CECAL
Instituto de Computación, Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay

PROYECTO DE GRADO EN INGENIERÍA EN COMPUTACIÓN

Planificación de recursos informáticos heterogéneos considerando la energía

Martín Da Fonte, Daniel Filgueiras

Febrero de 2017

Tutor del proyecto:
Sergio Nesmachnow, Universidad de la República.

Planificación de recursos informáticos heterogéneos considerando la energía
Da Fonte, Martín
Filgueiras, Daniel
Proyecto de grado en Ingeniería en Computación
CECAL
Instituto de Computación - Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay, Febrero de 2017

Planificación de recursos informáticos heterogéneos considerando la energía

Resumen

Este trabajo estudia el problema de la planificación de tareas y usuarios en sistemas heterogéneos (*heterogeneous computing scheduling problem*, HCSP) con un enfoque multiobjetivo, que busca minimizar el consumo energético de los recursos informáticos y mantener la calidad de servicio ofrecida. Se resuelven dos variantes del problema HCSP utilizando algoritmos evolutivos (AEs), una enfocada a la planificación de tareas en un cluster de computadoras, y la otra a la asignación de usuarios en un salón de informática.

El problema de planificación de tareas se modela en base al Cluster FING, de Facultad de Ingeniería, UDELAR. El objetivo de la calidad de servicio es considerado en base al makespan, el tiempo que toma ejecutar todas las tareas. El problema se resuelve implementando NSGA-II, un AE multiobjetivo explícito. La evaluación experimental se realiza sobre tres instancias de diferentes dimensiones, comparando el AE con una variante multiobjetivo de la heurística minmin. En los resultados de la evaluación el AE obtiene mejores resultados para ambos objetivos, además de brindar un conjunto amplio de soluciones ofreciendo distintos niveles de compromiso entre los dos objetivos considerados.

El problema de la asignación de usuarios modela un salón de informática de la misma facultad, y considera la calidad de servicio en base a minimizar la sobreasignación, una función que compara los requerimientos de los usuarios en un recurso y la capacidad máxima de este. Para el estudio del problema se implementa un sistema que permite recopilar información de uno de los salones de informática, reportando datos de las sesiones de los usuarios. Se implementa una heurística determinista, dos variantes de la heurística minmin, un algoritmo con política de round robin, una heurística que simula el comportamiento de los usuarios, un AE basado en NSGA-II y un algoritmo de backtracking. La evaluación experimental se realiza sobre un conjunto de 7 instancias creadas a partir de los datos recopilados del salón. Los resultados experimentales muestran que el AE obtiene las mejores soluciones en cuanto a distribución de usuarios, variedad de soluciones y mejores valores para ambos objetivos. En comparación con round robin se mejora entre un 3 % y 20 % el consumo energético y entre 15 % y 160 % la sobreasignación. Como parte de la solución se presenta una aplicación web que facilita la implantación del sistema en un entorno real.

Palabras claves: planificación, cluster, salón de informática, consumo energético, makespan, algoritmos evolutivos, calidad de servicio.

Índice general

1. Introducción **1**

2. Heurísticas y metaheurísticas para resolver problemas de optimización **4**
2.1. Problemas de optimización 4
2.1.1. Problema de optimización general de un solo objetivo 4
2.1.2. Problema de optimización multiobjetivo 5
2.1.3. Mínimo global MOP 9
2.2. Heurísticas 9
2.3. Metaheurísticas 9
2.3.1. Ejemplos y usos 10
2.4. Algoritmos evolutivos (AEs) 10
2.4.1. Conceptos básicos de algoritmos evolutivos 10
2.4.2. Esquema de algoritmo evolutivo 12
2.4.3. Optimización de un MOEA genérico 16
2.4.4. Análisis y pruebas en un MOEA 21
2.4.5. Parámetros de un MOEA 22
2.4.6. Evaluación de eficacia 24

3. Computación heterogénea y trabajos relacionados **26**
3.1. Sistema Computacional Heterogéneo 26
3.2. Consumo energético en computadoras 27
3.3. Planificación de recursos computacionales heterogéneos 30
3.3.1. Planificación de recursos computacionales 31
3.4. Trabajos relacionados 31
3.4.1. Estrategias generales para disminuir el consumo energético 31
3.4.2. Problema de la asignación de tareas en cluster o grid 32
3.4.3. Estudios de eficiencia energética en el ámbito empresarial 35
3.4.4. Resumen 36

4. Planificación de tareas en un cluster heterogéneo considerando el consumo energético **38**
4.1. Cluster heterogéneo 38
4.2. Presentación del problema 38
4.2.1. Formulación matemática del problema 39
4.3. Implementación 40
4.3.1. Biblioteca de desarrollo 41
4.3.2. Codificación de las soluciones 41

4.3.3. Metaheurística utilizada 42
4.4. Evaluación experimental 44
4.4.1. Definición de las instancias 45
4.4.2. Metodología de la evaluación experimental 46
4.4.3. Comparación de diferentes métodos de inicilización 47
4.4.4. Ajuste paramétrico 47
4.4.5. Comparación con heurística MinMin 48
4.5. Resultados 50
4.6. Resumen 53

5. Asignación de usuarios en un salón de informática 55
5.1. Presentación del problema 55
5.2. Modelo del problema 56
5.2.1. Formulación matemática del problema 57
5.3. Diseño de los algoritmos propuestos 61
5.4. Algoritmo DetParam 62
5.5. Algoritmo minmin 66
5.6. Backtracking 67
5.7. Algoritmo evolutivo 70
5.8. Implementación de la solución 71
5.8.1. Componentes de la solución 72
5.8.2. Sistema para la recopilación de datos 73
5.9. Evaluación experimental 76
5.9.1. Análisis de los datos recopilados 76
5.9.2. Definición de las instancias 78
5.10. Resultados 83
5.11. Resumen 88

6. Conclusiones y trabajo futuro 89
6.1. Conclusiones 89
6.2. Trabajo futuro 90

Bibliografía 92

Apéndices 98

A. Interfaz Sistema 100
A.1. Pseudocódigo de la interfaz del Sistema 100
A.2. Detalle de implementación del Sistema 102

B. Algoritmo Determinista 107
B.1. Detalle de implementación del algoritmo determinista 107

C. Aplicación web 114
C.1. Casos de uso 114

D. Estudio de comandos remotos **120**
D.1. Pruebas de librerías para Windows y Linux 120
D.1.1. Librería Paramiko 120
D.1.2. Librería Plink 121

Índice de figuras

2.1. Representación del frente de Pareto 8
2.2. Ejemplo de cruzamiento basado en Punto-Simple 11
2.3. Dos mecanismos de cálculo de ranking de dominancia, agrupando individuos con igual valor 16

4.1. Diagrama de codificación de la solución 42
4.2. Frente de Pareto calculado por el AE 51
4.3. Comparación de makespan para dos soluciones del AE 52
4.4. Comparación de la energía para dos soluciones del AE 53

5.1. Distribución de las sesiones por recursos del salón 77
5.2. Porcentaje de uso de los recursos del salón 78
5.3. Consumo energético por tipo de recurso 81
5.4. Frentes de Pareto calculados del problema HCSP 85
5.5. Distribución de usuarios para cada algoritmo 86

C.1. Pantalla inicial 115
C.2. Utilización de memoria y CPU 116
C.3. Máquinas disponibles 117
C.4. Formulario de ingreso 118
C.5. Resultado del formulario 118
C.6. Subir archivo 119

Índice de cuadros

4.1. Resultado de la comparación de la inicialización del AE 47
4.2. Resultados del estudio paramétrico del AE 49
4.3. Resultados de la comparación de AE y MinMin 51

5.1. Detalle de las isntancias definidas 80
5.2. Resultados del estudio paramétrico del AE 82
5.3. Resultados del estudio comparativo entre todos los algoritmos 87

Índice de algoritmos

1. Esquema Genérico de un Algoritmo Evolutivo 12
2. Tareas de un MOEA general 13
3. Esquema de un MOEA optimizado 18
4. NSGA 19
5. NSGA-II 20
6. Algoritmo de inicialización de la población 43
7. Pseudocódigo de HCTCrossover 44
8. Algoritmo MinMin 50
9. Algoritmo para el calculo de los objetivos del problema 60
10. Principales operaciones del algoritmo DetParam 65
11. Algoritmo minmin 67
12. Backtracking 69
13. Pseudocódigo del algoritmo para generar sesiones de usuario 79
14. Interfaz Sistema 101

Capítulo 1

Introducción

La computación heterogénea (*heterogeneous computing*, HC) es un modelo basado en el uso de múltiples recursos computacionales que varían en potencia, arquitectura, componentes e incluso en su propósito. Los sistemas de HC surgen como una alternativa a los sistemas homogéneos normalmente construidos a medida y con un costo mayor. Algunos sistemas HC son diseñados con el objetivo de contar con recursos especializados para diferentes tareas, mientras que otros resultan de un proceso de actualización parcial de una infraestructura, en el que varias generaciones de recursos trabajan en conjunto. Dependiendo de la agrupación lógica de un sistema HC, este puede ser catalogado como un cluster, un grid, cloud computing o ser simplemente un conjunto de recursos interconectados que trabajan de forma independiente (Foster and Kesselman, 2003; Zhao et al., 2008).

Estos sistemas compuestos por una gran cantidad de recursos informáticos, traen aparejado un gran consumo energético que aumenta a medida que crecen en tamaño y potencia (Casanova et al., 2012). El creciente costo de la energía consumida por data centers y clusters de computadoras, junto con el impacto ambiental de estos elevados niveles de consumo, ha despertado interés en el desarrollo de técnicas que disminuyan el consumo de energía de estos sistemas (Beloglazov et al., 2011).

Se han desarrollado diferentes aproximaciones al problema, algunas enfocándose en reducir el consumo energético de cada recurso de forma individual, y otras que incorporan técnicas para reducir el consumo energético en la administración del sistema. El enfoque principal de nuestro trabajo es encontrar la planificación óptima de tareas y usuarios en un sistema heterogéneo (*Heterogeneous Computing Scheduling Problem*, HCSP).

Para la aplicación práctica de un algoritmo de planificación, el tiempo de ejecución del mismo es importante ya que normalmente los usuarios esperan a que se complete la asignación para poder hacer uso del sistema. Dado que los problemas de asignación tradicional son NP-difíciles (Ullman, 1975), en la medida que las instancias del problema se hacen mayores es imposible mantener tiempos de ejecución reducidos utilizando métodos exactos. Las meta-heurísticas, en particular los algoritmos evolutivos (AEs), proveen de un método aproximado que permite obtener soluciones de buena calidad en tiempos de ejecución razonables (Nesmachnow, 2015).

En el trabajo que se presenta en este informe, se utilizan una combinación de técnicas para resolver el problema de HCSP: a nivel de los recursos individuales modificar el voltaje y frecuencia de funcionamiento de los circuitos de los recursos informáticos para

alterar su consumo y rendimiento dinámicamente. A nivel del sistema, se mejora la asignación de tareas y usuarios a los recursos, con el fin de lograr que estos trabajen a un nivel óptimo permitiendo que un mayor número de recursos puedan entrar en estado de reposo o sean apagados, disminuyendo su consumo.

El trabajo aquí presentado se enfoca en dos tipos de sistemas de HC, un cluster de computadoras que recibe tareas de larga duración, y un salón de informática donde los usuarios pueden ejecutar sus tareas de forma presencial o remota. El cluster de computadoras fue modelado en base al Cluster FING, Facultad de Ingeniería, UDELAR (Nesmachnow, 2010), mientras que el salón de informática modela uno de los salones de la misma institución. En ambos sistemas el objetivo de nuestro trabajo fue minimizar el consumo energético manteniendo a la vez la calidad de servicio ofrecida. En el cluster la calidad de servicio se midió como el tiempo total que le lleva al sistema completar la ejecución de las tareas. Mientras que en el salón de informática, la calidad de servicio fue medida como la sobreasignación, una función que compara los requerimientos de los usuarios de un recurso y la capacidad máxima de este, evaluando el nivel en el que no se puede cumplir con los requerimientos de estos usuarios.

Las principales contribuciones de este trabajo son:

1. El análisis de la literatura relacionado al problema de planificación de recursos heterogéneos, específicamente aquellos que tienen como objetivo minimizar el consumo energético.

2. La definición y formulación matemática de dos variantes del problema de planificación de recursos heterogéneos, una orientada a cluster de computadoras y la otra a un salón de informática.

3. La implementación de AEs para resolver ambas variantes del problema.

4. La recopilación de datos de un salón de informática de la Facultad de Ingeniería, UDELAR, brindando información sobre el uso real de la plataforma por parte de los usuarios.

5. La creación de un conjunto de instancias para la evaluación experimental, tres instancias sintéticas para el problema de cluster, y siete instancias basadas en datos reales para el problema del salón de informática.

6. La evaluación experimental de los AEs propuestos, comparándolos contra heurísticas ávidas diseñadas en base a las ideas presentadas en trabajos de la literatura relacionada, y en el caso del salón de informática también con una estrategia intuitiva.

7. El diseño de una plataforma web que permite la implementación de un algoritmo de planificación de usuarios en un salón de informática.

El resto del documento se estructura de la siguiente manera: en el Capítulo 2 se presentan los problemas de optimización y las heurísticas y metaheurísticas como herramientas para su resolución, haciendo un énfasis especial en los AEs, presentando su estructura, sus variantes, sus operadores y métricas particulares. En el Capítulo 3 se introduce el concepto de computación heterogénea, el problema de la planificación de recursos heterogéneos y una reseña de los principales trabajos relacionados. El trabajo

realizado para resolver el problema de la planificación de tareas en un cluster de computadoras, junto con la evaluación experimental de los algoritmos implementados para su resolución y la discusión de resultados, se presenta en el Capítulo 4. En el Capítulo 5 se presenta el trabajo realizado sobre los salones de informática, incluyendo la herramienta para la recopilación de información, el análisis de los datos recabados, un detalle de los algoritmos implementados y la discusión de los resultados. Por último, las conclusiones de este trabajo y las principales líneas de trabajo futuro se presentan en el Capítulo 6.

Capítulo 2

Heurísticas y metaheurísticas para resolver problemas de optimización

En este capítulo se presentan las heurísticas y metaheurísticas como forma de resolver problemas de optimización en sus variantes de un objetivo y multiobjetivo. Se introduce la terminología de Pareto aplicada a los problemas multiobjetivo y se describe en profundidad los algoritmos evolutivos, un tipo de metaheurísticas, y los principales operadores evolutivos utilizados por estos algoritmos, incluyendo los utilizados en nuestro trabajo. Finalmente se detallan las pruebas utilizadas para evaluar la eficacia de un algoritmo evolutivo.

2.1. Problemas de optimización

En esta sección se presenta la definición formal de los problemas de optimización, tanto en su versión de un solo objetivo como en su versión multiobjetivo.

