INSTITUTO POLITÉCNICO NACIONAL
CENTRO DE INVESTIGACIÓN EN COMPUTACIÓN

Algoritmos bioinspirados para generar redes de ordenamiento eficientes

TESIS

Que para obtener el grado de
Doctora en Ciencias de la Computación

Presenta
MCC. Blanca Cecilia López Ramírez

Directores de tesis
Dra. Nareli Cruz Cortés
Dr. Francisco José Rodríguez Henríquez

México D.F. Julio 2014

SIP-14 BIS

INSTITUTO POLITÉCNICO NACIONAL
SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

ACTA DE REVISIÓN DE TESIS

En la Ciudad de México, D.F. siendo las 12:00 horas del día 6 del mes de Junio de 2014 se reunieron los miembros de la Comisión Revisora de la Tesis, designada por el Colegio de Profesores de Estudios de Posgrado e Investigación del:

Centro de Investigación en Computación

para examinar la tesis titulada:

"Algoritmos bioinspirados para generar redes de ordenamiento eficientes"

Presentada por la alumna:

LÓPEZ	**RAMÍREZ**	**BLANCA CECILIA**
Apellido paterno	Apellido materno	Nombre(s)

Con registro: **B 1 0 2 2 5 0**

aspirante de: **DOCTORADO EN CIENCIAS DE LA COMPUTACIÓN**

Después de intercambiar opiniones los miembros de la Comisión manifestaron ***APROBAR LA TESIS***, en virtud de que satisface los requisitos señalados por las disposiciones reglamentarias vigentes.

LA COMISIÓN REVISORA

Directores de tesis

Dra. Nareli Cruz Cortés

Dr. Francisco José Rodríguez Henríquez

Dr. Carlos Fernando Aguilar Ibáñez

Dr. René Luna García

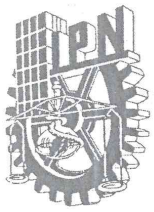
Dr. Carlos Artemio Coello Coello

Dr. Ricardo Barrón Fernández

PRESIDENTE DEL COLEGIO DE PROFESORES

Dr. Luis Alfonso Villa Vargas

INSTITUTO POLITECNICO NACIONAL
CENTRO DE INVESTIGACION
EN COMPUTACION
DIRECCION



INSTITUTO POLITÉECNICO NACIONAL

SECRETARÍA DE INVESTIGACIÓN Y POSGRADO

CARTA CESIÓN DE DERECHOS

En la Ciudad de _México, D.F. el día 20 del mes de junio del año 2014, el (la) que suscribe Blanca Cecilia López Ramírez alumno (a) del Programa de_Doctorado en Ciencias de la Computación con número de registro B102250, adscrito a Centro de Investigación en Computación , manifiesta que es autor (a) intelectual del presente trabajo de Tesis bajo la dirección de la Dra. Nareli Cruz Cortés y el Dr. Francisco José Rodríguez Henríquez y cede los derechos del trabajo intitulado Algoritmos bioinspirados para generar redes de ordenamiento eficientes, al Instituto Politécnico Nacional para su difusión, con fines académicos y de investigación.

Los usuarios de la información no deben reproducir el contenido textual, gráficas o datos del trabajo sin el permiso expreso del autor y/o director del trabajo. Este puede ser obtenido escribiendo a la siguiente dirección _bcelopez@gmail.com, nareli@cic.ipn.mx . Si el permiso se otorga, el usuario deberá dar el agradecimiento correspondiente y citar la fuente del mismo.

Blanca Cecilia López Ramírez

Nombre y firma

Resumen

El ordenamiento es una operación muy utilizada en el área de las Ciencias Computacionales. La frecuencia de uso (directa o indirecta) ha sido razón para buscar algoritmos *ad-hoc* que permitan mejorar el desempeño de las aplicaciones.

Se cuenta con una variedad de algoritmos de ordenamiento tradicionales conocidos como no adaptativos porque su desempeño depende del orden de los datos de entrada. Por otro lado, aquellos que tienen sus operaciones predefinidas antes de comenzar la ejecución del algoritmo de ordenamiento se les conoce como algoritmos no adaptativos. Dentro de este último grupo se encuentran las Redes de Ordenamiento. Además de contar con operaciones de comparación e intercambio predefinidas, éstas pueden ser ejecutadas al mismo tiempo siempre que cuenten con la independencia de sus elementos a ordenar.

Las Redes de Ordenamiento (RO) se han estudiado desde hace décadas y aunque se han encontrado resultados favorables, aún sigue siendo un reto encontrar RO con mínimo número de operaciones de comparación e intercambio con un mayor número de operaciones que puedan ser ejecutadas en una misma unidad de tiempo. Estos retos son los dos objetivos principales en este trabajo. El problema de construir una RO no es únicamente encontrar una combinación adecuada de comparadores que ordenen una secuencia de datos de entrada, también habrá que validar que ordenará cualquier permutación de un conjunto de datos de entrada y dicha operación de validación tiene un costo.

Las aplicaciones de las RO más conocidas se encuentran en hardware, sin embargo en este trabajo se aplica una RO en software para usarla en el ordenamiento de grandes cantidades de datos en unión con un algoritmo tradicional.

El trabajo de tesis consiste en los siguientes puntos: 1) Se propone diseñar RO con un mínimo número de operaciones de comparación e intercambio con una heurística llamada algoritmo de Sistema Inmune Artificial. Este método ayuda a encontrar diseños con diferentes configuraciones para diferentes tamaños de datos de entrada. 2) Se propone diseñar RO para una heurística bioinspirada llamada algoritmo de Optimización por Cúmulo de Partículas con representación binaria. La técnica ayudó a encontrar resultados favorables. 3) Se realiza una combinación de un algoritmo tradicional (Quicksort) más una RO con la intención de conocer si mejora el desempeño en comparación con el

Quicksort. Los resultados son muy interesantes, debido a que el uso de la RO demuestra que si mejora su desempeño empleando el Quick + RO.

Abstract

Sorting is a frequently studied procedure in Computer Science. Several approaches have been proposed to improve its performance. Sorting algorithms can be classified into two groups: the adaptive and the non adaptive ones. Adaptive algorithms are those whose operations depend on the input data. Non adaptive algorithms are those with their operations fixed before the algorithm execution. That is, the comparisons are independent to previous steps. Sorting Networks are an example of the non adaptive algorithms. The problem of finding optimal Sorting Networks is an open research area. In this thesis the problem of designing SN with minimal number of comparators and/or maximum parallelism is studied. A biological inspired heuristic called Artificial Immune Systems is utilized to find Sorting Networks with minimal number of comparators. Besides, a Particle Swarm Optimization algorithm is applied to design Sorting Networks with maximum parallelism. In addition to that, a strategy is proposed to join some Sorting Networks with the Quick-Sort algorithm to be able of sorting large input data with less comparisons. Some experimental work is designed to empirically demonstrate the proposals viability.

Agradecimientos

A Dios por su grandeza, entrega y amor. Por dejarme vivir una etapa más en mi proyecto de vida.
A Román Eduardo por ser mi cómplice, mi mejor amigo y el gran amor de mi vida. Gracias por soportar mis ausencias y todo lo que de esto deriva. A mis padres Don Román y Doña Coris por su compañía y apoyo.

A mis asesores la Dra. Nareli y el Dr. Francisco por creer en mi, por ser pacientes y persistentes con mi enseñanza. Su tiempo y dedicación a mi proyecto es invaluable, además de que agradezco la confianza que me brindaron para dejarme conocer a su linda familia. En especial quiero agradecerle a la Dra. Nareli por darme la oportunidad de ser amiga, no sólo por ser compañera en las buenas y en las malas, por escuchar y conceder un comentario como consejo, sino que me ha enseñado la valía de nuestro género.

A mis amigos y compañeros en este caminar. Luis, Alfonso, Christian e Israel, todos ellos han dejado en mi buenas experiencias, ahora son parte de mi familia. A Luis por su paciencia y su entereza, por confiar en mi y compartir tus conocimientos en más de una vez. Eres un gran amigo. Alfonso, gracias por ser mi confidente y amigo. Siempre pude contar contigo aunque antes tuviera que escuchar un comentario de reproche. Christian, eres un gran amigo que me brindó la confianza para acercarme a tu familia. Siempre me diste compañía y consuelo. A Israel, por dejarme conocerte y entrar en tu mundo, sabes ser un gran amigo.

A Sandra, por su amistad incondicional. Sé que siempre contaré contigo. A Paco, Obdus, Betty Garro y Elena por su amistad e impulso.

A Chivis y Carmelita agradezco por su apoyo en todas las cuestiones académicas, pero más su paciencia, amistad y comprensión.

A Giovanni por darme su tiempo y vivencias inesperadas tan agradables. Por su comprensión para mis cambios de humor en las buenas y en las malas. Ahora eres parte de mi historia.

Se agradece el apoyo a Conacyt con los proyectos 132073 y 180421. También al proyecto SIP20140147.

Índice general

Índice de figuras

Índice de cuadros

Capítulo 1

Introducción

1.1. Motivación

La operación de ordenamiento es quizá una de las más empleadas en las Ciencias de la Computación. Su uso frecuente y repetitivo hace necesaria la búsqueda de mejores técnicas para ordenar un conjunto de datos.

Actualmente se cuenta con una variedad de técnicas de ordenamiento *ad-hoc*. Su desempeño es evaluado por el conjunto de operaciones que efectúa entre los datos. Los métodos de ordenamiento se pueden clasificar en adaptativos y no adaptativos. En el primer grupo, el proceso es dependiente de los datos a ordenar y a este grupo pertenecen los algoritmos tradicionales ("quicksort", "shellsort", "burbuja", etc.). Por otro lado, los no adaptativos cuentan con una secuencia de operaciones predefinida, así que su desempeño siempre será el mismo. En esta clasificación se encuentran las Redes de Ordenamiento (RO).

Una RO es un método de ordenamiento sin memoria ("oblivious") que permite ordenar n objetos mediante el uso de un número fijo de operaciones binarias (llamados también comparadores) los cuales están distribuidas en un orden predefinido. Este es quizás uno de los algoritmos más eficientes de ordenamiento que, además de contar con operaciones preestablecidas, tiene la cualidad del paralelismo de las mismas. El desempeño de una RO se puede medir mediante:

1. El mínimo número de comparadores para un tamaño de entrada n.
2. El mínimo número de pasos (tiempos) que podría tardar la RO en terminar su tarea si fuera paralelo.

Cabe señalar que una RO eficiente es aquella con un mínimo número de comparadores conocido y, una RO que tenga un mínimo número de pasos para ordenar se identifica

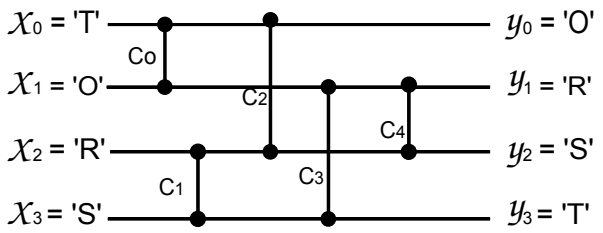


Figura 1.1: Ejemplo de una Red de Ordenamiento para 4 datos de entrada.

como una RO rápida.

Diseñar una RO óptima es un problema combinatorio que ha sido estudiado por algunos investigadores de las Ciencias Computacionales durante décadas. La construcción de una secuencia mínima de comparadores en una RO es un problema NP-hard [53].

Las RO son algoritmos altamente eficientes, sin embargo no se conocen redes óptimas para ordenar una gran cantidad de datos. Las RO más estudiadas son para tamaños de entrada $n = 10$, $n = 12$ y $n = 16$.

1.2. Planteamiento del problema

Un conjunto de comparadores conforman una RO. Un comparador c recibe dos datos de entrada (a, b) para ser comparados, si $a > b$ entonces se aplica la operación de intercambio tal que $a < b$. Generalmente las RO son representadas esquemáticamente como en la Figura 1.1. Cada línea horizontal representa el trayecto de un dato de entrada. Cada línea vertical es un comparador que evalúa los elementos entre su extremo superior e inferior. Si el valor del extremo superior es más grande que el extremo inferior entonces intercambia los datos. Por ejemplo, en una RO donde se quieren ordenar 4 datos de entrada, los valores de x_0, x_1, x_2, x_3 son colocados del lado izquierdo de la RO y un dato por cada línea horizontal, como en la Figura 1.1. Las líneas verticales etiquetadas como c_0, c_1, c_2, c_3, y c_4 son los comparadores. Cada comparador recibe dos valores, esto es, c_0 recibe los valores x_0 y x_1; c_1 recibe los valores de x_2 y x_3, y así sucesivamente. De esta forma, los comparadores son ejecutados de izquierda a derecha según su aparición. Finalmente, la lista ordenada y_0, y_1, y_2, y_3 se obtiene al lado derecho de la RO y debe cumplir con $y_0 \leq y_1 \leq y_2 \leq y_3$.

Usualmente el tiempo discreto requerido para el ordenamiento se refiere al número de pasos o capas que la RO debería considerar después de verificar la independencia de los comparadores que pueden ser ejecutados en paralelo. Una RO se considera *rápida* si tiene un bajo número de capas, donde cada capa está compuesta por un subconjunto de comparadores que puedan realizar sus operaciones concurrentemente sin interferir entre ellas.

Una RO como cualquier algoritmo, es necesario garantizar que ordenará cualquier per-

mutación de los datos de entrada.

En este trabajo se explora el área de las RO para conocer nuevas configuraciones de comparadores, y proponer mecanismos para construir RO. Además se implementa una RO grande en un problema real para grandes cantidades de datos.

1.3. Objetivos

1.3.1. Objetivo General

Diseñar RO eficientes y rápidas mediante técnicas evolutivas y bio-inspiradas.

1.3.2. Objetivos específicos

1. Analizar aspectos teóricos y prácticos para minimizar el número de comparadores en la construcción de RO.
2. Analizar aspectos teóricos y prácticos para minimizar el número de pasos en la RO.
3. Diseñar RO eficientes empleando heurísticas evolutivas.
4. Proponer un diseño de construcción eficiente para RO.
5. Estudiar otros métodos de construcción para RO grandes ($n > 16$).
6. Aplicar una RO en un problema práctico para grandes cantidades de datos.

1.4. Aplicaciones de una Red de Ordenamiento

Hoy en día el rendimiento de un equipo computacional puede ser medible por la capacidad de realizar múltiples tareas de manera simultánea. Recientemente se realizan trabajos de arquitecturas para el trabajo paralelo y cada vez más se requiere de la implementación de algoritmos *ad-hoc* [39]. Existen aplicaciones de RO para la elaboración de arquitecturas de hardware, en el diseño de simplificadores de circuitos mediante un ordenamiento paralelo. Otra aplicación posible es establecer la prioridad de solicitudes y respuestas de paquetes que soportan una diversa comunicación de tráfico en un conmutador de redes [5, 44, 31].

Aplicar RO grandes para trabajos donde se tiene grandes cantidades de datos no es atractivo por su alta complejidad con la implementación, sin embargo sus cualidades

son útiles para áreas donde los procesos aprovechan las cualidades de su diseño, tal como tiempos de ejecución independientes de la configuración de entrada, por ejemplo en el área de Criptografía.

1.5. Alcance y contribuciones

El estudio de esta tesis tiene las siguientes contribuciones:

1. Se diseñaron RO eficientes y rápidas con diferente configuración de comparadores a las conocidas en la literatura.
2. Se propuso una técnica de construcción competitiva para conocer la "calidad" de los comparadores.
3. Se llevó a cabo la implementación de dos técnicas bio-inspiradas para ayudar en el proceso de búsqueda. La primera, un algoritmo de sistema inmune artificial para diseñar RO eficientes y la segunda un algoritmo de optimización por cúmulo de partículas para buscar RO rápidas.
4. Se diseñaron RO grandes competitivas con las conocidas en la literatura especializada.
5. Se diseñó una RO grande a partir de dos RO eficientes de $n = 16$ (se obtiene una RO para $n = 256$). Ésta se emplea como herramienta para ayudar en el desempeño del algoritmo de Quicksort.
6. Se calcula el espacio de búsqueda en capas para una RO.

1.6. Publicaciones

1. Blanca López and Nareli Cruz-Cortés, "Designing Minimal Sorting Networks using a Bio-inspired Technique", MICAI, Consorcio Doctoral, Noviembre, 2012.
2. Blanca López y Nareli Cruz-Cortés,"Algoritmo de optimización por cúmulo de partículas", Research in Computing Science, vol 72, pp.99-122, 2014.
3. Blanca López and Nareli Cruz-Cortés,"On the Usage of Sorting Networks to Big Data", ABDA: International Conference on Advances in Big Data Analytics, Las Vegas, Nevada, E.U.A., 21-24 july, 2014
4. Blanca López and Nareli Cruz-Cortés,"Designing Minimal Sorting Networks using a Bio-inspired Technique". Computación y Sistemas ISSN: 1405-5546. Aceptado 2014

1.7. Estructura general del documento

En general el contenido del trabajo es el siguiente:

- El **Capítulo 2** es un apartado que presenta conceptos básicos de algoritmos de ordenamiento tradicionales, algunos ejemplos para su implementación y además sus ventajas y desventajas.
- En el **Capítulo 3** se describen los conceptos básicos de las RO y el trabajo previo.
- El **Capítulo 4** expone la propuesta de construcción de RO para encontrar comparadores prometedores. Se utiliza un algoritmo bio-inspirado (algoritmo del sistema inmune artificial) para realizar la búsqueda de RO con el menor número de comparadores.
- En el **Capítulo 5** se presenta un diferente planteamiento para encontrar RO maximizando la paralelización con ayuda de un algoritmo de optimización por cúmulo de partículas.
- En el **Capítulo 6** se propone el uso de RO en unión con una técnica clásica, con el objetivo de mejorar su comportamiento frente a la técnica clásica. En el apartado se pueden observar los experimentos y resultados.
- En el **Capítulo 7** se tienen las conclusiones de este trabajo.
- En el **Apéndice A** se muestran las Redes de Ordenamiento eficientes diseñadas por la técnica propuesta.
- En el **Apéndice B.** está ilustrada una de las Redes de Ordenamiento de tamaño extenso.
- En el **Apéndice C.** está el artículo aceptado titulado "Designing Minimal Sorting Networks using a Bio-inspired Technique".

Capítulo 2

Algoritmos de ordenamiento tradicionales

Una amplia variedad de algoritmos de ordenamiento han sido concebidos y aunque tienen el mismo propósito, cuentan con ciertas peculiaridades que su comportamiento es diferente frente a ciertos escenarios. En este apartado se describen algunos de los algoritmos de ordenamiento tradicionales, así como sus características ventajas y desventajas en ciertos escenarios.

2.1. Definición de ordenamiento

Ordenar es una operación la cual coloca en determinado orden un conjunto de elementos. El ordenamiento ha sido facilitador para otras tareas, como la búsqueda en una gran cantidad de datos. Según Hosam [42] las dos operaciones más comunes para el almacenamiento de datos en una computadora es el ordenamiento y la búsqueda. Sin el ordenamiento la tarea de búsqueda es más complicada.

El problema de ordenamiento se puede ver como una lista A con tamaño de datos n, los elementos de la lista son X_1, X_2, X_3, ... X_n que se quiere ordenar de forma ascendente, el objetivo es diseñar un algoritmo eficiente tal que se pueda crear una permutación $X_{(1)}$, $X_{(2)}$,$X_{(3)}$, ... $X_{(n)}$, donde $X_{(i)}$ es el i-ésimo más pequeño (i-ésimo orden estático) entre los elementos de la lista. El término más pequeño se refiere al valor más pequeño en relación con los otros elementos de la lista. Por ejemplo, si la lista $A = X_1, X_2, X_3, ...X_n$ son números enteros y " $\leq$" es la comparación aritmética, al final la lista ordenada A en este caso será $X_{(1)}, X_{(2)}, X_{(3)}, ...X_{(n)}$ y sus elementos estarán ordenados en forma creciente.

Al buscar un algoritmo de ordenamiento conveniente para cierto problema, es importante considerar ciertos factores para su implementación tanto en software como en

hardware [30, 47, 52]. La siguiente sección describe una manera de analizar un algoritmo.

2.1.1. Análisis de un algoritmo

La sencillez de un algoritmo facilita su verificación, su mantenimiento y el estudio de su eficiencia.

Cuando tenemos algoritmos con la misma finalidad pero cuentan con procesos diferentes es importante conocer su rendimiento o comportamiento para saber cuál de ellos es más eficiente y bajo qué términos puede tener un mejor desempeño.

El rendimiento es posible medirlo de varias maneras aunque las más comunes son el tiempo y el tamaño en memoria. El primero es el más difícil de medir, el segundo es posible medirlo con la cantidad de datos de entrada, siempre y cuando no se use recursividad o memoria dinámica.

Las tareas que realizan los algoritmos no utilizan la misma cantidad de recursos aunque el tamaño de entrada sea el mismo, depende de los datos a ordenar, de la calidad del código objeto generado, de la naturaleza del código fuente y la complejidad misma de la técnica. Lo que si es obvio es que la técnica emplea más recursos si el tamaño de entrada crece, y por esto, el rendimiento se mide con base en el tamaño de entrada.

En el supuesto de que alguien quisiera saber cuál algoritmo es mejor que otro para ordenar cierta cantidad de datos, es complicado determinarlo únicamente con el factor tiempo, a menos que todos los algoritmos se codificaran en la misma máquina y se realizara de manera justa la evaluación para cada uno de ellos. Un factor que puede ayudar a determinar cuál es mejor, es el número de comparaciones que realiza el algoritmo.

El análisis y estudio de los algoritmos es una herramientas de las Ciencias Computacionales considerado como abstracto cuando no existe un programa que pueda determinar de forma precisa dicho estudio. Así que generalmente se usan algunas disciplinas matemáticas para respaldar el análisis del estudio, de no ser así quedaría en términos abstractos [54, 64] .

Existen dos posibles maneras de analizar el tiempo que emplea un algoritmo:

1. *A priori* o teórica, consiste en obtener una función que acote con un límite superior e inferior el tiempo de ejecución dado un tamaño de entrada de datos.
2. *A posteriori*, consiste en medir el tiempo de ejecución del algoritmo para un tamaño de entrada de datos.

La primera nos ofrece estimaciones acerca del comportamiento de los algoritmos sin tener que implementarlos y probarlos en la máquina, mientras que la segunda otorga

un información más precisa y real del desempeño del algoritmo [41, 67].

El comportamiento de un algoritmo puede cambiar radicalmente para diferentes entradas y de hecho para muchos algoritmos el tiempo de ejecución es en realidad una función de la entrada específica y no sólo el tamaño de ésta. Si $C(n)$ representa el número de comparaciones que ejecuta un algoritmo de ordenamiento para ordenar n elementos, entonces se consideran tres medidas de velocidad:

- Peor caso: representa la cantidad más grande de operaciones de comparación que el algoritmo requiere para ordenar n elementos.
- Mejor Caso: representa la menor cantidad de operaciones de comparación que el algoritmo requiere para ordenar n elementos.
- Caso Promedio: es el promedio de la cantidad de operaciones de comparación para ordenar los n elementos.

2.2. Estrategia divide y vencerás

La técnica llamada *divide y vencerás* consiste en la división de problemas en subproblemas del mismo tipo pero de menor tamaño. Este mecanismo de dividir los problemas en subproblemas se aplica continuamente mientras su tamaño sea adecuado para resolverlo [24, 18]. Los pasos para resolver un problema con la técnica de divide y vencerás son los siguientes:

1. La división. El problema se descompone en k subproblemas del mismo tipo aunque de menor tamaño ($1 \leq k \leq n$).
2. La solución independiente de cada subproblema. Como el tamaño de los subproblemas es estrictamente menor que el problema original, esto garantiza la convergencia hacia los casos elementales o base.
3. Combinar las soluciones obtenidas en el paso anterior para solucionar el problema original.

Al utilizar el diseño de divide y vencerás es importante considerar el número de k subproblemas y el tamaño de k, pues es relevante para la eficiencia del algoritmo resultante. Se dice que es conveniente un número k pequeño e independiente de una entrada. Cuando se maneja el algoritmo de recursividad se conocen como algoritmos de simplificación, porque reduce el problema en otro del mismo tipo pero más pequeño, se le conoce también como estrategia de divide y vencerás. Su ventaja es que tienen unos tiempos de ejecución buenos, de orden logarítmico o lineal.

2.3. Algoritmo de ordenamiento burbuja (*bubblesort*)

Es uno de los algoritmos de ordenamiento más antiguos e ineficientes por la cantidad de comparaciones que realiza [3]. Su implementación es fácil pero es de los algoritmos más lentos que existen, según [43]. Su nombre proviene del hecho de que los elementos "flotan como una burbuja" y considerando que el ordenamiento que se quiere realizar es ascendente. En cada iteración los elementos pequeños ascienden o mientras los mayores se hunden.

En general, el algoritmo consiste en comparar e intercambiar entre elementos adyacentes de una lista de datos (A) de tamaño n. La comparación entre los elementos adyacentes ($A[j], A[j+1]$) realizan un intercambio siempre que $A[j] < A[j+1]$. El ordenamiento no mejora en las primeras iteraciones, los elementos mayores se irán colocando al final de la lista, mientras los elementos más pequeños subirán a las primeras posiciones.

Algoritmo 1 Ordenamiento burbuja

Entrada: A, n, j

```
1   begin
2       for (iteracion = 0; iteracion < (n-1); iteracion ++)
3           for (j=0; j<(tam-iteracion)-1; j++)
4               if ( A[j] > A[j+1])
5                   intercambia (A[i], A[j])
6               end
7           end
8       end
9   end
```

Salida: A

Veamos un ejemplo, en la Figura 2.1 se presenta una lista de elementos A que contiene $n = 11$ datos enteros desordenados, aplicando el Algoritmo 1 en la primera iteración al término de las operaciones, se obtiene una lista con el mayor elemento en su posición (valor del elemento: 9). Se puede observar como en cada lista se van comparando e intercambiando (números en un cuadrado punteado) los elementos en parejas, al término de la iteración, el mayor elemento de la lista se instala en su lugar. En la segunda iteración se ordena el elemento segundo mayor que es el 8, y así sucesivamente hasta que la última iteración sean comparados los elementos $A[0]$ y $A[1]$.

El número de iteraciones es de $n-1$. En cada iteración quedará en su lugar el elemento con mayor valor (desordenado).

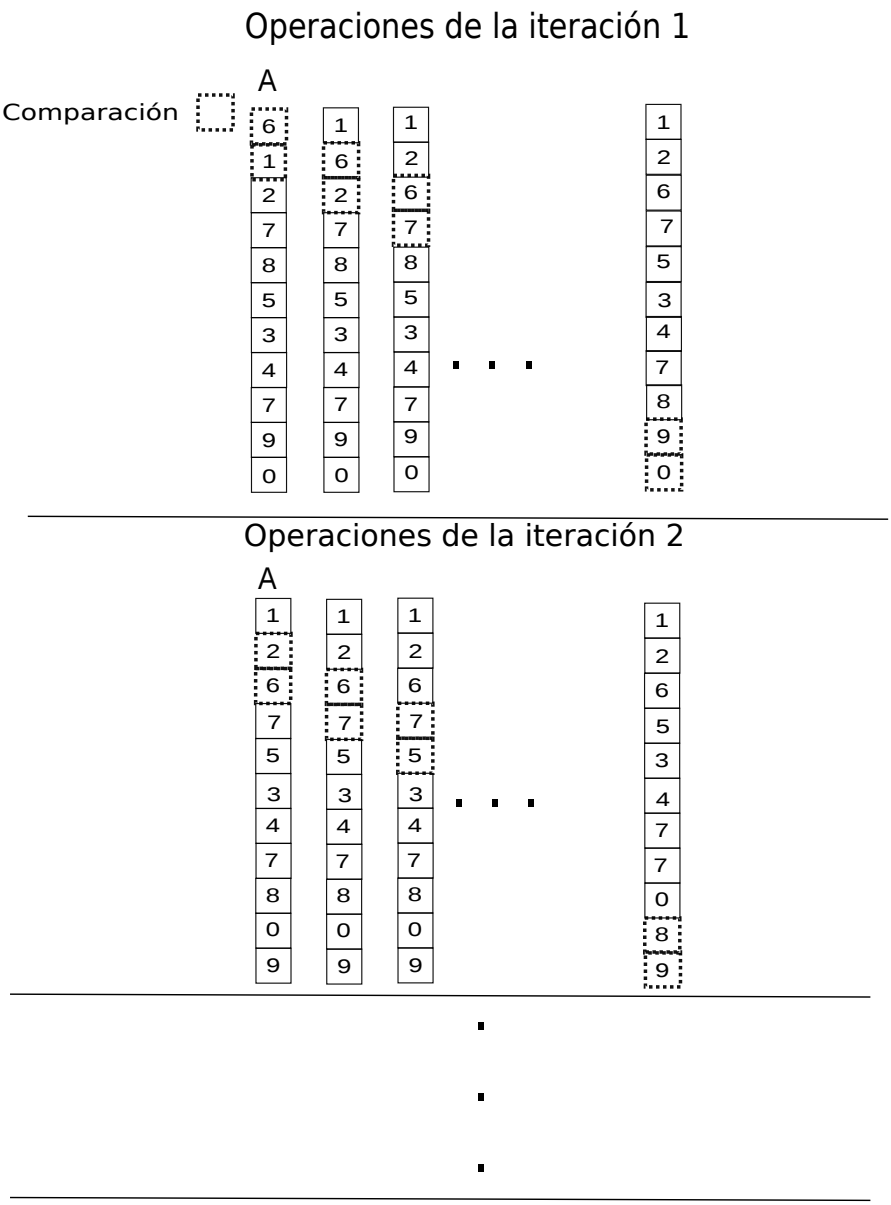


Figura 2.1: Ejemplo de un algoritmo de ordenamiento burbuja.

Tiempo de ejecución. El recorrido de la lista se ejecuta $n*n$ veces, por lo que su complejidad con respecto al número de operaciones es de $O(n^2)$.

- Peor caso: $O(n^2)$
- Mejor caso: $O(n)$
- Caso Promedio: $O(n^2)$
- Complejidad de espacio: $O(n)$

Ventajas:

- Fácil de implementar.
- No requiere de memoria adicional.

Desventajas:

- Muy Lento.
- Realiza un gran número de comparaciones.
- Realiza muchos intercambios.

2.4. Algoritmo de ordenamiento por selección

Es un algoritmo de ordenamiento sencillo de implementar aunque realiza muchas operaciones de comparación. Los pasos son los siguientes:

1. Por cada posición i de la lista A, busca el elemento de valor menor (min).
2. Intercambia el elemento de la posición min por el elemento de la posición i.
3. El paso 1 y 2 se repite hasta el elemento $n - 1$.

