analizó la forma de las soluciones obtenidas con respecto a la cantidad de contigs que generaban los algoritmos. El GPU-DFA obtuvo nuevamente resultados promisorios para todas las instancias.

Por último, se realizó una comparación entre la versión en CPU del DFA y nuestro GPU-DFA. La ganancia de tiempo reporta valores entre 1.73 a 10.14 lo que demuestra la capacidad de aceleración de nuestra aproximación al no tener un crecimiento del tiempo de forma lineal para instancias grandes del DNA-FAP en contraposición con la versión en CPU. El método propuesto logró disminuir en hasta 10 veces el tiempo de ejecución del modelo secuencial, lo que permitirá trabajar grupos de fragmentos de mayor magnitud en un tiempo razonable. Por todo esto el GPU-DFA constituye una alternativa atractiva para la resolución de instancias grandes del DNA-FAP.

Como trabajo futuro se espera realizar una hibridación del FA que permita abordar de forma inteligente el DNA-FAP y mejorar así la búsqueda de soluciones.

Agradecimientos

Los autores agradecen al Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET) por el Proyecto de Investigación Plurianual PIP 2015-2017, a la Agencia Nacional de Promoción Científica y Tecnológica por el proyecto PICT 2014-0430. Los autores también agradecen a la Universidad Nacional de la Patagonia Austral por su proyecto PI 29/B209.

References

[1] Baykasoglu, A. and Ozsoydan, F.B., An improved firefly algorithm for solving dynamic multidimensional knapsack problems. Expert Syst. Appl., 41(8), pp. 3712-3725, 2014. DOI: 10.1016/j.eswa.2013.11.040

[2] Bojic, I., Podobnik, V., Ljubi, I., Jezic, G. and Kusek, M., A self-optimizing mobile network: Auto-tuning the network with firefly-synchronized agents. Information Sciences, 182(1), pp. 77-92, 2012. DOI: 10.1016/j.ins.2010.11.017

[3] Chandrasekaran, K. and Sishaj, P.S., Network and reliability constrained unit commitment problem using binary real coded firefly algorithm. International Journal of Electrical Power & Energy Systems, 43(1), pp. 921-932, 2012. DOI: 10.1016/j.ijepes.2012.06.004

[4] Chen, T. and Skiena, S.S., A case study in genome-level fragment assembly. BIOINF: Bioinformatics, 16, 2000.

[5] Coull, S.E. and Szymanski, B.K., Sequence alignment for masquerade detection. Computational Statistics & Data Analysis, 52(8), pp. 4116-4131, 2008. DOI: doi.org/10.1016/j.csda.2008.01.022

[6] de Paula, L. et al., Parallelization of a modified firefly algorithm using GPU for variable selection in a multivariate calibration problem. International Journal of Natural Computing Research (IJNCR), 4(1), pp. 31-42, 2014. DOI: /10.4018/ijncr.2014010103

[7] Engle, M.L. and Burks, C., Artificially generated data sets for testing DNA sequence assembly algorithms, Genomics, Volume 16(1), pp. 286-288, 1993. DOI: 0.1006/geno.1993.1180

[8] Farhoodnea, M., Mohamed, A., Shareef, H. and Zayandehroodi, H., Optimum placement of active power conditioners by a dynamic discrete firefly algorithm to mitigate the negative power quality effects of renewable energy-based generators. International Journal of Electrical Power & Energy Systems, 61(0), pp. 305-317, 2014. DOI: 10.1016/j.ijepes.2014.03.062

[9] Firoz, J.S., Rahman, M.S. and Saha, T.K., Bee algorithms for solving dna fragment assembly problem with noisy and noiseless data. In Proceedings of the 14th Annual Conference on Genetic and Evolutionary Computation, GECCO'12, pp. 201-208, 2012. DOI: 10.1145/2330163.2330192

[10] Fister, I., Fister, I. Jr., Yang, X.-S. and Brest, J., A comprehensive review of firefly algorithms. CoRR, abs/1312.6609, 2013.

[11] Gandomi, A.H., Yang, X.-S., Talatahari, S. and Alavi, A.H., Firefly algorithm with chaos. Comm Nonlinear Sci Numer Simulat, 18(1), pp. 89- 98, 2013. DOI: 10.1016/j.cnsns.2012.06.009

[12] García-Nieto, J.M., Olivera, A.C. and Alba, E., Optimal cycle program of traffic lights with particle swarm optimization. IEEE Transactions on Evolutionary Computation, 17(6), pp. 823-839, 2013. DOI: 10.1109/TEVC.2013.2260755

[13] Green, P., Phrap, version 1.090518, [online], 2009, Available at: http://phrap.org.