2.1.1. Problema de optimización general de un solo objetivo

Un problema de optimización general de un solo objetivo se define como la minimización (ó maximización) de $f(x)$ sujeto a $g_i(x) \leq 0, i = 1 \cdots m$, y $h_j(x) = 0$, $j = 1 \cdots p$ con $x \in \Omega$. Una solución minimiza (ó maximiza) el escalar $f(x)$ donde x es un vector variable de decisión n-dimensional $x = (x_1 \cdots x_n)$ de algún universo Ω. (Coello et al., 2007)

Observe que $g_i(x) \leq 0$ y $h_j(x) = 0$ representan restricciones que deben cumplirse además de la optimización (minimizar o maximizar) $f(x)$. Ω contiene todas las posibles x que se pueden utilizar para satisfacer una evaluación de $f(x)$ y sus restricciones. Por supuesto, x puede ser un vector de variables continuas o discretas, así como f ser continua o discreta.

Los problemas de optimización de mínimo global son un caso particular de los problemas de optimización, que buscan determinar el óptimo global de una función, pudiendo ser en ocasiones más de uno.

Problema de optimización mínimo global simple objetivo: Dada una función $f : \omega \subseteq \mathbb{R}^{\ltimes} \rightarrow \mathbb{R}, \Omega = \phi$, para $x \in \Omega$, el valor $f^* \ f(x^*) > -\infty$ es llamado un mínimo global si y solo si $\forall x \in \Omega : f(x^*) \leq f(x)$. Por definición x^* es la solución mínima global, f la función objetivo y el conjunto Ω es la región factible de x. El objetivo de determinar la solución mínima global (s) es llamado el problema de optimización global para un objetivo.

2.1.2. Problema de optimización multiobjetivo

Según Osyczka (1985) el problema de optimización multiobjetivo (*Multiobjective optimization problem*, MOP) puede ser definido como: el problema de encontrar un vector de variables de decisión que cumple las restricciones impuestas, y además optimiza una función vectorial cuyos elementos representan las funciones objetivo. Estas funciones describen matemáticamente los criterios de rendimiento elegido. Por lo tanto, optimizar esta función vectorial significa encontrar valores para todas las variables que sean aceptables por el tomador de decisiones (*decision maker*, DM).

Aunque los problemas de optimización de un solo objetivo pueden tener una única solución óptima, los MOPs suelen presentar un conjunto posiblemente incontable de soluciones, que al ser evaluado, produce vectores cuyos componentes representan las concesiones en el espacio objetivo. Un DM debe escoger una o varias soluciones aceptables mediante la selección de uno o más de estos vectores.

Las *variables de decisión* son las cantidades numéricas que se eligen para un valor en un problema de optimización. Se indican como x_j, con $j = 1 \cdots n$. El vector x de n variables de decisión se representa como: $x = \lceil x_1 \cdots x_n \rceil$. Otra forma de escribirlo es $x = [x_1 \cdots x_n]^T$, donde T indica la transposición del vector columna al vector fila.

Una componente importante de los problemas de optimización son las limitaciones, comúnmente llamadas restricciones, que son impuestas a la solución. Estas restricciones pueden ser físicas, de tiempo, de recursos entre otras, y son particulares de cada problema y su ambiente, así como de los recursos que posee. Para qué una solución sea factible estas restricciones se deben cumplir; se pueden ver como dependencias entre las variables de decisión y las constantes o parámetros del problema.

La forma de escribir estas restricciones son como desigualdades matemáticas:

$$g_i(x) \leq 0,\ i = 1 \cdots m$$

o como igualdades:

$$h_j(x) = 0,\ j = 1 \cdots p$$

Notar que p es el número de restricciones de igualdad y debe ser menor que n, el número de variables de decisión. Cuando $p \geq n$ se dice que el problema es *overconstrained*, ya que no existen grados de libertad $(n - p)$ para optimizar las ecuaciones, hay más restricciones que incógnitas. Además, las restricciones pueden ser explícitas mediante expresiones algebraicas o implícitas donde el algoritmo para calcular $g_i(x)$ para cualquier vector dado x debe ser conocido.

Para saber que tan buena es una solución se define una función computable de las variables de decisión llamada función objetivo. Por definición, en los problemas multiobjetivo las funciones a optimizar están en conflicto entre si, es por esto que la búsqueda

del óptimo global de un MOP general es un problema NP-completo (Bäck, 1996). Las funciones objetivos medidas en las mismas unidades se llaman **Conmensurables**, y las que están en diferentes unidades, **No-Conmensurables**.

En investigación de operaciones es común diferenciar entre los *atributos*, *criterios*, *objetivos* y *metas* (MacCrimmon, 1973). Los *atributos* definen aspectos, características o propiedades de distintas alternativas de un individuo. Los *criterios* denotan medidas de evaluación, escalas o dimensiones que se utilizan para comparar mejores o peores valores. Los *objetivos* suelen verse de igual manera, pero también pueden denotar niveles deseados para cumplir un cierto criterio.

También suele utilizarse la noción de objetivo a largo plazo para designar niveles potencialmente alcanzables, y los restantes objetivos para designar las metas inalcanzables. Los vectores que resultan de la evaluación de la soluciones de un MOP que se trazan en el espacio de coordenadas son denotados por el espacio de las funciones objetivo.

Por esto, las funciones objetivo se designan:

$$f_1(x), f_2(x), ..., f_k(x)$$

donde k es el número de funciones objetivo en el MOP. Las funciones objetivo forman una función vectorial $f(x)$ definida como:

$$f(x) = [f_1(x), f_2(x)...f_k(x)]$$

Para resolver mediante un algoritmo de búsqueda es conveniente contar con una definición matemática del MOP para un mejor entendimiento del problema, la formulación de un MOP extiende la formulación de los problemas de optimización con un solo objetivo. Para determinar el conjunto de soluciones al MOP se utiliza la teoría de optimización de Pareto. (Ehrgott, 2006)

Tener en cuenta que los problemas multiobjetivo requieren un tomador de decisiones para elegir entre los valores x_i^*. La selección es esencialmente una solución de compromiso de una solución completa x sobre otra en el espacio multiobjetivo.

El objetivo real de un MOP es optimizar las k funciones objetivos simultáneamente. Esta optimización es independiente para cada función objetivo y puede ser tanto la minimización como una maximización. Según Van Veldhuizen (1999), Schlottmann and Seese (2004) y Coello (2002) se puede definir formalmente un MOP mínimo global (ó máximo) como:

MOP general: Minimizar (o maximizar) $F(x) = (f_1(x), ..., f_k(x))$ s.a. $g_i(x) \leq 0$, $i = \{1, ..., M\}$, y $h_j(x) = 0$, $j = \{1, ..., P\}$, y $x \in \Omega$. La solución de un MOP minimiza (ó maximiza) los componentes de un vector (x) donde x es un vector variable de decisión $n-dimensional$ con $x = (x_1, ..., x_n)$ de un universo Ω. Se observa que $g_i(x) \leq 0$ y $h_j(x) = 0$ representan restricciones que se deben cumplir y reducir al mínimo (ó maximizar) $F(x)$, y Ω contiene todas las posibles x que se pueden utilizar para satisfacer una evaluación de $F(x)$.

Por lo tanto, un MOP consiste en k objetivos que se representan en las k funciones objetivo, $m + p$ restricciones en las funciones objetivo y n variables de decisión. Las k funciones objetivo pueden ser lineales o no lineales y continuas o discretas. La función

de evaluación $F : \Omega \to \bigwedge$ es un mapeo entre el vector variable de decisión $(x = x_1, ...x_n)$ y los vectores de salida $(Y = a_1, ..., a_k)$. Notar que el vector de variables de decisión x_i también puede ser continuo o discreto (Van Veldhuizen, 1999). Existe un vector de variables que es el óptimo, llamado vector ideal. Según Coello et al. (2007) se puede definir como:

Vector ideal: Sea $x^0(i) = [x_1^0(i), x_2^0(i), ..., x_n^0(i)]^T$ un vector de variables que optimiza la i-ésima función objetivo $f_i(x)$. Osea, el vector $x_0(i) \in \Omega$ es tal que $f_i(x^0(i)) = opt f_i(x)$, $x \in \Omega$. El vector $f^0 = [f_1^0, f_2^0, ..., f_k^0]^T$, donde f_i^0 denota el óptimo de la i-ésima función, es el ideal para un MOP. El punto en $\mathbb{R}^{\mathbb{N}}$ que determinó este vector es la solución ideal (es en realidad utópico) y por consiguiente se llama el vector ideal. Se observa entonces que el vector ideal contiene el óptimo para cada uno por separados de los objetivos en el mismo punto $\mathbb{R}^{\mathbb{N}}$.

Optimalidad, dominancia y conjunto óptimo de Pareto

Según Van Veldhuizen (1999), una solución $x \in \Omega$ se dice que es *óptimo de Pareto* con respecto a Ω si y sólo si no existe un $x' \in \Omega$ para el que $v = F(x') = (f_1(x'), ..., f_K(x'))$ domine a $u = F(x) = (f_1(x), ..., f_k(x))$. Se considera el óptimo de Pareto respecto al total del espacio variable de decisión.

Si se supone un problema de minimización, se puede decir que x^* es un óptimo de Pareto si no existe un vector factible x que disminuya algún criterio sin aumentar otro al mismo tiempo, y viceversa para un problema de maximización.

Sumemos otras dos definiciones que son parte importante en la resolución de un MOP:

Dominancia de Pareto: Un vector $u = (u_1, ..., u_k)$ se dice que domina a otro vector $v = (v_1, ..., v_k)$ (denotado por $u \preceq v$) si y solo si u es parcialmente menor que v, i.e. $\forall i \in 1, ..., k, u_i \leq v_i \wedge \exists i \in 1, ..., k : u_i < v_i$.. (Schlottmann and Seese, 2004)

Conjunto óptimo de Pareto: Dado un MOP y $F(x)$, el conjunto óptimo de Pareto (P^*) está definido como:

$$P^* := \{x \in \Omega \mid \neg \exists x' \in \Omega \ \wedge \ F(x') \preceq F(x)\}$$

Las soluciones óptimas de Pareto son aquellas soluciones dentro del espacio de búsqueda cuyos componentes del vector objetivo no pueden ser mejorados todos simultáneamente, entonces sus correspondientes vectores se dicen no-dominados. Si se selecciona un vector a partir de este conjunto de vectores (FP^*, el frente de Pareto), implícitamente indica que contiene soluciones del óptimo de Pareto factibles.

Estos vectores de solución pueden no tener ninguna relación aparte de su pertenencia en el conjunto óptimo de Pareto. Juntos forman el conjunto de todas las soluciones cuyos vectores asociados son no-dominados.

Frente de Pareto: Dado un MOP, $F(x)$, y un conjunto P^* óptimo de Pareto, Coello (2002) define el frente de Pareto FP^* como:

$$FP^* := \{u = F(x) | x \in P^*\}$$

Siendo P^* un subconjunto del conjunto solución, cuando los vectores no-dominados se trazan en el espacio objetivo, conjuntamente se denominan el frente de Pareto.

Sus vectores objetivo evaluados forman el FP^*, donde cada uno de ellos es no-dominado respecto al resto de los vectores generados mediante la evaluación de todas las posibles soluciones en Ω.

Generar el frente de Pareto no es sencillo, un procedimiento normal y muy utilizado es crearlo a partir del cálculo de muchos puntos en Ω y su correspondiente evaluación en la función. Cuando se tienen suficientes puntos se está en condiciones de determinar los no-dominados y generar el frente de Pareto del problema. En la figura 2.1 se observa un ejemplo de frente de Pareto en la minimización de dos objetivos: costo y eficiencia.

Los MOPs a diferencia de los problemas de un solo objetivo, que usualmente presentan una única solución óptima, tienen un conjunto de soluciones en el frente de Pareto. Cada solución asociada a un punto en el FP^* es un vector donde sus componentes representan niveles de compromiso en el espacio de las soluciones. (Coello, 2002)

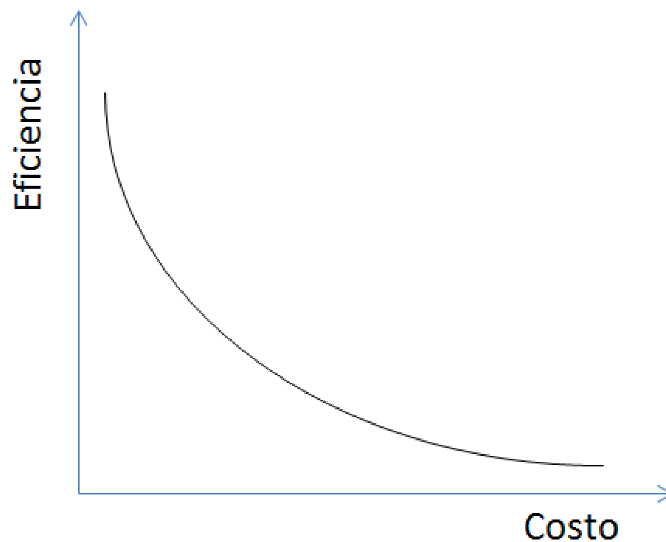


Figura 2.1: Problema con dos funciones objetivo: minimización de costo y eficiencia. El frente de Pareto está delimitado por una línea curva.

Optimalidad de Pareto

Según Coello et al. (2007) en la optimalidad de Pareto se distinguen dos tipos, la optimalidad estricta y la optimalidad débil.

Un punto $x^* \in \Omega$ es un óptimo débil de Pareto si no existe $x \in \Omega$ tal que $f_i(x) < f_i(x^*)$, para $i = 1, ..., k$.

Un punto $x^* \in \Omega$ es un óptimo estricto de Pareto si no existe $x \in \Omega, x = x^*$ tal que $f_i(x) \leq f_i(x^*)$, para $i = 1, ..., k$. Coello et al. (2007)

2.1.3. Mínimo global MOP

Una solución es óptima de Pareto si al ser evaluada no existe un vector que mejore su rendimiento sin afectar otro simultáneamente. El frente de Pareto FP^* determinado mediante la evaluación de P^* es fijo por el MOP definido, donde P^* representa la mejor solución disponible y permite la definición de un óptimo global del MOP. (Coello et al., 2007)

Un mínimo global se define entonces como: Dada una función $f : \Omega \subseteq R^n \rightarrow R^k, \Omega = \phi, k \geq 2$, para $x \in \Omega$ el conjunto $FP^*, f(x_i^*) > (\ -\infty, ..., +\infty)$ se llama mínimo global si y sólo si

$$\forall x \in \Omega : f(x_i^*) < f(x)$$

Entonces, x_i^* con $i = 1, ..., N$, es el conjunto solución mínimo global (P^*) donde f es la función objetivo múltiple y Ω la región factible. El problema de determinar el conjunto solución mínimo global se denomina problema de optimización global MOP.

2.2. Heurísticas

Las heurísticas son criterios, métodos o principios para elegir de entre un conjunto de alternativas la que permite acercarse más a un objetivo establecido. Las heurísticas como algoritmos se basan en buscar o descubrir soluciones, eligiendo en cada paso la opción que parezca más prometedora basada en información limitada disponible sobre el problema. Un compromiso habitual en las heurísticas es elegir un equilibrio entre la simplicidad de los criterios a utilizar y la efectividad de los mismos para encontrar buenas soluciones (Pearl, 1984).