El Algoritmo 2 muestra los pasos del ordenamiento por selección.

Como ejemplo se muestra en la Figura 2.2 una lista a ordenar de 11 elementos. Por cada elemento de la lista se realiza los siguientes dos pasos: 1) Se busca el menor de los valores de la lista; en la iteración 1 el menor de los valores es de 0, 2) el siguiente paso es intercambiar el valor encontrado en el paso 1 con el elemento correspondiente de la lista, en este caso, el $A[10] = 6$ por el $A[0] = 0$. Al término de la iteración el número con valor más pequeño de los elementos ya está en su lugar. Para la segunda iteración se buscará el segundo elemento menor del resto de los elementos. Así que la siguiente iteración (2) deberá realizar el paso 1, buscando el elemento con el menor valor (de los elementos restantes), en este ejemplo el elemento se encuentra en su lugar. Las iteraciones subsecuentes se realizan de la misma forma hasta $n - 1$.

Tiempo de ejecución. El recorrido de la lista se ejecuta n veces, y la búsqueda hace un recorrido de tamaño igual a los elementos por ordenar. La complejidad del algoritmo es de $O(n^2)$. El comportamiento de este algoritmo no depende del valor de los datos, sino del tamaño de entrada de los datos.

Algoritmo 2 Ordenamiento por selección

Entrada: A, n

```
1    begin
2        for (i=0; i<n; i++)
3            min = i;
4            for (j=i+1; j<n; j++)
5                if (A[j]<A[min])
6                    min = j;
7                end
8                temp = A[i]
9                A[i] = A[min]
10               A[min] = temp
11           end
12       end
13   end
```

Salida: A

- Peor caso: $O(n^2)$
- Mejor caso: $O(n^2)$
- Caso Promedio: $O(n^2)$
- Complejidad de espacio: $O(n)$

Ventajas:

- Fácil de implementar.
- No requiere de memoria adicional.
- Realiza pocos intercambios con respecto a otros algoritmos como el algoritmo burbuja.
- Rendimiento constante, no existe diferencia entre el peor caso o mejor caso.

Desventajas:

- Es lento.
- Realiza un gran número de comparaciones.

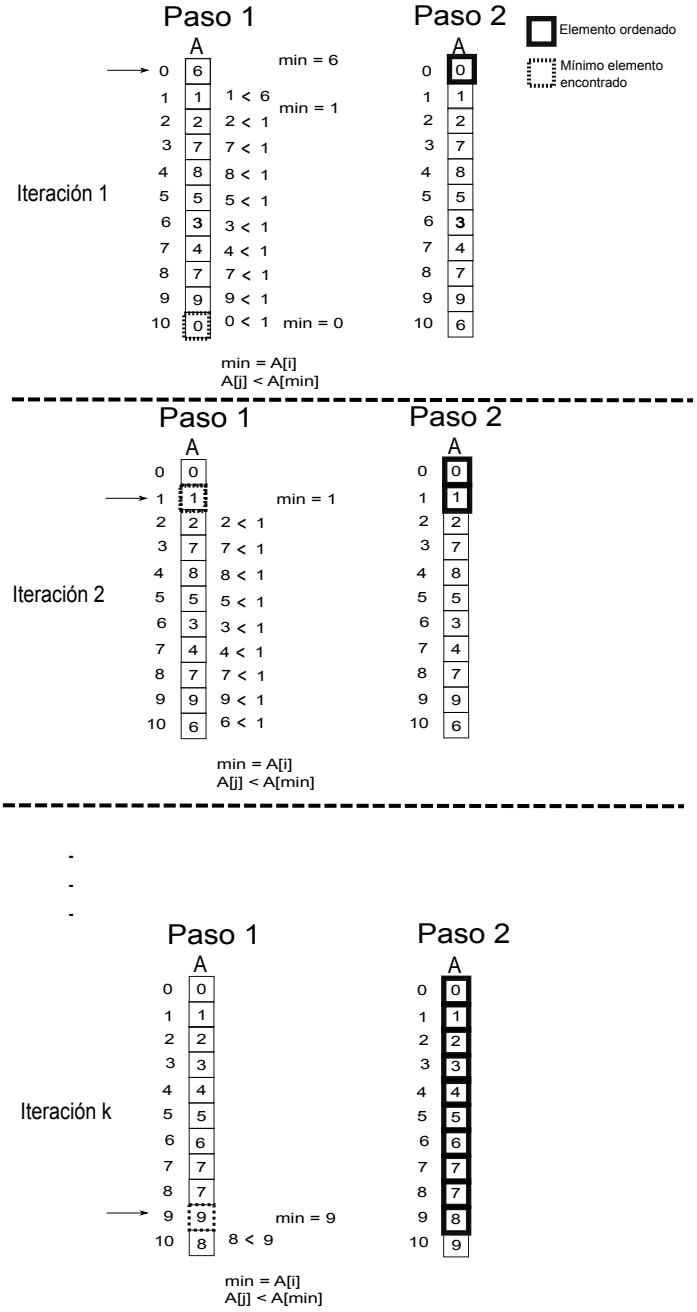


Figura 2.2: Ejemplo del algoritmo de ordenamiento por selección.

2.5. Algoritmo de ordenamiento por inserción

Este algoritmo consiste en tomar elementos de una lista desordenada e insertar cada elemento en una sublista que está ordenada. El Algoritmo 3 presenta los pasos para ordenar una lista A. No requiere de una estructura alterna y por cada elemento i de la lista A recorre todos los elementos de la sublista ordenada para encontrar su posición.

En la Figura 2.3 se muestra un ejemplo.

En la iteración 1 inicia con un elemento en la sublista ordenada (con el valor de 6), la sublista desordenada contiene $n - k$ elementos. El primer elemento de la lista desordenada tiene el valor de 1 y se compara con cada elemento de k (ver Figura 2.3 iteración 1 inciso b), así que se cumple que $A[k] < A[k-1]$, donde k corresponde a los índices de la sublista ordenada, entonces se intercambian los valores de los elementos (ver iteración 1 inciso b). Como se ha terminado el recorrido en la sublista desordenada entonces se termina la iteración 1 de la Figura 2.3. Una vez que ha sido incluido en la lista desordenada, la sublista ordenada incrementa en uno.

En la iteración 2 de la Figura 2.3, el siguiente elemento de la sublista desordenada es el elemento con valor de 2, este elemento es comparado con cada elemento de la sublista ordenada para encontrar su posición; en el inciso a) de la iteración 2 se compara con el valor 6 por lo cual se intercambian los elementos puesto que $2 < 6$, posteriormente se compara el 2 con el 1 pero no aplica el intercambio. En el inciso c) de la Figura 2.3 los elementos de la sublista terminan ordenados y termina la Iteración 2. En la iteración 3 de la Figura 2.3 el elemento que se quiere ordenar es el 7, y se realiza la comparación con cada elemento de la sublista ordenada sin intercambios. El proceso de las iteraciones continúa hasta el elemento $n - 1$ a ordenar, donde se observa las comparaciones y los intercambios que se llevan a cabo con cada elemento de la lista ordenada.

Algoritmo 3 Ordenamiento por inserción

Entrada: A, n

```
1   begin
2       for (c = 1 ; c ≤ n - 1; c++)
3            k = c
4           while (k > 0&& A[k] < A[k-1])
5               temp = A[i]
6               A[i] = A[min]
7               A[min] = temp
8               k−−
9           end
10      end
11  end
```

Salida: A

Tiempo de ejecución. El recorrido de la lista desordenada es de $n-1$ veces, y el recorrido de la sublista ordenada es 1,2,3 hasta $n-2$ en el peor caso. La complejidad del algoritmo es $O(n^2)$.

- Peor caso: $O(n^2)$

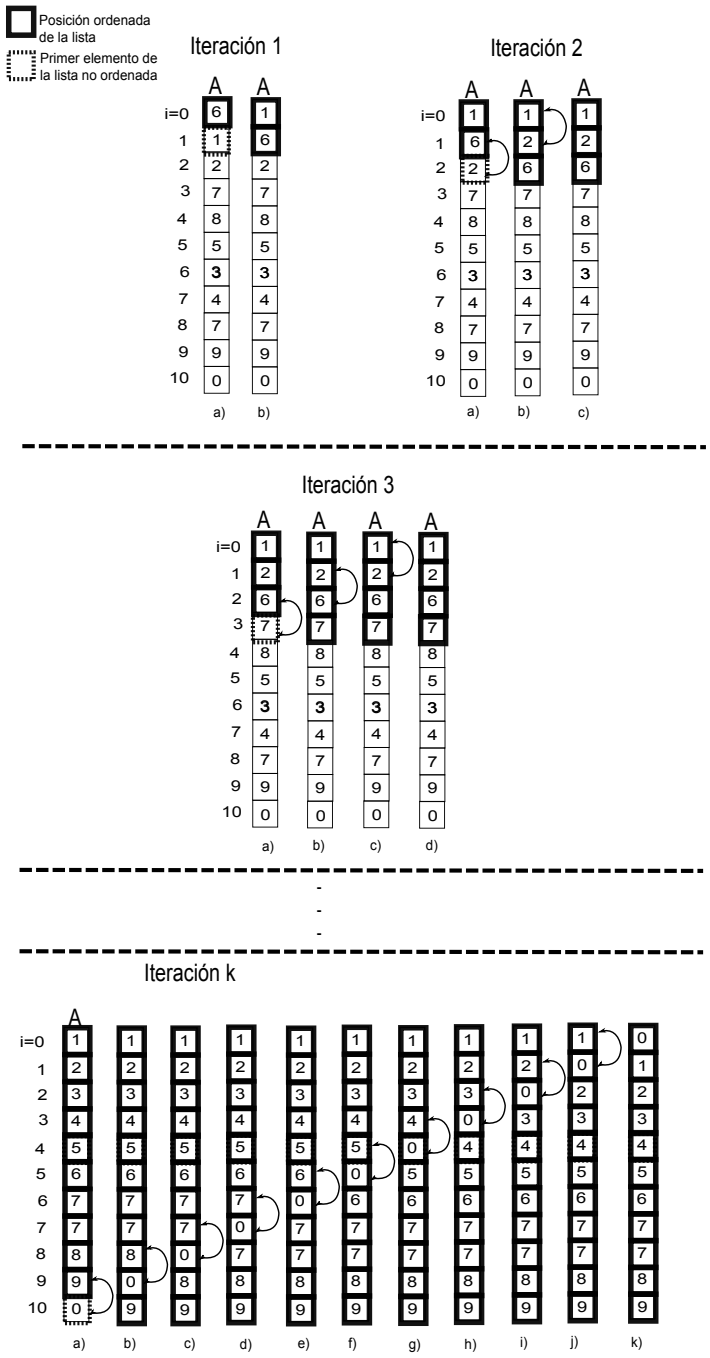


Figura 2.3: Ejemplo del algoritmo de ordenamiento por inserción.

- Mejor caso: $O(n)$
- Caso Promedio: $O(n^2)$
- Complejidad de espacio: $O(n)$

Ventajas:

- Fácil de implementar.
- Requiere de poca memoria.
- Realiza muchos intercambios.

Desventajas:

- Es lento.
- Realiza un gran número de comparaciones.

2.6. Algoritmo de ordenamiento por montículos (*Heapsort*)

Es un algoritmo desarrollado por J. Williams en 1964, tiene la característica de utilizar montículos *heap* para ordenar los elementos de un conjunto de datos de entrada. Un montículo es un árbol binario que puede ser vacío o tener ramificaciones y debe cumplir con:

1. El valor de cada nodo debe ser mayor o igual que los valores de los hijos.
2. El árbol está balanceado.

En general, el algoritmo consiste en organizar los elementos de una lista A en el montículo, donde al final el nodo raíz será el elemento de mayor valor (considerando que se quieren ordenar en forma ascendente) y, una vez que estén organizados el elemento raíz es extraído para que nuevamente sea reorganizado el montículo hasta terminar con todos los nodos.

Las Figuras 2.4 y 2.5 presentan un ejemplo del algoritmo por montículos. Dada una lista A con 11 elementos como datos de entrada, el primer paso es organizar los datos en un montículo y una vez que ha sido organizado el árbol (ver Figura 2.4) se extrae de la raíz el elemento con valor de 9 (el cual tiene el valor más grande). El paso 2 consiste en colocar el 9 en la posición $n - i$, donde $i = 1$ e incrementa con cada eliminación del nodo raíz en el montículo. El elemento que se encontraba en la posición $n - i$ es colocado en la primera posición de la lista, aquí termina la primera iteración. Para la segunda iteración se construye nuevamente el montículo de tamaño $n-i$, dejando fuera el elemento ya ordenado (9), en la Figura 2.5 se muestra el montículo como paso 1 y, en el paso 2 la lista A después de haber extraído la raíz del árbol (8) y colocarla en

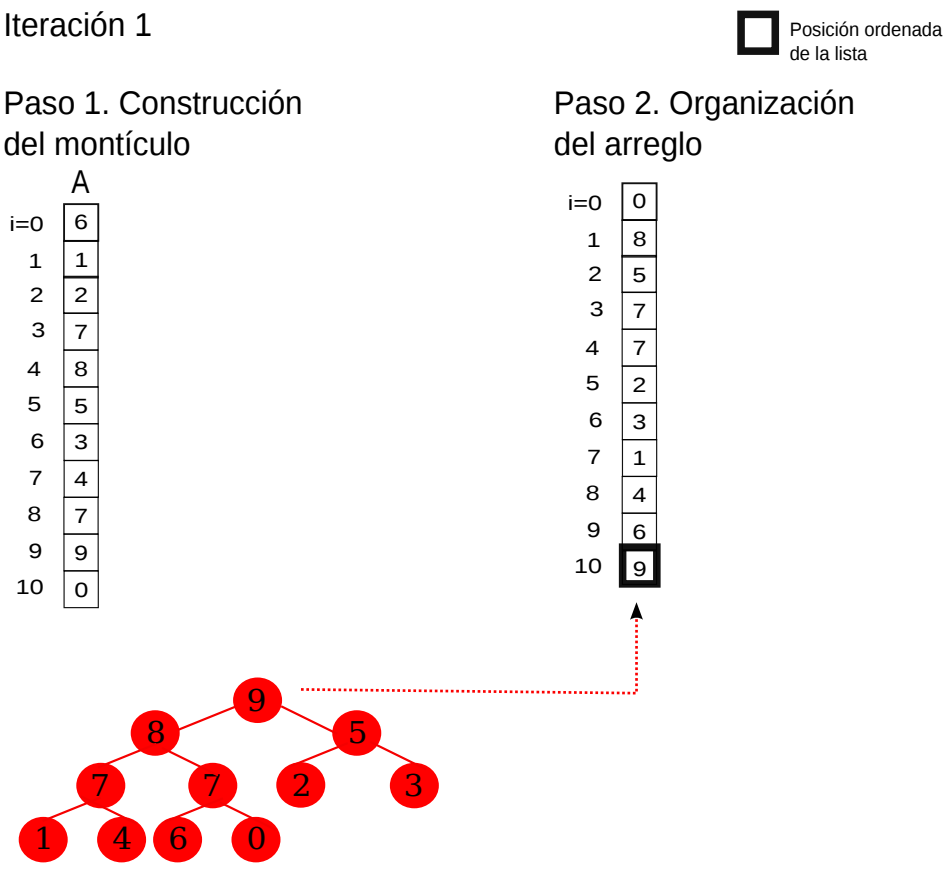


Figura 2.4: Ejemplo del algoritmo de ordenamiento por montículos. Iteración 1.

su posición ordenada. Estos pasos son repetidos hasta que se terminen los nodos del montículo.

Tiempo de ejecución. El método para construir el montículo es $log(n)$ y los recorridos para insertar y eliminar los elementos del montículo son de orden lineal (n), así que la complejidad del algoritmo es de $O(n\ log\ n)$.

- Peor caso: $O(n\ log\ n)$
- Mejor caso: $O(n\ log\ n)$
- Caso Promedio: $O(n\ log\ n)$
- Complejidad de espacio: $O(n)$

Ventajas:

- Trabaja mejor con datos desordenados.

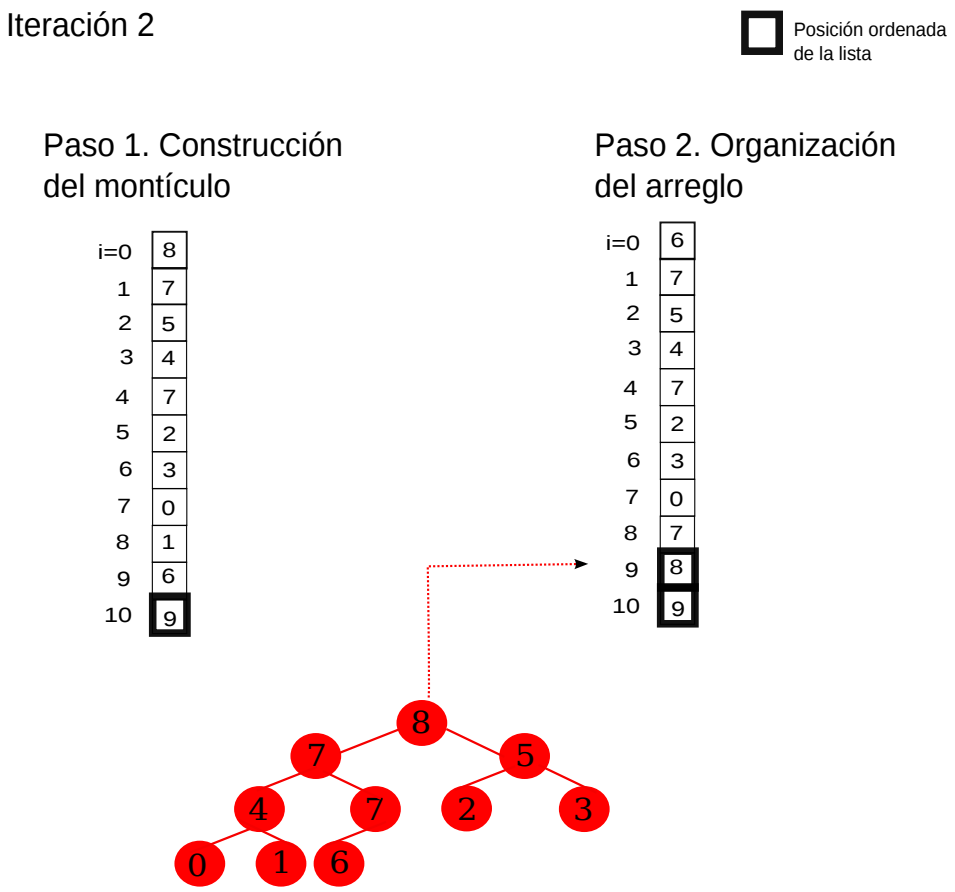


Figura 2.5: Ejemplo del algoritmo de ordenamiento por montículos. Iteración 2.

- No utiliza memoria adicional.

- Este algoritmo tiene una complejidad de $O(n\ log\ n)$ comparaciones y movimientos hasta en el peor de los casos [21]. Su desempeño en promedio es igual que el del *quicksort* y mejor en el caso peor.

Desventajas:

- Es lento.

- Es inestable con datos del mismo valor.

Algoritmo 4 Ordenamiento por montículos

Entrada: A, n

```
1     begin
2         temp = A[i]
3         j=i*2
4         while (j ≤ n)
5             if ((j<n)&&(A[j]¡A[j+1]))
6                 j++
7             end
8             if (A[j]<A[j/2])
9                 break
10            end
11             A[j/2]=A[j]
12        end
13         j=j*2
14         A[j/2]=temp
15    end
```

Salida: A

2.7. Algoritmo de ordenamiento por mezcla (*Mergesort*)

Surge en 1945 por John Von Newmann, la idea de este algoritmo es dividir la lista de elementos a ordenar en sublistas y combinarlas con el objetivo de llevar a cabo un ordenamiento parcial, este procedimiento se lleva a cabo de forma recursiva. El algoritmo de mezcla emplea la estrategia de divide y vencerás como el quicksort.

Los pasos para el algoritmo de mezcla son:

1. Si el tamaño de la lista es 0 ó 1 entonces ya está ordenada.
2. Se divide la lista no ordenada en dos sublistas de la mitad del tamaño.
3. Se mezclan las dos sublistas para formar una ordenada.

Para su implementación veamos el código del Algoritmo 5 de partición de la lista A la cual divide en dos cada sublista recursivamente hasta que el tamaño de la sublista es de 1. El procedimiento de mezcla (ver Algoritmo 6) ordena las sublistas comparando e intercambiando los elementos hasta que la lista inicial queda finalmente ordenada (ver ejemplo en la Figura 2.6.).

- Peor caso: $O(n\ log\ n)$

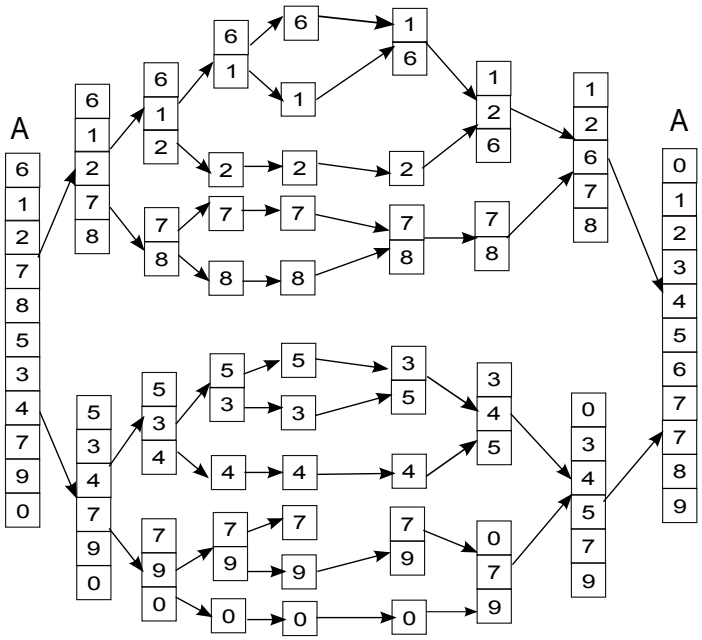


Figura 2.6: Ejemplo del algoritmo de ordenamiento por mezcla.

Algoritmo 5 Particion.

Entrada: A, min, max

```
1    begin
2        if (min < max)
3            mid = (min + max)/2
4            particion(A, min, mid)
5            particion(A, mid + 1, max)
6            mezcla(A, min, mid, max)
7        end
8    end
```

Salida: A

- Mejor caso: $O(nlogn)$
- Caso Promedio: $O(nlogn)$
- Complejidad de espacio: $O(2n)$

Ventajas:

1. Más rápido que el algoritmo de ordenamiento por montículos.
2. Ordenar recursivamente cada mitad.

Algoritmo 6 Mezcla

Entrada: A, n

```
1     begin
2         i,j,k,m
3         i = min
4         m = mid + 1
5         for (i=min; j≤ mid && m ≤ max ; i++)
6             if (arr[j] ≤ arr[m])
7                 temp[i]=arr[j]
8                 j++
9             else
10                tmp[i]=arr[m]
11                m++
12            end
13        end
14        if (j > mid)
15            for (k=m; k ≤ max; k++)
16                tmp[i]=arr[k]
17                i++
18            end
19        else
20            for (k=j; k ≤ mid; k++)
21             tmp[i]=arr[k]
22            end
23        end
24        for (k=min; k ≤ max; k++)
25             arr[k]=tmp[k]
26        end
27    end
```

Salida: A

3. Combinar las dos mitades ordenadas: $O(n)$.

Desventajas:

- Utiliza memoria extra.
- Controlar la posición del elemento más pequeño en cada mitad y tiene que añadir el elemento más pequeño de los dos a una lista auxiliar.

2.8. Algoritmo de ordenamiento por método rápido(*Quicksort*)

Quicksort es un algoritmo de divide y vencerás. La idea es dividir una lista de elementos a ordenar en dos sublistas. El elemento que divide la lista se llama pivote. Una vez dividida la lista, se realizan las operaciones de comparación e intercambio para que todos los elementos con valor menor al pivote queden del lado izquierdo al pivote y los elementos con mayor valor que el pivote sean colocados en la sublista del lado derecho. Este proceso es recursivo hasta que la lista quede de tamaño 0 o 1(ver Figura 2.7). El procedimiento recursivo se puede ver en el Algoritmo 8, describe los pasos. En la línea 4, se hace la llamada al Algoritmo 7 para las operaciones de comparación e intercambio de cada sublista.

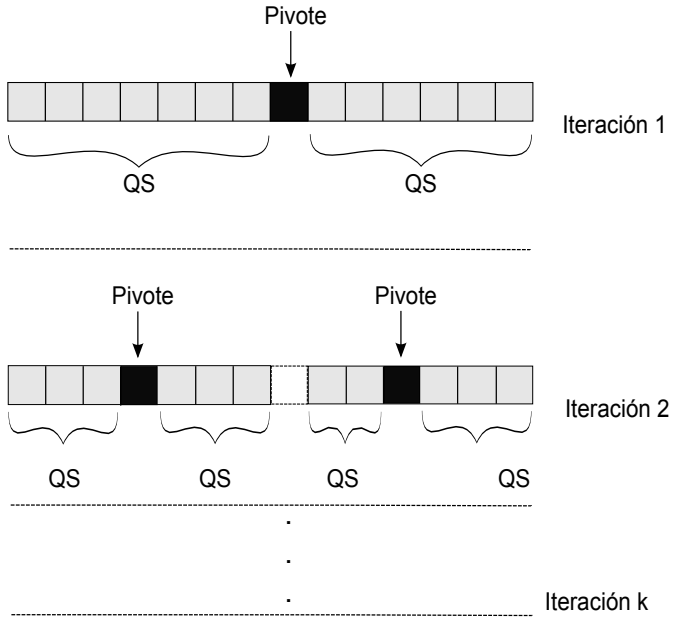


Figura 2.7: Esquema del algoritmo *quicksort*.

El algoritmo puede mostrar un comportamiento diferente con respecto a la elección de la ubicación del pivote. Tomar el pivote sin ningún cálculo adicional lo hace más rápido. Sin embargo, ésta elección ocasiona que tenga una complejidad de $O(n^2)$ para ciertas permutaciones. Otra opción es recorrer la lista e identificar el elemento que va a quedar en el medio, aunque implica un costo extra y terminará con un orden de $O(nlogn)$ en el peor de los casos, sin embargo para el caso promedio disminuirá su eficiencia. Otra opción para elegir el pivote es tomar tres elementos de la lista, por ejemplo el primero el de en medio y el último, compararlos y tomar como pivote el del valor central.

- Peor caso: $O(n^2)$

Algoritmo 7 Pivote

```
Entrada: A, izq, der
1    begin
2        pivote = izq
3        valor_pivote = A[pivote]
4        for (i=izq+1; i<=der; i++)
5            if (A[i] < valor_pivote)
6                pivote++
7                aux=A[i]
8                A[i]=A[pivote]
9                A[pivote]=aux
10       end
11       aux=A[izq]
12       A[izq]=A[pivote]
13       A[pivote]=aux
14   end
```

Salida: pivote

Algoritmo 8 quicksort

```
Entrada: A, n
1    begin
2        int pivote
3        if (izq < der)
4            pivote=Pivote(A, izq, der)
5            Quicksort(A, izq, pivote-1)
6            Quicksort(A, pivote+1, der)
7    end
```

Salida: A

- Mejor caso: $O(n\ log\ n)$
- Caso Promedio: $O(n\ log\ n)$
- Complejidad de espacio: $O(n)$

Ventajas:

- Muy rápido.

- No utiliza memoria adicional.

Desventajas:

- Implementación un poco complicada.
- En la recursividad emplea muchos recursos.
- Existe mucha diferencia entre el peor y el mejor caso.

2.9. Estudio comparativo

En esta sección se presenta un estudio comparativo de algunos algoritmos de ordenamiento. El objetivo es conocer su comportamiento en conjuntos de datos aleatorios con tamaños de entrada de datos pequeños y grandes. Los algoritmos implicados en el experimento fueron codificados en lenguaje C y son:

- El algoritmo de ordenamiento burbuja.
- El algoritmo de ordenamiento por selección.
- El algoritmo de ordenamiento por inserción.
- El algoritmo de ordenamiento por montículos.
- El algoritmo de ordenamiento por mezcla.
- El algoritmo de ordenamiento *quicksort*.

Las pruebas fueron ejecutadas en un procesador AMD Athlon a 2.4 GHz. con sistema operativo Ubuntu. Los conjuntos de datos son de tipo de datos entero con valores que van desde 0 a 1 millón. El conjunto de experimentos es para tamaños de datos de entrada de 100, 1000, 10000 y 100000.

La Tabla 2.1 muestra los resultados. En cada fila se identifica el tiempo obtenido en segundos por cada técnica de ordenamiento. En la segunda columna están los tiempos de las pruebas cuando fueron 100 datos de entrada los que ordenó la técnica. En tercera columna 1000 datos de entrada que se utilizaron para ordenar y así sucesivamente. En negritas se encuentran los resultados con menor tiempo que se encontraron. Por ejemplo, la técnica de ordenamiento por mezcla obtuvo el menor tiempo para ordenar 100 datos. Como peores tiempos tenemos al algoritmo de Burbuja para ordenar 10000 con 0.56 segundos mientras que la mejor técnica puede ordenar los mismos datos en 0.0062 segundos. En los resultados es notable como el algoritmo de ordenamiento por

Tabla 2.1: Tabla comparativa de tiempo (segundos) entre algunos algoritmos tradicionales de ordenamiento.

Algoritmo \ Cantidad Datos	100	1000	10000	100000
Burbuja	0.000202	0.020483	0.564711	54.995611
Selección	0.000113	0.009609	0.26164	25.571957
Inserción	0.000114	0.010066	0.289241	26.852729
Montículos	0.000054	0.000567	0.007349	0.033491
Mezcla	**0.00005**	0.000584	0.007575	0.03315
Quicksort	0.000054	**0.000471**	**0.006253**	**0.030825**
Red de ordenamiento	*0.000143	***0.000393**	-	-

mezcla como el algoritmo de ordenamiento por montículos tienen muy poca diferencia entre sus tiempos.

Además el experimento aplicó una prueba para una RO con tamaño de entrada de 128 y 1024. Las RO de tamaño extenso se obtienen de los experimentos realizados en el Capítulo 6.2. EL resultado es favorable cuando se utiliza una RO para 1024, el tiempo es menor que todos los algoritmos propuestos para las pruebas.