[14] Huang, X. and Madan, A., Cap3: A DNA sequence assembly program. Genome research, 9(9), pp. 868-877, 1999. DOI: 10.1101/gr.9.9.868

[15] Husselmann, A.V. and Hawick, K.A., Parallel parametric optimisation with firefly algorithms on graphical processing units. World Congress in Computer Science, Computer Engineering, and Applied Computing, 2012.

[16] Jati, G.K., Manurung, R. and Suyanto., Discrete firefly algorithm for traveling salesman problem: A new movement scheme. In Xin-She Y., Zhihua, C., Renbin, X., Amir, H.G. and Mehmet, K., eds., Swarm Intelligence and Bio-Inspired Computation, Elsevier, Oxford, pp. 295-312, 2013.

[17] Jones, N.C. and Pevzner, P.A., Preface. An Introduction to bioinformatics algorithms. Massachusetts Institute of Technology, 2004.

[18] Kirk, D.B. and Hwu, W.W., Programming massively parallel processors: A hands-on approach. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2010.

[19] Knuth, D.E., The art of computer programming, Volume 3: (2Nd Ed.) Sorting and Searching. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[20] Kothapalli, K., Shah, R. and Narayanan, P.J., GPU-accelerated genetic algorithms. Workshop on Parallel Architectures for Bio-ispired Algorithms in conjunction with Parallel Architectures and Compilation Techniques (PACT Workshop), pp. 27-34, 2010.

[21] Kubalik, J., Buryan, P. and Wagner, L., Solving the DNA fragment assembly problem efficiently using iterative optimization with evolved hypermutations. Proceedings of the 12th annual conference on Genetic and evolutionary computation, 2010, pp. 213-214. DOI: 10.1145/1830483.1830522

[22] Maher, B. et al., A firefly-inspired method for protein structure prediction in lattice models. Biomhc, 4(1), pp. 56-75, 2014. DOI: 10.3390/biom4010056