Estas técnicas suelen utilizarse para encontrar soluciones más rápidamente cuando métodos tradicionales demoran mucho tiempo, o soluciones aproximadas cuándo los métodos tradicionales fallan en encontrar soluciones exactas.

2.3. Metaheurísticas

Las metaheurísticas son estrategias genéricas que definen los algoritmos para el diseño de un conjunto de técnicas capaces de encontrar de manera eficiente y precisa soluciones aproximadas para los problemas de búsqueda, problemas de optimización y problemas de aprendizaje automático (Glover, 1986). Pueden ser utilizadas para una amplia variedad de problemas, normalmente utilizando una instancia de un esquema genérico de algoritmo que se adapta con facilidad para resolver el problema específico.

Las metaheurísticas, meta significa "alto nivel", generalmente tienen mejor desempeño que las heurísticas y se utilizan para desarrollos más complejos. (Gallego et al., 2008)

Las metaheurísticas se basan principalmente en dos componentes: explotación y exploración. La exploración significa generar soluciones diversas, producto de explorar el espacio de búsqueda a escala global, y la explotación significa concentrarse en una búsqueda local y explotar la información de la solución parcial encontrada en esa región. Esta combinación suele significar que las soluciones convergerán hacia el óptimo manteniendo diversidad en su conjunto (Nesmachnow, 2014; Yang, 2010a,b).

2.3.1. Ejemplos y usos

Según Osman and Kelly (1996), en muchas ocasiones se utilizan las metaheurísticas como técnicas para resolver problemas complejos cuando se busca un mejor rendimiento, debido a la buena precisión numérica de las soluciones y la eficiencia computacional del método.

Las metaheurísticas incorporan conceptos de distintos campos de estudio tales como la genética, la biología, la inteligencia artificial, matemáticas y física, neuro-ciencia, entre otros. (Cordon et al., 2002)

Algunos ejemplos de metaheurísticas incluyen búsqueda local iterativa, búsqueda variable de vecinos, GRASP (greedy randomized adaptive search procedure) (Feo and Resende, 1995; Festa and Resende, 2002) y algoritmos evolutivos.

Las técnicas metaheurísticas son algoritmos aproximados no determinísticos, que intentan evitar los óptimos locales y determinan los criterios utilizados en la búsqueda en la medida que el método avanza, pudiendo cambiar el balance entre exploración y explotación (Blum and Roli, 2003). Inician con una solución o un conjunto de ellas, típicamente alejada del óptimo, y van obteniendo soluciones cercanas, de las cuales elige una o varias que cumplan algún criterio y con las que comienzan una nueva búsqueda. El método se detiene cuando se cumple alguna condición predefinida. (Nesmachnow, 2014)

2.4. Algoritmos evolutivos (AEs)

Según Coello et al. (2007) los problemas multiobjetivo surgen naturalmente cada vez más en diversas disciplinas y presentan un desafío constante para las investigadores. El área de investigación de operaciones ha sido quien ha desarrollado diversas técnicas para su estudio, pero debido a la creciente complejidad de sus soluciones han surgido enfoques alternativos que también abordan estos problemas.

Los algoritmos evolutivos multi objetivo (*Multiobjective evolutionary algorithms*, MOEA) suelen ser elegidos para resolver este tipo de problemas debido a su naturaleza basada en poblaciones que permite generar varios individuos candidatos a soluciones en una sola ejecución.

Los principales objetivos de un MOEA son:

- Preservar puntos no dominados en el espacio objetivo y los puntos de solución asociados en el espacio de decisión.
- Avanzar en el algoritmo hacia el frente de Pareto en el espacio de la función objetivo.
- Mantener la diversidad de puntos en el frente de Pareto (espacio fenotipo) y/o de soluciones óptimas de Pareto (espacio de decisión: espacio genotipo).

2.4.1. Conceptos básicos de algoritmos evolutivos

En algoritmos evolutivos cada candidato a solución es representado como un individuo en la población. Un individuo es representado como una cadena que corresponde a un genotipo biológico, donde este genotipo define un organismo individual decodificado en un fenotipo. Un genotipo está compuesto por uno o más cromosomas donde cada uno de ellos contiene genes separados (alelos) que toman valores de un alfabeto genético. Al

conjunto de cromosomas se le denomina población.

Para ir creando una población más apta en cada generación, existen operadores que trabajan sobre esta población y se denominan operadores evolutivos (*Evolutionary Operators*, EVOPs). Los principales EVOPs asociados a los AE son los de mutación, cruzamiento y selección.

En la Figura 2.2 se muestra un ejemplo de recombinación punto-simple (cruzamiento) que actúa sobre dos cadenas binarias. Se toma cada padre, se divide y cruza con una parte de otro. Los individuos que son superiores a la media se eligen (selección) con más frecuencia que los inferiores, para ser parte de la próxima generación. Es por esto que la selección contribuye eligiendo las cadenas con mayor aptitud y probabilidad de aportar más descendientes a la siguiente generación.

Existen diversas variaciones de estos EVOPs y todos los AEs utilizan algún subgrupo de estos. La elección de este subgrupo de EVOPs depende de las restricciones del problema de cada AE y afecta la estructura cromosómica y los alelos (Bäck, 1996).

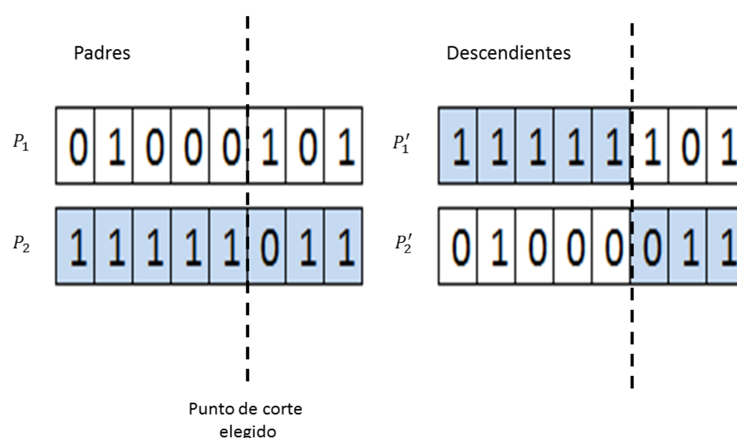


Figura 2.2: Ejemplo de cruzamiento basado en Punto-Simple

Un AE necesita de una función objetivo y una función de *fitness* o aptitud. La función objetivo opera en el dominio del problema y define la condición de optimalidad del AE, mientras que la función de fitness opera en el dominio del algoritmo, y mide qué tanto satisface una solución particular esa condición y le asigna un valor real correspondiente a esa solución.

Existen diversas técnicas de selección, las más utilizadas son por clasificación y por torneo. La selección por clasificación asigna probabilidades de selección solo en el rango de un individuo, ignorando los valores absolutos de fitness. La selección por torneo toma un número de individuos de la población al azar, y elige el mejor individuo que formará parte de la próxima generación. Los torneos binarios (q = 2) son los más implementados en AE. (Bäck, 1996; Goldberg and Deb, 1991)

2.4.2. Esquema de algoritmo evolutivo

Según Merkle and Lamont (1997) y Bäck (1996) un Algoritmo Evolutivo se define formalmente como: Sea I un conjunto no vacío (el espacio individual), y

- $\{\mu^{(i)}_{i\in\mathbb{N}}\}$ una subsecuencia en $\mathbb{Z}^+$ (tamaño de la población de los padres)
- $\{\mu^{*(i)}_{i\in\mathbb{N}}\}$ una subsecuencia en $\mathbb{Z}^+$ (tamaño de la población de los hijos)
- $\phi : I \to \mathbb{R}$ una función de fitness y $\iota : \bigcup_{i=1}^{\infty} (I^{\mu})^{(i)} \to \{true, false\}$ el criterio de terminación
- r una subsecuencia $\{r^{(i)}\}$ del operador de cruzamiento $r^{(i)} : X_r^{(i)} \to \tau(\Omega_r^{(i)})$, $\chi \in \{true, false\}$, $\tau(I^{\mu^{(t)}})$ y $I^{\mu^{*(t)}}$
- m una subsecuencia $\{m^{(i)}\}$ del operador de mutación $m^{(i)} : X_m^{(i)} \to \tau(\Omega_m^{(i)})$, $tau(I^{\mu^{*(t)}})$ y $I^{\mu^{*(t)}}$
- s una subsecuencia $\{s(i)\}$ del operador de selección $s(i) : X_s^{(i)} \times \tau(I, \mathbb{R}) \to \tau(\Omega_s^{(i)}, \tau(I^{\mu^{*(t)}+\chi\mu^{(t)}})$ y $I^{\mu^{(t+1)}}$
- $\Theta_r^{(i)} \in X_r^{(i)}$ (los parámetros de cruzamiento)
- $\Theta_m^{(i)} \in X_m^{(i)}$ (los parámetros de mutación)
- $\Theta_s^{(i)} \in X_s^{(i)}$ (los parámetros de selección)

Para facilitar la comprensión se muestra el pseudocódigo correspondiente a un AE en el Algoritmo 1.

Algoritmo 1 Esquema Genérico de un Algoritmo Evolutivo

```
t := 0
inicializar P(0) := {a_1(0), ..., a_u(0)} ∈ I^{μ(0)}            ▷ inicializa la población
while (ι({P(0), ..., P(t)}) ≠ true) do
    P*(t) := r^(t)_{θ_r^(t)}(P(t))            ▷ cruzamiento entre dos individuos seleccionados
    P**(t) := m^(t)_{θ_r^(t)}(P*(t))                    ▷ mutación de los hijos generados

    if χ then
      P(t+1) := s^(t)_{(θ_s^(t),φ)}(P**(t))
    else
      P(t+1) := s^(t)_{(θ_s^(t),φ)}(P**(t) ∪ P(t)) ▷ selección de individuos para nueva población
    end if
    t := t + 1;                                   ▷ avanzar a siguiente generación
end while
```

Técnicas que utilizan los AEs

Los algoritmos evolutivos resuelven en forma precisa los problemas de optimización, sobre todo los problemas multiobjetivo debido a que este algoritmo es capaz de trabajar simultáneamente con un conjunto factible de soluciones. Trabajar de esta manera le permite al algoritmo encontrar con una alta probabilidad distintos miembros del óptimo de Pareto en una sola ejecución, en vez de tener que realizar varias ejecuciones independientes como en las técnicas tradicionales (Coello, 1999). Otra ventaja de los algoritmos evolutivos para resolver estos problemas frente a la programación matemática tradicional es que son más flexibles a la forma o continuidad del frente de Pareto.

Algoritmo evolutivo multiobjetivo: Un MOEA es aquel AE cuya función de fitness es de la forma $\phi : I \rightarrow R^k$, con $k \geq 2$, una función multiobjetivo. Una característica importante de los algoritmos evolutivos está en su estructura. Si se realiza una descomposición de tareas se observa poca diferencia estructural entre un MOEA y un algoritmo evolutivo de un solo objetivo. Sea un AE general y un MOEA, ambos utilizando un esquema de tareas como se muestra en el algoritmo 2, en el MOEA la tarea 2 calcula los valores correspondientes a las funciones de fitness. Tener en cuenta que los MOEAs esperan un único valor de fitness con el que se realiza la selección, por lo que a veces es necesario un procesamiento para transformar soluciones de fitness vectoriales en un escalar (tarea $2a$). Aunque las diversas técnicas de transformación varían en su impacto algorítmico, el resto del MOEA es estructuralmente idéntico al de un solo objetivo. (Coello et al., 2007).

Algoritmo 2 Tareas de un MOEA general

1. Iniciar la población
2. Evaluar la función de fitness
    2a. Transformación del vector de fitness
3. Implementar el cruzamiento
4. Generar la mutación
5. Realizar la selección

Implementación

En cada generación del AE se determina un nuevo conjunto de soluciones que busca mejorar (dominar en términos de Pareto) al conjunto de soluciones actual. A la población actual se le denomina $P_{current}(t)$, donde t representa el número de generación. Existen varias implementaciones del MOEA que mantienen una población secundaria donde almacenan soluciones no dominadas que encuentran a través de las generaciones. (Van Veldhuizen, 1999; Van Veldhuizen and Lamont, 1998).

La solución $P_{current}(t)$ debe ser actualizada periódicamente dado que la clasificación de una solución como óptimo de Pareto depende del contexto del problema. Por tanto, los vectores que corresponden a las soluciones deben ser probados, y en caso de estar dominados, removidos de la solución actual.

La población secundaria mencionada anteriormente se denomina $P_{known}(t)$. Al igual que $P_{current}$, se denota con t para marcar los posibles cambios en la composición durante la ejecución del MOEA. $P_{known}(0)$ se define como el conjunto vacío (ϕ) y P_{known} como el conjunto final de soluciones resueltas por el MOEA. Para implementar el almacenamiento de la población secundaria existen diversas técnicas, la más sencilla es cuando se añade $P_{current}(t)$ en cada generación (i.e. $P_{current}(t) \cup P_{known}(t-1)$). $P_{known}(t)$ es el conjunto de soluciones óptimas de Pareto encontrado hasta el momento por el MOEA durante la generación t, en cualquier momento dado. El conjunto P_{true}, el verdadero conjunto óptimo de Pareto, no es explícitamente conocido para cualquier problema complejo. Este conjunto que es fijo y no cambia está implícitamente definido por las funciones que componen el MOP. Al definir por optimalidad de Pareto, el conjunto $P_{current}(t)$ resulta un conjunto no vacío, y por tanto, los conjuntos $P_{current}(t)$, P_{known}, y P_{true} son conjuntos de genotipos del MOEA donde cada conjunto de fenotipos correspondientes forman un frente de Pareto. El frente de Pareto asociado para cada uno de estos conjuntos de soluciones se llama $PF_{current}(t)$, PF_{known}, y PF_{true}. Entonces, cuando se utiliza un MOEA para resolver un MOP, la suposición implícita es que se cumple una de las siguientes afirmaciones:

$$P_{known} = P_{true}$$

$$P_{known} \subset P_{true}$$

$$u_i \in PF_{know} \; donde \; \{u_j \in PF_{true} \mid \forall i, \forall j, min\,(distancia\,(u_i, u_j)) < \epsilon\}$$

La distancia se define sobre una norma (e.g., euclidiana, RMS).

Diseño

En el diseño de un algoritmo evolutivo, la tarea principal radica en la asignación de fitness basada en Pareto para así identificar los vectores no-dominados de la población actual de un MOEA. Sobre esto, los principales objetivos a alto nivel para resolver MOPs son:

1. Preservar puntos no-dominados (elitismo vs. no-elitismo): por ejemplo se pueden armar rankings de dominancia basado en el fitness de cada solución.

2. Progresar o guiar PF_{known} hacia PF_{true}: se busca converger al verdadero frente de Pareto computacional.

3. Generar y mantener la diversidad de puntos en PF_{known} y en P_{known}: para lograr esta diversidad se puede implementar *Niching/fitness sharing* y *crowding* en el frente de Pareto, junto con sus variaciones.