También se presentan las gráficas comparativas en la Figura 2.8 y en la Figura 2.9. En la primera gráfica están las técnicas que compitieron en el experimento con los tamaños de datos de entrada de 100 y 1000. La complejidad de los algoritmos con la estrategia de divide y vencerás muestran ventaja sobre los algoritmos de burbuja, selección e inserción. Los tiempos que resultaron de ejecutar las pruebas para todos los tamaños de entrada se presentan en la gráfica de la Figura 2.9. El crecimiento del tiempo con respecto al tamaño de los datos es más notable para los algoritmos de ordenamiento burbuja, selección e inserción. El algoritmo que se comporta mejor para grandes cantidades de datos es el *quicksort*.

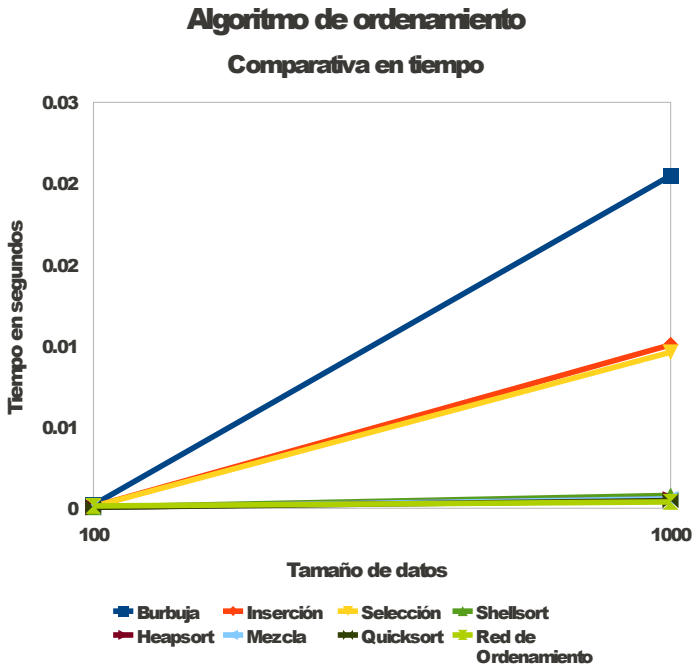


Figura 2.8: Comparativa en tiempo (segundos) de algoritmos tradicionales hasta con tamaño de datos 100 y 1000 aleatorios.

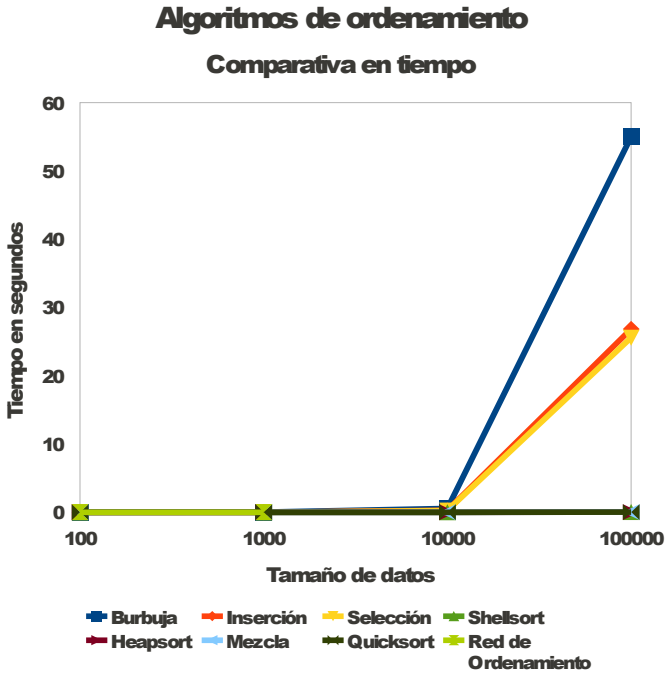


Figura 2.9: Comparativa en tiempo (segundos) de algoritmos tradicionales con tamaño de datos 100, 1000, 10000 y 100000 aleatorios.

Capítulo 3

Redes de Ordenamiento

Este apartado está dedicado a los conceptos básicos de las RO. También se describen sus principales características y se presentan algunos ejemplos ilustrativos.

En la segunda parte del capítulo son presentadas herramientas útiles para garantizar que una RO está bien diseñada. Otro aspecto importante es conocer el trabajo realizado desde sus inicios y el avance que se ha venido presentando hasta el momento, detalle que se encuentra en la parte final del capítulo.

3.1. Conceptos básicos

Diseñar Redes de Ordenamiento (RO) es un problema combinatorio que ha sido estudiado durante décadas [63, 23]. Las RO son un ejemplo de algoritmos llamados *oblivious*, esto es, que los pasos necesarios para ordenar no dependen de pasos anteriores. La operación fundamental en una RO es la llamada comparación-intercambio y se etiqueta como *comparador*. En la Figura 3.1 se ilustra un comparador c representado esquemáticamente, éste recibe dos datos de entrada (i, j) para ser comparados, si $i > j$ entonces se aplica la operación de intercambio tal que cumpla $i < j$. Cada línea horizontal representa el trayecto de un dato de entrada y la línea vertical es un comparador que evalúa los elementos entre su extremo superior e inferior. Si el valor del extremo superior es más grande que el extremo inferior entonces intercambia los datos.

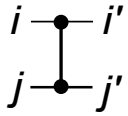


Figura 3.1: Ejemplo de un comparador c. Recibe una lista binaria de entrada (i, j). Si $i > j$ entonces intercambian los datos donde $i < j$

Un conjunto de comparadores conforman una RO. Generalmente las RO son repre-

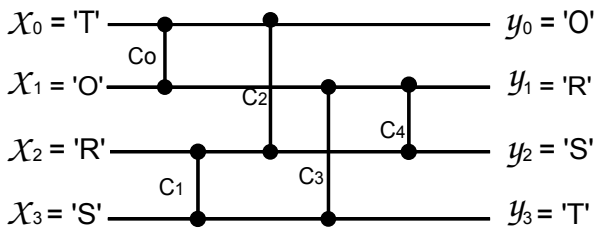


Figura 3.2: Ejemplo de una Red de Ordenamiento para 4 datos de entrada.

sentadas esquemáticamente como en la Figura 3.2. Por ejemplo, se quieren ordenar 4 datos de entrada, los valores de x_0, x_1, x_2, x_3 son colocados del lado izquierdo de la RO y un dato por cada línea horizontal, como en la Figura 3.2. Las líneas verticales etiquetadas como c_0, c_1, c_2, c_3, y c_4 son los comparadores. Cada comparador recibe dos valores, esto es, c_0 recibe los valores x_0 y x_1; c_1 recibe los valores de x_2 y x_3, y así sucesivamente. De esta forma, los comparadores son ejecutados de izquierda a derecha según su aparición. En este ejemplo primero se ejecuta c_0, posteriormente c_1, en seguida c_2, entonces c_3 y finalmente c_4. Esto es, c_0 evalúa 'T'>'R' entonces, los valores de x_0 y x_1 son intercambiados. El siguiente comparador es c_1 que evalúa 'O'>'S' mantiene a x_0 y x_1 sin intercambio. En seguida c_3 evalúa 'R'>'O' entonces, los valores x_0 y x_2 son intercambiados. Este proceso continúa hasta que todos los comparadores son aplicados. Finalmente, la lista ordenada y_0, y_1, y_2, y_3 se obtiene al lado derecho de la RO y debe cumplir con $y_0 \leq y_1 \leq y_2 \leq y_3$.

La configuración interna es característica primordial de las RO para no depender de los datos de entrada, es por ello que se encuentran en la clasificación de algoritmos no adaptativos.

Las principales diferencias entre las RO y los otros algoritmos clásicos de ordenamiento (tales como el *bubblesort*, *sellsort*, *quicksort*) son las siguientes:

1. La RO cuenta con una secuencia fija de comparadores que ordena cualquier permutación de datos de entrada;

2. Los comparadores pueden ejecutarse en paralelo, así que el número de pasos, o tiempo necesario para ordenar, es también fijo.

3.1.1. Red de Ordenamiento eficiente

El diseño de una RO se lleva a cabo seleccionando comparadores que ayuden en el ordenamiento de los datos de entrada. Los comparadores que pueden ser seleccionados se obtienen de la ecuación binomial:

$$C_i = \binom{n}{2} \tag{3.1}$$

Tabla 3.1: Número de comparadores de las RO rápidas conocidas desde $n = 1$ hasta $n = 16$.

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Número de comparadores	0	1	3	5	9	12	16	19	25	29	35	39	45	51	56	60
Límite mínimo	0	1	3	5	9	12	16	19	20	22	26	29	33	37	41	45

Es posible construir diferentes redes de ordenamiento para el mismo tamaño de entrada n, es decir con diferentes configuraciones. Por ejemplo las Figuras 3.3 y 3.4 son redes para 4 datos de entrada con 5 comparadores, sin embargo, la configuración de los comparadores es diferente.

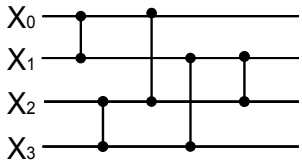


Figura 3.3: RO para $n = 4$ con 5 comparadores.

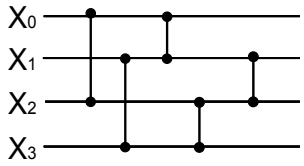


Figura 3.4: RO para $n = 4$ con 5 comparadores.

Así como es posible diseñar RO para una determinada n con diferentes configuraciones, también es posible diseñar RO para un particular n con diferente número de comparadores. Por ejemplo las Figuras 3.5 y 3.6 son redes también para 4 datos de entrada con diferente configuración y, donde la segunda RO tiene un comparador más que la primera. Entonces, si consideramos el número de operaciones como medida de eficiencia en una RO, es preferible escoger la RO de la Figura 3.5 que tiene sólo 5 comparadores. Por lo tanto, una RO eficiente es aquella con el menor número de comparadores dada una entrada de datos n.

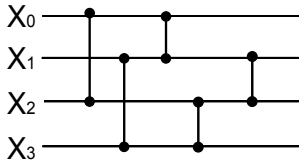


Figura 3.5: RO para $n = 4$ con 5 comparadores.

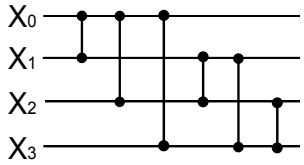


Figura 3.6: RO para $n = 4$ con 6 comparadores.

En la Tabla 3.1 se exhiben las diferentes RO con un número de comparadores mínimo para tamaños de entrada $1 \leq n \leq 16$ [36, 51, 63]. La primera fila es el tamaño de dato de entrada para la RO, la segunda fila contiene el número de comparadores mínimo que se conoce en la literatura especializada por cada RO, éstos fueron obtenidos de trabajos previos con diferentes técnicas (ver sección 3.3). Finalmente, en la tercera fila se tienen los límites mínimos de comparadores que han sido obtenidos de información teórica en [51, 63]. El límite mínimo de comparadores se obtiene del redondeo hacia arriba de $log_2(N!)$ [58].

Tabla 3.2: RO rápidas conocidas y límites mínimos de las RO rápidas obtenidas de información teórica.

n	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Número de capas	3	3	5	5	6	6	7	7	8	8	9	9	9	9
Límite mínimo	3	3	5	5	6	6	7	7	7	7	7	7	7	7

3.1.2. Red de Ordenamiento rápida

Una ventaja de las RO es que algunos de sus comparadores podrían ser ejecutados en paralelo. Usualmente, el tiempo discreto requerido para el ordenamiento se refiere al número de pasos o capas que la RO debería considerar después de verificar la independencia de los comparadores que pueden ser ejecutados en paralelo. El máximo número posible de comparadores por capa es de $n/2$. Una RO se considera *rápida* si tiene un bajo número de capas, donde cada capa está compuesta por un subconjunto de comparadores que puedan realizar sus operaciones concurrentemente sin interferir entre ellas.

Por ejemplo, si consideramos el total de capas por el número de operaciones de la RO para entradas de tamaño $n = 4$ de la Figura 3.2, el resultado es 5. En el diagrama presentado en la Figura 3.7 que corresponde a la RO de la Figura 3.2, es posible observar la ejecución de los comparadores donde c_0 y c_1 pueden ser aplicados al mismo tiempo (son independientes), lo que permite realizar las operaciones de forma concurrente. Entre los siguientes comparadores c_2 y c_3 de la misma forma son independientes y pueden ser ejecutados en la misma unidad de tiempo. El máximo número de comparadores que pueden estar en una capa es $n/2$. Finalmente, en este ejemplo se tiene una RO con 5 comparadores con 3 capas.

Las RO rápidas conocidas para n entradas entre $1 \leq n \leq 16$ con su respectivo número de capas, así como sus límites mínimos de capas se pueden ver en la Tabla 3.2. La segunda fila presenta el número de capas de las RO rápidas conocidas. En la tercera fila se tienen los límites mínimos de capas que han sido obtenidos de información teórica en [49, 51, 62]. Si $Z(n)$ es el mínimo número de comparadores, entonces los límites mínimos de capas de n se obtienen de la función techo de $Z(n)/(n/2)$. Cuando n es impar, cada capa tiene $(N-1)/2$ comparadores, así que el límite mínimo se obtiene con $Z(n)/((n-1)/2)$ [58].

3.2. Validez de una Red de Ordenamiento

De la misma forma que un algoritmo de ordenamiento, una RO debe cumplir su propósito para cualquier permutación de datos de entrada. La validación obvia consiste en verificar si todas las permutaciones de los datos de entrada son correctamente ordenados

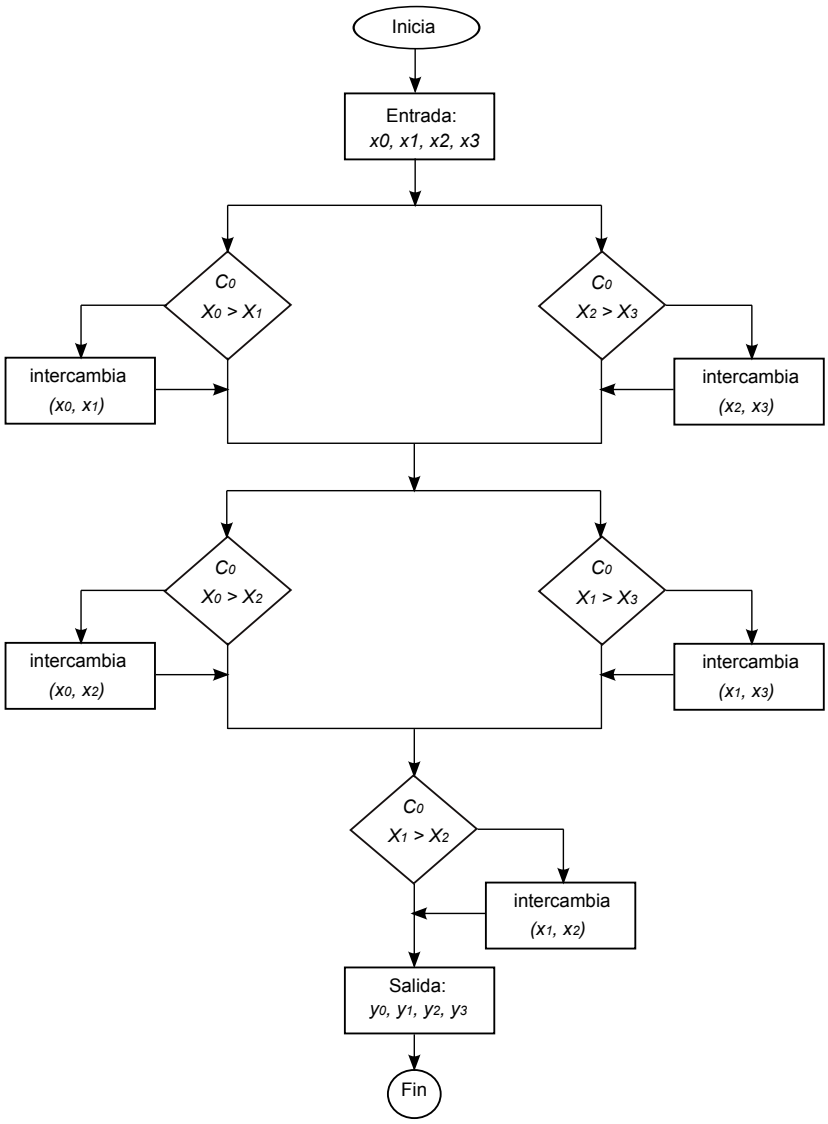


Figura 3.7: Diagrama de flujo de la Red de Ordenamiento para 4 datos de entrada.

por la RO.

Para un tamaño de dato de entrada n es necesario probar $n!$ veces la RO, esto implica que la tarea de la verificación incrementa su complejidad computacional cuando n tiende a incrementar, inclusive es intratable [50, 12, 36].

De tal manera que, para un tamaño de entrada relativo pequeño como $n = 12$ se requiere evaluar $479,001,600$ permutaciones, para $n = 16$ se requieren $2,0922,789,888,000$ evaluaciones.

Con la finalidad de reducir la complejidad de la verificación, los investigadores utilizan el Principio cero-uno (*Theorem Z*). Este principio que fue empleado por primera vez para validar una RO por Batcher en 1968 [5, 36].

Principio cero-uno[36]: "Si una red de comparadores ordena cualquier secuencia de 0's y 1's, entonces ordenará cualquier secuencia arbitraria de n números".

Demostración: Supongamos que existe una RO con n datos de entrada $\langle x_1, ..., x_n \rangle$ y datos de salida $\langle y_1, ..., y_n \rangle$. Supongamos que existe una función monotónica f, esto es, para cualquier x, y, se tiene que $x \leq y$ cuando $f(x) < f(y)$. Entonces si los datos de entrada de N son $\langle f(x_1), ..., f(x_n) \rangle$, la salida debe ser $\langle f(y_1), ..., f(y_n) \rangle$. Si $y_i > y_{i+1}$ para cierta i, consideremos la función monotónica f que toma todos los números $< y_i$ como cero y todos los números $\geq y_i$ como 1; esto define una secuencia $\langle f(x_1), ..., f(x_n) \rangle$ de ceros y unos que no son ordenados por n. Por lo tanto, si se ordenan todas las secuencias binarias, tenemos $y_i < y_{i+1}$ para $1 \leq i < n$.
■

Por lo tanto, el número de evaluaciones puede ser reducido significativamente de $n!$ a 2^n, utilizando el Principio Cero-Uno [36]. Chung and Ravikumar en su trabajo [51, 12] afirman que es suficiente evaluar $2^n - n - 1$ entradas de secuencias de ceros y unos desordenados. Por lo que para $n > 2$, $n! > 2^n$, la aplicación de este principio permite contar con una forma razonable de verificación en las RO.

Por ejemplo, la validación de la RO para $n = 4$ se puede ver en la Figura 3.8. Todos las entradas binarias resultan ordenadas a la salida, por lo tanto la RO es válida.

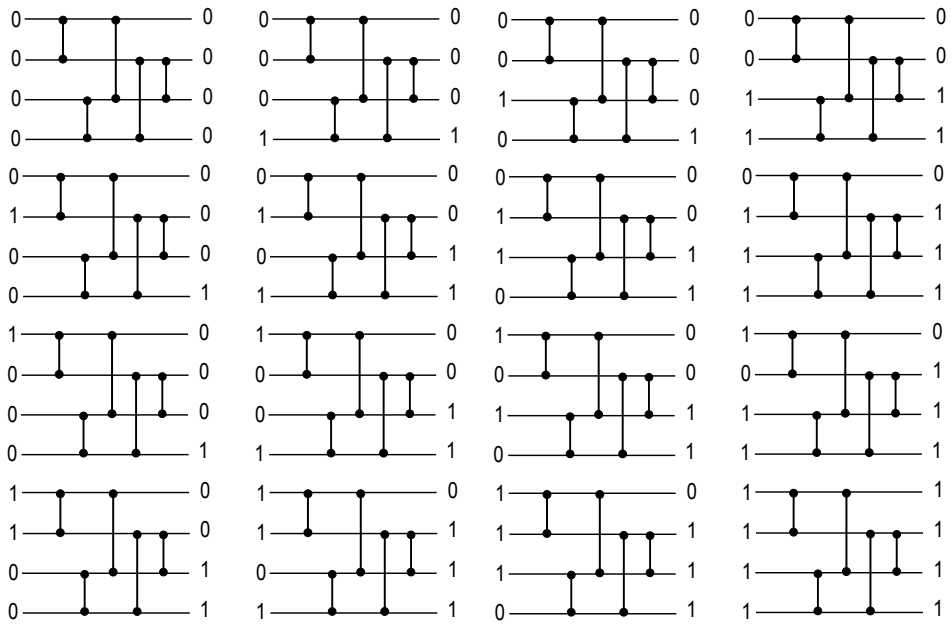


Figura 3.8: Operación de validación para la RO de la Figura 3.2. $2^n = 16$ permutaciones de la entrada de datos.

3.3. Trabajo previo en Redes de Ordenamiento

Los primeros diseños fueron presentados por P. N. Armstrong, R. J. Nelson y D. G. O'Connor en 1945 [48]. Ellos construyeron las RO para $n = 4$, $n = 5$, $n = 6$, $n = 7$ y $n = 8$ con 5, 9, 12, 18 y 19 comparadores respectivamente. Posteriormente, D. Knuth y R. W. Floyd en $1964 - 1966$ [36] afirmaron que éstas eran las más eficientes (con el menor número de comparadores). También en 1964, R. W. Floyd diseña la RO para $n = 9$ (ver Figura 3.9). Unos años más tarde, A. Waksman en 1968 [66, 65] diseñó la RO para $n = 10$ mediante permutaciones de $\{1, 2, ..., 10\}$ (ver Figura 3.10). Con respecto a la RO para $n = 11$, fue encontrada a partir de la RO eficiente para $n = 12$, eliminando los comparadores que tienen en su lista el último índice [36]. Un año más tarde los algoritmos presentados por K. E. Batcher en [5] fueron empleados por G. Shapiro y M. W. Green para trabajar con las RO para $n = 12$ donde se encontraron 39 comparadores.

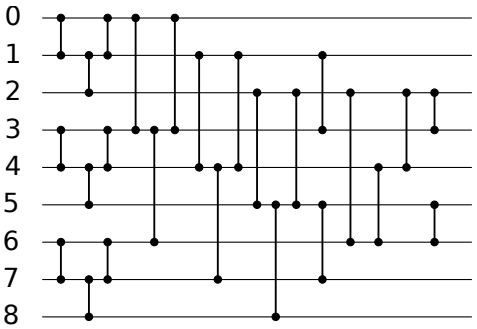


Figura 3.9: RO para $n = 9$ con 25 comparadores y 10 capas, descubierta por R. W. Floyd y D. Knuth.

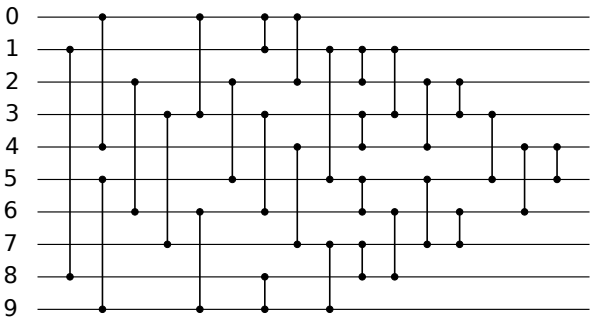


Figura 3.10: RO para $n = 10$ con 29 comparadores y 9 capas, descubierta por A. Waksman.

En los años 60's los investigadores trabajaron intensamente sobre la RO para $n = 16$. Los resultados más relevantes fueron: R. C. Bose y Nelson en 1962 [8] encontraron una RO con 65 comparadores. En 1964 K. E. Batcher y D. Knuth [5, 58] presentaron una RO con 63 comparadores, dos años más tarde G. Shapiro obtuvo una RO con 62 comparadores. En el mismo año (1969), M. W. Green logró el diseño de una RO con sólo 60 comparadores, que sigue siendo hasta el momento la RO más eficiente para $n = 16$.

En cuanto a las RO para $n = 14$ y $n = 15$, se encontraron a partir de la mejor RO conocida para $n+1$, eliminando el último índice y todos los comparadores que lo lleven.

Desde los años 90's, surgen nuevas propuestas. Hillis en 1990 [27] utilizó por primera vez evolución simulada en combinación con una coevolución de especie de parásitos. La población llamada "host" coevoluciona con otra población llamada parásitos que son los casos de prueba. La aptitud de los casos de prueba era medida a partir del fallo de los "host". El objetivo de la coevolución fué evitar quedar atrapado en un óptimo local. Los "host" eran una población de RO y la otra población eran las secuencias de prueba. Las dos poblaciones compiten por la supervivencia. Fue el primero en utilizar una

inicialización previa de *comparadores primarios* en una RO para $n = 16$, encontrando 61 comparadores. El concepto de *comparadores primarios* se explica en la siguiente sección.

En 1995 Hugues Jullié [29] con su método conocido como END *(*Evolving Non - Determinism) construye soluciones como rutas en una estructura de árbol, donde las hojas representan las soluciones válidas. El proceso de búsqueda es similar a la búsqueda *beam* [61] asignando aptitudes a los nodos internos para entonces seleccionar los más prometedores. La aptitud de un nodo interno es estimada mediante la construcción de una ruta incremental y aleatoria a una de las hojas. Esta técnica encuentra las redes de tamaño mínimas conocidas para $9 \leq n \leq 16$ y además descubre una red para $n = 13$ con 45 comparadores, que hasta el momento es la mejor conocida (ver Figura 3.11 y Figura 3.12).

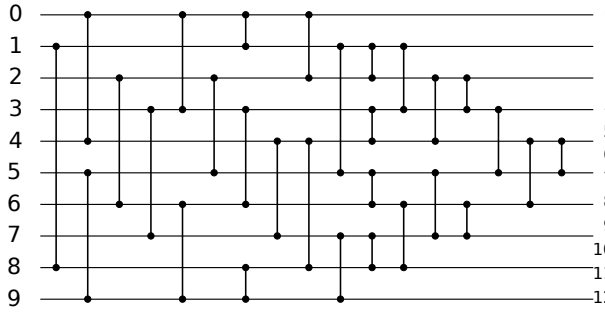


Figura 3.11: RO de Jullié para $n = 10$ con 29 comparadores y 9 capas.

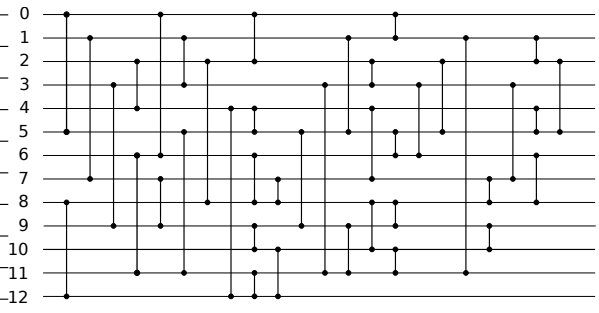


Figura 3.12: RO de Jullié para $n = 13$ con 45 comparadores y 10 capas.

Recientes estrategias orientadas a reducir la complejidad del algoritmo y el tiempo de procesamiento se han publicado desde entonces, sin embargo no han mejorado el número de comparadores en los resultados encontrados. Las trabajos más relevantes se presentan a continuación. En 1997 J. Koza [38] diseñó la RO para $n = 7$ con Programación Genética reduciendo el número de comparadores en 2. M. Harrison et al. en el 2004 [26] al igual que Koza usaron programación genética para evolucionar la construcción del diseño de redes. En 2005, Choi y Moon [11] publicaron un Algoritmo Genético para diseñar RO para $n = 10$, $n = 12$, y $n = 16$. Los autores argumentaron que su trabajo disminuye el tiempo de procesamiento de una manera impactante para encontrar las RO mejores conocidas en segundos con un simple procesador. Sin embargo, su técnica disminuye drásticamente el espacio de búsqueda debido a que también utiliza *comparadores primarios* como Hillis. Por ejemplo, para $n = 12$, su técnica añade los primeros 18 comparadores, mientras que para $n = 16$, toma 32 comparadores con los que inicializa su algoritmo. En el 2005 L. Sekanina [56, 55] propuso una técnica para construir RO largas en la cual un objeto construido por un grupo de comparadores se describe como un embrión que puede crecer. En 2011 Valsalam [60] trabajó con la generación de redes grandes a partir de simetrías y mediante la selección de comparadores de acuerdo a una distribución Gaussiana. Él obtuvo RO para $9 \leq n \leq 23$. Inicialmente genera la RO con un grupo de comparadores simétricos y al dividir el problema en

subproblemas se pierde dicha simetría, así que utiliza una técnica *greedy* para utilizar la información estadística de comparadores en redes mínimas. Sus resultados obtienen el mejor conocido de las redes de $9 \leq n \leq 16$. Para RO grandes se comparó con las de Batcher $17 \leq n \leq 23$ mejorando los resultados por uno o dos comparadores, aunque igualando el tamaño de las RO para $n = 21$ y $n = 23$ de Batcher.

3.3.1. Comparadores primarios

Los comparadores que conforman una RO válida afectan en alguna medida al conjunto de secuencias de prueba. El conjunto de éstos comparadores tienen un orden predefinido y puede ser usado un subconjunto de ellos para buscar nuevas RO. A este subconjunto de comparadores útiles los definimos como "comparadores primarios". Usar comparadores primarios tiene como ventaja reducir el espacio de búsqueda del problema. Por ejemplo, la RO con $n = 16$ debe evaluar 2^n secuencias binarias, Hillis utilizó los 32 comparadores primarios de la RO de Green, que están ilustrados en la Figura 3.13. Así que la técnica de Hillis diseña los comparadores para ordenar 151 secuencias de prueba. Los comparadores encontrados por la técnica co-evolutiva están en la Figura 3.14.

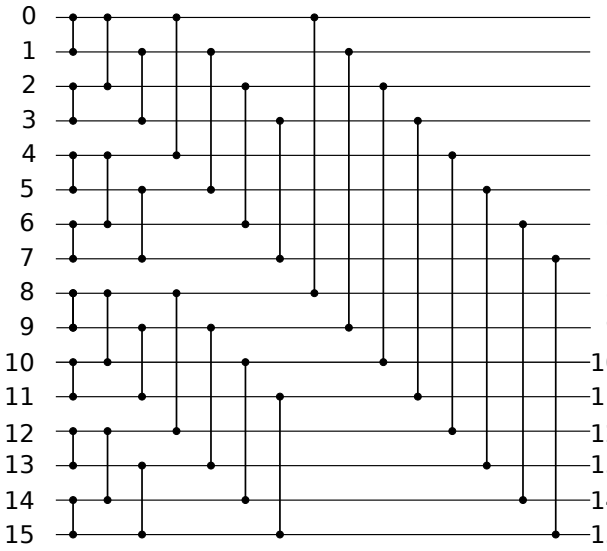


Figura 3.13: Primeros 32 comparadores de la RO de Green. RO completa (ver Figura 3.15).