[23] Mallén-Fullerton, G.M., Hughes, J.A., Houghten, S. and Fernández-Anaya, G., Benchmark datasets for the DNA fragment assembly problem. International Journal of Bio-Inspired Computation, 5(6), pp. 384-394, 2013. DOI: 10.1504/IJBIC.2013.058912
[24] Mallén-Fullerton, G.M. and Fernández-Anaya, G., DNA fragment assembly using optimization. IEEE Congress on Evolutionary Computation, Cancun, 2013, pp. 1570-1577. DOI: 10.1109/CEC.2013.6557749
[25] Meybodi, M.R., Farahani, S.M. and Nasiri, B., A multiswarm based firefly algorithm in dynamic environments. In Third international conference on signal processing systems (ICSPS 2011), pp. 68-72, Yantai, China, 2011.
[26] Minetti, G. and Alba, E., Metaheuristic assemblers of DNA strands: Noiseless and noisy cases. Proceedings of the IEEE Congress on Evolutionary Computation (CEC2010), 2010, pp. 1-8. DOI: 10.1109/CEC.2010.5586524
[27] Minetti, G., Leguizamón, G. and Alba, E., SAX: A new and efficient assembler for solving DNA fragment assembly problem. 2012 Argentine Symposium on Artificial Intelligence, 2012.
[28] Mussi, L., Daolio, F. and Cagnoni, S., Evaluation of parallel particle swarm optimization algorithms within the CUDA architecture. Inf. Sci., 181(20), pp. 4642-4657, 2011. DOI: 10.1016/j.ins.2010.08.045
[29] Myers, E.W., Sutton, G.G., Delcher, A.L., Dew, I.M., Fasulo, D.P., Flanigan, M.J., Kravitz, S.A., Mobarry, C.M., Reinert, K.H.J, Remington, K.A., et al., A whole-genome assembly of drosophila. Science, 287(5461), pp. 2196-2204, 2000. DOI: 10.1126/science.287.5461.2196
[30] NVIDIA Corporation. NVIDIA CUDA C Programming Guide, 2011.
[31] Osaba, E., Yang, X.-S., Díaz, F., Onieva, E., Masegosa, A.D. and Perallos, A., A discrete firefly algorithm to solve a rich vehicle routing problem modelling a newspaper distribution system with recycling policy. Soft Computing, pp. 1-14, 2016.
[32] Pevzner, P., Computational molecular biology: An Algorithmic approach. MIT Press, 2000.
[33] Parsons, R.J., Forrest, S. and Burks, C., Genetic algorithms, operators, and DNA fragment assembly. Machine Learning, 21(1-2), pp. 11-33, 1995. DOI: 10.1007/BF00993377, 10.1023/A:1022613513712
[34] Peters, H., Schulz-Hildebrandt, O. and Luttenberger, N., Fast in-place sorting with CUDA based on bitonic sort. LNCS, 6067, pp. 403-410, 2009.
[35] Pop, M., Shotgun sequence assembly. Advances in Computers, 60, pp. 193-248, 2004. DOI: 10.1016/S0065-2458(03)60006-9
[36] Saito, M. and Matsumoto, M., Variants of Mersenne twister suitable for graphic processors. ACM Trans. Math. Softw., 39(2), pp. 1-12, 2013. DOI: 10.1145/2427023.2427029
[37] Sayadi, M.K., Hafezalkotob, A. and Naini, S.G.J., Firefly-inspired algorithm for discrete optimization problems: An application to manufacturing cell formation. Journal of Manufacturing Systems, 32(1), pp. 78-84, 2013. DOI: 0.1016/j.jmsy.2012.06.004
[38] Sutton, G.G., White, O., Adams, M.D. and Kerlavage, A.R., Tigrassembler: A new tool for assembling large shotgun sequencing projects. Genome Science and Technology, 1(1), pp. 9-19, 1995. DOI: 10.1089/gst.1995.1.9
[39] Vidal, P., Luna, F. and Alba, E., Systolic neighborhood search on graphics processing units. Soft Computing, 18(1), pp. 125-142, 2014. DOI: 10.1007/s00500-013-1041-7
[40] Vidal, P. and Olivera, A.C., A parallel discrete firefly algorithm on GPU for permutation combinatorial optimization problems. High Performance Computing, Communications in Computer and Information Science, 485, pp. 191-205. Springer, 2014.
[41] Yang, X.-S., Nature-inspired metaheuristic algorithms. Luniver Press, 2008.
[42] Yang, X-S., Firefly algorithms for multimodal optimization. In: Watanabe, O. and Zeugmann, T., eds., Stochastic algorithms: Foundations and applications, Lecture Notes in Computer Science, 5792, pp. 169-178, 2009. DOI: 10.1007/978-3-642-04944-6_14
[43] Yang, X-S., Firefly algorithm, stochastic test functions and design optimisation. Int. J. Bio-Inspired Comput., 2(2), pp. 78-84, 2010. DOI: 10.1504/IJBIC.2010.032124
[44] Yang, X-S. and He, X., Firefly algorithm: Recent advances and applications. Int. J. Swarm Intelligence, 1, pp. 36-50, 2013. DOI: 10.1504/IJSI.2013.055801
[45] Yang, X-S., Hosseini, S.S.S. and Gandomi, A.H., Firefly algorithm for solving non-convex economic dispatch problems with valve loading effect. Appl. Soft Comput., 12(3), pp. 1180-1186, 2012. DOI: 10.1016/j.asoc.2011.09.017

P.J. Vidal, received his BSc. degree in Computing Engineering, in 2008 from Universidad Nacional de la Patagonia Austral, Argentine; the degree of Dr. in Software Engineering and Artificial Intelligence, in 2106, from Universidad de Málaga, Spain. He is currently an adjunct professor at the Universidad de la Patagonia Austral, Unidad Académica Caleta Olivia, Argentine. Currently has a postdoctoral scholarship from National Council of Scientifics and Technological Researches from the Ministerio de Ciencia y Tecnología de la Nación, Argentine. He has experience in software engineering with emphasis on automation and graphic processing units research. His main research topics are: parallel and distributed computing, evolutionary algorithms and metaheuristics and numerical modeling.
ORCID: 0000-0001-6502-8010

A.C. Olivera, is an Adjunct Researcher at National Council of Scientifics and Technological Researches from the Ministerio de Ciencia y Tecnología de la Nación, Argentine. Dr. in Computer Science from Universidad Nacional del Sur, in Bahía Blanca, Argentine. She is adjunct professor at the Department of Exact and Natural Sciences of Universidad Nacional de la Patagonia Austral. She directs in several national and international projects. Her research focuses on bio-inspired algorithms and optimization. She has published several book chapters, articles in indexed journals and proceedings of refereed international conferences.
ORCID: 0000-0001-7825-1959