4. Proporcionar al tomador de decisiones un número limitado de puntos PF_{known}.

Observando los objetivos anteriores se desprende que un MOEA debería orientar la búsqueda hacia PF_{true}, generar y mantener la diversidad de puntos PF_{known}, y evitar la pérdida de "buenas" soluciones mediante sistemas de almacenamiento.

Según Coello et al. (2007) la forma más clara del diseño de un MOEA idealizado o genérico sería dividirlo en los siguientes niveles.

Procedimiento general

Paso 0: Definir el MOP, establecer $F(x) = [f_(x), f_2(x), ..., f_k(x)]$ y la representación cromosómica de x. Definir las restricciones dinámicas, estáticas, lineales, no lineales, etc. Definir el modelo en un proceso de búsqueda algorítmico MOEA.

Paso 1: El MOEA genera el frente de Pareto PF_{known} y determina los conjuntos no-dominados durante las generaciones. Este converge en forma cerrada hacia el verdadero frente de Pareto, aquel que se obtiene computacionalmente y se denomina PF_{true}. Aquí tener en cuenta que en realidad lo que se obtiene es una aproximación del verdadero frente de Pareto. Ejecutar este proceso MOEA para un cierto número determinado de generaciones o hasta que alguna métrica alcance un umbral predefinido.

Paso 2: El MOEA genera una distribución uniforme en PF_{known} al final de cada generación.

Paso 3: Seleccionar varios puntos óptimos del PF_{known} que se considerarán en la etapa del tomador de decisiones (DM).

Paso 4: Determinar el conjunto P_{known} e implementar los valores de las variables de decisión seleccionada por el DM.

Paso 5: Visualizar el procesamiento de algoritmos y resultados para mejorar el rendimiento del MOEA (eficiencia y eficacia).

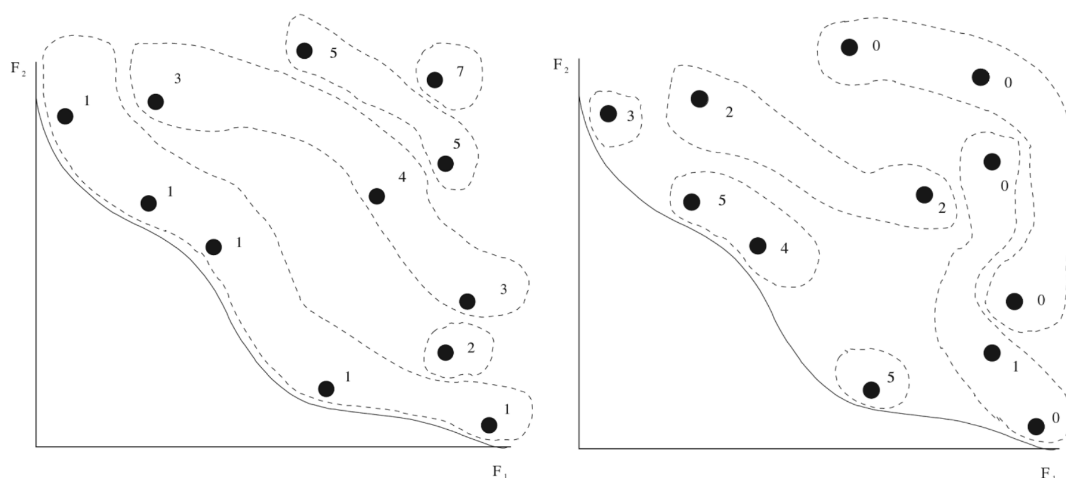
Para modelar un MOP mediante un software MOEA además de conocer el dominio del problema, es necesario definir la implementación de los operadores. Esto ayuda a mantener un diseño detallado con una implementación específica.

Dominancia de Pareto

La relación de dominancia es un operador binario, ya que se aplica sobre dos soluciones. Al comprobar la dominancia entre dos soluciones existen dos posibilidades: que una solución domine a otra, o que las soluciones no se dominen entre sí. En los operadores de dominancia existen relaciones matemáticas, tal como sucede con otro tipo de operadores binarios. En este caso se cumple la relación transitiva y reflexiva, por lo que podemos decir que está ordenado, y que no es un orden total pero sí un orden parcial estricto. Por definición entonces se puede decir que dado un punto en el espacio de la función objetivo, este puede ser dominado o no por otro punto del espacio.

Si se utiliza un AE para generar y seleccionar el conjunto óptimo de Pareto, se requiere de una técnica ordenada donde los valores de fitness son tuplas de dimensión N considerando n objetivos. Esta técnica debe ser escalable a través de las tuplas mediante un orden parcial estricto, para que se generan las soluciones no dominadas. En la mayoría, estas técnicas ordenan los individuos en el espacio de la función objetivo antes de la selección, y a cada miembro de los posibles individuos en el espacio de la función objetivo se le asigna un rango referido al orden de dominancia, al recuento de dominancia y la profundidad de la misma.

Según el dominio del problema, la implementación de un diseño específico de MOEA



(a) Ranking de dominancia, cuenta la cantidad de individuos que dominan una solución.

(b) Conteo de dominancia, cuenta la cantidad de individuos que una solución domina.

Figura 2.3: Dos mecanismos de cálculo de ranking de dominancia, agrupando individuos con igual valor

puede derivar en diferentes rendimientos (eficiencia y eficacia). El resultado de la clasificación de dominancia, en la Figura 2.3a, es una estricta lista ordenada parcial que se utiliza para clasificar los puntos antes de implementar el operador de selección. En la Figura 2.3b se observa un ejemplo de recuento de dominancia que minimiza dos objetivos y presenta un orden parcial distinto. Para ambos ejemplos, las diferentes comparaciones se muestran en las regiones punteadas. El ordenamiento está basado en el ranking de profundidad, y es diferente para las dos relaciones de dominancia. (Coello et al., 2007).

A continuación se nombran ejemplos de MOEAs que utilizan el ranking de dominancia:

- MOGA, NPGA: orden de dominancia (Deb et al., 2002; Horn et al., 1994)
- NSGA/NSGA-II: profundidad de dominancia (Deb et al., 2002; Srinivas and Deb, 1994)
- SPEA/SPEA2: recuento de dominancia y ranking de dominancia (Zitzler and Thiele, 1999; Zitzler et al., 2001)
- MOMGA-II: orden de dominancia (Van Veldhuizen, 1999)

Uno de los principales objetivos de diseño de un MOEA, que aún no fue mencionado, es proporcionar al tomador de decisiones una *diversidad de los puntos* de PF_{known} o P_{known} y que tengan una distribución uniforme a través del Frente de Pareto conocido. Existen diversas técnicas para mantener la diversidad en un MOEA, dentro de las que se destacan el enfoque de ponderación del vector, el enfoque de Fitness Sharing/Niching, Crowding/Clustering, Cruzamiento Restringido, y la Dominancia Relajada. (Sareni and Krahenbuhl, 1998)

2.4.3. Optimización de un MOEA genérico

Para lograr un MOEA eficiente, este debería incorporar las siguientes operaciones genéricas asumiendo operaciones en individuos completos (Coello et al., 2007, 87).

- Crear una población inicial de N individuos y una evaluación de fitness; esta etapa suele denominarse fase de inicialización. Cada individuo que es codificado en el dominio del problema, puede ser binario, entero o real.
- Retirar individuos dominados del Frente Pareto de P, basándonos en las evaluaciones de la función multiobjetivo escalar; $P \rightarrow P_i$.
- Utilizar un estimador de densidad para restringir el número de individuos que se encuentran en P_i en regiones del PF_{known} actual o P_{known}. Algunas de estas técnicas son *niching*, *sharing* y *crowding* con los valores de los parámetros asociados.
- Realizar las operaciones evolutivas (recombinación, mutación, etc.) para generar nuevos individuos que utilicen valores de los parámetros adecuados; $P_i \rightarrow P_{ii}$. Algunas técnicas de selección de individuos para una recombinación son ranking, binary tournament, o selección proporcional.
- Seleccionar individuos para la próxima generación (población P_{iii}), uno podría operar en $[P_{ii}]$ o $[P_i P_{ii}]$ utilizando el ranking. P_{iii} es el $P_{current}$. También existen técnicas para restringir el tamaño de P_{iii} con el operador de selección, como por ejemplo el operador de selección por torneo binario con reemplazo o elitismo. El elitismo suele generar los mejores resultados debido a que consiste en retener los mejores individuos del espacio objetivo.
- Si una condición de terminación no se cumple, como puede ser el número máximo de generaciones o algún criterio de convergencia, establecer P_{iii} a P como $P_{current}$.
- Retirar los individuos dominados de Pareto y no factibles desde P_{iii}, o intentar reparar los individuos no factibles. Establecer P_{iii} a P como $P_{current}$.
- Mantener un archivo de los individuos no dominados y factibles mediante el almacenamiento de P_{iii} en un archivo P_{iv}. A medida que la nueva población P_{iii} se une con el archivo, el operador de no dominación se aplica a la combinación resultante de la fusión. Los archivo P_{iv} contienen los puntos P_{known} y PF_{known} asociados.
- Las operaciones de búsqueda local en MOEAs híbridos o meméticos pueden proporcionar un buen rendimiento mediante la exploración de las regiones límite en el espacio objetivo. Esto significa que solo se mueve hacia regiones específicas en el frente de Pareto.

En el algoritmo 3 se presenta el pseudocódigo para un MOEA genérico.

Bloques de construcción (BBs)

Para entender cómo funcionan los bloques de construcción, primero debe estudiarse que son los esquemas. Según (Murias Rodríguez, 2007, p,2) un esquema es un patrón que se utiliza para representar un subconjunto de una parte de la codificación de soluciones. Entonces, en una cadena binaria algunas posiciones se fijan mientras otras se dejan libres, y cuando está ocurriendo la evolución, en los individuos de la población

Algoritmo 3 Esquema de un MOEA optimizado

```
Inicializar población P y P_iv
Evaluar los valores objetivos F(x) en la población
Asignar el Fitness Compartido o el Crowding
while no se cumple condición de terminación (nro. de generaciones u otro) do
    Seleccionar los mejores individuos de P → P^i
    Cruzar y mutar individuos de P^i → P^ii
    Evaluar valores objetivos de los hijos P^ii
    Rankear (P^i ∪ P^ii) → P^iii basado en la dominancia de Pareto;
    Calcular el Niche
    Asignar el Fitness Compartido o el Crowding
    Reducir y descartar P^iii → P
    Copiar P^iii → P^iv basado en la dominancia de Pareto
end while
```

aparecen los esquemas que contienen soluciones buenas. Estos esquemas son combinados para crear mejores cadenas y se denominan bloques *constructivos o de construcción.*

Un bloque constructivo es un esquema de longitud definida y corta, con orden pequeño y un fitness alto (superior al fitness medio).

Un BB es considerado una solución a un subproblema del problema global de optimización. El primero, se subdivide en subproblemas más simples de tratar, para luego combinarlos, siendo cada vez más complicados hasta encontrar una solución. (Murias Rodríguez, 2007)

La mayoría de los MOEAs generacionales procesan Bloques de Construcción (BBs) implícitamente, y algunos pocos algoritmos, como el MOMGA, procesan los BBs de forma explícita. Las estructuras de BBs son distintas en diferentes vectores en el espacio objetivo. Esto significa que dado un problema multiobjetivo, para diferentes MOEAs la performances de los objetivos es distinta.

NSGA

Srinivas and Deb (1994) propusieron una variación del enfoque de Goldberg llamado el *Nondominated Sorting Genetic Algorithm* (NSGA, algoritmo genético de ordenamiento no dominado). NSGA está basado en varias capas de clasificaciones de individuos. Antes de realizar la selección, la población es clasificada en base a la dominancia: todos los individuos no dominados son clasificados en la misma categoría con un valor de fitness simulado, proporcional al tamaño de la población, que le proporciona un potencial reproductivo igual para estos individuos. El pseudocódigo de NSGA se presenta en el algoritmo 4, el mismo se basa en el presentado en (Coello et al., 2007, 91).

NSGA-II

Deb et al. (2002) propuso una versión mejorada del algoritmo NSGA llamado NSGA-II. En el Algoritmo 5 se muestra el pseudocódigo del NSGA-II. Este algoritmo se presenta como una mejora al diseño original del NSGA; es un BB genérico no explícito aplicado a

Algoritmo 4 NSGA

```
function NSGA(ℕ, 𝔾, f(x))
    ▷ ℕ miembros evolucionan 𝔾 generaciones para resolver f(x)
    P_0 = InicializarPoblacion(ℕ)
    gen = 0
    while gen < 𝔾 do
        Selección proporcional al valor de fitness
        Cruzamiento en un punto
        Operador de mutación
        Evaluar función objetivo f(x) sobre las soluciones
        Asignar ranking basado en dominancia de Pareto
        Calcular el niche
        Asignar el fitness compartido
        gen = gen + 1
    end while
end function
```

MOP. En el algoritmo se construye una población de individuos competentes, se clasifica y ordena cada individuo de acuerdo al nivel de no dominación, se aplican las EVOPs para crear nuevo grupo de descendientes (offspring), y luego se combinan los padres e hijos antes de la partición del nuevo offspring. Luego se realiza el niching agregando la distancia de crowding a cada miembro. Esta distancia se utiliza en su operador de selección para mantener un frente diverso, asegurándose que cada miembro mantiene una separación de distancia de crowding. Así, la población se mantiene diversa y ayuda al algoritmo a explorar el espacio de fitness.

Restricciones

Para que un MOP no converja hacia soluciones no factibles se deben definir restricciones a incorporar en la búsqueda.

El conjunto de restricciones suele definirse por el vector $g(x) \leq 0$. Suelen utilizarse solo restricciones de desigualdad debido a que restricciones de igualdad pueden transformarse fácilmente en una de desigualdad.

Las restricciones más utilizadas para un AE simple y multiobjetivo son funciones de penalización (Richardson et al., 1989). Su fórmula general es la siguiente:

$$\phi(x) = f(x) \pm [\sum_{i=1}^{n} r_i \times G_i + \sum_{j=1}^{p} c_j \times L_j]$$

donde $\phi(x)$ es el nueva ampliada función objetivo, G_i y L_j son funciones de las restricciones $g_i(x)$ y $h_j(x)$, respectivamente, y r_i y c_j son constantes positivas normalmente llamadas ”factores de penalización”. La forma más común de G_i y L_j es:

$$G_i = max[0, g_i(x)]^{\beta}$$
$$L_j = \mid h_j(x) \mid^{\gamma}$$

donde β y γ son normalmente 1 o 2.