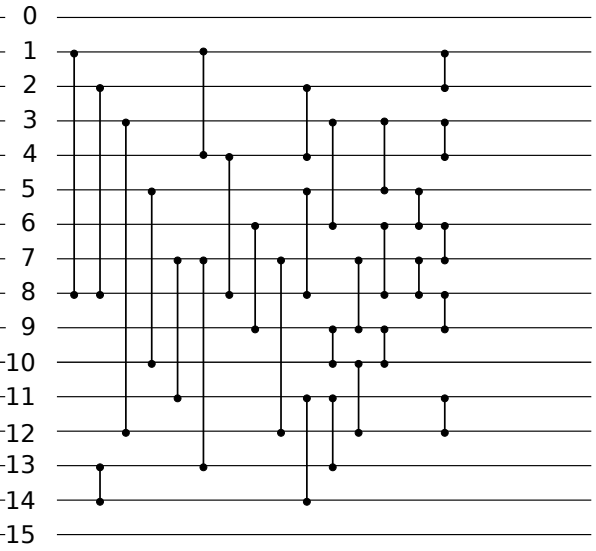


Figura 3.14: Comparadores de la RO de Hillis. Para $n = 16$ con 29 comparadores que busca la técnica de co-evolución.

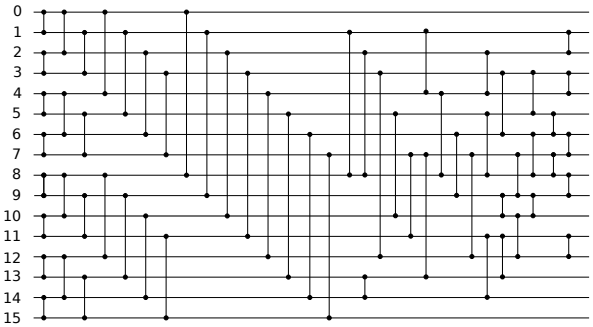


Figura 3.15: RO de W. Daniel Hillis para $n = 16$ con 61 comparadores y 11 unidades de tiempo.

3.4. Métodos guía para construir una Red de Ordenamiento

Existen métodos para construir RO a partir de otras conocidas y válidas. En la primera subsección de este apartado se tienen dos métodos para incrementar el tamaño de entrada en $n + 1$ mientras que en el segunda parte se presentan dos mecanismos que son útiles para incrementar el tamaño de n a $2n$.

3.4.1. Construcción de una Red de Ordenamiento en n + 1

La construcción de una RO conocida válida para un tamaño de entrada n puede ser base para incrementar en uno el tamaño de entrada $(n + 1)$. Considerando que ya se cuenta con cierto número de comparadores para n, sólo habrá que agregar los comparadores para el índice agregado. Se conocen al menos dos formas de incrementar en $n + 1$ el tamaño de una RO: mediante el principio de inserción o el principio de selección. Si $|\phi|$ es el número de comparadores que tiene la RO válida, el total de comparadores resultantes de aplicar éstos principios es: $|\phi|+n$.

3.4.1.1. Construcción de una Red de Ordenamiento mediante el principio de *inserción*

El método consiste en insertar el elemento por ordenar en la posición $n + 1$. Los comparadores anteriores ya han sido colocados para ordenar n datos de entrada (con otra RO válida), así que para garantizar que la RO ordenará el $n + 1$ dato, se agrega un comparador entre el elemento n y el elemento $n+1$, posterior a los comparadores que ya ordenan. Y para garantizar que el elemento será ubicado en el lugar que le corresponde en el proceso de ordenamiento, se agregan los comparadores que se requieren, es decir n comparadores. Por ejemplo, se quiere construir una RO para $n = 5$ considerando la RO para $n = 4$ presentada en la Figura 3.16 inciso a. A la RO se le agrega el último

índice (X_4) junto con los comparadores necesarios para que el último dato pueda ser ordenado.

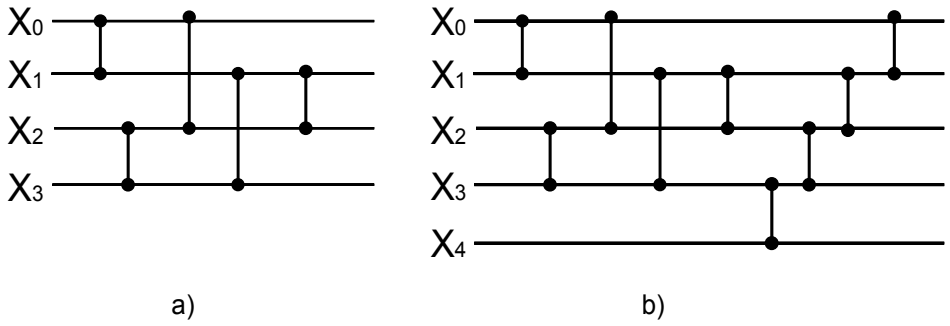


Figura 3.16: Ejemplo de una construcción mediante el principio de inserción. A partir de una RO para $n = 4$ se agrega un elemento para la construcción de la RO para $n = 5$

3.4.1.2. Construcción de una Red de Ordenamiento mediante el principio de *selección*

Otra forma de incrementar en $n + 1$ el tamaño de una RO, es mediante el principio de selección. Toda vez que ya se tiene una RO con los comparadores que ordenan n elementos, se agrega un elemento por ordenar, éste es colocado antes de la RO que ya se tiene. La RO de la Figura 3.17 inciso b es diseñada a partir de la RO para $n = 4$ (inciso a), el elemento $n + 1$ es colocado en la parte inferior, se incorpora el comparador antes de que inicie el conjunto de comparadores de la RO que ya ordena, entre el elemento n y el elemento $n + 1$.

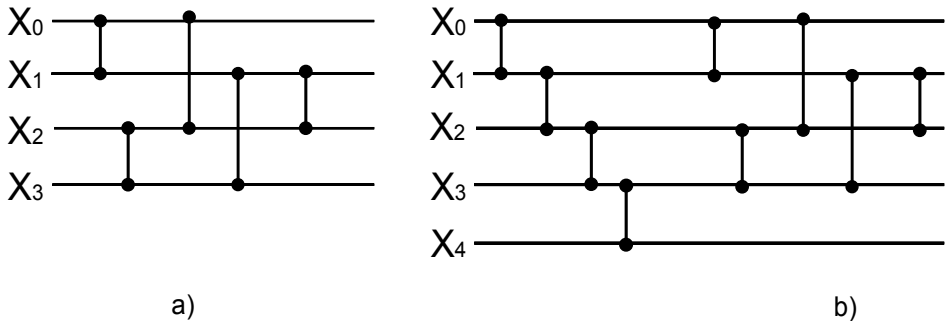


Figura 3.17: Ejemplo de una construcción mediante el principio de inserción. A partir de una RO para $n = 4$ se agrega un elemento para la construcción de la RO para $n = 5$

3.4.2. Red de Ordenamiento mediante fusión

Los métodos de fusión conocidos son dos:

1. Impar-par (*Odd-Even Merge Sorting*) y,

2. Fusión Bitónica ("Bitonic Sorting")

Presentados por K.E. Batcher en 1968 [5, 58] son implementados con frecuencia en hardware o en un arreglo de procesadores paralelo [39, 2, 57]. A continuación se explica brevemente cada uno de ellos.

3.4.2.1. Red de Ordenamiento bitónica

Una secuencia bitónica tiene dos subsecuencias, una monotónicamente creciente y otra decreciente. Por ejemplo, $p_0 < p_1 < p_2, ..., p_{i-1} < p_i > p_{i+1} > p_{i+2}, ..., p_{n-2} > p_{n-1}$

La mezcla bitónica consiste de una operación de comparación e intercambio con los elementos p_i y $p_{i+n/2}$ y se obtienen dos subsecuencias bitónicas; una con un número de secuencias menores que los de la otra (ver Figura 3.18). Aplicando recursivamente la operación de comparar e intercambiar a las subsecuencias, se llega a las listas bitónicas de tamaño uno, y así la lista inicial de n elementos estará ordenada.

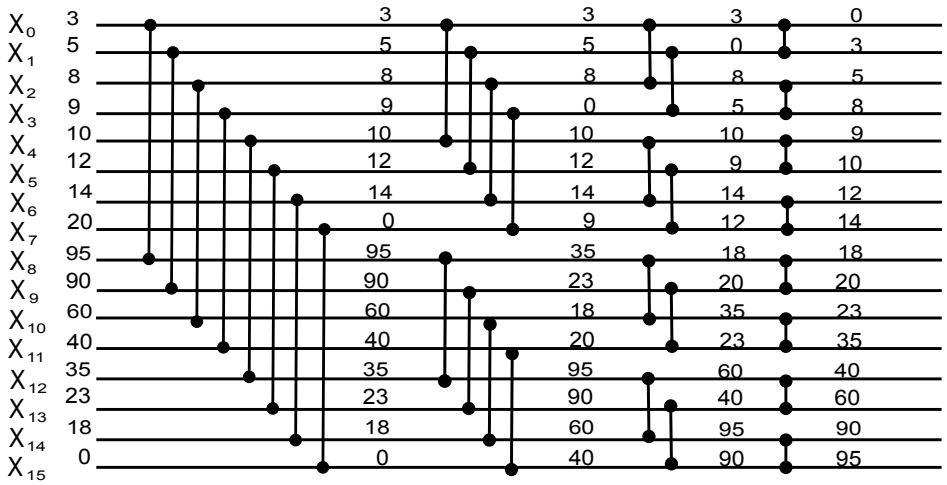


Figura 3.18: Ejemplo de una RO construida mediante una fusión bitónica.

Una desventaja es que la secuencia de datos iniciales debe ser bitónica, de no ser así, se procede a preparar los datos de la siguiente manera:

1. Convertir cada par de elementos adyacentes en una secuencia incremental y la siguiente en una decremental

2. Dividir secuencias bitónicas de $n/2$ sublistas de elementos en secuencias bitónicas. Ordenar cada secuencia de $n/2$ elementos (incrementando o decrementando) y mezclar de forma bitónica en una secuencia de n elementos.

3. Ordenar la secuencia bitónica de n elementos.

3.4.2.2. Red de Ordenamiento de fusión impar-par

Las redes derivadas de este método tienen un tamaño de comparadores $O(n\,(log\,n)^2)$ y un tiempo de ejecución $O((log\,n)^2)$. La idea principal del algoritmo es exactamente el mismo que el del algoritmo de ordenamiento de fusión estándar (*mergesort*). El algoritmo es recursivo, existen dos listas A y B a ordenar. Se identifican las listas de índices pares e impares, donde:
$A = [a_0, ..., a_{n-1}]$ y $B = [b_0, ..., b_{n-1}]$ y;
$P(A) = [a_0, a_2, ..., a_{n-2}]$, $O(A) = [a_1, a_3, ..., a_{n-1}]$
$P(B) = [a_0, a_2, ..., a_{n-2}]$, $O(B) = [a_1, a_3, ..., a_{n-1}]$

Algoritmo 9 Fusión impar-par

```
1    begin
2        Fusión P(A), O(B) = D = [d_o, ...d_{n_1}]
3        Fusión I(A), P(B) = F = [f_o, ...f_{n_1}]
4        Fusión (A, B) = [d_0, f_0, d_1, f_1, ..., d_{n-1}, f_{n-1}]
5    end
```

La última lista de elementos deberá ser ordenada por pares $\{d_0, f_0\}$, $\{d_1, f1\}$, así sucesivamente.

Por ejemplo, para construir una RO con $n = 8$ es posible utilizar una RO para $n = 4$, como se muestra en la Figura 3.19

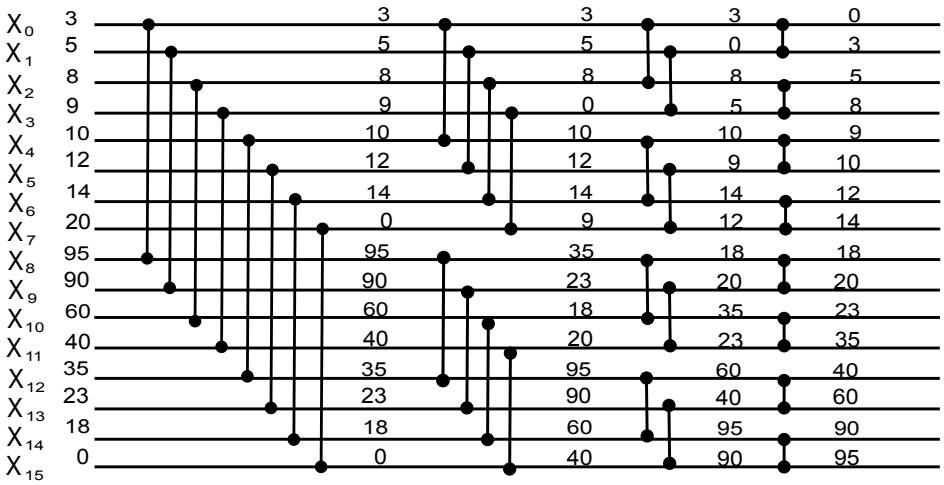


Figura 3.19: Ejemplo de un diseño mediante el uso de fusión impar-par.

¿Porqué seguir buscando RO cuando ya se tienen las RO de K. E. Batcher y se pueden utilizar para la construcción de grandes RO?.

A pesar de que se cuenta con RO en las que se conoce y es posible estimar el tiempo de ejecución, K. E. Batcher describe que aún es desconocido el costo óptimo de una RO en paralelo si se construye el circuito de ordenamiento en hardware. La RO bitónica y la fusión impar-par son utilizadas en las Unidades de Procesamiento Gráficas (GPU's) o en las arquitecturas de hardware. A. Baddar y K. E. Batcher, por mencionar algunos, proponen continuar con las investigaciones para encontrar RO más eficientes y rápidas [58, 5, 52]. Rene Mueller et al. mencionan que las RO son apropiadas para la implementación en hardware [46]. La implementación de éste tipo de algoritmos de ordenamiento en hardware es más fácil y su crecimiento asintótico de las RO es de $O(log\ n)$ [58].

Capítulo 4

Algoritmo del Sistema Inmune Artificial para encontrar Redes de Ordenamiento eficientes

Este capítulo expone la implementación de un modelo computacional inspirado en el sistema inmune biológico para ayudar en el diseño de RO eficientes. El algoritmo del sistema inmune artificial es un paradigma que ha sido utilizado para resolver problemas complejos con éxito. Tanto los componentes como los mecanismos de la heurística son atractivos para usarlos en la búsqueda de RO. En consecuencia, este apartado proporciona una breve descripción del funcionamiento del sistema inmune artificial y sus principales componentes, con la finalidad de ahondar más en el mecanismo de principio de selección clonal, que es el modelo computacional implementado por su sencillez y potencialidad para ayudar a resolver el problema en la búsqueda de RO eficientes.

Aún cuando los algoritmos con inspiración biológica han mostrado un desempeño satisfactorio en la solución de problemas de optimización [4] y específicamente el sistema inmune artificial en problemas de combinatoria [4, 15], el problema con el diseño de las RO es que la cantidad de comparadores puede estar tan alejada del mínimo conocido que implica el uso de más recursos por parte de la heurística, así que para ayudarle y reducir de cierta manera el espacio de búsqueda, se proponen diferentes funciones de aptitud en un mecanismo de construcción que descarta comparadores inútiles.

4.1. Sistema Inmune Artificial (SIA)

El Sistema Inmune Artificial (SIA) es un paradigma computacional que toma ideas del sistema inmune natural (especialmente el que corresponde a los mamíferos) para

resolver problemas complejos. En años recientes, el SIA ha sido aplicado con éxito para resolver problemas en diferentes áreas como en las Ciencias de la Computación y seguridad en redes [35, 25], detección de fallas, programación de horarios, máquinas de aprendizaje y optimización. Los problemas de optimización reportados resueltos por el SIA han sido de tipo multimodal [19], numérico, y optimización combinatoria, como entre otros muchos [16, 14].

Desde un punto de vista biológico, el sistema inmune del humano es muy complejo, conformado por un gran número de células, moléculas y diversos mecanismos. Algunos inmunologistas argumentan que una de las principales funciones de este sistema es proteger nuestro cuerpo de la invasión de microorganismos externos. Éste está compuesto de dos líneas principales de defensa: la inmunidad innata y la inmunidad adaptativa. La primera es no específica lo que significa que es independiente del antígeno externo. La inmunidad adaptativa tiene memoria y capacidad de aprendizaje y es dependiente del antígeno, lo que significa que cada tipo diferente de antígeno provocará una respuesta inmmune diferente. Los principales componentes de la inmunidad adaptativa son las células llamadas linfocitos B o simplemente células B. Cuando los linfocitos B son simulados mediante un antígeno específico producirán un gran número de moléculas llamadas anticuerpos, las cuales juegan un papel principal en la respuesta inmune adaptativa.

Desde el punto de vista del procesamiento de información, el sistema inmune es visto como un sistema paralelo y adaptativo distribuido. Es capaz de aprender o, usar la memoria y realizar la recuperación de información asociativa. Particularmente, éste aprende cómo reconocer patrones. Su comportamiento global es una propiedad emergente de muchas interacciones locales.

Diversos autores han propuesto definiciones de lo que se entiende por un sistema inmune artificial. Algunas de ellas se presentan a continuación:

"Los sistemas inmunes artificiales son sistemas adaptativos, inspirados en la teoría, modelos y principios inmunológicos y las funciones inmunes observadas, que se aplican a la solucines de problemas" [20].

"Los sistemas inmunes artificiales son metodologías inteligentes inspiradas en el sistema inmune, enfocadas a resolver problemas del mundo real" [17].

Como ya se ha mencionado, el sistema inmune es muy complejo, probablemente comparable a la complejidad del cerebro. Haciendo una sobre simplificación, se usan dos elementos del sistema inmune en algunos modelos computacionales: los llamado antígenos (microorganismo externo) y los anticuerpos (actores principales de la respuesta inmune adaptativa).

El algoritmo presentado en este documento está basado en un mecanismo llamado

principio de selección clonal, que explica la forma en la cual los anticuerpos eliminan un antígeno externo [15].

4.1.1. Principio de selección clonal

La Teoría de la Selección Clonal fué propuesta por Burnet en 1959 [9] y establece la idea de que sólo aquellos anticuerpos que poseen en su membrana receptores capaces de reconocer y unirse específicamente a él proliferan. El proceso de simulación y clonación de anticuerpos más aptos, su hipermutación y auto-regulación son conocidos como el principio de selección clonal. Esto es una simplificación de lo que sucede en una respuesta inmune natural. Esta idea se ilustra en la Figura 4.1. En el algoritmo tenemos un grupo de anticuerpos que son estimulados por un antígeno o el valor de una función objetivo para ser clonado, entonces la población de clones sufre mutaciones proporcionales a su afinidad. Los clones con más alta afinidad son seleccionados para conformar el conjunto de memorias y los clones con más baja afinidad son remplazados [19]. Un valor numérico (afinidad) es asignado a cada antígeno, éste representa que tan bien puede resolver el problema.

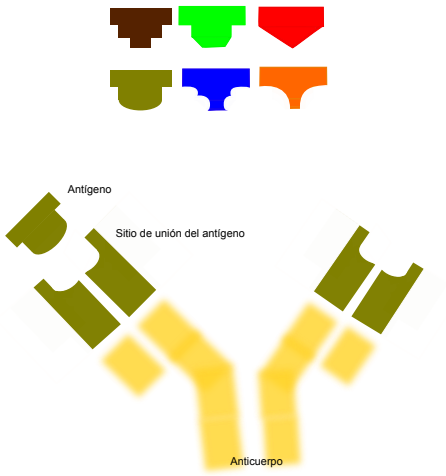


Figura 4.1: Esquema representativo del principio de selección clonal.

4.2. Definición del problema

Una RO ordena un entrada de datos $x = (x_0, x_1, ..., x_{n-1})$ de tamaño n. Una RO tiene un conjunto de comparadores (i, j) representado por ϕ donde $0 \leq i \leq j \leq n-1$. Si los datos de salida están en orden para todas las entradas de x, entonces ϕ es válida.

Para la validación, al igual que otros investigadores utilizamos el principio cero-uno para la validación. La validación consiste en evaluar un conjunto de secuencias binarias B_n de

tamaño 2^n. Por ejemplo, si se quiere validar una RO para $n = 5$, se evalúan 32 secuencias binarias. B_n consiste del conjunto formado por {00000,00001,00010,00011,00100, ..., 11111}.

Las secuencias binarias son evaluadas como se muestra en la Figura 4.2, cada secuencia es colocada del lado izquierdo del esquema, cada elemento de la secuencia binaria se coloca en una línea horizontal o bus. Una secuencia ordenada S_n es aquella donde en primer lugar están los 0's y al final los 1's.

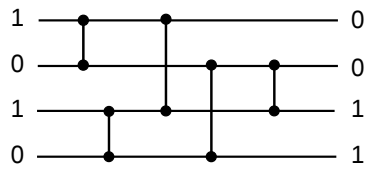


Figura 4.2: Ejemplo de validación de una RO para $n = 5$ utilizando el principio cero-uno.

La función de validación sería:

$$f_\phi(|B_n|) = S_n \tag{4.1}$$

Entonces, se quiere minimizar el conjunto de comparadores ϕ tal que cumpla con la validación según la Ecuación 4.1. La función para encontrar una RO eficiente válida es la siguiente:

$$|\phi| = min\{|\phi||f_\phi(|B_n|) = S_n\} \tag{4.2}$$

4.2.1. Comparadores candidatos

El conjunto de posibles comparadores C se obtiene de la Ecuación 3.1. Al subconjunto C' que contiene elementos de C con la característica de ser más aptos al evaluar un conjunto de secuencias de prueba se les nombra comparadores candidatos.

A continuación se describe la implementación del algoritmo con respecto al problema de diseñar RO eficientes.

4.3. Propuesta: Algoritmo de sistema inmune artificial para encontrar Redes de Ordenamiento eficientes.

En esta sección, se explica la propuesta para encontrar RO eficientes y para ello se divide en dos partes. En la primera, se explican los elementos del algoritmo del sistema inmune artificial, que aunque , no cuenta con un esquema propio, se toma el esquema de los algoritmos bioinspirados. Mientras, que en la segunda parte se explica el algoritmo propuesto y sus algoritmos secundarios.

4.3.1. Elementos del algoritmo de sistema inmune artificial

Al igual que cualquier otro sistema computacional con inspiración biológica, se definen los elementos del SIA [20, 15]:

1. Representación de los componentes del sistema.
2. El mecanismo de evaluación de los individuos con su ambiente. El ambiente es simulado por un conjunto de estímulos en una o más funciones de aptitud.
3. El proceso de adaptación que gobierna la dinámica del sistema.

En la Figura 4.3 se encuentran en síntesis los componentes del sistema con relación a nuestro problema.

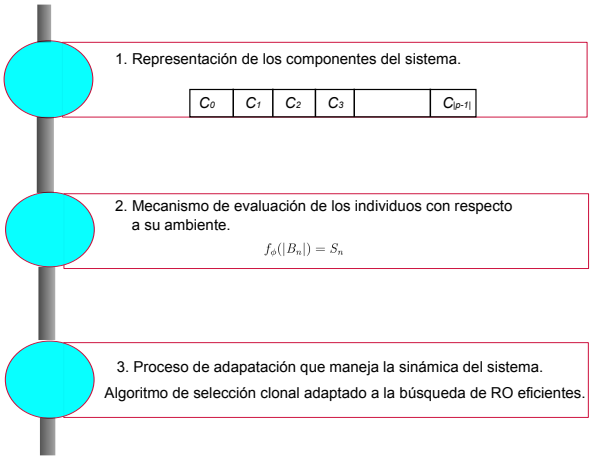


Figura 4.3: Componentes del SIA aplicado al problema de RO.

4.3.1.1. Representación de los componentes del sistema

Un anticuerpo está compuesto de una lista de comparadores. Cada comparador consiste de una pareja a, b. Por ejemplo, la representación para una RO $n = 4$ se puede ver en la Tabla 4.1.

[0-1]	[0-2]	[0-3]	[1-2]	[1-3]	[2-3]

Tabla 4.1: Representación de un anticuerpo para una RO con $n = 4$.

4.3.1.2. El mecanismo de evaluación de la iteración de los individuos con su ambiente y/o con otros individuos

Se emplea una función que mide la afinidad de una solución (ver en la Ecuación 4.2).

4.3.1.3. El proceso de adaptación que gobierna la dinámica del sistema

El proceso de adaptación lo dirige el principio de selección clonal que se puede ver en el diagrama de la Figura 4.4. Primero se genera una población de anticuerpos, a ellos se les mide su afinidad. Posteriormente son seleccionados los anticuerpos para la clonación. A cada clon se le aplica una mutación en su estructura para posteriormente ser evaluados. Una vez que se tiene la población de clones, son seleccionados un grupo de ellos para que conformen el grupo de anticuerpos. Los pasos del proceso se repiten hasta que se cumpla cierta condición de salida.

4.3.2. Algoritmo del sistema inmune artificial para generar Redes de Ordenamiento eficientes

La idea es usar un algoritmo de sistema inmune mediante el algoritmo de selección clonal como una estrategia global para minimizar el tamaño de una RO. Por otro lado, se diseña una estrategia local para seleccionar los comparadores que conformarán la RO (definido como el valor de afinidad).

El algoritmo maneja un conjunto inicial R de RO válidas. Cada elemento r, con $r \in R$ está representado como una lista de comparadores. El conjunto R es tomado por el algoritmo de selección clonal para minimizar el tamaño de cada r. Lo que equivale a minimizar el número de comparadores que conforman la RO.

El algoritmo de selección clonal para un tamaño de datos de entrada n se puede ver en Algoritmo 10. Se genera un conjunto de soluciones válidas R (línea 2). El número

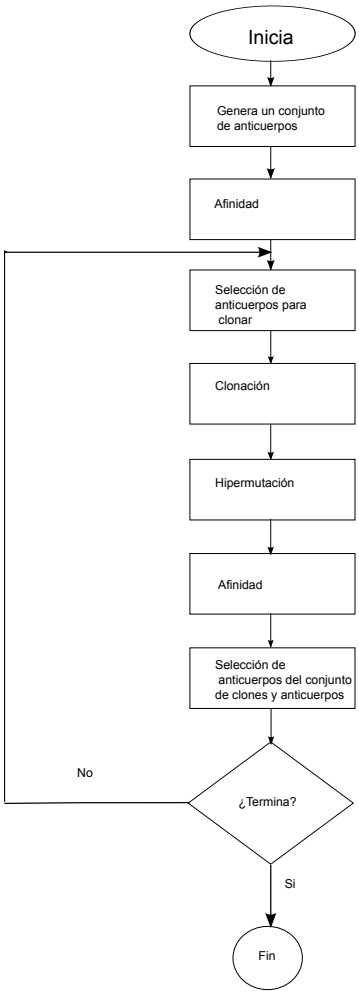


Figura 4.4: Esquema representativo del principio de selección clonal.

de comparadores que forma un elemento r es asignado a su valor de aptitud (línea 3). Posteriormente, los elementos r son clonados (línea 4) y mutados (línea 5). Las mejores soluciones son seleccionadas para actualizar R (línea 7). Éste proceso es repetido un número de veces.

A continuación se describen a más detalle de la representación, la generación de la población inicial y la mutación.

4.3.2.1. Representación

Las soluciones son representadas como una lista de comparadores (RO válida). Cada comparador se describe como una pareja de elementos (x, y) donde x y y son los índices de los buses, superior e inferior respectivamente, con valores $0 \leq x, y \leq n-1$ y $x < y$. Esta representación permite soluciones con tamaños de comparadores variables.

Algoritmo 10 Propuesta de algoritmo de selección clonal para generar RO con un mínimo número de comparadores.

Entrada: n = tamaño de entrada de datos para la RO

1 **begin**
2 Generar una población inicial de forma aleatoria $\{R\}$ de RO válidas (con tamaño $|R|$).
3 Asignar un valor de afinidad a cada elemento r con $r \in R$.
4 Clonar los elementos de $\{R\}$ proporcional a su valor de afinidad. Éstos elementos conforman el conjunto de clones $\{K\}$.
5 Aplicar la hipermutación para k con $k \in K$ con una probabilidad inversamente proporcional a su valor de afinidad.
6 Asignar un valor de afinidad a cada elemento en K.
7 Seleccionar las mejores $|R|$ soluciones a partir de la unión $R \cup K$ para actualizar R .
8 Repetir a partir del paso 3 un determinado número de veces.
9 **end**

Salida: El elemento $\{R\}$ con el valor de mejor afinidad.

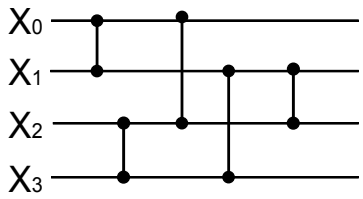


Figura 4.5: Ejemplo de una Red de Ordenamiento para 4 datos de entrada.

Por ejemplo, la RO de la Figura 4.5 puede ser representado como:

$(0,1)(2,3)(0,2)(1,3)(1,2)$

4.3.2.2. Generación de la población inicial

El conjunto inicial R de RO (ver Algoritmo 11) es generado de forma aleatoria pero restringido para contener RO válidas. Esto es, cada $r \in R$ debe ordenar[1] cualquier representación binaria de los números del 0 hasta $2^n - 1$ (ver Principio cero-uno en la Sección 3.2). El algoritmo recibe los siguientes elementos de entrada:

- El conjunto B de todos los número enteros entre 0 hasta $2^n - 1$ representados como cadenas binarias con un tamaño n, y

[1]Una cadena binaria ordenada en B es aquella que tiene todos los 0's al inicio y al final 1's.

- El conjunto C de todos los comparadores. Cada comparador $c \in C$ se describe como un par (x, y) donde x y y son los índices de la actual posición (ver Sección 3.1), x representa la posición superior y, y la posición inferior del comparador. Los valores de x y y se encuentran entre $0 \leq x, y \leq n-1$ y $x < y$, esto es:

  $C = \{(0,1),(0,2),...,(0,n-1),(1,2),(1,3),...,(1,n-1),...,(n-2,n-1)\}$

Este algoritmo regresa un conjunto R de tamaño $|R|$. Es posible que los elementos de S sean de diferente tamaño.

Algoritmo 11 Generación de la población inicial aleatoria $\{R\}$ (línea 2 del Algoritmo 10)

Entrada: n, $\{B\}$, $\{C\}$

```
1   begin
2       for i=0 hasta |R|.
3           j=0
4           Copiar {B} en {B'}.
5           Seleccionar aleatoriamente un elemento c con c ∈ C.
6           Agregar c en S_{i,j}.
7           Aplicar c a cada elemento en B'.
8           Remover las secuencias binarias ordenadas de B'
9           j++
10          Repetir a partir del paso 4 hasta que {B'} se encuentre vacío.
11      end
12  end
```

Salida: $\{R\}$

4.3.2.3. Función de afinidad

La afinidad esta definida como el número de comparadores que conforman una r con $r \in R$. El objetivo es minimizar esta función.

4.3.2.4. Clonación

La clonación (Línea 4 del Algoritmo 10) consiste en la producción de copias de las soluciones actuales r. Las soluciones con mayor afinidad producirán más copias de ellas y, por el contrario, las malas soluciones se clonarán en menor número. El tamaño de clones $(\#C)$ es calculado como sigue,

- Ordenar los elementos de r ($r \in R$) de acuerdo con su afinidad, iniciando con el r de mejor afinidad hasta el peor.

- El número de clones # para el $i-$ésimo elemento r_i está compuesto como:

$$\#r_i = \sum_{i=1}^{|r|} \frac{|R|}{i} \tag{4.3}$$

Donde $|R|$ es el número de elementos de un conjunto R.

4.3.2.5. Hipermutación

La hipermutación (mutación) es un proceso que se aplica a los clones para introducir un cambio en su configuración. Este proceso es aplicado seleccionando un punto m en s y eliminando todos los elementos que vienen después de él, esta parte es reconstruida nuevamente. El porcentaje de mutación debe ser alto para clones con baja afinidad y al contrario para clones con afinidad alta. Un cambio fuerte en s es cuando el punto m se encuentra cercano al comienzo de s (más a la izquierda), mientras que un pequeño cambio es cuando el punto m está cercano al final. El proceso de mutación se puede ver en el Algoritmo 12. Se proponen dos maneras para seleccionar el comparador en cada etapa. La primera (llamada $F1$) considera la calidad de las secuencias binarias en B que se encuentran desordenadas después de que el comparador es aplicado. La segunda (llamada $F2$) considera la cantidad total de bits en B que se encuentran desordenados después de aplicar el comparador. Cada una de estas funciones de aptitud son seleccionadas con una probabilidad de $pF1$ (línea 8 a 23). Los datos de entrada de este algoritmo son:

- El clon k a ser mutado.

- L es el tamaño del clon original k, que es el número de comparadores que tiene k.

- La probabilidad $pF1$ para seleccionar la función de aptitud $F1$.

- El conjunto de $\{B\}$ que constituye el número enteros entre 0 hasta $2^n - 1$ representados como cadenas binarias con un tamaño n

- El conjunto de $\{C\}$ de todos los posibles comparadores.

A continuación se describen ejemplos de los criterios $F1$ y $F2$.

Algoritmo 12 Proceso de hipermutación aplicado a un clon k (línea 4 del Algoritmo 10)

Entrada: k = Un clon a mutar.
L = Tamaño de k.
pF1 = Probabilidad para seleccionar F1 como función de aptitud.
C, B

```
1   begin
2       Copiar {B} en {B'}.
3       Seleccionar un punto de mutación m con 0 ≤ m ≤ L.
4       j = m.
5       Eliminar todos los elementos en k a partir de m hasta L.
6       Aplicar a B' los comparadores de k.
7       Remover las secuencias binarias ordenadas de B'.
8       if pF1
9           for i=0 hasta |C|
10              Copiar {B'} a {Temp}.
11              Aplicar el comparador c_i a las cadenas en Temp.
12              Remover las secuencias binarias ordenadas de Temp.
13              Asignar la aptitud de c_i como el número de cadenas en Temp.
14          end
15      else
16          for i=0 hasta |C|
17              Copiar {B'} a {Temp}.
18              Aplicar el comparador c_i a las cadenas en Temp.
19              Remover las secuencias binarias ordenadas de Temp.
20              Obtener la distancia de Hamming entre los elementos de Temp
                    y su correspondiente secuencia ordenada.
21              Asignar la aptitud de c_i, el valor de la distancia de Hamming.
22          end
23      end
24      Seleccionar el comparador c* con el mínimo valor de aptitud.
25      Agregar el comparador c* en k_j.
26      j++.
27      Aplicar el comparador c* a todos los elementos en B'.
28      Remover las secuencias binarias ordenadas de B'.
29      Repetir de la línea 6 hasta que {B'} esté vacío.
30      k' = k.
31  end
```

Salida: k'

- *F1*

  Como la RO debe realizar la operación de validación esta operación es oportuna para medir la calidad de un comparador. Así que el conjunto de secuencias binarias es la herramienta que nos ayuda a determinar la aptitud de un comparador.En la Figura 4.7 presenta un ejemplo de cómo el comparador $0-1$ afecta a un conjunto B_n. Eliminando las secuencias que resultaron ordenadas y aquellas que se repiten queda un nuevo conjunto de secuencias binarias. Al aplicar este criterio, identifica comparadores que minimizan el conjunto de secuencias por ordenar.

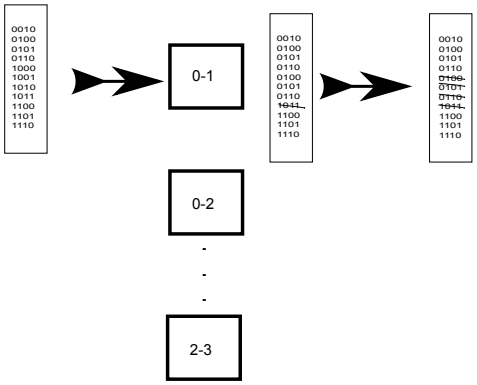


Figura 4.6: Impacto de un comparador sobre el conjunto de secuencias binarias B'.

Con respecto a la cantidad de secuencias binarias que son eliminadas toda vez que causa un impacto el comparador, es importante mencionar que no es lo mismo si se elige un comparador por el tamaño de secuencias que ordena, a que sea elegido por el tamaño de secuencias que deja desordenadas.

- *F2*

  Durante el análisis fue posible observar la influencia del orden que llevan los elementos de B', por ejemplo 1100 y 0110 la primer secuencia binaria requiere de al menos dos comparadores para ser ordenada, mientras que la segunda, requiere de un comparador en el mejor de los casos. El grado de desorden se obtiene al evaluar una determinada secuencia b con su s ordenada, al sumar las posiciones que son desiguales por cada elemento binario.

  La Tabla 4.2 es un ejemplo para obtener el grado de desorden de la secuencia 1100 donde cada bit es comparado con su respectiva secuencia ordenada 0011 etiquetada en la fila $t_{ordenada}$, cada columna verifica si los valores de los bits son desiguales. El grado de desorden de una secuencia es la suma de los bits desiguales con su secuencia ordenada.

  Ahora es necesario conocer qué tan cerca se encuentran las secuencias a su estado ideal, para ello se calcula el grado de desorden de cada secuencia en el conjunto de comparadores candidatos. Entonces se suma por cada comparador candidato c el grado de desorden del conjunto B.

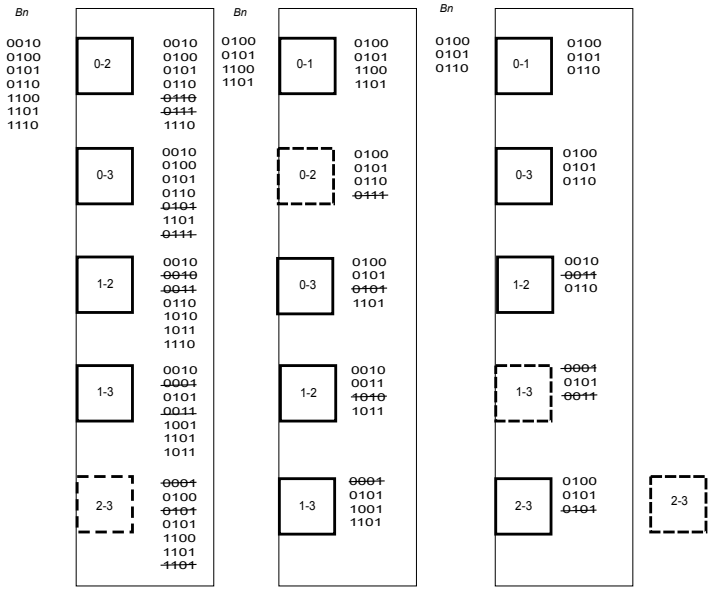


Figura 4.7: Diseño de una RO mediante el criterio F1 en B_n.

b	1	1	0	0
$b_{ordenada}$	0	0	1	1
total desorden Σ	1	1	1	1

Tabla 4.2: Ejemplo del grado de desorden de una secuencia. Se suman las posiciones que son desiguales por cada elemento binario.

Por ejemplo, en la Tabla 4.3 cada columna contiene un vector de dos elementos; la secuencia por ordenar y el grado de desorden (gd) de dicha secuencia. Por cada comparador se acumula el total de grado de desorden y es seleccionado el comparador cuya suma del grado de desorden en su evaluación sea la menor. El comparador $[0-1]$ tiene un grado de desorden igual a 6, mientras que $[1-3]$ suma un total de grado de desorden con el valor de 4, lo que significa que este comparador es el prometedor.

b \ c	0-1	gd	1-3	gd
10001	01001	2	10001	2
11000	11000	4	10010	2
Total desorden	2	6	2	4

Tabla 4.3: Ejemplo de elección de comparadores que disminuyen el grado de desorden de B.

4.4. Experimentos y Resultados

El algoritmo propuesto fue usado para diseñar RO eficientes para tamaños de entrada desde $n = 9$ hasta $n = 17$. Los experimentos se llevaron a cabo en una PC con procesador Intel core i5 2.2 Ghz. con 4 GB de RAM y el código fue escrito en lenguaje C en sistema operativo Linux.

El algoritmo fue ejecutado 30 veces. En general, los parámetros experimentalmente probados fueron:

- Número de generaciones del SIA : $100 - 3000$
- Tamaño de la población de anticuerpos $10 - 300$
- $pF1 = (0.30 - 0.99)$

Particularmente, para las RO de $n = 9$ bastaron 10 individuos con 100 generaciones para encontrar la mejor solución conocida. En los experimentos para la RO con $n = 10$ bastaron con 20 individuos de población inicial, al igual que la RO para $n = 11$ con 100 generaciones. Con la RO para 12 se obtuvieron las mejores soluciones hasta 300 generaciones y con un mínimo de 30 elementos en la población. Para las RO con $n = 13$, $n = 14$ y $n = 15$, el tamaño de población fue de $100 - 120$ con $1000 - 2000$ generaciones y, finalmente para la RO de $n = 16$ se ocuparon 300 individuos de población y 3000 generaciones.

Los tiempos que tardaron los experimentos por corrida, con los parámetros arriba mencionados son:

- $n = 10$ 1.07 min
- $n = 11$ 1.47 min
- $n = 12$ 58.79 min
- $n = 13$ 236.27 min
- $n = 14$ 1036.27 min
- $n = 15$ 3036.27 min

Las RO encontradas se pueden consultar en el Apéndice A.

A continuación se describe puntual los resultados encontrados:

1. Se encontraron nuevas configruaciones para las RO con $n = 9$, $n = 10$, $n = 11$, $n = 12$, $n = 13$, $n = 14$, $n = 15$, $n = 16$ y $n = 17$.

Tabla 4.4: Tabla de medidas estadísticas de las 30 ejecuciones del SIA (número de comparadores).

Medidas \ n	9	10	11	12	13	14	15	16
Mejor	25	29	35	39	45	51	56	60
Peor	25	29	36	40	46	54	57	63
Promedio	25	0	35.1111	39.1852	45.2592	53.5185	56.8518	61.5556
Varianza	0	0	0.1026	0.1567	0.1994	1.1823	0.1305	0.9487
Desv. Std.	0	0	0.3202	0.3958	0.04467	1.0874	0.3620	0.9740

2. Todas las soluciones encontradas como eficientes tienen una configuración en sus comparadores diferente a las del estado del arte.

3. Las RO para $n = 10$ y $n = 12$, fueron descubiertas con el mínimo número de capas conocido respecto de n.

4. En los experimentos para la RO con $n = 12$, al aplicar únicamente el criterio FP1 para construir el anticuerpo, se encontró la RO que presenta Knuth en su Libro con el mínimo conocido de comparadores.

5. Se encuentra una RO para $n = 10$ con 30 comparadores y 8 capas. La RO eficiente conocida tiene 29 comparadores y 9 capas, mientras que la RO rápida tiene 31 comparadores y 7 capas.

6. Las RO para $n = 13$ y $n = 14$ utilizan *comparadores primarios* y encuentran el mínimo conocido con diferente configuración.

En la Tabla 4.4 se ilustran los valores estadísticos de las 30 corridas independientes del SIA. La primera fila presenta los diferentes tamaños de datos de entrada para las RO y, en la primera columna están las medidas estadísticas con las que se evalúan los resultados. Con éstos, la técnica del SIA demuestra su consistencia para encontrar las RO con entradas de datos $n = 9$ y $n = 10$. Además el SIA también consigue calidad en los resultados, esto es, que para todos los valores de n, se obtiene el resultado del mejor conocido.

4.5. Conclusiones

El algoritmo propuesto es viable para encontrar RO eficientes con diferentes configuraciones en tiempos razonables. La primera hipótesis propuesta es confirmada, pero además también se confirma que empleando los *comparadores primarios*, esta técnica, puede encontrar el resto de los comparadores de la RO con una configuración diferente.

Se confirma que únicamente con el SIA no es posible encontrar las RO mínimas, es necesario emplear criterios para encontrar comparadores útiles que hagan que la búsqueda se ajuste a soluciones convenientes. Finalmente se encontraron las RO eficientes y hasta con el mínimo número de capas conocido en la literatura especializada.

Capítulo 5

Algoritmo de optimización por cúmulo de partículas para la construcción de redes de ordenamiento rápidas

En este apartado presentamos la propuesta para construir RO rápidas mediante una heurística bioinspirada llamada algoritmo de optimización por cúmulo de partículas. Esta heurística es reconocida en el estado del arte como una de las mejores para resolver algoritmos de alta complejidad y aplicada para solucionar problemas en una amplia variedad de áreas.

En primer lugar se mencionan los conceptos básicos de la heurística bioinspirada y su funcionamiento. Posteriormente se presenta la propuesta junto con el mecanismo de construcción de la RO.

5.1. Introducción

En las redes de ordenamiento el tiempo discreto requerido para ordenar se refiere al número de pasos (capas) que la RO debería ejecutar después de verificar la independencia de los comparadores. Una RO es *rápida* si tiene un bajo número de capas, donde cada una de ellas esta compuesta por un subconjunto de comparadores que puedan realizar sus operaciones concurrentemente sin interferir entre ellas.

El Algoritmo 13) contiene las operaciones que necesita una RO para $n = 4$ (ver Figura 5.1). Este ejemplo es únicamente para mostrar cómo se identifican los comparadores independientes para formar una capa: las operaciones de las líneas 2 y 4 se pueden desempeñar en paralelo y de igual forma para las líneas 6 y 8. Por lo tanto, al final

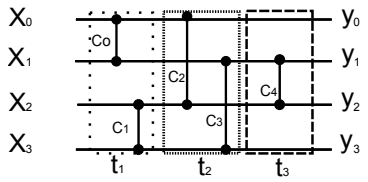


Figura 5.1: Ejemplo de una Red de Ordenamiento para 4 datos de entrada.

con 3 unidades de tiempo o pasos es posible ordenar toda la secuencia si aplicamos el paralelismo mencionado.

Algoritmo 13 Algoritmo de ordenamiento para $n = 4$ correspondiente a la RO ilustrada en la Figura 5.1

Entrada: x_0, x_1, x_2, x_3

```
1    begin
2        if (x0 > x1)
3            Intercambia (x0, x1);
4        if (x2 > x3)
5            Intercambia (x2, x3);
6        if (x0 > x2)
7            Intercambia (x0, x2);
8        if (x1 > x3)
9            Intercambia (x1, x3);
10       if (x1 > x2)
11           Intercambia (x1, x2);
12       y0 = x0; y1 = x1; y2 = x2; y3 = x3;
13   end
```

Salida: y_0, y_1, y_2, y_3

Además en la mayoría de la literatura especializada las publicaciones están orientadas a la construcción de RO que minimizan el número de comparadores, existiendo poco trabajo dedicado al diseño de RO rápidas, esto es, RO con máximo paralelismo. La medida de paralelismo en una RO se determina con el número de capas [49].

Diferentes trabajos se han encontrado para construir RO rápidas, como en los trabajos presentados en [53] y [7] se aplican técnicas clásicas de ordenamiento, tales como el *sellsort* y *mergesort* respectivamente. Además en [1] fue propuesta una estrategia recursiva.

Es posible construir RO extensas (RO de tamaño de entrada mayores que 16) median-

te una técnica conocida (por ejemplo, mezcla par-impar), no obstante, el número de comparadores no es el número óptimo y debido a ésto, las RO extensas para n llegan a ser ineficientes en la práctica.

En este capítulo, se propone una representación compacta y novedosa que ayuda a la construcción de RO rápidas. La búsqueda es mediante la técnica bioinspirada llamada algoritmo de Optimización por Cúmulo de Partículas (*Particle Swarm Optimization* PSO.

Los resultados presentados muestran que la estrategia es eficiente y capaz de encontrar rápidas RO. Además, la cantidad de comparadores encontrados en las RO resultantes son aceptables de acuerdo con el estado del arte.

5.2. Algoritmo de Optimización por Cúmulo de Partículas (PSO)

El PSO [33, 32] es una heurísticas bioinspirada inspirada por el comportamiento social del vuelo de las aves o el movimiento de los bancos de peces. Ésta se ha convertido en un técnica popular para resolver problemas de optimización complejos.

El PSO fue diseñado para solucionar problemas de optimización con números reales y binarios [34, 40]. En la literatura especializada son pocos los trabajos que utilizan la heurística para resolver problemas en espacios discretos con una codificación binaria [45, 59]. En [22] afirma que la limitación del PSO para adaptarse a problemas discretos de combinación se debe a la falta de principios en la generalización de la heurística.

El PSO se describe generalmente como una población de vectores (llamadas partículas) cuyas trayectorias oscilan en torno a una región. En el caso más general, teniendo en cuenta la configuración del enjambre (población), hay dos versiones del PSO: globales y locales. En la versión global (*gbest*) la trayectoria de cada partícula está influenciada por el mejor punto encontrado por cualquier miembro de todo el enjambre. En la versión local (*lbest*) la trayectoria de cada partícula puede ser influenciada sólo por el mejor miembro en el vecindario. Además, cada partícula se ve influida por su mejor éxito previo.

El PSO con representación binaria es el que aplicamos en nuestra propuesta y tiene dos características, la primera es que la representación de la partícula es binaria, mientras que la segunda es que su velocidad debe ser transformada en una probabilidad de cambio a uno.

Se denota a Np como el número de partículas en la población, cada una está compuesta por $Y_i^t = (y_{i1}^t, y_{i2}^t, \ldots, y_{iD}^t)$, $y_{id}^t \in [0, 1]$. i es una partícula con D *bits* en la iteración t, donde Y_i es una solución potencial que tiene una tasa de cambio llamada velocidad.

La velocidad es $V_i^t = (v_{i1}^t, v_{i2}^t, ..., v_{iD}^t)$, $v_{id}^t \in \mathbb{R}$. La mejor solución i que se ha obtenido en la iteración t esta representada por $P_i^t = (p_{i1}^t, p_{i2}^t, ..., p_{iD}^t)$, y $p_g^t = (p_{g1}^t, p_{g2}^t, ..., p_{gD}^t)$ representa la mejor solución obtenida a partir de P_i^t en la población (*gbest*) o del vecindario local (*lbest*), en la iteración t.

Cada partícula ajusta su velocidad de acuerdo a la siguiente ecuación:

$$v_{id}^t = v_{id}^{t-1} + c_1 r_1 (p_{id}^t - y_{id}^t) + c_2 r_2 (p_{gd}^t - y_{id}^t) \tag{5.1}$$

donde c_1 es el factor de aprendizaje cognitivo, c_2 es el factor de aprendizaje social, y r_1 y r_2 son números generados en forma aleatoria con distribución uniforme entre $[0, 1]$. Los valores c y r determinan los pesos de dos partes, donde la suma de ellos es limitada normalmente a 4 [33].

La velocidad representa la probabilidad de que el valor actual tome el valor de uno. Para mantener el valor en el intervalo $[0, 1]$ se emplea la función sigmoidal:

$$sig(v_{id}^t) = \frac{1}{1 + exp(-v_{id}^t)} \tag{5.2}$$

Algoritmo de PSO

for $i = 0$ to Np **do**
  Calculate the fitness value for each p_i
  if $Y_i^t > P_i^t$ **then**
    for $d = 1$ to D bits **do**
      $P_{id}^t = Y_i^t$
    end for
  end if
  $g = i$
  for j=indices of neighbors **do**
    if $P_j^t > P_g^t$ **then**
      $g = j$
    end if
  end for
  for $d = 1$ to D **do**
    $v_{id}^t = v_{id}^{t-1} + c_1 r_1 (P_{id}^t - Y_{id}^t) + c_2 r_2 (P_{gd}^t - Y_{id}^t)$
    $v_{id}^t \in [-V_{max}, +V_{max}]$
    if random number $[0, 1] < sig(v_{id}^t)$ **then**
      $Y_i^{t+1} = 1$
    else
      $Y_i^t = 0$
    end if
  end for
end for

5.3. Propuesta

5.3.1. Planteamiento del problema

El problema que aborda este trabajo es el diseño de RO rápidas para tamaño de los datos de entrada de $n = 10$, $n = 12$, y $n = 14$.

5.3.2. Metodología

La idea general es limitar el tamaño de la representación (el número de variables de decisión) al mínimo número de capas vacías que tiene una RO válida, entonces nuestra técnica procede a llenar cada una de estas capas con una *adecuada* selección de los comparadores. En este caso una *adecuada* selección de los comparadores significa que éstos garantizarán una RO válida.

Se establece que para una entrada de datos de tamaño n, la siguiente información es conocida:

- El máximo número de comparadores en una capa es ($\lfloor n/2 \rfloor$),
- El número de capas en la RO rápida (ver Tabla 3.2 en la Sección 3.1.2).

Por lo tanto, en este escenario, el problema es determinar qué comparadores deberían poblar cada una de las capas. Debido a la gran cantidad de posibilidades, se decidió utilizar una heurística para encontrar tales RO. Para este caso se presenta la versión binaria canónica del PSO.

La representación de las partículas, la función de aptitud, y la descripción general del algoritmo se presenta a continuación.

5.3.3. Representación de la partícula

Una partícula representa una RO candidata compuesta por l capas (donde l es la más rápida teórica). Cada capa se considera como una variable de decisión.

Recordemos que para que una determinada RO con entrada n, se conoce su número mínimo de capas (dato extraído de la Tabla 3.2 en la Sección 3.1.2), por tanto éste será el tamaño de la partícula.

Para seleccionar los comparadores que poblarán la capa, todas las posibles configuraciones de capas son generadas y enumeradas, y sus números (utilizando codificación binaria) funcionarán como identificadores.

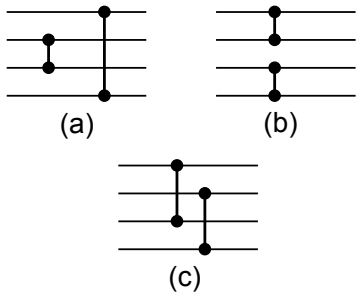


Figura 5.2: Total de configuraciones de comparadores para una RO de $n = 4$. (a) Muestra la configuración $\{1,2\}\{0,3\}$, (b) $\{0,1\}\{2,3\}$, y (c)$\{0,2\}\{1,3\}$.

Por ejemplo,supongamos que estamos buscando la RO para $n = 4$. El mínimo número de capas conocido es 3 (es el más rápido, ver (ver Tabla 3.2 en la Sección 3.1.2) tercera fila, tercera columna). Por lo tanto, una partícula P se compone de 3 variables de decisión que es equivalente a 3 capas: $P = (L1, L2, L3)$

Ahora, dentro de cada capa L podemos colocar 2 comparadores paralelos (basados en que $n/2 = 2$). Es evidente que sólo hay tres posibles maneras de poner 2 comparadores paralelos, que son los siguientes (se ilustra en la Figura 5.2):

- $\{1,2\}\{0,3\}$
- $\{0,1\}\{2,3\}$
- $\{0,2\}\{1,3\}$

Las configuraciones son etiquetadas como: 1,2 y 3. Por lo tanto, las variables de decisión pueden tomar uno de éstos tres valores. Continuando con el ejemplo, la partícula

$$P = (1, 3, 1)$$

puede representar la RO

$$(L1 = (\{1,2\}\{0,3\}), (L2 = \{0,2\}\{1,3\}), (L3 = \{1,2\}\{0,3\}))$$

que se puede ver en la Figura 5.3. En este trabajo los valores son codificados con representación binaria.

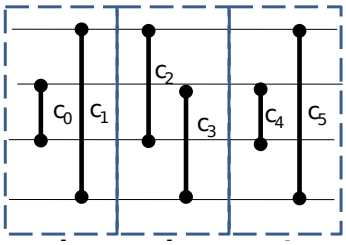


Figura 5.3: Ejemplo de una RO para $n = 4$

5.3.4. Función de aptitud

La RO representada por una partícula *debe* ordenar todas las permutaciones de datos de entrada $x = (x_0, x_1, ..., x_{n-1})$ para reconocer su validez.

Esto es equivalente a ordenar todas las 2^n secuencias binarias (Principio cero-uno mencionado en la Sección 3.2). De acuerdo con este principio, se quiere minimizar el número de secuencias binarias desordenadas que deja una partícula. Así que, el mínimo valor esperado es cero.

La *aptitud* de una partícula es el número de secuencias binarias que no es capaz de ordenar. Las secuencias binarias son representadas por B_n y, S_n el conjunto de secuencias binarias ordenadas, por lo tanto $S_n \in B_n$.

Retomando el ejemplo con la RO para $n = 4$, los datos de entrada son $B_4 = \{0000, 0001, 0010, 0011, 0100, ...1111\}$ y, S_4 es el conjunto de secuencias binarias ordenadas de B_4, así que $S_4 = \{0000, 0001, 0011, 0111, 1111\}$. El objetivo es que el conjunto de B_n llegue a ser igual que el conjunto de S_n, pero no al contrario.

La partícula presentada en la Figura 5.3 tiene un valor de aptitud igual a 2 porque existen dos secuencias binarias $0010, 1011$ en b_4, que la RO no puede ordenar.

Como una única manera de reconocer si una RO ordena un conjunto de secuencias, es aplicando todos sus comparadores a cada secuencia de B_n y supervisando que la salida sea correctamente ordenada.

Un algoritmo de Optimización por Cúmulo de Partículas binario canónico (presentado en la Sección 5.2) es utilizado para resolver este problema.

5.3.5. Tamaño del espacio de búsqueda

De acuerdo con este tipo de representación y considerando la manera en la que se plantea el problema, nos permite encontrar un espacio de búsqueda bajo las siguientes condiciones:

1. Que el número de capas es fijo,

2. Que se conoce el número mínimo de capas para n,
3. Que se conoce el número máximo de comparadores por capa y,
4. Que n es un número par.

y en consideración de los puntos anteriores, se tiene la Ecuación:

$$\text{Tamaño de espacio de búsqueda} = \left(\frac{\prod_{k=0}^{\frac{n}{2}-1} \binom{n-2k}{2}}{\frac{n}{2}!} \right)^{\#C} \tag{5.3}$$

Donde $\#C$ es el número de capas mínimo que conforma la RO y n es el tamaño de datos de entrada.

5.4. Resultados experimentales

Nuestros experimentos fueron dirigidos a diseñar RO rápidas para $n = 10$, $n = 12$ y $n = 14$ mediante la ayuda del PSO binario canónico. Fueron aplicadas las dos versiones: *lbest* y *gbest*, para cada n. La versión *lbest* del PSO tuvo un mejor desempeño en todos los experimentos. Todos los experimentos fueron ejecutados en un procesador Core i5-430M (2,26 GHz). Para cada caso se realizaron 20 ejecuciones independientes. A continuación se detallan los experimentos y resultados para cada n.

5.4.1. Red de Ordenamiento para $n = 10$ datos de entrada

El diseño de la partícula es sin *comparadores primarios* (es decir, sin tomar comparadores de otra RO conocida)y los parámetros del algoritmo son los siguientes:

- Número de partículas 100
- Iteraciones 1000
- $V_{max} = 4.0$
- $c_1 =$ aleatorio $(0, 4)$
- $c_2 = 4.0 - c_1$
- Tamaño de vencindario = 10

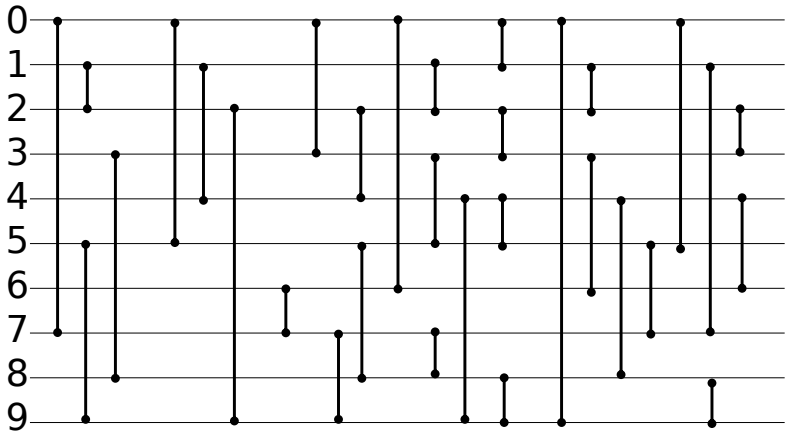


Figura 5.4: La mejor RO $n = 10$ encontrada por el algoritmo PSO con el mínimo número de capas conocido igual a 7.

El mejor resultado conocido es una RO para $n = 10$ con 7 capas y 31 comparadores, resultado que coincide con la RO rápida y óptima en el número de comparadores. Además cuenta con una configuración de comparadores diferente a las conocidas (nuevo diseño). Su esquema esta presentado en la Figura 5.4. El tiempo promedio por cada corrida en promedio fue de $44.58min$.

5.4.2. Red de Ordenamiento para $n = 12$ datos de entrada

El diseño de la partícula es sin *comparadores primarios* y los parámetros del algoritmo son los siguientes:

- Número de partículas 100
- Iteraciones 3000
- $V_{max} = 4.0$
- $c_1 =$ aleatorio $(0, 4)$
- $c_2 = 4.0 - c_1$
- Tamaño de vencindario = 10

La técnica encuentra una RO con 51 comparadores y 9 capas. El esquema de esta RO se encuentra en la Figura 5.5. Las capas inicialmente se fijaron con 8 pasos y al no encontrar RO válidas en la evaluación $300,000$ se agregó una capa más. Los tiempos por cada corrida en promedio fue de $8hrs$.