Algoritmo 5 NSGA-II

```
function NSGA-II(ℕ, 𝔾, f(x))
    ▷ ℕ miembros evolucionan 𝔾 generaciones para resolver f(x)
    P(0) = InicializarPoblacion(ℕ)
    gen = 0
    EvaluarFuncionObjetivo(f(x), P(0))
    OrdenamientoNoDominado(P(0))
    padres = SeleccionarPadres(P(0), ℕ)                    ▷ Torneo binario
    hijos = CruzamientoYMutacion(padres)         ▷ Aplica operadores evolutivos
    while gen < g do
        EvaluarFuncionObjetivo(f(x), P(gen))
        union = padres ∪ hijos
        frentes = OrdenamientoNoDominado(union)       ▷ Vectores de dominancia
        nuevaPob = ∅, i = 1
        while |nuevaPob| + |frentes(i)| ≤ ℕ do
            CalcularDistanciaDeCrowding(frentes(i))
            nuevaPob = nuevaPob ∪ frentes(i)
        end while
        OrdenamientoPorDistanciaCrowding(frentes(i))
        ▷ Tomo los primeros elementos del último frente para completar ℕ
        nuevaPob = nuevaPob ∪ frentes(i)[1 : (ℕ − |nuevaPob|)]
        gen = gen + 1
        P(gen) = nuevaPob
        padres = SeleccionarPadres(P(gen), ℕ)              ▷ Torneo binario
        hijos = CruzamientoYMutacion(padres)     ▷ Aplica operadores evolutivos
    end while
    return P(gen)
end function
```

Operador de selección torneo binario

En esta sección se presenta uno de los operadores de selección utilizado por los algoritmos NSGA y NSGA-II, y que utilizamos en nuestro trabajo.

Según Bäck et al. (2000) un grupo de q individuos es elegido con o sin reemplazo de forma randómica del conjunto población. Estos individuos elegidos compiten en un torneo donde se determina el ganador según su valor de fitness. El individuo con el mayor valor (mejor) es usualmente elegido de forma determinística, aunque a veces se puede implementar una selección estocástica. Sea por una técnica u otra, solo el ganador del torneo es agregado en la siguiente población. Este proceso se repite λ veces para obtener la nueva población. En AEs, usualmente este torneo se implementa con dos individuos y es conocido como *Binary Tournament*, aunque también se puede generalizar a un tamaño de grupo arbitrario más grande conocido como tamaño del torneo.

Como la población no necesita ser ordenada, la selección por torneo puede ser implementada muy eficientemente en un tiempo de orden $O(\lambda)$. A pesar de esto, el algoritmo anterior tiende a producir una alta varianza en el número esperado de descendientes a medida que se ejecutan ensayos independientes λ.

Según Maza and Tidor (1993) la selección por torneo es de ajuste invariante, por lo tanto un ajuste del valor del fitness no afecta el comportamiento de la selección. Es por esto que las técnicas de ajuste no son necesarias y simplifican la aplicación del método de selección.

Variación de torneo binario utilizando crowding Una variación sobre el operador de torneo binario es propuesta en Deb et al. (2002), esta versión del torneo binario utiliza también el factor de distancia de crowding. En el algoritmo del torneo binario, si dos soluciones que compiten por ser seleccionadas empatan en el torneo, la función devuelve de forma aleatoria cual elegir en la selección. En esta variante, en el caso de empate, se elije la solución con mayor distancia de crowding, y luego si también empatan en esta comparación se retorna una al azar.

Cada individuo $a \in I$ (normalmente representado como un vector a) representa una solución candidata, donde las dimensiones del vector son análogas a un cromosoma y sus genes. El algoritmo deja dimensiones de cada individuo sin especificar y se modifican según sea necesario para cada instancia AE en particular.

Cuando se define la transformación de una población (generacional) se denota según la siguiente relación: $\tau : I_\mu \to I_\mu$, donde $\mu \in N$. Sin embargo, algunas variantes de AEs obtienen poblaciones resultantes cuyo tamaño no es igual a sus predecesores. Este algoritmo también representa a todos los tamaños de población, operadores evolutivos y parámetros como secuencias debido a que diferentes AEs utilizan estos factores de forma distinta.

2.4.4. Análisis y pruebas en un MOEA

Cuando se construye un MOEA hay que tener especial cuidado en la arquitectura, debe ser una representación precisa, fiable, consistente y no arbitraria. Estos procedimientos, al igual que cuando se definen criterios estándar, son para tratar de minimizar la influencia de sesgo o prejuicio del experimentador al probar una hipótesis de un MOEA.

El diseño detallado está basado principalmente en los esquemas presentados por Barr et al. (1995). Estos dos artículos discuten experimentos computacionales sobre el diseño de métodos heurísticos, y plantean las directrices para la presentación de resultados asegurando la reproducción. Los pasos que plantean para un buen diseño son los siguientes:

1. Definir objetivos experimentales
2. Elegir medidas de rendimiento (métricas)
3. Diseñar y ejecutar el experimento
4. Analizar los datos y sacar las conclusiones
5. Registrar los resultados del experimento

El objetivo más importante de todas las pruebas es comparar la eficacia de un MOEA sobre distintos MOPs elegidos mediante la medición de calidad de la solución. Para obtener el resultado de estas pruebas se deben seleccionar los MOPs específicos y métricas. Generalmente las métricas de rendimiento elegidas son la eficiencia y eficacia.

La eficiencia es el esfuerzo computacional que necesita para obtener las soluciones. Pueden ser por ejemplo, tiempo de CPU, número de evaluaciones/iteraciones, uso de los recursos espaciales y temporales, etc.

La eficacia mide la exactitud y convergencia de las soluciones. También incluye medidas de robustez, escalabilidad y facilidad de uso. La robustez mide qué tan bien se recupera el código de una entrada incorrecta, la escalabilidad mide la capacidad del algoritmo de resolver una clase de problemas en la medida que la dimensión de este se hace mayor, y la facilidad de uso mide la cantidad de esfuerzo que un usuario requiere para utilizar el código.

El principal objetivo cuando se prueba un MOEA es comparar algoritmos bien diseñados en términos de eficacia y eficiencia utilizando las métricas apropiadas. Estos dos enfoques de medición deberían ser suficientes para validar la factibilidad y confiabilidad de un MOEA.

Si se desea estudiar y analizar la dinámica de ejecución de un MOEA, se necesita medir la población generacional de PF_{known} y P_{known}. Estos indicadores están basados generalmente en el concepto de dominancia de Pareto. Otras variables a utilizar son la mediciones utilizando primer orden (media) y de segundo orden (varianza) y estadística de orden superior, que pueden desprenderse de las distribuciones estocásticas empíricas para la eficacia y eficiencia.

2.4.5. Parámetros de un MOEA

Con el fin de lograr la implementación más eficaz y eficiente para cierta clase del problema, suele variarse los parámetros del algoritmo y algunas de las principales características asociadas intentando buscar los mejores resultados. El objetivo de estas pruebas es buscar el mejor rendimiento general del MOEA y explorar el dominio del algoritmo específico.

Representación y operadores evolutivos (EVOPs): Generalmente un MOEA utiliza una representación binaria, formada por una cadena de $l - bit$ ($l = 24$) para cada solución y valores mínimos/máximos idénticos en cada dimensión de variable de decisión, asegurando la misma accesibilidad de los algoritmos de prueba para un MOP dado. La dimensión de la solución puede variar según se desee, por ejemplo, para examinar un mayor espacio de búsqueda se podría aumentar l a 64 bits. A pesar de esto, algunos MOEAs utilizan diferentes valor-binario para asignar valor-real.

No se ha definido un valor (tasa) de cruzamiento por defecto en un MOEA, pero sí la mayoría lo utilizan en el rango de $P_c \in [0{,}7, 1{,}0]$ (Horn et al., 1993). En cuanto al operador de mutación, todos los algoritmos relevados en este trabajo excepto el MOMGA se implementan con una tasa de mutación de $P_m = 1/l$, donde l es el número de dígitos binarios.

Tamaño de la Población: Diversos algoritmos utilizan una inicialización aleatoria de población (e.g., NSGA-II, NPGA, MOMGA, MOMGA-II, SPEA2, PAES), dado la representación genética, todas las soluciones en la población inicial son seleccionadas uniformemente del espacio de soluciones.

NSGA-II distribuye una población constante a través de una secuencia de frentes clasificados, en cambio, el MOMGA utiliza un esquema más determinista. En cada generación este algoritmo genera todos las posibles bloques de construcción (BBs) de tamaño k. Esto determina que la población inicial siempre sea conocida. Por otro lado, el algoritmo IMOEA basándose en el número actual de individuos en frente de Pareto conocido, y en la densidad de población que se desee, utiliza una técnica que consiste en encolar los individuos a una población dinámica (Chen, 2004). Además, existen los micro-MOEAs que limitan el tamaño de la población a unos pocos individuos debido al alto costo computacional de la función objetivo.

Asignación de fitness: El NSGA evalúa y clasifica la población por rango. Lo que hace es asignar una aptitud alta inicial para todas las soluciones del mejor rango. Luego, asigna un valor menor de aptitud de prueba a las soluciones del siguiente mejor rango, y así sucesivamente.

El MOMGA y NPGA utilizan selección por torneo y no requieren de modificación en la solución de fitness. El MOGA evalúa todas las soluciones y luego asigna el fitness mediante ordenamiento por clasificación de la población ('0' es la mejor, hasta 'N'). El fitness se asigna a cada solución linealmente ordenado, y el fitness final se determina promediando los valores de fitness para soluciones igual clasificadas y después realiza el intercambio de fitness.

Restricciones de cruzamiento: Las restricciones de cruzamiento son sobre ambos individuos. Existen diversos experimentos sobre este tema, algunos indican que varias implementaciones del MOEA funcionan bien sin él, mientras que otros resultados arrojan que son necesarios para el buen rendimiento.

El problema que existe en agregar restricciones de cruzamiento en el software del MOEA experimental, es que requiere de importantes modificaciones en el código, y debido a su utilidad incierta en el dominio del MOP, es que estas restricciones no suelen agregarse en la mayoría de los MOEAs experimentales.

Compartiendo el fitness: Los MOEAs en su mayoría incorporan lo que se llama *phenotypic based sharing*, lo que hacen es utilizar distancia entre los vectores objetivo para la consistencia.

En el caso del MOGA y NSGA, se calcula σ_{share} y se forma una matriz de intercambio a través de la ecuación de intercambio estándar (Goldberg, 1989). El intercambio del fitness se produce sólo entre soluciones con el mismo rango.

Se han desarrollado otros enfoques, el más conocido es la distancia de crowding (utilizada por el algoritmo NSGA-II). Se utiliza σ_{share} para representar cómo dos individuos cercanos deben estar en orden para disminuir el fitness de cada uno. Este valor suele depender del óptimo en el espacio de búsqueda. Al ser desconocido este óptimo, así como también el frente de Pareto real en el espacio objetivo, el valor de σ_{share} se asigna mediante el método de Fonseca (Fonseca and Fleming, 1998):

$$N = \frac{\prod_{i=1}^{k}(\Delta_i + \sigma_{share})}{\sigma_{share}^{k}}$$

Terminación: Un algoritmo debe parar su ejecución cuando converge, pero no es posible saber cuándo se está produciendo la convergencia. La mejor estimación que suele implementarse, es establecer banderas de terminación adecuadas. En diversos MOPs, lo que se utiliza es ejecutarlo primero y determinar el número de evaluaciones de soluciones por ejecución. En cambio, otros MOEAs (por ejemplo, cada uno con tamaño de población de N = 50) se pueden fijar al ejecutar el mismo número de evaluaciones (N multiplicado por el número de generaciones), asegurando de esta forma que para cada prueba del MOEA se emplea el mismo esfuerzo computacional.

2.4.6. Evaluación de eficacia

Para poder utilizarlos comparativamente, algunos indicadores necesitan ser normalizados y escalados (según sea lineal o no lineal).

Si se quiere formar métricas de calidad formales, es necesario seleccionar un conjunto apropiado de indicadores de calidad. A continuación se nombran los más importantes.

Relación de error (ER): O Tasa de Errores (ER) reporta el número de vectores en PF_{known} que no son miembros de PF_{true}. Es compatible con la terminología de Pareto, y requiere que el frente de Pareto real (PF_{true}) sea conocido, y que el MOEA se aproxime al FP. Matemáticamente, se representa con la siguiente ecuación:

$$ER \triangleq \frac{\sum_{i=1}^{|PF_{know}|} e_i}{|PF_{know}|}$$

donde $e_i = 0$ cuando el vector i-ésimo de PF_{know} es un elemento de PF_{true}, o $e_i = 1$ si el vector i-ésimo de PF_{known} no es un elemento de PF_{true}. (Van Veldhuizen, 1999)

Distancia generacional (GD): Esta métrica reporta en qué medida (promedio) PF_{known} es de PF_{true}. No cumple las normas de Pareto y requiere que se conozca PF_{true}. Matemáticamente, se representa con la siguiente ecuación:

$$GD \triangleq \frac{(\sum_{i=1}^{n} d_i^p)^{\frac{1}{p}}}{|PF_{know}|}$$

donde $| PF_{know} |$ es el número de vectores en PF_{know}, $p = 2$, y d_i es la distancia fenotípica euclideana entre cada miembro i de PF_{known} y el miembro más cercano en PF_{true} a ese miembro i. (Van Veldhuizen, 1999)

Hiperárea: El hiperárea (HA) y la medida de relación de HA son compatibles con Pareto y están relacionados con el área de cubrimiento del PF_{known} con respecto al espacio objetivo para una MOP que sea bi-objetivo. Esto equivale a la suma de todas las áreas rectangulares, limitada por algún punto de referencia y (f1(x), f2(x)). Matemáticamente, se representa con la ecuación siguiente:

$$HA \triangleq \{\bigcup_i area_i | vec_i \in PF_{know}\}$$

donde vec_i es un vector no dominados en PF_{known} y $area_i$ es el área entre el origen y el vector vec_i. Notar que si PF_{known} no es convexa, los resultados pueden engañar. El punto de referencia tomado para calcular el HA es el valor mínimo para cada objetivo. (Zitzler and Thiele, 1998)

Hipervolumen (HV): La medida de HV es similar a HA, es compatible con Pareto y se define como el volumen de cubrimiento de PF_{known} con respecto al espacio objetivo para una MOP bi-objetivo. Esto equivale a la suma de todas las áreas rectangulares, limitada por algún punto de referencia y $(f_1(x), f(x))$. (Zitzler and Thiele, 1998)

Matemáticamente, se representa con la siguiente ecuación:

$$HV \triangleq \{\bigcup_i vol_i | vec_i \in PF_{know}\}$$

La diferencia con la métrica HA es que excede las dos dimensiones y sustituye vol_i por $area_i$ en la ecuación.

Espaciado: describe numéricamente el espaciado (*spread*, S) de los vectores en PF_{known}. A diferencia de las anteriores no es compatible con Pareto. Mide la distancia de varianza entre vectores vecinos en PF_{known}. Matemáticamente, se representan con las dos ecuaciones siguientes:

$$S \triangleq \sqrt{\frac{1}{|PF_{know}|-1} \sum_{i=1}^{|PF_{know}|} (\bar{d} - d_i)^2}$$

y

$$d_i = min_j(| f_1^i(x) - f_1^j(x) | + | f_2^i(x) - f_2^j(x) |)$$

donde $d_i = min_j(| f_1^i(x) - f_1^j(x) | + | f_2^i(x) - f_2^j(x) |), i, j = 1, ..., n, \bar{d}_i$ es la media de todos los d_i, y n es el número de vectores en PF_{know}. Cuando $S = 0$, todos los miembros están espaciados uniformemente. Este indicador no requiere que se conozca PF_{true}, aunque se supone el MOEA ya convergió antes de aplicar esta medida. (Coello et al., 2007)

Capítulo 3

Computación heterogénea y trabajos relacionados

En este capítulo se presenta el concepto de computación heterogénea (*heterogeneous computing*, HC), discutiendo algunas de sus virtudes y desventajas. A continuación se presenta el problema del consumo energético asociado a los sistemas de HC, y algunas técnicas utilizadas para regular el mismo.