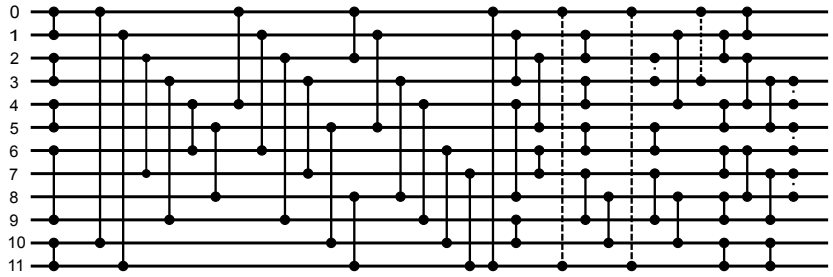


Figura 5.5: La mejor RO rápida encontrada por el PSO con 51 comparadores y 9 capas.

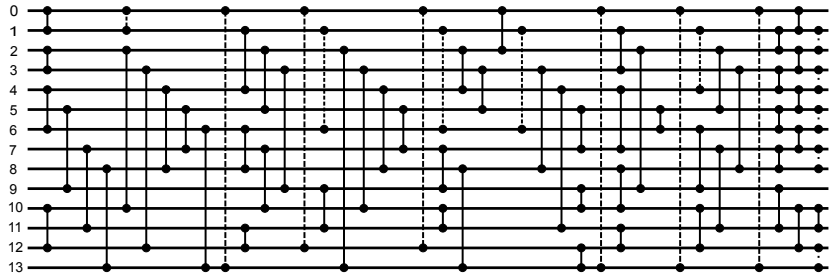


Figura 5.6: La mejor RO para $n = 14$ encontrada por el PSO con 62 comparadores

.

5.4.3. Red de Ordenamiento para $n = 14$ datos de entrada

El diseño de la partícula es sin *comparadores primarios* y los parámetros del algoritmo son los siguientes:

- Número de partículas 100
- Iteraciones 5000
- $V_{max} = 4.0$
- $c_1 =$ aleatorio $(0, 4)$
- $c_2 = 4.0 - c_1$
- Tamaño de vencindario = 10

Con respecto a los experimentos para la RO con $n = 14$, la heurística encontró una RO con 11 capas y 62 comparadores. Esta solución se encuentra en la Figura 5.6. Esta RO es un nuevo diseño.

Finalmente, en la Tabla 5.1 se ilustran los valores estadísticos de las 20 corridas independientes del PSO para buscar RO rápidas. La primera fila presenta los diferentes tamaños de datos de entrada para las RO y, en la primera columna se muestran las medidas estadísticas obtenidas de las 20 ejecuciones. En los resultados puede verse una

Tabla 5.1: Tabla de medidas estadísticas de las 20 ejecuciones independientes del PSO (número de capas).

Medidas \ n	10	12	14
Mejor	7	9	11
Peor	12	13	14
Promedio	9.2963	10.1852	12.5926
Varianza	1.13960	2.0028	1.78917
Desv. Std.	1.0675	1.4152	1.3376

distancia de entre 1 y 2 comparadores en promedio, de la RO rápida. Sin embargo, los datos muestran que la heurística es capaz de encontrar calidad en las soluciones para la $n = 10$ y $n = 12$. Los tiempos por cada corrida en promedio fue de $16hrs$.

5.4.4. Comparadores redundantes

Los comparadores redundantes en una *RO* son aquellos que no realizan ninguna operación de intercambio en las secuencias de prueba. Nuestra técnica diseña una RO posiblemente con algunos comparadores redundantes. Por ejemplo con la RO para $n = 10$ que se obtiene cuenta con 35 comparadores en 7 capas, de las cuales 4 comparadores son redundantes, estos comparadores no aplican la operación de intercambio durante el desempeño de la RO. Por lo tanto se consideran inútiles y son ignorados. En la Figura 5.4, 5.5 y 5.6, las líneas verticales punteadas son comparadores redundantes.

5.5. Conclusiones

En general, existen dos formas como medida de eficiencia en una RO. La primera es el número de comparadores, y la segunda es el número de pasos o capas para identificar qué tan rápida es la RO. Considerando este último punto, se presenta una estrategia para diseñar RO de maximiza paralelización de comparadores. La idea principal es fijar el tamaño de capas igual a la mínima conocida. De esta forma, los comparadores por cada capa son buscados y agregados a la partícula, la cual usa una representación compacta. La técnica de búsqueda es adaptada al PSO. La RO para $n = 10$,$n = 12$ y $n = 14$ con ayuda de la heurística se encontraron nuevos diseños, los resultados fueron muy cercanos a las RO rápidas y la $n = 10$ además de una nueva configuración alcanza los mínimos conocidos en ambos objetivos.

Capítulo 6

Uso de Redes de Ordenamiento para ordenar grandes cantidades de datos

Las RO óptimas conocidas actualmente ordenan tamaños de datos de entrada muy pequeños si los comparamos con las grandes cantidades de datos que usan la mayoría de las aplicaciones del mundo real. Aunque su desempeño es óptimo en términos del número de comparaciones que efectúan, las RO tienen la desventaja de falta de flexibilidad pues para cada tamaño de entrada, la RO es completamente diferente y debe ser específicamente diseñada.

Por otro lado, los algoritmos tradicionales como "quicksort", "bubble sort", "merge sort", "shell sort", "heapsort", "insertion", "introsort", "shear sorting", etc. tienen la ventaja de adaptarse a cualquier cantidad de entrada de datos lo que los hace muy flexible, aunque como se sabe, no son óptimos en cuanto al número de comparaciones que realizan.

En este capítulo se propone hacer uso de una RO para mejorar el desempeño a un algoritmo tradicional (*quicksort*) en grandes cantidades de datos. En la primera parte se presenta el algoritmo *quicksort*. Se aprovecha su estrategia *divide y vencerás* para agregar la implementación de una RO. En la segunda parte se detallan los experimentos realizados para obtener las RO con un tamaño de datos de entrada extenso y posteriormente se describe la propuesta. En general, los resultados demuestran que la RO ayuda a reducir el tiempo de ejecución.

6.1. Conceptos básicos: Algoritmo quicksort

El algoritmo quicksort (también conocido como "*Partition-Exchange Sort*") fue presentado en 1960 por Tony Hoare [28, 13]. Utiliza la estrategia de divide y vencerás en la cual una lista es dividida en dos sublistas más pequeñas. A una sublista le son asignados los elementos con los valores más pequeños y la otra con los elementos de valor mayor. Luego, cada sublista es ordenada de forma recursiva de la misma forma.

La secuencia de pasos es la siguiente:

1. Seleccione un elemento de la lista que se llamará "pivote".
2. Ordenar la lista de tal manera que todos los valores que son menores que el pivote estarán situados a su izquierda (antes del pivote). Además, todos los valores mayores que el pivote (ver Algoritmo 14) estarán ubicados a su derecha (después del pivote). De esta manera, el valor del pivote estará en su posición.
3. Para cada sublista, se repiten los pasos anteriores en forma recurrente hasta que el tamaño de las sublistas es cero o uno (ver Algoritmo 15).

Esta idea se ilustra en la Figura 6.1. Quicksort es un algoritmo en promedio muy eficiente, y en el mejor de los casos hace $O(n\,log\,n)$ comparaciones para ordenar n elementos. En el peor de los casos hace $O(n^2)$. Se sabe que la elección del pivote es un factor que determina en gran medida el desempeno del algoritmo.

Algunas variantes de este algoritmo son presentadas en [37, 10], donde los autores proponen algunas modificaciones para reducir el en segundos de ejecución.

6.2. Construcción de una Red de Ordenamiento a partir de otra Red de Ordenamiento existente

Las RO eficientes conocidas son para un tamaño de datos de entrada pequeño con respecto a la gran cantidad de información que los sistemas modernos manejan. Con motivo de incrementar el tamaño de datos de entrada, algunos autores proponen mecanismos como; incrementar en $n+1$, la mezcla bitónica y la técnica "Odd-Even Merging". Existe trabajo en la literatura donde se utiliza la técnica "Odd-Even Merging" propuesta por Batcher que usa exactamente $(n^2-n+4)^{n-2}-1$ comparadores para ordenar un arreglo de (2^n). Esta técnica es para ordenar dos listas de tamaño "g" y "h", como se muestra en la Figura 6.2, la primera tiene como elementos $\{t_1, t_2..., t_g\}$ ordenados, y por otro lado, una segunda lista con elementos $\{w_1, w_2, ..., w_h\}$. Después de la operación de mezcla la lista resultante tiene "g + h" elementos de salida en orden que son $\{u_1, ..., u_{g+h-1}, u_{g+h}\}$, como se muestra en la Figura 6.3. La mezcla se construye

Algoritmo 14 partición

```
Entrada: A, izq, der
 1    begin
 2     i, d
 3     pivot = A[(der + izq)/2]
 4     i = izq
 5     d = (der)
 6         while ((rm-lm) > 0)
 7             while (A[d] > pivot)
 8                 d−−
 9             while (A[d] < pivot)
10                 r−−
11             if (i < d)
12                 temp = A[d]
13                 A[d] = A[i]
14                 A[i] = temp
15             else
16                 return i
17         end
18    end
```

Salida: d

Algoritmo 15 quicksort

```
Entrada: A, izq, der
 1    begin
 2     i, d
 3         if (izq < der)
 4             limiteParticion = particion(A, izq, der)
 5             quicksort (A, izq, limiteParticion)
 6             quicksort (A, limiteParticion, der)
 7         end
 8    end
```

Salida: A

intercalando con los índices de número par y los índices de número impar de dos listas de entrada. El menor de los elementos del conjunto de datos posterior a la mezcla es el elementos más pequeño al final de la lista. Los pasos para ordenar dos listas son:

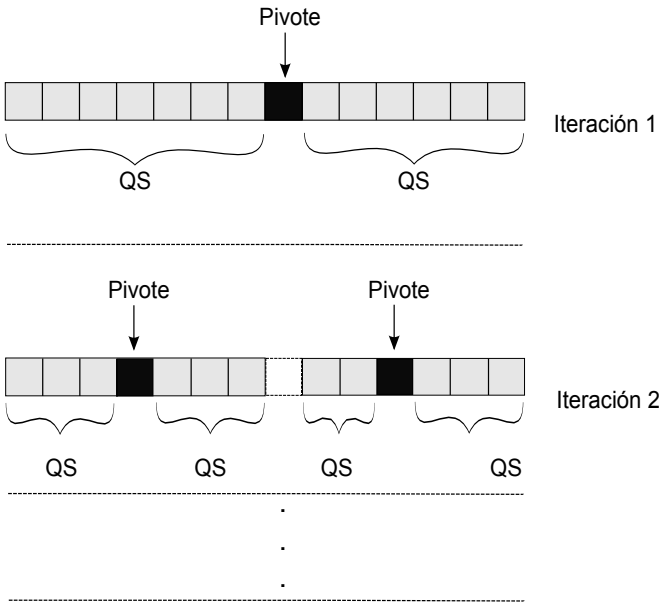


Figura 6.1: Operación de partición recursiva del algoritmo quicksort.

1. Mezcla de los índices de número par $\{t_1, t_3, t_5, ...\}$ con $\{w_1, w_3, w_5, ...\}$ para formar las secuencias $\{x_1, x_2, x_3, ...\}$ de índices en orden.

2. Mezcla de los índices de número impar $\{t_2, t_4, t_6, ...\}$ con $\{w_2, w_4, w_6, ...\}$ para formar las secuencias $\{y_1, y_2, y_3, ...\}$ de índices en orden.

3. Entonces $u_1 = x_1$ y el conjunto de comparadores $u_{2i} = min(x_{i+1}, y_i)$ y $u_{2i+1} = max(x_{i+1}, y_i)$ $i = \{1, 2, 3, ...\}$

Los pasos 1 y 2 se pueden aplicar en paralelo de acuerdo al teorema en [5, 36, 58].

Al unir dos RO para $n = 4$ se obtiene la RO para $2n$ entradas que esta en la Figura 6.4. El número completo de comparadores es de 19. El esquema de esta RO esta dividida en 6 pasos: Los pasos del $1-3$ ordenan dos listas para $n = 4$, ambos en orden decreciente, en el paso 4 son empleados 4 comparadores para mezclar los 8 elementos de las listas y entonces, dos comparadores que están en el paso 5 ordenan las 4 listas que resultaron del paso anterior. En el último paso se usan 3 comparadores para mezclar las secuencias ordenadas para formar una lista ordenada de 8 elementos.

Este método trabaja mejor para un pequeño tamaño de entrada de datos, y disminuye su desempeño como incrementa su tamaño de entrada, esto es, para un tamaño de entrada más grande la RO tendrá más comparadores de un valor óptimo.

Para obtener una RO con un tamaño de datos de entrada extenso, a la se identificará como m mediante la técnica "Odd-Even Merging", se lleva a cabo un incremento

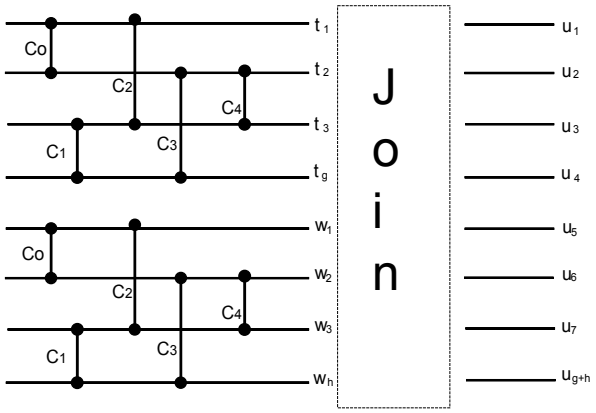


Figura 6.2: Esquema "Odd-Even Mergesort" para ordenar $n = 8$ datos a partir de una RO para $n = 4$.

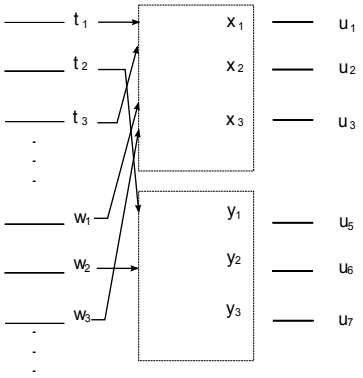


Figura 6.3: Esquema "Odd-even" merging.

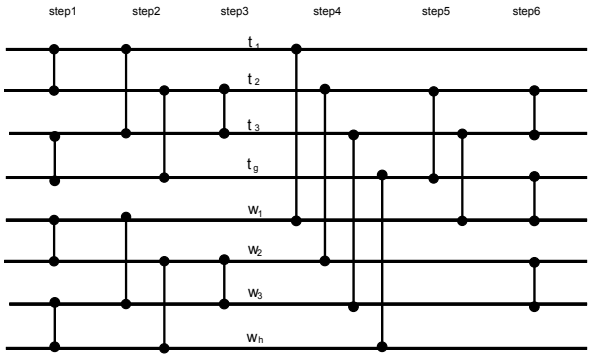


Figura 6.4: Esquema de una RO para $n = 8$ entradas a partir de dos RO para $n = 4$.

de $2n$ partiendo de una RO de tamaño de entrada pequeño. La RO de tamaño n resultante se vuelve a utilizar para incrementar el tamaño en $2n$ nuevamente, hasta que $m = 2n$.

Observamos que las RO construidas con esta metodología utilizan una mayor cantidad de comparadores en lugar de emplear una RO eficientes para diseñar otra RO para un tamaño de entrada extenso [6]. Para conocer la configuración y el número de comparadores se llevaron a cabo los siguientes experimentos:

- **Experimento 1**. A partir de una RO para $n = 4$, se quiere construir una RO para $n = 8$, $n = 16$, $n = 32$, $n = 64$, $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$. La metodología es la siguiente:

  1. La RO para $n = 8$ se obtiene a partir de la RO para $n = 4$ utilizando la técnica "Odd-Even Merging".
  2. Para obtener la RO para $n = 16$, se emplea la RO obtenida del punto anterior (RO para $n = 8$) y se consigue $2n$ mediante la técnica "Odd-Even Merging".
  3. La RO para $n = 32$ utiliza la RO que se obtiene del punto anterior (RO para $n = 16$) y mediante la técnica "Odd-Even Merging" se obtiene el incremento a $n = 32$.

  El proceso continúa siempre tomando la RO resultante del punto anterior y aplicando la técnica "Odd-Even Merging" hasta formar la RO para $n = 4096$.

- **Experimento 2**. Se quiere construir una RO para $n = 32$, $n = 64$, $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$. La metodología es la siguiente:

  1. Para obtener la RO para $n = 32$ se considera la RO de Green (ver Figura 6.5) la cual tiene 60 comparadores y es la más eficiente para $n = 16$, se aplica la técnica "Odd-Even Merging" para incrementar la entrada en $2n$.
  2. A partir de la RO para $n = 32$ que se obtuvo del punto anterior, se aplica la técnica "Odd-Even Merging" para obtener la RO para $n = 64$.
  3. Nuevamente, con la RO para $n = 64$ se aplica la técnica"Odd-Even Merging" y se obtiene la RO para $n = 128$.

  El proceso continúa siempre tomando la RO resultante del punto anterior y aplicando la técnica "Odd-Even Merging" hasta formar la RO para $n = 4096$.

- **Experimento 3**. Se quiere construir una RO para $n = 32$, $n = 64$, $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$. La metodología es la siguiente:

  1. Para obtener la RO para $n = 32$ se considera dos RO (diferente a la de Green) con 60 comparadores para $n = 16$.

2. A partir de la RO para $n = 32$ se aplica la técnica "Odd-Even Merging" para obtener la RO para $n = 64$.
3. Con la RO para $n = 64$ se aplica la técnica"Odd-Even Merging" y se obtiene la RO para $n = 128$.

El proceso continúa siempre tomando la RO resultante del punto anterior y aplicando la técnica "Odd-Even Merging" hasta formar la RO para $n = 4096$.

- **Experimento 4**. Se quiere construir una RO para $n = 32$, $n = 64$, $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$. La metodología es la siguiente:

  1. Para obtener la RO para $n = 32$ se considera dos RO para $n = 16$ diferentes con 60 comparadores cada una.
  2. A partir de la RO para $n = 32$ se aplica la técnica "Odd-Even Merging" para obtener la RO para $n = 64$.
  3. Con la RO para $n = 64$ se aplica la técnica"Odd-Even Merging" y se obtiene la RO para $n = 128$.

  El proceso continúa siempre tomando la RO resultante del punto anterior y aplicando la técnica "Odd-Even Merging" hasta formar la RO para $n = 4096$.

- **Experimento 5**. Se quiere construir una RO para $n = 32$, $n = 64$, $n = 128$, $n = 256$, $n = 512$, $n = 1024$, $n = 2048$ y $n = 4096$. La metodología es la siguiente:

  1. Para obtener la RO para $n = 64$ se considera dos RO para $n = 32$ diferentes; la RO para $n = 32$ que resulta del experimento 2 y del experimento 4.
  2. A partir de la RO para $n = 64$ se aplica la técnica "Odd-Even Merging" para obtener la RO para $n = 128$.
  3. Con la RO para $n = 128$ se aplica la técnica"Odd-Even Merging" y se obtiene la RO para $n = 256$.

  El proceso continúa siempre tomando la RO resultante del punto anterior y aplicando la técnica "Odd-Even Merging" hasta formar la RO para $n = 4096$.

- **Experimento 6**. Se quiere construir una RO para $n = 16$ mediante la mezcla de dos RO, una para $n = 6$ y otra para $n = 10$.

En la Tabla 6.1 se puede ver como disminuye el número de comparadores empleando la construcción con RO extensas a partir de RO eficientes, así que es factible y más eficiente una de las RO extensa obtenida a partir del experimento 2 a 5. La primera fila son las entradas de las RO para una n y, cada fila contiene los datos resultantes de cada experimento. Aunque las RO fueron experimentalmente diseñadas con diferentes RO eficiente el mínimo número de comparadores no cambio. Así que es mejor emplear las RO de la fila 2.

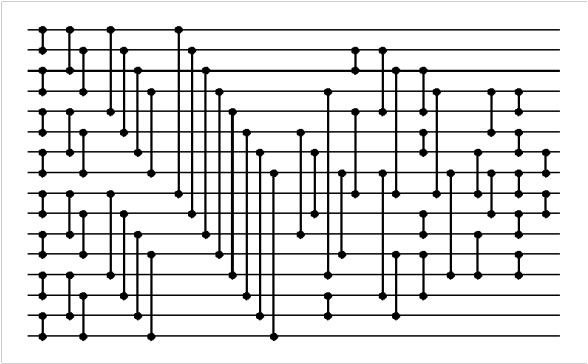


Figura 6.5: RO para $n = 16$ diseñada por Green, es la mejor conocida con 60 comparadores.

Tabla 6.1: Tabla de resultados de los experimentos propuestos para encontrar una RO de la forma $2n$.

Experimento \ n	8	16	32	64	128	256	512	1024	2048	4096	
1	19	63	191	543	1471	3839	9727	24063	58367	139263	
2	-	-	**185**	**531**	**1447**	**3791**	**9631**	**23871**	**57983**	**138485**	
3	-	-	**185**	**531**	**1447**	**3791**	**9631**	**23871**	**57983**	**138485**	
4	-	-	**185**	**531**	**1447**	**3791**	**9631**	**23871**	**57983**	**138485**	
5	-	-	**185**	**531**	**1447**	**3791**	**9631**	**23871**	**57983**	**138485**	

6.3. Propuesta: Quick-RO

Considerando que el algoritmo quicksort efectúa divisiones de una lista de forma iterativa donde las sublistas resultantes llegan a ser cada vez más pequeñas que la original y que las RO eficientes conocidas son para un tamaño de datos de entrada pequeño, la propuesta consiste en acoplar una RO en el algoritmo quicksort para ordenar grandes cantidades de datos y verificar que su desempeño es mejorable. El algoritmo propuesto será llamado (Quick + RO).

La idea general es aplicar el algoritmo quicksort en su forma tradicional, llevar a cabo las sublistas las veces que sean necesarias hasta que alguna de ellas sea tan pequeña que el tamaño de entrada sea igual que la RO. Esta idea está ilustrada en la Figura 6.6.

Con este pequeño cambio en el algoritmo Quicksort se quiere mejorar su desempeño manteniendo la característica de su flexibilidad. Así el algoritmo puede ordenar cualquier tamaño de datos de entrada.

Supongamos que se ha seleccionado la RO para trabajar con un tamaño $n = 16$. Dada una entrada de datos A de tamaño Z (donde Z es un número grande) que será ordenado, el algoritmo Quick+RO es definido como sigue:

1. Se divide el arreglo A en 2 partes (a_i y a_d), seleccionando el elemento cuya posición esta en medio del arreglo, al llamado pivote y al que denotamos como m.

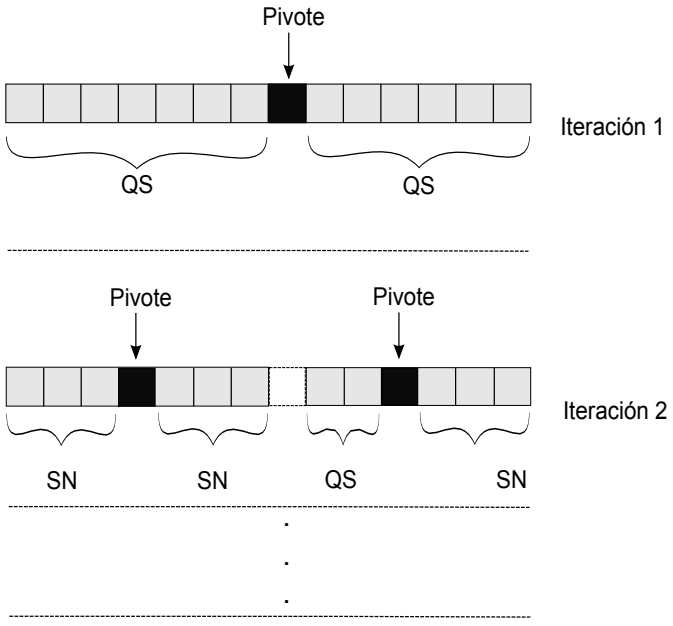


Figura 6.6: Esquema del algoritmo propuesto Quick+RO

2. Se compara cada elemento de a_i con los elementos de a_d hasta el pivote y se mueven todos los elementos menores que m a la izquierda, en el arreglo a_i, y los elementos más grandes que m a la derecha, en el arreglo a_d.

3. Se verifica si las sublistas son de tamaño n:

   - Si la lista es de tamaño de entrada igual a n entonces la sublista es ordenada mediante la RO.
   - sino entonces regresar al paso 1 recursivamente.

La salida de este algoritmo es la lista A totalmente ordenada. Es importante mencionar que el Quick+RO utiliza una RO específica, es decir, que el tamaño de datos de entrada es fijo. Así que únicamente si la sublista tiene exactamente n elementos, entonces la RO puede ser aplicada. No es posible conocer los tamaños resultantes de los subarreglos que deja el quicksort al dividir la lista, por lo que no es posible determinar a priori el número exacto de veces que la RO es aplicada.

6.4. Experimentos y Resultados

Con el fin de evaluar la eficiencia del algoritmo propuesto, un conjunto de experimentos fueron diseñados. El Quick+RO se aplicó a listas de datos (números) con diferentes tamaños de entrada, también con diferentes permutaciones de datos de entrada. Además,

se experimentó utilizando diferentes RO. La configuración por cada experimento es una combinación de las siguientes opciones:

- Tamaño de datos de entrada: Dos listas conformadas por $1,000,000$ y $10,000,000$ de números para ser ordenados los cuales fueron utilizados como entrada.
- Permutación de datos de entrada: Tres tipos de diferentes configuraciones de listas de entrada fueron utilizadas:
  - Números generados de forma aleatoria.
  - Números ordenados.
  - Números en orden invertido.
- Tamaño de dato de entrada para las RO: $n = 16$ y $n = 256$.

Por lo tanto, se llevaron a cabo 12 experimentos con diferentes configuraciones. Para obtener suficientes datos estadísticos cada experimento fue ejecutado en 30 ocasiones. Tanto el tiempo (en segundos) como el número de comparaciones desempeñadas fueron valorados para el quicksort y para el Quick+RO.

Los resultados estadísticos, en cuanto al tiempo, se encuentran en la Tabla 6.2. La primera columna presenta la configuración de entrada de los datos, en la segunda columna se muestra el tamaño de los datos de entrada, en las siguientes dos columnas están los tiempos consumidos (en segundos) usando Quick+RO utilizando $n = 16$ y $n = 256$, respectivamente y en la última columna el tiempo consumido por el quicksort.

En cuanto a los resultados estadísticos del número de comparaciones expuestos en la Tabla 6.3, las columnas están asignadas de la misma forma, exceptuando que las columnas 3 y 4 tienen las comparaciones efectuadas por el Quick+RO para $n = 16$ y $n = 256$, respectivamente y la última columna contiene el número de comparaciones que realiza el algoritmo quicksort. Con respecto a las RO utilizadas, se empleó la RO mínima conocida para $n = 16$. Para la RO extensa ($n = 256$) se obtuvo de la forma que se menciona en el apartado 6.2 en este capítulo.

Se puede observar que el desempeño en cuanto al tiempo, la RO para $n = 16$ en la configuración de números aleatorios es mejor en comparación con el quicksort, sin embargo en las configuraciones donde los datos están ordenados e invertidos, el quicksort ocupa menos tiempo que el Quick+RO.

Los resultados del Quick+RO para $n = 256$ mejoran notablemente, comenzando con los tiempos, los cuales son mejores que los del quicksort para todas las configuraciones propuestas. En cuanto al número de comparaciones, en la configuración de números aleatorios el Quick+RO con $n = 256$ tiene un mejor desempeño que el quicksort.

En todos los casos de prueba se cuenta con una ganancia en tiempo y comparaciones por parte del algoritmo Quick+RO. En cuanto a los datos con una configuración aleatoria

Tabla 6.2: Resultados estadísticos (en segundos) del Quick+RO y quicksort.

Configuración de entrada de los datos	Tamaño de entrada de los datos Z		**QS+RO** con $n = 16$	**QS+RO** con $n = 256$	Original quicksort
Aleatorio	1,000,000	Promedio	0.287025	**0.269315**	0.284484
		Mediana	0.283842	**0.266893**	0.278212
		Mejor	0.278647	**0.262631**	0.283074
		Peor	0.316102	0.305107	**0.278919**
	10,000,000	Promedio	3.505109	**3.456773**	3.503901
		Mediana	3.495797	**3.446392**	3.491349
		Mejor	3.463172	**3.393128**	3.465855
		Peor	**3.574376**	3.632529	3.590459
Ordenado	1,000,000	Promedio	0.111039	**0.109753**	0.112652
		Mediana	0.106884	**0.108799**	0.110282
		Mejor	0.106884	**0.107760**	0.109995
		Peor	0.126193	**0.115464**	0.138364
	10,000,000	Promedio	1.40588	**1.358374**	1.416539
		Mediana	1.402868	**1.355140**	1.413907
		Mejor	1.395423	**1.346074**	1.410352
		Peor	1.414223	**1.393968**	1.431942
Invertido	1,000,000	Promedio	0.115451	**0.113330**	0.1184194
		Mediana	0.114292	**0.114311**	0.117748
		Mejor	0.113429	**0.112885**	0.116315
		Peor	0.124456	**0.116046**	0.122324
	10,000,000	Promedio	1.470613	**1.422433**	1.487349
		Mediana	1.465786	**1.419184**	1.486568
		Mejor	1.461634	**1.409075**	1.474512
		Peor	1.498166	**1.459109**	1.517120

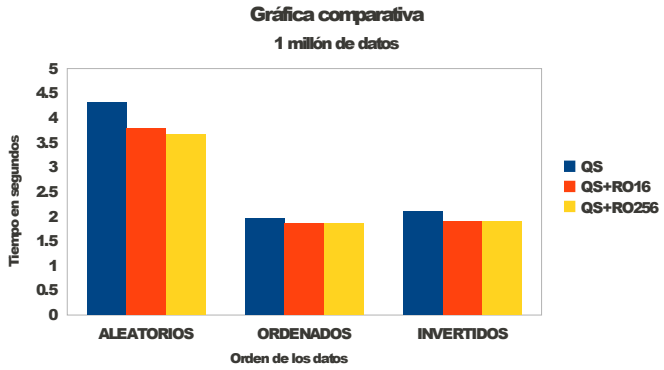


Figura 6.7: Gráfica comparativa en tiempo con 1 millón de datos entre las técnicas de ordenamiento quicksort y Quick+RO.

Tabla 6.3: Resultados estadísticos en número de comparaciones del Quick+RO y quicksort.

Configuración de entrada de los datos	Tamaño de entrada de los datos Z		**QS+RO** con $n = 16$	**QS+RO** con $n = 256$	Original quicksort
Aleatorio	1,000,000	Promedio	31563203.9	**31047573.2**	43199483.9
		Mediana	30614366	**30866951**	43042794
		Mejor	33404132	**30153976**	42246335
		Peor	**31354258**	32913228	45044477
	10,000,000	Promedio	466673967.3	**461857715.5**	544013791.6
		Mediana	466072537	**457137848**	542430812
		Mejor	474876907	**455052517**	536471351
		Peor	**466072537**	476971359	559793832
Ordenado	1,000,000	Promedio	18491248.9	**18485467.8**	19548810.2
		Mediana	18491096	**18485411**	19548662
		Mejor	18489821	**18485171**	19548053
		Peor	18492491	**18485760**	19549746
	10,000,000	Promedio	241277124.5	**235340529.6**	245704189.5
		Mediana	241268483	**235338829**	245701131
		Mejor	241196844	**235330901**	245690300
		Peor	241337327	**235348489**	245722497
Invertido	1,000,000	Promedio	18991031.3	**18985450.4**	21048727.0
		Mediana	18990840	**18985429**	21048675
		Mejor	18989911	**18985171**	21048051
		Peor	18992609	**18985738**	21049458
	10,000,000	Promedio	251278637.1	**245340656.8**	260704211.4
		Mediana	251287707	**245339256**	260702951
		Mejor	251264881	**245330352**	260689641
		Peor	251275019	**245354164**	260722491

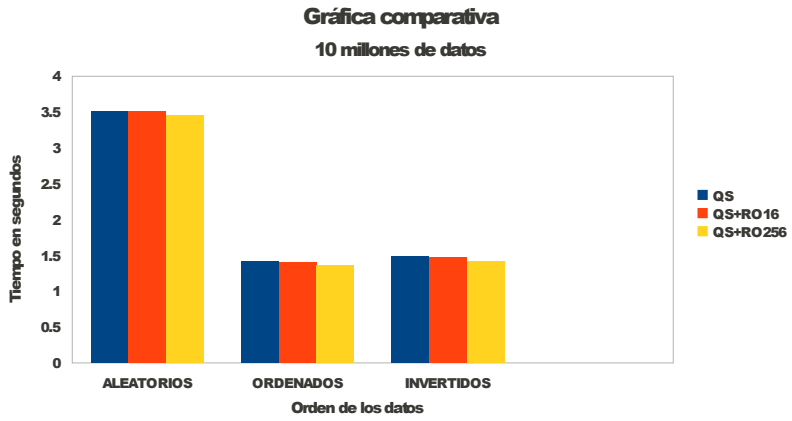


Figura 6.8: Gráfica comparativa en segundos con 10 millones de datos entre las técnicas de ordenamiento quicksort y Quick+RO..

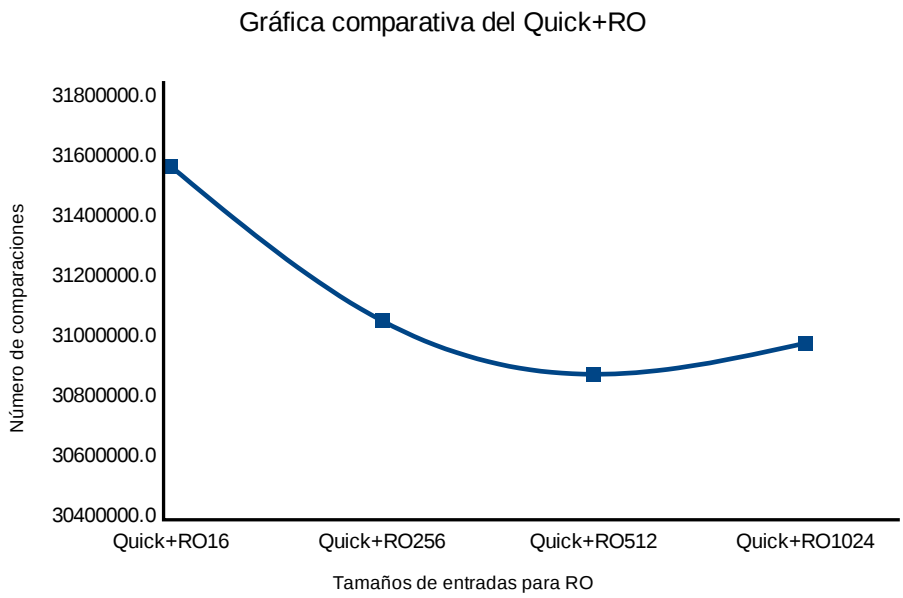


Figura 6.9: Gráfica comparativa en número de comparaciones del Quick+RO para 1 millón de datos aleatorios con tamaños de entrada para la RO de $n = 16$, $n = 256$, $n = 512$ y $n = 1024$.

Tabla 6.4: Desempeño en número de comparaciones del Quick+RO con diferentes tamaños de entrada para la RO para 1 millón de datos aleatorios

RO para n / Experimento	16	256	512	1024
Núm. comparaciones	31563203.9	31047573.2	**30869133.0**	30973064.0

para un millón de datos, el método Quick+RO (con $n = 256$) mejoró en segundos con respecto al quicksort.

En cuanto a los datos ordenados e invertidos el algoritmo obtiene una mejoría cuando los datos de entrada están en un orden aleatorio. Durante el desempeño del Quick+RO requiere de un número menor de particiones, ya que, cuando la sublista requiera ser ordenada se manda llamar a la RO. Esta disminución de operaciones de partición es lo que la hace ser más eficiente.

En estos experimentos el pivote se elige en la mitad de la lista para que exista una mayor probabilidad de utilizar una RO.

Con los datos obtenidos hasta el momento, la técnica Quick+RO con una RO para $n = 256$ requiere de un menor número de comparaciones. No obstante, para conocer si el comportamiento de la técnica Quick+RO continúa mejorando su desempeño, se incrementó el tamaño de datos de entrada, se extenso se experimentó con el mismo conjunto de datos de 1 millón de datos aleatorios y con tamaño de datos de entrada extenso en la RO para $n = 512$ y $n = 1024$. Los resultados se pueden observar en la gráfica de la Figura 6.9.

La gráfica de la Figura 6.9 ilustra el desempeño (en número de comparaciones) de la técnica Quick+RO en 1 millón de datos aleatorios. En el eje de las "abscisas" se encuentra la técnica Quick+RO con los diferentes tamaños de n (16, 256, 512 y 1024), en el eje de las "ordenadas' se presentan el número de comparaciones en escala logarítmica. Este trabajo empírico desarrollado ayudó a determinar que una RO para $n = 512$ mejora aún más el número de comparadores. Por el contrario, si a la técnica Quick+RO se le añade una RO para $n = 1024$ ocupará un número mayor de comparadores que si se aplica una de RO para $n = 512$ o una RO para $n = 256$ datos de entrada. También los datos obtenidos del experimento se encuentran en la Tabla 6.4

6.5. Conclusiones

Se realizó un acoplamiento entre el algoritmo quicksort y una RO al que llamamos Quick+RO. Se concluye que la unión de los algoritmos es una estrategia ventajosa que resulta en un algoritmo de ordenamiento más eficiente. Las debilidades de cada uno de los algoritmos por separado se ven superadas hasta cierto punto con las fortalezas del otro. La falta de flexibilidad de las RO para adaptarse a diferentes cantidades de entrada de datos se ve superada al unirlo al quicksort. Por otro lado, la optimalidad de las RO respecto al número de comparaciones, resulta en una disminución de comparaciones del algoritmo Quick+RO.

Capítulo 7

Conclusiones

En esta tesis el objetivo principal fue encontrar RO eficientes y rápidas competitivas con la literatura especializada, se implementaron dos heurísticas un algoritmo SIA y un PSO. Además se presentó una propuesta para mostrar que las RO también pueden ser útiles en el ordenamiento de grandes cantidades de datos. Las propuestas en este documento fueron:

1. Diseñar RO eficientes con ayuda de un algoritmo de Sistema Inmune Artificial.

2. Diseñar RO rápidas con ayuda de un algoritmo de optimización por cúmulo de partículas.

3. Usar RO en el ordenamiento de grandes cantidades de datos.

A continuación se describen las conclusiones puntuales por cada propuesta, así como el trabajo futuro.

7.1. Propuesta 1

1. Se implementó un mecanismo de construcción donde un comparador es seleccionado con base en cierta aptitud. Este tipo de mecanismo ayudó a disminuir en gran medida el espacio de búsqueda.

2. El algoritmo SIA fue implementado en su versión más sencilla y ayudó en la búsqueda de RO eficientes.

3. El tiempo para encontrar las RO eficientes, es competitivo con el estado del arte.

4. Las RO eficientes encontradas son diferentes a las conocidas en la literatura especializada. La única RO que se encuentra idéntica a otra que se encuentra en el libro de D.Knuth es una para una entrada de datos de 12.

5. Aunque la metodología en esta propuesta no manejó la estructura o representación para maximizar los comparadores en paralelo, algunos los resultados coinciden con el mínimo número de capas o en algunos casos muy cercas al mejor conocido.

6. Se encontró una RO interesantes para $n = 10$ donde se encontraron 30 comparadores en 8 capas, ya que en la literatura se tiene como una RO eficiente para $n = 10$ con 29 comparadores y 9 capas, y una RO rápida con 7 capas y 31 comparadores.

7. Entre los resultados se tiene: que se encontraron las RO para $n = 9$ hasta $n = 16$ con el número de comparadores igual al mínimo conocido. En cuanto al número de capas la RO para $n = 9$ y $n = 12$ alcanzaron el mínimo conocido.

8. En comparación con otras técnicas que emplean como un algoritmo de optimización local o algoritmos greedy, esta heurística es sencilla de implementar y los resultados encontrados son competitivos.

9. El uso de "comparadores primarios" no fue limitante para encontrar distintas configuraciones de RO e inclusive hasta pudieron encontrarse distintas RO eficientes.

10. Trabajo a futuro:

    - Considerar distintas medidas de aptitud para seleccionar nuevos comparadores en la construcción de la RO.
    - Aunque se probaron distintos grupos de "comparadores primarios" de diferentes RO, es recomendable continuar con la exploración de nuevos espacios del problema, es decir, continuar proponiendo nuevos "comparadores primarios".

7.2. Propuesta 2

1. En lo referente al método de construcción, se propone uno nuevo y eficiente, consiste en agregar capas con máxima paralelización de comparadores durante la construcción de la partícula.

2. Se identifica la versión local del PSO como la de mejor desempeño para resolver el problema.

3. Se encontraron RO rápidas para $n = 10$ donde su tiempo es igual a la mejor conocida en el estado de arte. Teóricamente demostrado que no existe una RO más rápida.

4. Las RO para $n = 12$ y $n = 14$ tienen tiempos muy cercanos a los mejores conocidos

5. Las configuraciones de comparadores y tiempos de las RO encontradas son distintas a las conocidas en la literatura especializada.

6. Trabajo a futuro:

   - Probar con nuevas técnicas de selección de la capa para formar la partícula.
   - Probar "comparadores primarios" y mediante la técnica buscar RO de tamaño de entrada extenso.

7.3. Propuesta 3

1. Se propone la adherencia de una RO en un algoritmo Quicksort para verificar si ayuda en el desempeño de un algoritmo tradicional.

2. Se mostró que las RO ayudan en el ordenamiento de grandes cantidades de datos. Se utilizaron RO para $n = 16$ y $n = 256$ para probar el desempeño del Quicksort, al que llamamos Quick+RO.

3. Los resultados para la RO para $n = 256$ tiene un mejor desempeño que la RO para $n = 16$ en la mayoría de los casos.

4. Trabajo a futuro:

   - Probar los mismos escenarios para otra técnica de ordenamiento clásica y averiguar si también la RO ayuda en el desempeño.
   - Probar con diferentes tipos de datos el punto anterior, al igual que la técnica presentada en este documento.
   - Probar con RO de diferentes tamaños de n.

Apéndice A

Redes de ordenamiento eficientes

Aquí se presentan las RO obtenidas de la técnica del sistema inmune artificial presentado en el Capítulo 4. Las RO se presentan de una forma esquemática y otra numérica.

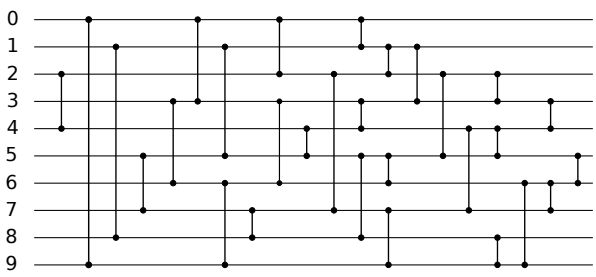


Figura A.1: Red para $n = 10$ con 29 comparadores y 9 capas.

1—[2-4][0-9][1-8][5-7][3-6]
2—[0-3][1-5][6-9][7-8]
3—[0-2][3-6][4-5]
4—[2-7][0-1][3-4][5-8]
5—[1-2][5-6][7-9]
6—[1-3][2-5][4-7][8-9]
7—[2-3][4-5][6-8]
8—[3-4][6-7]
9—[5-6]

Tabla A.1: Comparadores de la RO con $n = 10$, cuyo esquema se puede observar en la Figura A.2. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

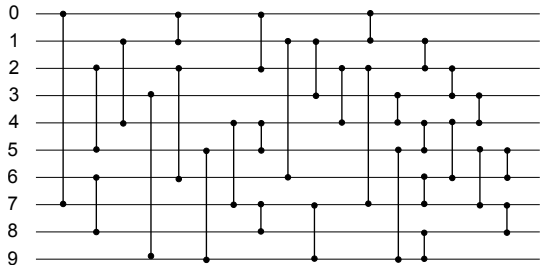


Figura A.2: Red para $n = 10$ con 30 comparadores y 8 capas.

1—[0-7][6-8][2-5][1-4][3-9]
2—[2-6][5-8][4-7][0-1]
3—[0-2][1-6][4-5][7-8]
4—[1-3][7-9][2-4]
5—[0-1][2-7][3-4][5-9]
6—[6-7][1-2][4-5][8-9]
7—[4-6][5-7][2-3]
8—[5-6][3-4][7-8]

Tabla A.2: .Comparadores de la RO con $n = 10$ correspondiente a la Figura A.2. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

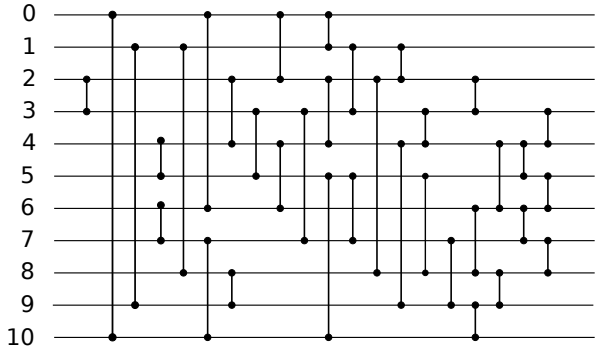


Figura A.3: Red para $n = 11$ con 35 comparadores y 11 capas.

1 —[2-3][0-10][1-9][6-7][4-5]
2 —[1-8][0-6][7-10][2-4]
3 —[3-5][8-9][0-2][4-6]
4 —[3-7][2-4][0-1][5-10]
5 —[1-3][5-7][2-8][4-9]
6 —[1-2][5-8][3-4][7-9]
7 —[6-8][2-3][9-10]
8 —[4-6][8-9]
9 —[4-5][6-7]
10 —[5-6][7-8][3-4]

Tabla A.3: Comparadores de la RO con $n = 11$, cuyo esquema se puede observar en la Figura A.3. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

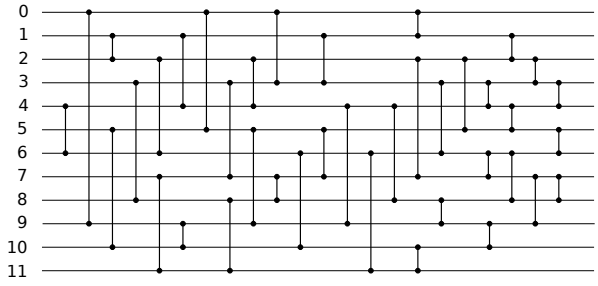


Figura A.4: Red de ordenamiento para $n = 12$ con 39 comparadores y 9 unidades de tiempo.

1 —[4-6][0-9][5-10][1-2][3-8][7-11]
2 —[2-6][1-4][9-10][0-5][3-7][8-11]
3 —[2-4][5-9][7-8][0-3][6-10]
4 —[5-7][4-9][1-3][6-11]
5 —[4-8][2-7][0-1][10-11][3-6]
6 —[8-9][2-5][3-4][6-7]
7 —[9-10][4-5][1-2][6-8]
8 —[2-3][7-9]
9 —[3-4][7-8][5-6]

Tabla A.4: Comparadores de la RO con $n = 12$, cuyo esquema se puede observar en la Figura A.4. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador

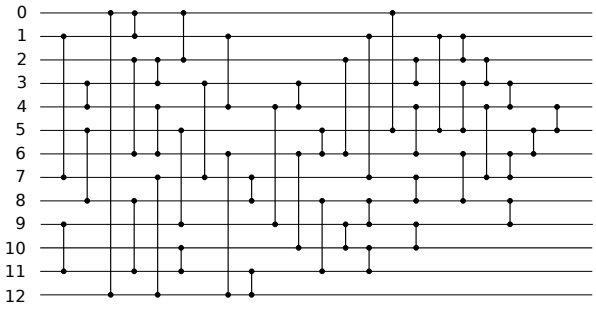


Figura A.5: Red de ordenamiento para $n = 13$ con 45 comparadores y 11 capas.

1 —[1-7][9-11][3-4][5-8][0-12][2-6]
2 —[0-1][2-3][4-6][8-11][7-12][5-9]
3 —[0-2][3-7][10-11][1-4][6-12]
4 —[7-8][11-12][4-9][6-10]
5 —[3-4][5-6][8-11][9-10]
6 —[2-6][1-7][0-5][8-9][10-11]
7 —[4-6][2-3][7,8][1,5][9-10]
8 —[3-5][1-2][4-7][6-8]
9 —[2-3][6-7][5-6]
10 —[8-9][3-4]
11 —[4-5]

Tabla A.5: Comparadores de la RO con $n = 13$, cuyo esquema se puede observar en la Figura A.5. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

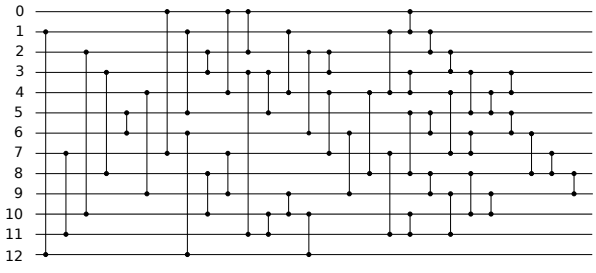


Figura A.6: Red de ordenamiento para $n = 13$ con 47 comparadores y 16 capas.

1 —[1-12][7-11][2-10][3-8][5-6][4-9]
2 —[0-7][1-5][6-12][2-3][8-10]
3 —[0-4][3-11][7-9]
4 —[0-2][3-5][11-12][9-10][1-4]
5 —[2-6][10-12][4-7]
6 —[2-3][6-9][4-8][7-11]
7 —[1-4][5-8][10-11]
8 —[0-1][3-4][5-6][8-9]
9 —[1-2][4-7][9-11][8-10]
10 —[2-3][6-7]
11 —[3-5][9-10]
12 —[4-5]
13 —[3-4][5-6]
14 —[6-8]
15 —[7-8]
16 —[8-9]

Tabla A.6: Comparadores de la RO con $n = 13$, cuyo esquema se puede observar en la Figura A.6. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

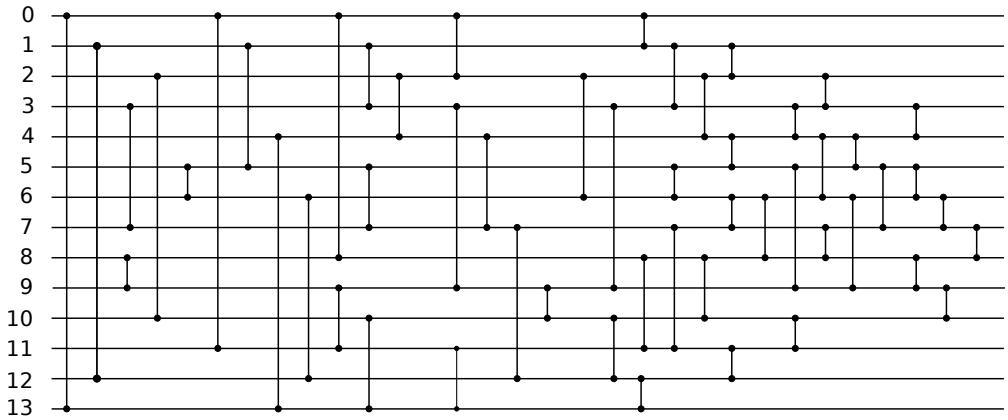


Figura A.7: Red de ordenamiento para $n = 14$ con 52 comparadores y 13 capas.

1 —[0-1][2-3][4-5][6-7][8-9][10-11][12-13]
2 —[0-2][4-6][8-10][1-3][5-7][9-11]
3 —[0-4][8-12][1-5][9-13][2-6][3-7]
4 —[0-8][1-9][2-10][3-11][4-12][5-13]
5 —[5-10][9-12][7-13][3-6][1-4][2-8]
6 —[6-12][3-9][4-8][7-11][1-2]
7 —[6-9][2-4][5-8][7-10][11-13]
8 —[3-5][6-8][7-9][10-12]
9 —[3-4][5-6][11-12]
10 —[6-7]
11 —[7-8]
12 —[8-10]
13 —[9-10]

Tabla A.7: Comparadores de la RO con $n = 14$, cuyo esquema se puede observar en la Figura A.7. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador.

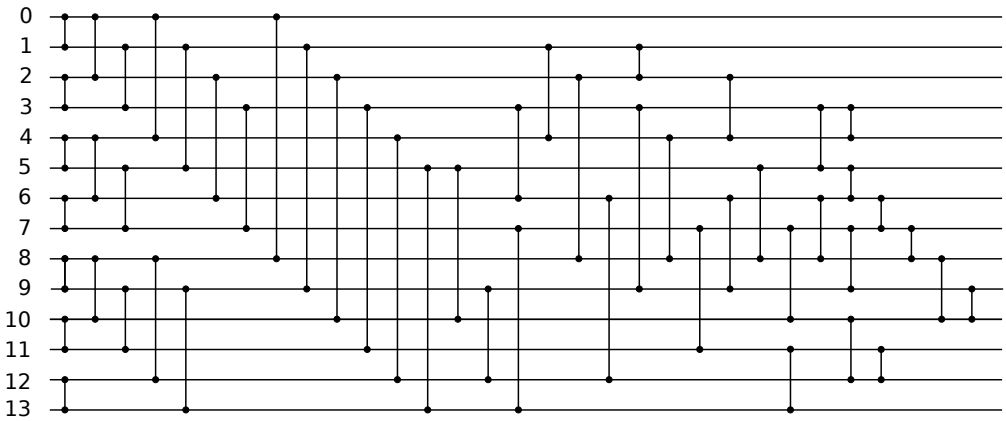


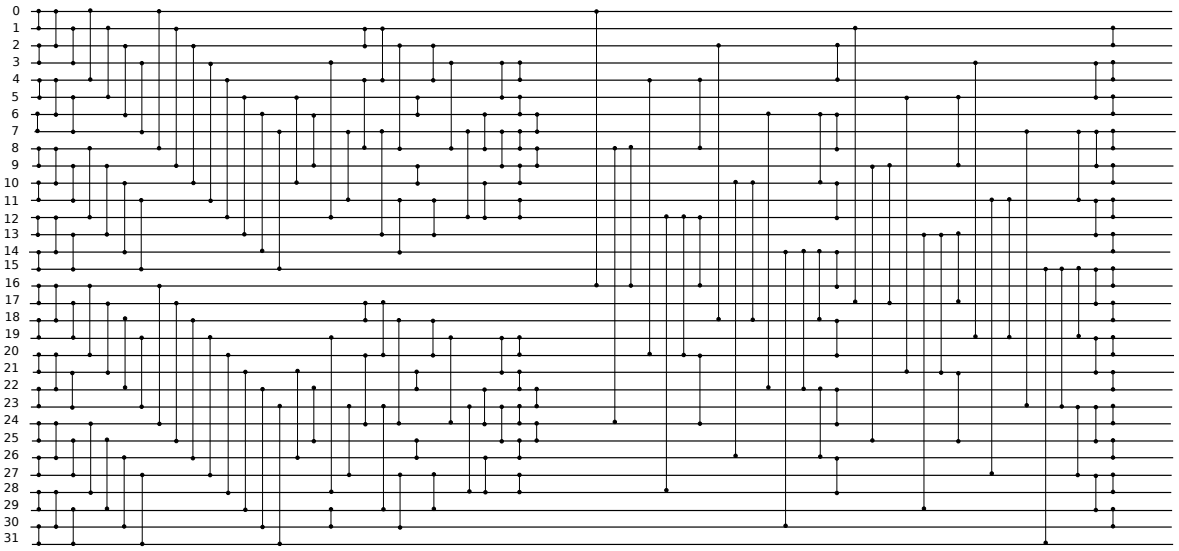
Figura A.8: Red de ordenamiento para $n = 14$ con 13 unidades de tiempo.

1 —[0-13][1-12][8-9][3-7][2-10][5-6][0-11]
2 —[1-5][4-13][6-12][9-11][0-8][1-3]
3 —[2-4][10-13][5-7][3-9]
4 —[0-2][4-8][11-13][7-12][9-10]
5 —[2-6][3-9][10-12][8-11][0-1][12-13]
6 —[5-6][1-3][7-11][8-10][2-4]
7 —[6-7][4-5][1-2][11-12]
8 —[6-8][5-9][3-4][10-11]
9 —[4-6][2-3][7-8]
10 —[4-5][6-9][5-7]
11 —[5-6][3-4][8-9]
12 —[6-7][9-10]
13 —[7-8]

Tabla A.8: Comparadores de la RO con $n = 14$ con 53 comparadores, cuyo esquema se puede observar en la Figura A.8. Cada línea horizontal representa una unidad de tiempo (capa). Cada pareja de números entre [ ] es un comparador..

Apéndice B

Red de Ordenamiento extensa



Apéndice C

"Designing Minimal Sorting Networks using a Bio-inspired Technique"

Designing Minimal Sorting Networks using a Bio-inspired Technique

Blanca C. López and Nareli Cruz-Cortés
Centro de Investigación en Computación
Instituto Politécnico Nacional
México City, Mexico

Abstract

Sorting Networks (SN) are efficient tools to order an input data sequence. They are composed by a set of comparison-exchange operations called *comparators*. The comparators are *a priori* fixed for a determined input size. The comparators are independent of the input configuration. To find SN with minimal number of comparators results on an optimal manner to sort. It is a classical NP-hard problem studied since more than 50 years. In this paper we adapted a biological inspired heuristic called Artificial Immune System to evolve candidate set of SN. Besides a local strategy is proposed to build the SN by considering the local information regarding the comparators and the sequences to be ordered at a determined building stage. New optimal Sorting Networks designs for input sizes from 9 to 15 are presented.

1 Introduction

Sorting Networks (SN) are tools utilized to sort a fixed-size input data. A SN is composed by a set of *comparators*. Each comparator executes an action *compare-interchange* between two elements (a, b). The element a must be not grater than b. If so, the values must be interchanged to (b, a). So, for a given input list with size n, the set of comparators conforming the SN are applied to it, then the output is a monotonically non decreasing ordered list. The SN are called *oblivious* that means that their comparisons do not depend on the input data or the previous comparisons [4, 5]. Unlike other well known sorting algorithms (like bubble sort, quicksort, heapsort, etc.), the sequence and number of comparisons are exactly the same no matter the input configuration (permutations).

Typically, the SN are graphically represented by n horizontal lines (called *buses*) representing the n input data. Besides, some vertical lines that represent comparisons between the value at its top extreme and the value at its bottom. If the value at the top is grater than the value at the bottom, these values must be swapped. The input data are placed at the left, then they go through the

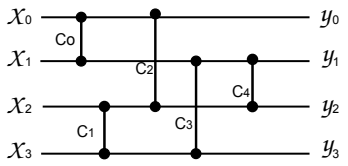


Figure 1: Example of a Sorting Network with $n = 4$ inputs.

horizontal lines executing the comparisons found. The output is obtained at the right. The data must be ascendant sorted from top to bottom.

See for example a SN for $n = 4$ inputs illustrated in Figure 1. Each input data is placed on the horizontal lines (buses) labelled as x_0, x_1, x_2, x_3. The vertical lines are the *comparators* c_0, c_1, c_2, c_3, c_4, each receiving two values. Therefore, the comparators c_0 and c_1 are executed first, then c_2 and c_3, and finally c_4. c_0 evaluates $4 > 2$, thus the values of x_0 and x_1 are swapped. c_1 evaluates $1 < 3$, then the values of x_2 and x_3 remain without change. This process continues until all the comparators are applied, so the final sorted list y_0, y_1, y_2, y_3 at the right accomplishes $y_0 \leq y_1 \leq y_2 \leq y_3$.

To design optimal SN is a hard classical problem largely studied during decades by the researchers. The optimality of the SN can be considered according to different criteria, such as parallelism or number of comparisons performed, in this work we will refer to optimal SN as those with minimal number of comparators.

Actually, only SN for small input sizes (less than 16) have been found. The most intensively studied SN are those for input size $n = 16$.

To verify if a determined SN is valid or not, it is necessary to verify if it is able to sort any input configuration (permutation), which is a NP-hard process.

The Artificial Immune System is a population-based meta-heuristic utilized to solve complex problems. Some algorithms have been developed based on mechanisms observed on the biological immune system. Specifically the Clonal Selection Algorithm was proposed to solve numerical and combinatorial optimization problems. For our goals this algorithm seems to be adequate mainly because its main variation operator is the *mutation*, that results less disruptive than others such that the *crossover*. Besides, the mutation designed in this work can be controlled to allow small variations in the possible solutions.

The Clonal Selection Algorithm is adapted to find efficient SN, that is, SN with low number of comparators. There were found new designs for SN with sizes from 9 to 15 with minimal comparators.

The remaining of this paper is organized as follows. In Section 2 some concepts about Sorting Networks and previous Related Works are presented. In Section 3 the Clonal Selection Algorithm of the Artificial Immune System is explained. Section 4 presents the proposal. In Section 5 the Results and the new SN designs are presented. Finally in Section 6 some Conclusions and Future Work are shown.