Luego se presenta el problema de la planificación de recursos en sistemas heterogéneos, y los trabajos relacionados clasificados según su enfoque en: estrategias generales para disminuir el consumo energético, trabajos sobre sistemas de cluster o grid y trabajos en el ámbito empresarial.

3.1. Sistema Computacional Heterogéneo

Según Khokhar et al. (1993), un sistema de computación heterogéneo es un conjunto bien orquestado y coordinado de los elementos de proceso, también llamados recursos, interconectados por una red y pudiendo incluir también máquinas paralelas.

Un sistema HC puede incluir máquinas heterogéneas, redes de alta velocidad, interfaces, sistemas operativos, protocolos de comunicación y ambientes de programación, todos combinados para un mejor rendimiento y facilidad de uso del sistema. Es una técnica eficiente para resolver problemas de uso computacional intensivo (Foster and Kesselman, 2003). Observar que un sistema HC puede ser visto como un computador virtual que está constituido por un conjunto de máquinas heterogéneas distribuidas que ofrecen su potencia de cómputo individual a la potencia computacional del sistema agregado total.

Según (Kshemkalyani and Singhal, 2011), se puede caracterizar a un sistema HC por:

1. Los recursos funcionan de forma asíncrona, no existe un reloj global al sistema.

2. No se utiliza memoria compartida en el sistema, por lo que se requiere de un modelo definido de comunicación y envío de mensajes.

3. Los recursos individuales son autónomos y heterogéneos, pudiendo tener distinta velocidad de cómputo, ejecutar diferentes sistemas operativos y tener distintas capacidades específicas.

4. Es habitual que los recursos se encuentren separados geográficamente, aunque no es un requisito que se comuniquen a través de una red WAN (red de área extensa).

Existen sistemas HC compuestos por pocos recursos informáticos o estaciones de trabajo que se conectan a través de redes locales, pero también existen sistemas que contienen miles de recursos esparcidos geográficamente conectados a través de redes WAN. Estos sistemas distribuidos de HC se utilizan cada vez en más áreas de aplicación, por ser herramientas muy útiles que brindan la potencia computacional necesaria para aplicaciones científicas y de alto rendimiento, con un costo que suele ser inferior al de los sistemas homogéneos (Foster and Kesselman, 2003; Zhao et al., 2008).

Debido a que existen diversos tipos de heterogeneidad, al definir un modelo de problema muchas veces se considera únicamente la heterogeneidad en las velocidades de procesamiento. Sin embargo existen otros tipos de heterogeneidad, diferencias en el hardware: diferentes grupos de instrucciones, incompatibilidades en la representación de datos, diferencias e incompatibilidades generales. Diferencias de red: diferencias entre medios de comunicación, técnicas de señalización, interfaces. Diferencias en el software: sistema operativo, disponibilidad de librerías o aplicaciones.

Los sistemas de HC muchas veces brindan servicio a una gran cantidad de usuarios, haciendo necesario planificar el uso de los recursos. La planificación consta de la asignación de recursos informáticos a las tareas que ingresen los usuarios, buscando cumplir con los objetivos definidos para el sistema. Los objetivos que se definan suelen ser a nivel de uso de los recursos, calidad de servicio, tiempo de ejecución de las tareas, consumo energético, entre otros (Buyya, 2002; Nesmachnow et al., 2010).

Una desventaja de los sistemas HC es que la planificación de la asignación de tareas debe tener en cuenta la diferencias de los recursos del sistema, ya que es posible que el tiempo de ejecución o incluso la capacidad de ejecutar una tarea varíe entre recursos.

Debido a que muchos sistemas HC se encuentran geográficamente distribuidos, muchas veces para su uso se requieren mecanismos de acceso remoto a los recursos. Los mecanismos pueden ser sistemas para ingresar y administrar tareas específicamente diseñadas o protocolos estándar como SSH, un protocolo que permite acceder a una terminal remota a través de la red. Este protocolo permite ejecutar comandos en el recurso remoto, acceder a archivos e incluso ejecutar programas.

3.2. Consumo energético en computadoras

En la actualidad, los sistemas de cómputo de altas prestaciones ofrecen un gran rendimiento pero consumen enormes cantidades de energía eléctrica (Casanova et al., 2012). En esta sección se presentan algunos mecanismos que mejoran la eficiencia energética de los recursos intentado minimizar la degradación en la calidad de servicio del sistema.

Regular el consumo energético

En una computadora el consumo energético se puede dividir en el consumo dinámico y el consumo estático. El consumo estático corresponde al consumo base del recurso encendido, permanece constante ante distintos niveles de carga del recurso, mientras que

el consumo dinámico varía según el nivel de carga del recurso. Cuando un recurso se encuentra inactivo su consumo corresponde al consumo estático.

Cuando el sistema tiene una baja carga de utilización, el consumo dinámico disminuye y el estático se mantiene, lo que hace que el factor del consumo estático sobre el dinámico aumente. Al bajar el consumo dinámico, disminuye la eficiencia energética debido a que el número de operaciones ejecutadas por unidad de energía es menor.

Consumo energético en servidores

Según Minas and Ellison (2009), tanto la capacidad de los servidores como su consumo energético han aumentando considerablemente con el tiempo, siendo los procesadores y los módulos de memoria RAM los componentes que contribuyen en mayor medida al incremento del consumo.

Los módulos de memoria utilizados en un servidor han aumentado, lo que conlleva a un aumento del consumo energético. En el trabajo de Minas and Ellison (2009) se menciona que las fuentes de alimentación que se utilizan en los servidores son ineficientes y desperdician gran parte de la energía en la transformación del voltaje de la corriente eléctrica al utilizado por el computador. La mayoría de los servidores de hoy en día utilizan el 20-40 % de lo suministrado por las fuentes, cuando el nivel de eficiencia máximo de las mismas se da en el rango entre 50 % y 70 % de carga. En su trabajo determinan que el uso de una fuente un 25 % más eficiente permitiría disminuir el consumo energético en un 17 %.

Un inconveniente de los circuitos electrónicos, es que cuando se encuentran en funcionamiento generan calor, y un elevado consumo energético suele llevar consigo grandes cantidades de calor que debe ser disipado. Mecanismos para enfriar y disipar el calor de los recursos son necesarios para mantener los componentes físicos a una temperatura segura, aunque este proceso de enfriamiento suele consumir energía adicional, ya sea mediante ventiladores, aire acondicionado, o cualquier otro sistema de refrigeración.

Los data centers de hoy en día contienen cada vez más servidores, lo que implica una mayor cantidad de electricidad consumida. La electricidad que utilizan los servidores se duplicó entre los años 2000 y 2005, de 12 mil millones de kilovatios hora a 23 mil millones. Este crecimiento de casi el doble de consumo de electricidad fue debido al aumento de la cantidad de servidores instalados en los data centers y a la infraestructura elegida de refrigeración (Koomey, 2008).

El consumo energético de los servidores ha aumentado con el tiempo. Antes del año 2000, los servidores consumían unos 50 vatios de electricidad en promedio, y en el año 2008 ya consumían un promedio de hasta 250 vatios. Los data centers aumentan a un ritmo más rápido su consumo de energía conforme van cambiando sus factores de forma de servidor a una distribución física de mayor densidad.

Una gran parte de la energía eléctrica que entra en una computadora se convierte en calor. La cantidad de calor generada por un circuito integrado depende en gran parte de que tan eficientemente esté diseñado el componente, además de la tecnología que se utiliza en el proceso de fabricación y de la frecuencia y voltaje a los que funcionan los circuitos. Disipar este calor generado en el servidor o en un data center, requiere de energía adicional.

Los procesadores, unidades de memoria y fuentes de alimentación de un servidor, generan grandes cantidades de calor cuando se encuentran en funcionamiento. Este calor

debe ser disipado para mantener los componentes dentro de temperaturas de funcionamiento seguras. Las piezas que se sobrecalientan generalmente tendrán una vida útil más corta. Una menor vida útil de los componentes del sistema puede llegar a producir problemas esporádicos, bloqueos del sistema o incluso fallos del mismo.

Debido a la demanda por recursos más eficientes, muchos fabricantes han incluido funciones que permiten administrar el consumo de energía de los recursos, y además permiten también a los usuarios evaluar el rendimiento por unidad de energía consumida.

C-States y P-States

ACPI (*Advanced Configuration and Power Interface*) es un conjunto de interfaces estándar de la industria desarrollado en conjunto por Hewlett-Packard, Intel, Microsoft, Phoenix, y Toshiba, para permitir al sistema operativo administrar el consumo energético y la gestión térmica de los distintos componentes de un computador. (ACPI, 2015).

En las interfaces definidas en ACPI (2015), se definen dos tipos de estados que serán utilizados en este trabajo, los C-States y los P-States, los primeros administran los componentes del sistema cuando este se encuentra ocioso, y los segundos regulan los distintos estados de performance del sistema cuando se encuentra activo.

Cada estado C tiene características diferentes de ahorro de energía y rendimiento (a causa de la latencia requerida para activar nuevamente el componente). La ACPI establece distintos estados, el estado C-0 representa el estado activo y los restantes definen diferentes estados de ociosidad, a mayor número menor será el consumo y mayor el tiempo que le tome volver al estado C-0.

Los P-State de las CPU, sirven como herramienta para regular el nivel de performance y el consumo energético de un recurso activo. Estos estados controlan la velocidad del procesador, relacionada con la cantidad de operaciones por segundo que puede ejecutar, y se encuentran definidos en la mayoría de las arquitecturas de procesadores modernos.

Escalado dinámico de la frecuencia

La tecnología de escalado dinámico de la frecuencia (DFS - Dynamic Frequency Scaling) es una técnica estándar que permite reducir el consumo energético mediante cambios dinámicos de la frecuencia de reloj de las CPUs. Esta técnica está relacionada con los P-States de ACPI, y es utilizada en varios trabajos para regular el consumo energético, como Casanova et al. (2012); Weiser et al. (1994). Ejemplos comerciales de DFS son la tecnología SpeedStep de Intel, y PowerNow! y Cool'n'Quiet de AMD.

Lo que se utiliza para ahorrar energía eléctrica resulta de la ecuación de potencia (P), que es proporcional al cuadrado del voltaje y la frecuencia del reloj (Rabaey et al., 2002; Snowdon et al., 2005). Se asume que los ciclos de reloj para un cierto cálculo es independiente de la frecuencia, entonces el tiempo de ejecución es inversamente proporcional a la frecuencia. Por lo tanto, la energía total (E) utilizada por la CPU para realizar el cálculo es proporcional al cuadrado del voltaje.

Una forma de utilizar este modelo es cuando baja el uso de la CPU un cierto umbral definido, entonces se disminuye el voltaje y la frecuencia de la CPU ahorrando así energía eléctrica. Por el contrario, cuando el uso de la CPU supera cierto umbral, se aumenta el voltaje y la frecuencia de la CPU para mejorar el rendimiento del sistema. Balladini et al. (2010)

Frecuencias de reloj más bajas requieren menos potencia del procesador, reduciendo así el calor generado y por tanto la energía necesaria para la refrigeración, así como también abre la posibilidad de aumentar la densidad de componentes. Normalmente, los algoritmos DFS buscan reducir el consumo energético sin reducir el rendimiento, no obstante, en algunas situaciones particulares es recomendable llevar el consumo energético a niveles inferiores aunque se deteriore el rendimiento del sistema.

Esta tecnología también puede utilizarse para hacer que los recursos funcionen a una frecuencia mayor de la normal, aumentando su potencia de cómputo y su consumo. En el trabajo de Casanova et al. (2012) se utiliza DFS tanto para disminuir como para aumentar la potencia de un recurso, aumentando la potencia de los recursos cuando la carga del sistema es baja buscando terminar la ejecución de las tareas lo más pronto posible, para poder pasar el recurso a un estado inactivo y disminuir su consumo.

3.3. Planificación de recursos computacionales heterogéneos

En esta sección presentamos los problemas de planificación, en particular el problema de la planificación de recursos en un ambiente computacional heterogéneo (*Heterogeneous Computing Scheduling Problem*, HCSP).

La planificación del uso de recursos en ambientes computacionales, pueden modelarse como un caso particular del problema de planificación de proyectos con recursos limitados (*resource-constrained project scheduling problem*, RCPSP). De forma simple, RCPSP se basa en determinar el orden en el que se ejecutarán un conjunto de actividades, y la asignación de un conjunto limitado de recursos a las actividades, buscando minimizar el tiempo total que tomará ejecutar todas éstas. Se presenta la expresión matemática del problema tal como se presenta en Hartmann and Briskorn (2010), se cuentan con n actividades $i = 1 \dots n$, y m recursos reutilizables $j = 1 \dots m$, de cada recurso se conoce la cantidad de unidades disponibles r_j. Cada actividad requiere t_i tiempo para ser completada, y requiere r_{ij} unidades del recurso j. Puede existir precedencia entre las actividades, si una actividad y debe ser ejecutada antes que otra actividad z se denota como $y \rightarrow z$. El objetivo del problema es determinar la asignación de recursos y el tiempo en el que se comienza a ejecutar cada una de las actividades s_i, para minimizar el makespan, definido como el tiempo en el que se termina de ejecutar la última actividad, respetando la disponibilidad de recursos y la precedencia de las actividades.
El makespan se define como $Mk = \max\limits_{i=1 \dots n} (s_i + p_i)$.

Existen diversas propiedades del RCPSP que permiten clasificarlo, puede ser por la cantidad de unidades disponibles de cada recurso, si $r_j = 1$ el problema es disyuntivo, y únicamente una actividad puede utilizar un recurso a la vez. En caso contrario el problema es acumulativo y permite a varias actividades compartir el recurso. Otra clasificación es según se conozcan o no todas las actividades de forma previa a comenzar la asignación. Si todas las actividades son conocidas de antemano se dice que el problema es de asignación *offline*, en cambio si las actividades se conocen de forma dinámica en el tiempo, y se debe comenzar la planificación antes de conocer todas las actividades, se trata de un problema de asignación *online*. Otro criterio utilizado es la información que se tenga de las actividades, especialmente el tiempo de ejecución de las mismas, si se tiene información completa de todas las actividades el problema es *clairvoyant* (clarividente), y en caso

contrario es un problema *non-clairvoyant*.

3.3.1. Planificación de recursos computacionales

Un problema habitual en computación es el de administrar un conjunto de recursos informáticos que son utilizados para la ejecución de tareas. En nuestro trabajo estudiamos la planificación de recursos en sistemas computacionales heterogéneos, que puede ser modelado como un caso de RCPSP: donde cada recurso de cómputo corresponde a un recurso, la cantidad de unidades del recurso dependerá de si el mismo puede ser utilizado de forma compartida o no, las actividades son las tareas a ejecutar, y se desea determinar el orden de ejecución de las tareas y el recurso que se le asignará a cada una para minimizar el makespan.

En nuestro trabajo estudiamos dos casos, el primero se basó en la ejecución de tareas en un cluster de computadoras. Dado que la información de todas las tareas a ejecutar son conocidas de antemano, se trata de un problema de planificación *offline* clarividente, y el modelo utilizado es disyuntivo ya que los recursos son utilizados de forma exclusiva. El segundo caso estudia la asignación de recursos informáticos a usuarios en una sala de computación. En este caso también se conoce la información de todos los usuarios que llegarán al salón, por lo que también se trata de un modelo *offline* y clarividente. En este caso el modelo es acumulativo ya que los recursos pueden ser compartidos por varios usuarios.

En los dos casos estudiados los recursos informáticos son heterogéneos, variando su capacidad de cómputo y consumo energético. Los objetivos considerados fueron minimizar el consumo energético del sistema y mantener la calidad de servicio ofrecida. En el caso de estudio del cluster la calidad de servicio se consideró como el makespan, y en el salón de informática la calidad de servicio se midió como la sobreasignación de los recursos, una función que compara los requerimientos de los usuarios de un recurso y la capacidad máxima de este, evaluando el nivel en el que no se puede cumplir con los requerimientos de estos usuarios.

3.4. Trabajos relacionados

En esta sección se presentan trabajos relacionados a la gestión de usuarios y tareas en sistemas de computación heterogéneos. Algunos de los trabajos se enfocan en la asignación de tareas en sistemas de computación cluster o grid, otros trabajos tomaron como objeto de estudio los recursos computacionales a nivel empresarial, donde se cuenta con gran cantidad de PCs que se utilizan únicamente en una franja horaria, y otros trabajos abordan el problema desde una perspectiva más amplia.

3.4.1. Estrategias generales para disminuir el consumo energético

A continuación, se presentan algunos de los trabajos que han estudiado el problema del consumo energético de los recursos computacionales, presentando o resumiendo soluciones que atacan el problema de forma general, pudiendo aplicarse a escenarios muy variados.

En Beloglazov et al. (2011) se plantea el creciente consumo energético de los recursos computacionales como un problema importante a resolver. Siendo sus principales efectos el impacto del mismo sobre las emisiones globales de CO2, aportando el 2 % de las emisiones actuales, y el aumento de los costos operacionales de clusters y data centers, que iguala en poco tiempo al costo de adquisición. Se dividen las técnicas estudiadas en tres niveles, la programación de programas que sean eficientes, la administración de los recursos computacionales y las mejoras en el consumo de los componentes de hardware. Las principales soluciones investigadas son: el uso de escalado dinámico de frecuencia y voltaje (DVFS) y la administración inteligente del uso de los recursos en data centers mediante la administración de tareas, control de la performance ofrecida, consolidación de trabajos buscando minimizar los nodos activos, entre otras.

Orgerie et al. (2014) estudia trabajos recientes, buscando minimizar el consumo energético de grandes infraestructuras computacionales mediante la coordinación del uso de los recursos, que permite explotar de manera adecuada las características de ahorro de energía de los recursos individuales. Para recursos individuales las principales soluciones planteadas son el uso de DVFS, el uso de estados de bajo consumo para recursos idle, mejoras en la eficiencia de software, y el apagado o disminución del rendimiento de dispositivos de hardware como discos y memorias. A nivel de infraestructura, se presentan soluciones que buscan localizar los nuevos data centers en zonas donde se puede acceder a energía barata y renovable, algoritmos de scheduling que tienen en cuenta la temperatura, y disminuir la cantidad de nodos activos en la infraestructura.

Agarwal et al. (2009) presenta una novedosa solución que busca disminuir la energía consumida por computadoras, que necesitan mantener su presencia en la red, ocasionando que permanezcan activas todo el tiempo. Las principales razones para mantenerlas activas son: permitir el acceso remoto, la rápida disponibilidad o aplicaciones que corren de fondo, incluyendo descargas, torrent y compartir archivos. Para atacar este problema presenta un prototipo de una interfaz de red, que mantiene la presencia de una PC activa en la red mientras ésta se encuentra en estados de bajo consumo, lo que en las pruebas realizadas implicó ahorros en el consumo energético de entre 60 % y 80 % en escenarios realistas. Al utilizar el dispositivo se puede mantener el PC en sleep aún mientras se utilizan aplicaciones de chat, torrent y descargas web. Su implementación tiene como ventaja que no implica cambios en las aplicaciones de servidor, sistemas operativos o demás componentes del PC.

De las propuestas relevadas, en nuestro trabajo incluimos el uso de DVFS para controlar el consumo y rendimiento de los recursos, así como la planificación de usuarios y tareas, buscando mantener la mínima cantidad de nodos activos, y maximizar el uso de la infraestructura.

3.4.2. Problema de la asignación de tareas en cluster o grid

A continuación, se presentan trabajos que proponen soluciones a la planificación de tareas y la gestión de los recursos en clusters o grids, los mismos se seleccionaron por compartir objetivos con nuestra propuesta o por utilizar una metodología similar.

Kim et al. (2007) presenta dos variantes de un algoritmo que realiza el scheduling de bolsas de tareas con deadlines, donde a intervalos irregulares de tiempo arriban trabajos,

compuestos por un conjunto de tareas que comparten un deadline. Los algoritmos propuestos utilizan escalado dinámico de voltaje en los recursos disponibles para minimizar el consumo energético a la vez que cumplen con los niveles de servicio acordados. Esta técnica también es utilizada en nuestro trabajo, implementando un modelo de consumo energético similar, basado en el uso de CPU, definiendo estados de performance en la CPU, donde cada estado tiene un consumo por unidad de tiempo y ejecuta una determinada cantidad de instrucciones por segundo. Las tareas se especifican en base a la cantidad de instrucciones del procesador que implican, y junto con el estado de ejecución del procesador el consumo energético de la misma. A diferencia de nuestro trabajo, el algoritmo propuesto no conoce todos los grupos de tareas que llegarán al sistema a priori, y no consideran el consumo en idle de los recursos, cuando no se encuentran ejecutando ninguna tarea. Los algoritmos propuestos en el trabajo son deterministas, uno ejecuta una tarea por recurso y otro permite que varias tareas compartan un recurso. En las simulaciones realizadas obtuvieron mejoras de al menos un 33 % en el consumo energético, en comparación con no utilizar DFVS, teniendo una degradación en las tareas aceptadas para su ejecución del 14 %, mejorando estos resultados cuando el tiempo entre el arribo de un trabajo y el siguiente era mayor.

En el trabajo de Bridi et al. (2016) se propone una solución al problema de planificación de tareas en un sistema de HPC heterogéneo. El algoritmo propuesto está basado en la programación con restricciones, enfocado a un sistema que utiliza colas de prioridades para la ejecución de las tareas. Este tipo de sistemas se suele encontrar en aplicaciones comerciales para la gestión de las tareas en sistemas de HPC, el objetivo del algoritmo propuesto es disminuir el tiempo que las tareas esperan en las colas a ser ejecutadas, respetando las prioridades asignadas a cada cola por los administradores del sistema. El algoritmo desarrollado se evaluó tanto en ambientes simulados como en un ambiente productivo, ejecutando como un plug-in del software *PBS Professional* utilizado en sistemas de HPC, donde disminuyó el tiempo promedio de espera en las colas, manteniendo a la vez el nivel de utilización de los recursos. El principal problema del algoritmo propuesto radica en la diferencia entre el tiempo esperado de ejecución, y el tiempo real que le toma ejecutar a la tarea. Uno de los principales resultados de este trabajo radica en la prueba efectiva del mismo en un ambiente productivo, sin embargo no tiene en cuenta el consumo energético de la ejecución de las tareas.

Murana et al. (2014) ataca el problema de la planificación de tareas en sistemas heterogéneos, buscando minimizar el tiempo total de ejecución de las tareas, manteniendo a la vez los niveles de servicio acordados en forma de fechas límites para la ejecución de las tareas. Para resolver el problema plantea una versión paralela del algoritmo evolutivo CHC con una combinación lineal ponderada de los dos objetivos del problema. En su análisis experimental su utilizaron instancias de hasta 2048 tareas y 64 recursos, y obtienen mejoras del entorno del 10 % para el tiempo total de ejecución, respetando estrictamente el nivel de servicio acordado para instancias pequeñas, y con pequeñas desviaciones para las instancias más grandes. El problema resuelto en este trabajo es similar al atacado en la primera parte de nuestro trabajo, enfocándose en objetivos diferentes para las soluciones. Para resolver el problema también se optó por un algoritmo evolutivo, eligiendo en este caso CHC en lugar de NSGA-II, y los resultados obtenidos también se compararon contra la heurística MinMin.

En Iturriaga et al. (2012) los autores analizan tres algoritmos para la resolución del problema de la asignación de tareas en sistemas computacionales heterogéneos, considerando a la vez los objetivos de minimizar el tiempo total de ejecución de las tareas y la energía consumida por el sistema en el proceso. La formulación del problema es muy similar a la utilizada en nuestro trabajo, considerando una función que devuelve la energía consumida para cada dupla de tarea y recurso, y teniendo el cuenta el consumo de los recursos cuando no ejecutan ninguna tarea. Aunque en el trabajo no se consideró el uso de mecanismos adicionales para el control del consumo energético y la performance de los recursos. Para resolver el problema los autores analizaron tres algoritmos utilizando un enfoque completamente multi-objetivo aplicando el concepto de dominancia de Pareto. Proponen utilizar NSGA-II, SPEA2 y un algoritmo multi-hilo de búsqueda local (MLS) basado en poblaciones. Los tres algoritmos fueron comparados con algoritmos heurísticos deterministas, considerando instancias de hasta 2048 tareas y 64 recursos. Obtuvieron mejoras de al menos 6 % en el objetivo de energía y 10 % en makespan, para todas las instancias comparando contra el mejor algoritmo determinista. En el estudio NSGA-II y SPEA2 obtuvieron resultados mejores que MLS, pero con tiempos de ejecución 6 veces mayores.

Li et al. (2009) propone un algoritmo basado en la heurística Min-Min, para resolver el scheduling de tareas en un cluster heterogéneo, teniendo en cuenta el consumo energético y el makespan. La solución propuesta también incorpora la definición de diferentes estados energéticos en los recursos del sistema, donde el estado 0 es aquel en el que se pueden ejecutar tareas, y los estados subsiguientes son distintos estados de inactividad, cada uno con un consumo energético asociado, y un costo de transición al estado 0 medido en tiempo y energía. El algoritmo fue probado con instancias de hasta 10.000 tareas y 256 recursos, y probó escalar correctamente con el aumento del tamaño de las instancias. Comparado con Min-Min logró mejoras de hasta un 34 % en el consumo energético, sin aumentar más de un 1 % el tiempo de ejecución. En nuestro trabajo implementamos una heurística inspirada en este trabajo para incorporar el objetivo del consumo energético en el algoritmo Min-Min.

Tarplee et al. (2016) propone una solución a la asignación de tareas en un sistema de HPC, enfocándose en minimizar el makespan y el consumo energético, incluyendo el consumo de los recursos inactivos. La solución modela el caso de asignación de bolsa de tareas, en el que todas las tareas a asignar son conocidas de forma previa, y las mismas son independientes entre si. La definición general del problema es muy similar a la empleada en nuestro trabajo, con la excepción de no considerar múltiples estados de ejecución para cada recurso, y utilizar un modelo del tiempo de ejecución y consumo energético basado en datos históricos. A diferencia de nuestra aproximación, los autores tomaron la decisión de agrupar todos los recursos similares en grupos de recursos, para los que asumen todas las características son idénticas, y dividen las tareas en clases a las que se determina un tiempo de ejecución y consumo estimado para cada categoría de recurso, basados en datos históricos obtenidos de plataformas de HPC. Esta decisión les permitió resolver instancias mucho más grandes del problema. El problema es resuelto en dos instancias, primero encuentra un límite inferior para el frente de Pareto, realizando una simplificación del problema que les permite aplicar programación lineal. La simplificación se basa en

expresar el problema en cuántas tareas de cada clase se asignan a cada grupo de recursos. El problema es resuelto no en su forma entera, sino considerando los número reales, es por esto que las soluciones encontradas no son factibles, pero igualmente permiten determinar un límite inferior para el frente de Pareto. A partir de estas soluciones no factibles, el trabajo propone mecanismos eficientes para encontrar soluciones factibles, aproximando las soluciones a valores enteros que cumplan con las condiciones necesarias, y utilizan técnicas de *convex filling* para mejorar la aproximación del conjunto de soluciones al frente de Pareto. El algoritmo propuesto fue comparado con la metaheurística NSGA-II, obteniendo mejores resultados y mejores tiempos de ejecución, en instancias de hasta 3.600 recursos. En el trabajo se reportan tiempos de ejecución mucho mayores para NSGA-II, pero esta diferencia se debe en parte a que el algoritmo evolutivo resolvió directamente la asignación da cada tarea a cada recurso, sin resolver previamente la asignación global de cantidad de tareas de cada tipo a ejecutar en cada grupo de recursos.

3.4.3. Estudios de eficiencia energética en el ámbito empresarial

A continuación, se presentan algunas de las soluciones que se han propuesto para disminuir el consumo energético de los computadores utilizados en el ámbito empresarial. Presenta similitudes con el ambiente estudiado en nuestro proyecto, ya que cuenta con una gran cantidad de computadores que permanecen sin ser utilizados durante largos períodos de tiempo, pero que deben mantener una presencia activa en la red para poder ser accedidos de forma remota. Los trabajos presentados no apuntan a la asignación de recursos a usuarios, sino que se enfocan en soluciones de hardware o virtualización para disminuir el consumo energético, ya que no encontramos publicaciones que tomen un enfoque similar al presentado en este trabajo, que utilicen la distribución de los usuarios en los recursos como método para disminuir el consumo energético.

En Das et al. (2010) se ataca el problema de las computadoras en ámbitos empresariales, que permanecen idle durante largos períodos de tiempo, pero deben mantener una presencia en la red. LiteGreen es una solución alternativa al uso de proxys para mantener la presencia de una computadora en la red mientras ésta entra en modo de reposo. En lugar de configurar un proxy, se virtualiza el ambiente de trabajo, permitiendo migrar esta máquina virtual entre el equipo de escritorio y un servidor dedicado a la virtualización, dependiendo de si el equipo se encuentra idle o activo. Los estudios sobre esta solución realizados a pequeña escala en ambientes reales muestran un ahorro de energía adicional del 40 % frente a la configuración de modos de energía de Windows y controles manuales.