2 Sorting Networks

To better understand the SN illustrated in Figure 1 it can be interpreted as an algorithm presented next:

Require: $\{X_0, X_1, X_2, X_3\}$
1: **if** $(X_0 > X_1)$ **then**
2: swap (X_0, X_1)
3: **end if**
4: **if** $(X_2 > X_3)$ **then**
5: swap (X_2, X_3)
6: **end if**
7: **if** $(X_0 > X_2)$ **then**
8: swap (X_0, X_2);
9: **end if**
10: **if** $(X_1 > X_3)$ **then**
11: swap (X_1, X_3);
12: **end if**
13: **if** $(X_1 > X_2)$ **then**
14: swap (X_1, X_2);
15: **end if**
16: $Y_0 = X_0$;
17: $Y_1 = X_1$;
18: $Y_2 = X_2$
19: $Y_3 = X_3$;
20: **return** $\{Y_0, Y_1, Y_2, Y_3\}$;

Algorithm 1: Sorting Network Algorithm to $n = 4$ corresponding to the Figure 1.

Generally speaking, the SN efficiency can be measured according to two criteria: 1) the number of comparators needed to order n input data, and 2) the parallel execution time spent by the SN. Of course, this last criterion makes sense only if the SN can be executed on a parallel architecture where the independent comparators are executed at the same time. A set of independent comparators is called a *layer*. Therefore, SN with less number of layers are considered as fast.

If an optimal SN for input size n can be designed (i. e. with minimal number of comparators), it means that is the best manner to sort n data. Designing SN with minimal number of comparators and/or high parallelism is a classical and interesting open problem in Computer Science. Actually, nowadays only the optimal SN for input sizes $4 \leq n \leq 14$ are known.

In order to verify the SN validity, i. e. it really sorts any input configuration, it is necessary to evaluate all the $n!$ possible permutations. However, the so called *Theorem Zero-One* [4] is utilized to verify the SN validity reducing the number of evaluations. This theorem affirms that, if a SN sorts all 2^n sequences

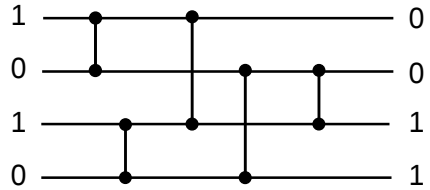


Figure 2: Example of the SN for $n = 4$ to sort the binary sequence 1010

of zeros and ones into non decreasing order, then it will sort any arbitrary sequence of n numbers.

In our example with $n = 4$, to prove the SN is valid by means of the Theorem Zero-One, we have to test if all the 2^4 binary sequences with length n from 0 to 3 are correctly sorted by the SN, that is, all the zeros appear at the top followed by all the ones. That is, we have to test each of the next the sequences $\{0000, 0001, 0010, ..., 1111\}$. In the Figure 2 the sequence $\{1010\}$ is tested by the SN; the output at the right side is correctly sorted. So, if all the 2^4 sequences can be correctly sorted then the SN is valid.

This paper proposes a scheme to design minimal SN without using predefined comparators. This approach is based on the Clonal Selection Algorithm of the Artificial Immune System. Some modifications are proposed to make it suitable for the SN optimization problem. The results show the capability of our technique to generate SN differently to those obtained by previous works.

3 The Clonal Selection Algorithm

From the information processing point of view, the biological immune system shows some interesting features such as intruder detection, memory, fault tolerance, pattern recognition, among others. The immune system functioning is very complex and not completely understood by the scientific community, however some models and theories trying to explain a specific process there exist. For example, the Clonal Selection Theory was proposed by Burnet in 1959 [1]. In the most general terms, it establishes that only the antibodies with high affinity regarding the external antigens will proliferate by cloning themselves. Then, these clones suffer some random changes named *somatic hypermutation* to improve their affinity and increase their capability to attach and eliminate the antigens.

The biological immune system has inspired a set of meta-heuristics to solve difficult problems in Computer Science and Engineering. Specifically, the Clonal Selection Algorithm has been proposed in [3] to solve mainly Computer Security and Optimization problems. The general idea of this algorithm is that a set of antibodies are the potential solutions, and the antigen is the problem at hand. A numerical value (called *affinity*) is assigned to each antibody that represents how well it solves the problem. The antibody's cloning probability is assigned

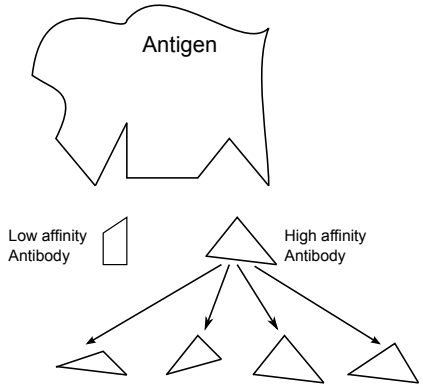


Figure 3: In the Clonal Selection Theory only the high affinity antibodies are allowed to be cloned. Then some mutations are induced to them.

proportional to its affinity value. That is, for the antibodies with high affinity values, high cloning probabilities are assigned, and viceversa. In other words, the best antibodies receive a higher quantity of clones meanwhile the worst create only a small quantity of clones.

Besides, the antibodies's hypermutation probabilities are inversely proportional to their affinity values. So, the high affinity antibodies suffer small changes through the hypermutation, and viceversa. After these cloning and hypermutation processes, a large quantity of cells will exist, then only the best of them are allowed to survive to initiate a new algorithm's iteration. This idea is illustrated in Figure 3.

4 Proposal

The general idea is to use an Artificial Immune System by means of the Clonal selection Algorithm as global strategy to minimize the number of comparators of the SN (which is defined as the *affinity* value). Besides, a local strategy is designed to select the comparators that will conform the SN. That is a process that assigns a fitness value to candidate comparators to be selected at a determined building stage.

The algorithm handles a set S of valid SN. Each element s, with $s \in S$, is a SN represented as a list of comparators. The set S is evolved by the Clonal Selection algorithm to minimize the length of each s. This is equivalent to minimize the number of comparators that conform the SN.

The Clonal Selection Algorithm for a given input size n is shown in the Algorithm 2. It generates a set S of valid SN (Line 1). The quantity of comparators of an element s is assigned as its affinity value. Next, the elements s are cloned (Line 3), and the resulting clones are mutated (Line 4). The best solutions are

selected to update S (Line 6). This process is repeated a predetermined number of times.

More details about the representation, initial population, cloning, and mutation are explained next.

Require: n = input size of the SN.
1: Generate a random initial set $\{S\}$ of valid SN (of size $|S|$);
2: Assign affinity value to each element s with $s \in S$;
3: Clone each s proportionally to their affinity value to conform the set of clones $\{K\}$;
4: Apply mutation to k, with $k \in K$, with probability inversely proportional to its affinity value;
5: Assign affinity value to each k;
6: Select the best $|S|$ solutions from $\{S \cup K\}$ to update S;
7: Repeat from Step 3 a determined number of times;
8: **return** The element of $\{S\}$ with the best affinity value

Algorithm 2: Proposed Clonal Selection Algorithm to generate minimal SN

4.1 Representation

The potential SN are represented as a list of valid comparators. Each comparator is written as a pair (x, y) where x and y are indexes to the top and bottom buses respectively, with $0 \leq x, y \leq n-1$ and $x < y$. This representation allows solutions with variable length that depends on the number of comparators conforming the SN.

For example, the SN shown in Figure 1 would be represented as:
(0,1)(2,3)(0,2)(1,3)(1,2)

4.2 Initial set S of SN

The initial set $\{S\}$ of SN (Algorithm 2 Step 1) is randomly built but restricted to contain only *valid* SN. That is, each $s \in S$ should sort[1] any binary representation of the numbers from 0 to $2^n - 1$ (See Theorem Zero-One in Section 2). For that sake the Algorithm 3 is proposed. The inputs of this algorithm are the following:

- The set $\{B\}$ conformed by the integer numbers from 0 to $2^n - 1$ represented as binary chains of length n, and
- The set $\{C\}$ of all the possible comparators. Each comparator $c \in C$ is written as a pair (x, y) where x and y are the comparator indexes to the top and bottom buses, with $0 \leq x, y \leq n-1$ and $x < y$. The complete

[1]Recall that a sorted binary chain in B means that all the zeros are at the top and the ones at the bottom.

list is this way:
$C = \{(0,1),(0,2),...,(0,n-1),(1,2),(1,3),...,(1,n-1),...,(n-2,n-1)\}$

This algorithm returns the set $\{S\}$ of size $|S|$ with the random initial population. Notice that the elements of S can be different length.

Require: n, $\{B\}$, $\{C\}$
1: **for** i=0 **to** $|S|$ **do**
2: $j = 0$;
3: Copy $\{B\}$ to $\{B'\}$
4: Randomly select an element c with $c \in C$;
5: Add c into $S_{i,j}$;
6: To apply c to each element in B' ;
7: Remove the sorted binary chains of B';
8: j++;
9: Repeat from Step 4 until B' is empty;
10: **end for**
11: **return** $\{S\}$

Algorithm 3: Generation of the random initial set $\{S\}$ (from Step 1 Algorithm 2)

4.3 Affinity Function

The affinity is defined as the number of comparators that conform a s with $s \in S$. We are trying to minimize this function.

4.4 Cloning

Cloning (Line 3 Algorithm 2) consists on producing copies of the current solutions s. Those with better affinity value will produce more copies of themselves, and viceversa. The number of clones is computed as follows,

- Sort the elements s ($s \in S$) according to their affinity value from the best to the worst.
- The number of clones # for the i-th element s_i is computed as follows:

$$\#s_i = \sum_{i=1}^{|S|} \frac{|S|}{i} \tag{1}$$

where $|S|$ is the number of elements of the set S.

4.5 Mutation

The hyper-mutation (or just mutation) is a process applied to the clones to induce some random changes to their configuration. This process is applied by choosing a point m of s and wiping out all the comparators after it, then the missing part is built again. The mutation rate should be high for clones with the worse affinities values, and viceversa. A big change is introduced if the point m is close to the begin of s (at the left), and small changes if m is closer to the end. The mutation process is shown in Algorithm 4. Two different fitness functions are proposed to select the comparator at each stage. The first (named $F1$) considers the quantity of binary chains in B that remains unsorted after the comparator is applied. The second (named $F2$) considers the total quantity of bits in B that are wrong after the comparator is applied. Each of these fitness functions is randomly selected with a probability $pF1$ (Lines 7 to 22). The inputs of this algorithm are the next:

- The clone k to be mutated
- L which is the length of the original clone k, that is the number of comparators that form k.
- The probability $pF1$ to choose the fitness function F1.
- The set $\{B\}$ conformed by the integer numbers from 0 to 2^n-1 represented as binary chains of length n
- The set $\{C\}$ of all the possible comparators

5 Experimental results

The proposed algorithm was used to design efficient SN for input sizes from $n = 9$ to $n = 15$. The Table 1 shows a comparison against the algorithms presented in Valsalam [6], and Choi and Moon [2]. The columns labelled with L show the number of comparators in the SN. The symbol P denotes the number of layers, that represents the SN parallelism. Besides, the columns with *help* indicates the number of initial comparators that the algorithm takes from Green's SN [4].

All the SN found by the proposal are new designs. They are presented in Figures 4, 5, 6, 7, 8, 9, and 10, for $n = 9$ to $n = 15$ respectively.

For the case of $n = 10$ in D. Knuth's book [4] two different SN are presented: the first with 29 comparators and 9 layers and, the second with 31 comparators and 7 layers. Our algorithm found a SN with 30 comparators and 8 layers which is a compromise solutions between these two.

6 Conclusions

The proposed Clonal Selection algorithm works as a global strategy to look for SN with minimal number of comparators. However it was necessary the

Require: k = A clone to be mutated
L = Length of k
$pF1$= Probability to select F1 as fitness function
$\{C\}$, $\{B\}$
1: Copy $\{B\}$ to $\{B'\}$
2: Select a mutation point m with $0 \leq m \leq L$
3: $j \leftarrow m$
4: Wipe out all the elements in k from m to L
5: Apply the comparators of k to B'
6: Remove the sorted elements from B'
7: **if** pF1 **then**
8: **for** i=0 **to** $|C|$ **do**
9: Copy B' to $Temp$
10: Apply the comparator c_i to the chains in $Temp$
11: Remove the sorted chains from $Temp$
12: Assign the fitness of c_i as the number of chains in $Temp$
13: **end for**
14: **else**
15: **for** i=0 **to** $|C|$ **do**
16: Copy B' to $Temp$
17: Apply the comparator c_i to the chains in $Temp$
18: Remove the sorted chains from $Temp$
19: Compute the Hamming distance between the elements of $Temp$ and their corresponding already sorted sequences
20: Assign the fitness of c_i as that Hamming distance
21: **end for**
22: **end if**
23: Select the comparator $c*$ with the minimal fitness value
24: Aggregate the comparator $c*$ in k_j
25: j++;
26: Apply the comparator $c*$ to all the elements in B'
27: Remove the sorted binary chains of B'
28: Repeat from Line 7 until B' is empty
29: $k' \leftarrow k$
30: **return** $\{k'\}$

Algorithm 4: Hypermutation process applied to a clone k (from Line 4 Algorithm 2)

n	Best Known		Choi Moon			Valsalam			SIA		
	L	P	L	*help*	P	L	*help*	P	L	*help*	P
9	**25**	**7**	-	-	-	25	-	9	**25**	-	9
10	**29**	**9**	29	-	-	29	-	10	**30**	-	**8**
11	**35**	**8**	-	-	-	35	-	11	**35**	-	11
12	**39**	**9**	39	18	-	39	-	11	**39**	-	**9**
13	**45**	**10**	-	-	-	45	-	16	**45**	22	11
14	**51**	**9**	-	-	-	51	-	16	**51**	24	13
15	**56**	**9**	-	-	-	56	-	15	**56**	29	15

Table 1: Comparison of Choi and Moon [2], Valsalam [6], and the proposed technique for SN with input sizes from $n = 9$ to $n = 15$.

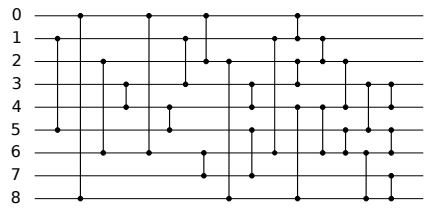


Figure 4: New SN design for $n = 9$ with 25 comparators and 9 layers.

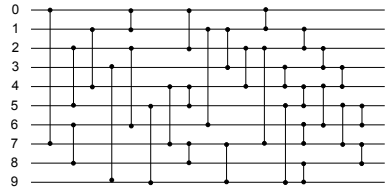


Figure 5: New SN design for $n = 10$ with 30 comparators and 8 layers.

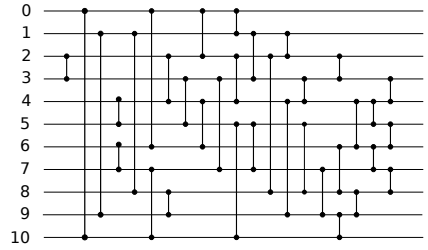


Figure 6: New SN design for $n = 11$ with 35 comparators and 11 layers.

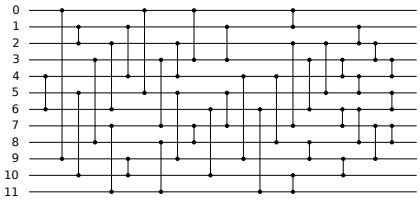


Figure 7: New SN design for $n = 12$ with 39 comparators and 9 layers.

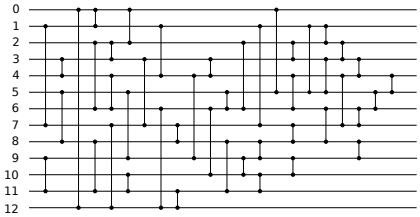


Figure 8: New SN design for $n = 13$ with 45 comparators and 11 layers.

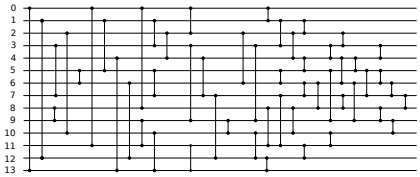


Figure 9: New SN design for $n = 14$ with 51 comparators and 13 layers.

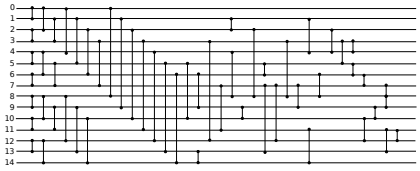


Figure 10: New SN design for $n = 15$ with 56 comparators and 15 layers.

incorporation of a local strategy to improve the results. It was designed by means of the *mutation*. It considers a fitness function related to a specific comparator at a determined building stage. The local information is related to the quantity of sequences that a determined comparator can sort at that moment and, how disordered are these sequences too. The experimental results showed the strategy is able to find new optimal SN in terms of the number of comparators and parallelism.

As Future Work is necessary to experiment with SN for larger input sizes, and to stablish the problem as a bi-objective one, where the parallelism is also considered in the objective function.

References

[1] F.M. Burnet. A modification of jerne's theory of antibody production using the concept of clonal selection. *A Cancer Journal for Clinicians*, 26:119–121, 1976.

[2] Sung-Soon Choi and Byung Ro Moon. A graph-based lamarckian-baldwinian hybrid for the sorting network problem. *IEEE Trans. Evolutionary Computation*, 9(1):105–114, 2005.

[3] Leandro N. de Castro and Fernando J. Von Zuben. Learning and optimization using the clonal selection principle. *IEEE Trans. on Evolutionary Computation*, 6(3):239–251, June 2002.

[4] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[5] Nelson Raymond J. O'Connor, Daniel G. Sorting system with n-line sorting switch. *United States Patent number 3,029,413*, 6, April 1962.

[6] Vinod K. Valsalam and Risto Miikkulainen. Utilizing symmetry and evolutionary search to minimize sorting networks. Technical Report AITR-11-09, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2011.

Bibliografía

[1] M. Ajtai, J. Komlós, and E. Szemerédi. An 0(n log n) sorting network. In *Proceedings of the fifteenth annual ACM symposium on Theory of computing*, STOC '83, pages 1–9, New York, NY, USA, 1983. ACM.

[2] Hussein M. Alnuweiri. Optimal bounded-degree vlsi networks for sorting in a constant number of rounds. In Joanne L. Martin, editor, *SC*, pages 732–739. IEEE Computer Society / ACM, 1991.

[3] Owen Astrachan. Bubble sort: An archaeological algorithmic analysis. *SIGCSE Bull.*, 35(1):1–5, January 2003.

[4] Thomas Back, David B. Fogel, and Zbigniew Michalewicz, editors. *Handbook of Evolutionary Computation*. IOP Publishing Ltd., Bristol, UK, UK, 1st edition, 1997.

[5] K. E. Batcher. Sorting networks and their applications. In *Proceedings of the April 30–May 2, 1968, spring joint computer conference*, AFIPS '68 (Spring), pages 307–314, New York, NY, USA, 1968. ACM.

[6] Daniel J. Bernstein, Tung Chou, and Peter Schwabe. Mcbits: fast constant-time code-based cryptography. In Guido Bertoni and Jean-Sébastian Coron, editors, *Cryptographic Hardware and Embedded Systems CHES 2013*, volume 8086 of *Lecture Notes in Computer Science*, pages 250–272. Springer-Verlag Berlin Heidelberg, 2013.

[7] G.E. Blelloch. Scans as primitive parallel operations. *Computers, IEEE Transactions on*, 38(11):1526–1538, Nov 1989.

[8] R. C. Bose and R. J. Nelson. A sorting problem. *J. ACM*, 9(2):282–296, April 1962.

[9] F.M. Burnet. A modification of jerne's theory of antibody production using the concept of clonal selection. *A Cancer Journal for Clinicians*, 26:119–121, 1976.

[10] D. Cantone and G. Cincotti. Quickheapsort, an efficient mix of classical sorting algorithms. In Gambosi G. Bongiovanni G.C. and Petreschi R., editors, *CIAC*, volume 1767 of *Lecture Notes in Computer Science*, pages 150–162. Springer, 2000.

[11] Sung-Soon Choi and Byung Ro Moon. A graph-based lamarckian-baldwinian hybrid for the sorting network problem. *IEEE Trans. Evolutionary Computation*, 9(1):105–114, 2005.

[12] Moon-Jung Chung and Bala Ravikumar. Bounds on the size of test sets for sorting and related networks. In *ICPP*, pages 745–751, 1987.

[13] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms (3. ed.)*. MIT Press, 2009.

[14] Nareli Cruz Cortés and Carlos A. Coello Coello. Multiobjective optimization using ideas from the clonal selection principle. In *GECCO*, pages 158–170, 2003.

[15] Nareli Cruz Cortés, Francisco Rodríguez-Henríquez, and Carlos A. Coello Coello. An artificial immune system heuristic for generating short addition chains. *IEEE Trans. Evolutionary Computation*, 12(1):1–24, 2008.

[16] Vincenzo Cutello and Giuseppe Nicosia. An immunological approach to combinatorial optimization problems. In *Proceedings of the 8th Ibero-American Conference on AI: Advances in Artificial Intelligence*, IBERAMIA 2002, pages 361–370, London, UK, UK, 2002. Springer-Verlag.

[17] Dipankar Dasgupta. *Artificial Immune Systems and Their Applications*. Springer, Verlag, Berlin, 1999.

[18] S. Dasgupta, C. Papadimitriou, and U. Vazirani. *Algorithms*. McGraw-Hill Education, 2006.

[19] Leandro N. de Castro and Fernando J. Von Zuben. Learning and optimization using the clonal selection principle. *IEEE Trans. on Evolutionary Computation*, 6(3):239–251, June 2002.

[20] Leandro Nunes de Castro and Jonathan Timmis. *Artificial immune systems - a new computational intelligence paradigm*. Springer, 2002.

[21] Ronald D. Dutton. Weak-heap sort. *BIT*, 33(3):372–381, 1993.

[22] Tao Gong and Andrew L Tuson. Binary particle swarm optimization: A forma analysis approach. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 172–172, New York, NY, USA, 2007. ACM.

[23] Lee Graham, Hassan Masum, and Franz Oppacher. Statistical analysis of heuristics for evolving sorting networks. In *Proceedings of the 2005 conference on Genetic and evolutionary computation*, GECCO '05, pages 1265–1270, New York, NY, USA, 2005. ACM.

[24] Vineet Agarwal Gupta Prabhakar and Manish Varshney. *Design and Analysis of Algorithms*. PHI Learning Private Limited, New Delhi., Asoke K. Ghosh, 2010.

[25] Paul K. Harmer, Paul D. Williams, Gregg H. Gunsch, and Gary B. Lamont. An artificial immune system architecture for computer security applications. *IEEE Transactions on Evolutionary Computation*, 6:252–280, 2002.

[26] Michael L. Harrison and James A. Foster. Co-evolving faults to improve the fault tolerance of sorting networks. In Maarten Keijzer, Una-May O'Reilly, Simon M. Lucas, Ernesto Costa, and Terence Soule, editors, *EuroGP*, volume 3003 of *Lecture Notes in Computer Science*, pages 57–66. Springer, 2004.

[27] W. Daniel Hillis. Co-evolving parasites improve simulated evolution as an optimization procedure. *Phys. D*, 42(1-3):228–234, June 1990.

[28] C. A. R. Hoare. Algorithm 64: Quicksort. *Commun. ACM*, 4.

[29] Hugues Juillé. Evolution of non-deterministic incremental algorithms as a new approach for search in state spaces. In *Third European Conference on Artificial Life*, pages 27–32, 1995.

[30] Nabil Kahale, Tom Leighton, Yuan Ma, C. Greg Plaxton, Torsten Suel, and Endre Szemerédi. Lower bounds for sorting networks. In *Proceedings of the Twenty-seventh Annual ACM Symposium on Theory of Computing*, STOC '95, pages 437–446, New York, NY, USA, 1995. ACM.

[31] Rajgopal Kannan and Sibabrata Ray. Sorting networks with applications to hierarchical optical interconnects. In *ICPP Workshops*, pages 327–, 2001.

[32] James Kennedy. Population structure and particle swarm performance. In *Proceedings of the Congress on Evolutionary Computation (CEC 2002*, pages 1671–1676. IEEE Press, 2002.

[33] James Kennedy and Russell C. Eberhart. Particle swarm optimization. In *Neural Networks, 1995. Proceedings., IEEE International Conference on*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[34] James. Kennedy and Russell C. Eberhart. A discrete binary version of the particle swarm algorithm. In *Systems, Man, and Cybernetics, 1997. Computational Cybernetics and Simulation., 1997 IEEE International Conference on*, volume 5, pages 4104–4108 vol.5, Oct 1997.

[35] Jungwon Kim, Peter J. Bentley, Uwe Aickelin, Julie Greensmith, Gianni Tedesco, and Jamie Twycross. Immune system approaches to intrusion detection - a review. *CoRR*, abs/0804.1266, 2008.

[36] Donald E. Knuth. *The Art of Computer Programming, Volume 3: (2Nd Ed.) Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

[37] U. Kocamaz. Increasing the efficiency of quicksort using a neural network based algorithm selection model. *Inf. Sci.*, 229:94–105, April 2013.

[38] John R. Koza, Stephen L. Bade, and Forrest H Bennett Iii. Evolving sorting networks using genetic programming and rapidly reconfigurable field-programmable gate arrays. In *Workshop on Evolvable Systems. International Joint Conference on Artificial Intelligence*, pages 27–32. IEEE Press, 1997.

[39] M. Kumar and D.S. Hirschberg. An efficient implementation of batcher's odd-even merge algorithm and its application in parallel sorting schemes. *IEEE Transactions on Computers*, 32(3):254–264, 1983.

[40] Ching-Jong Liao, Chao-Tang Tseng, and Pin Luarn. A discrete version of particle swarm optimization for flowshop scheduling problems. *Comput. Oper. Res.*, 34(10):3099–3111, October 2007.

[41] David J. C. MacKay. *Information Theory, Inference & Learning Algorithms*. Cambridge University Press, New York, NY, USA, 2002.

[42] Hosam M. Mahmoud. *Sorting: A Distribution Theory*. John Wiley and Sons, Inc., 2000.

[43] M. McMillan. *Data Structures and Algorithms Using C#*. Cambridge University Press, 2007.

[44] Kenneth Yun Member, Kenneth Y. Yun, Kevin W. James, Student Member, Robert H. Fairlie-cuninghame, Supratik Chakraborty, Rene L. Cruz, and Senior Member. A self-timed real-time sorting network. In *C) 2003 IEEE IEEE INFOCOM 2003*, pages 356–363, 2000.

[45] Alberto Moraglio and Julian Togelius. Geometric particle swarm optimization for the sudoku puzzle. In *Proceedings of the 9th Annual Conference on Genetic and Evolutionary Computation*, GECCO '07, pages 118–125, New York, NY, USA, 2007. ACM.

[46] Rene Mueller, Jens Teubner, and Gustavo Alonso. Sorting networks on fpgas. *The VLDB Journal*, 21(1):1–23, February 2012.

[47] David E. Muller and Franco P. Preparata. Bounds to complexities of networks for sorting and for switching. *J. ACM*, 22(2):195–201, April 1975.

[48] Nelson Raymond J. O'Connor, Daniel G. Sorting system with n-line sorting switch. *United States Patent number 3,029,413*, 6, April 1962.

[49] Ian Parberry. A computer assisted optimal depth lower bound for sorting networks with nine inputs. In *Proceedings of the 1989 ACM/IEEE conference on Supercomputing*, Supercomputing '89, pages 152–161, New York, NY, USA, 1989. ACM.

[50] Ian Parberry. Single-exception sorting networks and the computational complexity of optimal sorting network verification. *Mathematical Systems Theory*, 23(2):81–93, 1990.

[51] Ian Parberry. *Circuit Complexity and Neural Networks*. MIT Press, Cambridge, MA, USA, 1994.

[52] Christine Rüb. On batcher's merge sorts as parallel sorting algorithms. In Michel Morvan, Christoph Meinel, and Daniel Krob, editors, *STACS*, volume 1373 of *Lecture Notes in Computer Science*, pages 410–420. Springer, 1998.

[53] R. Sedgewick. Analysis of shellsort and related algorithms. In *ESA '96: Fourth Annual European Symposium on Algorithms*, pages 25–27. Springer, 1996.

[54] R. Sedgewick and K. Wayne. *Algorithms*. Pearson Education, 2011.

[55] Lukas Sekanina. Evolving constructors for infinitely growing sorting networks and medians. In *SOFSEM 2004: Theory and Practice of Computer Science*, volume 2932 of *Lecture Notes in Computer Science*, pages 314–323, Czech Republic, 24-30 January 2004. Springer-Verlag.

[56] Lukas Sekanina and Michal Bidlo. Evolutionary design of arbitrarily large sorting networks using development. *Genetic Programming and Evolvable Machines*, 6(3):319–347, September 2005. Published online: 17 August 2005.

[57] N. K. Sharma. Modular design of a large sorting network. In *ISPAN*, pages 362–368. IEEE Computer Society, 1997.

[58] Kenneth E. Batcher Sherenaz W. Al-Haj Baddar. *Designing Sorting Networks: A new Paradigm*. Springer, 2011.

[59] M. Fatih Tasgetiren, Yun-Chia Liang, Mehmet Sevkli, and Gunes Gencyilmaz. A particle swarm optimization algorithm for makespan and total flowtime minimization in the permutation flowshop sequencing problem. *European Journal of Operational Research*, 177(3):1930 – 1947, 2007.

[60] Vinod K. Valsalam and Risto Miikkulainen. Utilizing symmetry and evolutionary search to minimize sorting networks. Technical Report AITR-11-09, Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2011.

[61] Thé Van Luong, Nouredine Melab, and El-Ghazali Talbi. Gpu-based multi-start local search algorithms. In *Proceedings of the 5th international conference on*

Learning and Intelligent Optimization, LION'05, pages 321–335, Berlin, Heidelberg, 2011. Springer-Verlag.

[62] David C. Van Voorhis. Toward a lower bound for sorting networks. In Raymond E. Miller and James W. Thatcher, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 119–129. Plenum Press, New York.

[63] David C. Van Voorhis. An economical construction for sorting networks. In *Proceedings of the May 6-10, 1974, national computer conference and exposition*, AFIPS '74, pages 921–927, New York, NY, USA, 1974. ACM.

[64] Vijay V. Vazirani. *Approximation Algorithms*. Springer-Verlag New York, Inc., New York, NY, USA, 2001.

[65] Abraham Waksman. A permutation network. *J. ACM*, 15(1):159–163, January 1968.

[66] Abraham Waksman. A model of replication. *J. ACM*, 16(1):178–188, January 1969.

[67] David P. Williamson and David B. Shmoys. *The Design of Approximation Algorithms*. Cambridge University Press, New York, NY, USA, 1st edition, 2011.