Bila et al. (2015) presenta una solución similar a LiteGreen al problema de minimizar la energía consumida por equipos de oficina. Se basa en la noción de que una computadora en estado sleep accede únicamente a una pequeña porción de su memoria y disco, por lo que propone realizar migraciones parciales de los equipos de escritorio a un servidor de virtualización, donde se solicitarán páginas de memoria y disco a demanda, permitiendo al equipo de escritorio permanecer en reposo. La migración parcial disminuye la latencia en la transición a activo, y el tiempo de migración. En comparación con migración total de máquinas virtuales, la solución presentada es muy competitiva, ya que permite obtener entre el 85 % y el 105 % de los ahorros energéticos, teniendo latencias de migración 2 o 3 órdenes de magnitud más pequeñas y uso de la red un orden de magnitud menor.

En Nordman and Christensen (2009) se presenta la necesidad de mantener una presencia activa en la red como la principal razón por que los equipos de oficina se mantienen encendidos 24 horas al día. Para solucionar este problema se plantean dos vías, implementar proxys y otros mecanismos de red que le permitan al equipo mantener su presencia en la red mientras está en sleep, y reeducar a los usuarios sobre las opciones de manejo de energía que presenta un PC, particularmente sobre las ventajas del modo sleep y como configurarlo correctamente. Utilizando ambas aproximaciones se espera que en un futuro se maximice el tiempo que un computador idle pasa en modo sleep, sin que esto afecte las capacidades del mismo para mantener sus funcionalidades de red. Se prevé que una vez que la utilización de proxys sea más frecuente, los protocolos de comunicación, y programas P2P, sean programas para entender su utilización y procurar no despertar de forma innecesaria a otro computador.

3.4.4. Resumen

En este capítulo se presentó el concepto de computación heterogénea, técnica utilizada en clusters de computadoras, infraestructuras grid y computación en la nube entre otros ambientes. La característica distintiva de HC es que implica el uso de recursos no homogéneos, lo que permite tener recursos especializados para distintas tareas o varias generaciones de computadores conviviendo en la infraestructura. También se presentaron algunas técnicas que suelen utilizar los recursos informáticos para controlar el consumo energético. Los C-States y P-States definidos en la interfaz del estándar de ACPI, definen diferentes estados en los que se puede encontrar un computador, cada uno de ellos con un nivel de performance y un consumo energético asociado. Los C-States, definen los estados en los que se puede encontrar un recurso ocioso, en el primer estado el recurso se encuentra activo, y en los siguientes estados el recurso apaga sus componentes progresivamente, disminuyendo su consumo y aumentando el tiempo que le toma volver a estar activo. Los P-States definen estados de performance de un recurso activo, a cada estado asocia un nivel de performance o velocidad, y su correspondiente aumento en el consumo. Estos estados permiten que un recurso no ejecute a su máximo potencial cuando no es requerido, permitiendo conservar energía.

Estos conceptos son utilizados en los trabajos relacionados presentados, los cuales fueron clasificados según su enfoque en: trabajos enfocados en sistemas de cluster o grid, sistemas enfocados a redes empresariales y sistemas que toman un enfoque más global.

Los trabajos sobre sistemas de cluster se basaron en resolver el problema de la asignación de tareas, tomando distintos casos particulares del problema y enfocándose en distintos objetivos. Los algoritmos propuestos para resolver el problema fueron variados, incluyendo algoritmos evolutivos, heurísticas multiobjetivo, y algoritmos deterministas más simples.

Los trabajos presentados sobre redes de computadoras en ambientes empresariales se enfocaron en disminuir el consumo energético de los recursos manteniendo a la vez la presencia en la red. Las soluciones propuestas son diversas, incluyendo la virtualización y migración de los recursos, y el diseño de sistemas de software que permitan al recurso permanecer inactivo por más tiempo.

Los trabajos que toman un enfoque general se basaron en la técnica de DFVS para controlar el consumo, y en el diseño de dispositivos de hardware que reemplazan el puerto de red tradicional, agregando funcionalidades que permiten al recurso permanecer inactivo y mantener actividad en la red. Además, presentan recomendaciones de carácter

general para la planificación de data centers y redes de computadoras.

Capítulo 4

Planificación de tareas en un cluster heterogéneo considerando el consumo energético

Este capítulo presenta el trabajo realizado sobre la optimización de la distribución de tareas en un cluster heterogéneo o Heterogeneous Cluster Task Scheduling (HCTS), un caso particular de HCSP. Para resolver el problema se plantea un algoritmo evolutivo multiobjetivo basado en el algoritmo NSGA-II de Deb et al. (2002), y se comparan los resultados obtenidos con MinMin una heurística multiobjetivo, presentada por Luo et al. (2007). El capítulo se organiza de la siguiente forma: primero se presenta el concepto de un cluster heterogéneo y la necesidad detrás del estudio, luego se define formalmente el problema, en la tercera sección se describe la implementación y en la cuarta se explica el proceso de evaluación experimental. Finalmente se presentan y discuten los resultados.

4.1. Cluster heterogéneo

Según Sadashiv and Kumar (2011), un cluster es un conjunto de recursos computacionales conectados entre si, que se presentan al usuario como un único recurso virtual que permite resolver tareas de gran magnitud. Es común que este conjunto de recursos sea utilizado por múltiples usuarios al mismo tiempo, por lo que es necesario realizar una asignación de estos recursos a los usuarios y sus tareas, permitiendo que los usuarios obtengan la mejor calidad de servicio posible, a la vez que se respeten las restricciones de los costos operativos. Un cluster heterogéneo es un caso particular, en el que los recursos que componen el cluster son diferentes entre si. Pueden convivir recursos viejos con nuevos, de diferente capacidad de procesamiento, diferentes costos operativos e incluso propósitos diferentes, cómo: recursos orientados al uso del procesador, al uso de disco, a cálculos que requieran tarjetas gráficas, etc.

4.2. Presentación del problema

El problema a resolver modela la siguiente realidad: en un cluster de computadoras heterogéneo se desea ejecutar un conjunto de tareas independientes (no se impone ningún orden en su ejecución). El problema de optimización consiste en realizar la asignación

de las tareas a los recursos del cluster, terminando la ejecución de las tareas en el menor tiempo posible y minimizando a su vez el consumo energético del cluster al realizar la ejecución. Además de realizar una asignación de tareas óptima, se pueden utilizar otras herramientas para regular el consumo energético a nivel del recurso individual, como puede ser: apagar componentes, entrar en estado idle disminuyendo el consumo a un mínimo, y el control de rendimiento de componentes individuales, como el disco, la CPU (unidad central de procesamiento), etc.

En nuestro modelo se considera que los recursos pueden encontrarse activos o idle, dependiendo de si se encuentra ejecutando una tarea o no. El consumo de un recurso en estado idle es menor al consumo en estado activo y se tomará como parte de la definición de las instancias. El consumo de los recursos activos podrá variar según los estados de ejecución del mismo. Utilizamos los P-State de las CPU de los recursos como herramienta para regular el nivel de performance y el consumo energético de un recurso activo, presentados en la página 29. La cantidad de estados que se utilizan es parametrizable, por lo que nuestro modelo también es aplicable a aquellas realidades que no utilicen diferentes niveles de rendimiento, considerándose como un único estado posible.

El modelo de consumo energético de los recursos se basa en los distintos niveles de performance que éstos ofrecen, cada uno asociado con una cantidad de operaciones que puede ejecutar por segundo y un consumo energético por segundo. Los datos específicos de esos valores se tomarán de la definición de la instancia. Las tareas a ejecutar son modeladas como una cantidad de operaciones de CPU, se asume que la cantidad de operaciones necesarias para completar la tarea no varía de un recurso a otro. Las tareas no incluyen datos de requerimientos de red, memoria o disco, se asumirá que estos recursos permanecen activos durante la ejecución de la tarea, y la variación en el consumo de estos recursos se deberá modelar en los distintos niveles de performance ofrecidos. No fueron considerados recursos con procesadores gráficos o GPU (Graphics Processor Unit).

Dado que un recurso que se encuentre en un estado de rendimiento reducido ejecuta menos operaciones por segundo, le tomará más tiempo ejecutar la misma tarea, aunque consumirá menos energía por unidad de tiempo en el proceso. Esto genera un conflicto entre los dos objetivos del problema, el tiempo total de ejecución y el consumo energético. Para contemplar a la vez ambos objetivos sin preponderar uno sobre el otro, se debe emplear un algoritmo multiobjetivo.

4.2.1. Formulación matemática del problema

Para desarrollar una solución al problema se realizó un modelo de la realidad. En este todas las tareas a ejecutar son independientes, se encuentran disponibles desde el comienzo de la planificación y no tienen una fecha límite para completar su ejecución. Cada tarea requiere de un único recurso de cómputo y un recurso no puede ejecutar más de una tarea a la vez. Se asume también que todos los recursos poseen la misma cantidad de estados, y la transición de estados en un recurso es instantánea y no consume energía. El estado de ejecución de un recurso permanecerá constante durante la ejecución de una tarea.

En el modelo se definen las siguientes variables:

- Se cuenta con M recursos informáticos $R = r_1 \cdots r_M$
- Se cuenta con N tareas independientes $T = t_1 \cdots t_N$
- Se cuenta con S estados posibles para los recursos $P = p_1 \cdots p_S$

Para representar las soluciones se utiliza un arreglo de vectores $A = [a_1 \dots a_M]$, donde a cada recurso le corresponde un vector de tuplas $a_j = [(t_i, p_s)]$, indicando que la tarea t_i se ejecutará en el recurso r_j en el estado p_s. Las tareas se ejecutarán en el mismo orden del vector.

Para la ejecución del algoritmo se asume contar con la información para poder implementar las siguientes funciones, la función *tiempo* : $(t_i, r_j, p_s) \longrightarrow \overline{T}$, que retorna el tiempo que tomaría ejecutar la tarea t_i en el recurso r_j en el estado p_s. La función *energia* : $(r_j, p_s) \longrightarrow \overline{E}$, que devuelve la energía consumida por unidad de tiempo por el recurso r_j en el estado p_s. Y la función *energia idle* : $(r_j) \longrightarrow \overline{EI}$ que retorna la energía consumida por el recurso r_j en estado idle.

Objetivos del problema

Se busca minimizar el makespan (Mk) y la energía consumida (Ec). Para poder definir el makespan, definiremos el tiempo total de ejecución del recurso j, representado como TE_j como la suma de los tiempos de ejecución de todas las tareas asignadas a dicho recurso, ver ecuación 4.1. El makespan se define como el máximo de los tiempos de ejecución de todos los recursos considerados, ver ecuación 4.2.

$$a_j \in A$$

$$TE_j = \sum_{(t_i,p_s)\in a_j} tiempo\,(t_i, r_j, p_s) \tag{4.1}$$

$$Mk = \max_{r_j \in R}(TE_j) \tag{4.2}$$

La energía consumida representa la suma de la energía consumida por cada uno de los recursos, ver ecuación 4.5. La energía consumida por el recurso j se divide en la energía consumida en la ejecución de las tareas ETr_j, más la energía consumida mientras se encontraba en idle EIr_j, calculada en base al consumo en idle del recurso y la diferencia entre el tiempo de ejecución del recurso y el makespan, ver ecuaciones 4.3 y 4.4.

$$r_j \in R$$

$$ETr_j = \sum_{(t_i,p_s)\in a_j} (energia(r_j, p_s) \times tiempo(t_i, r_j, p_s)) \tag{4.3}$$

$$EIr_j = energia\ idle(r_j) \times (Mk - TE_j) \tag{4.4}$$

$$Ec = \sum_{r_j \in R} (ETr_j + EIr_j) \tag{4.5}$$

4.3. Implementación

Según Ullman (1975) los problemas de planificación tradicional son NP-Hard, por lo que los algoritmos exactos pueden resultar poco prácticos a medida que aumenta el tamaño de las instancias consideradas, debido a elevados tiempos de ejecución. Es por esto que optamos por resolver el problema utilizando una metaheurística, basada en el algoritmo evolutivo NSGA-II.

4.3.1. Biblioteca de desarrollo

Para la implementación del algoritmo evolutivo se utilizó el framework JMetal 4.5 implementado en Java propuesto por Durillo et al. (2010). El desarrollo del framework es parte de un proyecto open source cuyo objetivo es brindar a los usuarios una herramienta flexible, extensible y fácil de usar, que simplifique el proceso de realizar investigaciones con metaheurísticas para problemas multiobjetivo.

En la presentación Durillo et al. (2011) destacaron que el diseño del framework se enfoca en la simplicidad y facilidad de uso, lo que permite utilizar el mismo con un mínimo tiempo de aprendizaje. También es importante la flexibilidad de ejecutar algoritmos diversos, pudiendo ingresar parámetros genéricos o específicos de cada situación, y cada variación que se realice sobre un algoritmo, como cambiar el tipo de una codificación, debe tener un mínimo impacto. Al haber sido desarrollado en Java la herramienta puede ser ejecutada en distintos sistemas operativos y arquitecturas. Por último, también es muy importante que la herramienta pueda ser extendida, para incorporar nuevos algoritmos, problemas y operadores que no hayan sido previstos en su desarrollo inicial.

En el framework se incluye la implementación de numerosos problemas, heurísticas, operadores evolutivos, indicadores de calidad, y funciones utilitarias que facilitan la ejecución de estudios comparativos, automatizando el proceso de realizar múltiples ejecuciones con distintos problemas o parámetros y compilar los resultados para su comparación.

4.3.2. Codificación de las soluciones

La información de las soluciones se codificó utilizando dos cromosomas, uno conteniendo la asignación de las tareas a los recursos y el otro conteniendo la información del estado en el que se ejecutará cada tarea en el recurso.

El estado se representó mediante un arreglo de enteros de largo N (cantidad de tareas), donde cada espacio del arreglo lo ocupa un entero entre 0 y $P-1$, siendo P la cantidad de estados disponibles, donde un valor p_i en la posición n implica que la tarea t_n se ejecutará en el estado p_i.
La asignación de tareas se codificó mediante una permutación de los valores enteros $0 \ldots (N+M-2)$, siendo M la cantidad de recursos. En esta permutación los valores entre 0 y $N-1$ representan una tarea, y los valores entre N y $N+M-2$ representan a los recursos del sistema, exceptuando el primero que tiene una representación implícita en la codificación.

En la permutación se encontrarán intercalados los valores que representan tareas y aquellos que representan recursos, asumiendo una posición virtual al comienzo de la permutación que siempre se encuentra ocupada por el primer recurso r_0. Todas las tareas que se encuentren entre un valor de un recurso r_i y otro r_j, serán asignadas para su ejecución al recurso que aparece primero r_i.

Por la forma en la que se estructuró la codificación no existen codificaciones que no representen una solución ni viceversa, por lo que no es necesario realizar correcciones en los operadores de mutación y cruzamiento.

Se presenta a continuación un ejemplo de la codificación:
Suponiendo el conjunto de tareas $T = [t_0, t_1, t_2, t_3, t_4, t_5]$, el conjunto de recursos $R = [r_0, r_1, r_2]$ y el conjunto de estados $P = [p_0, p_1]$, una posible solución formada por los cromosomas $A = [t_2, t_3, r_1, t_4, t_5, r_2, t_0, t_1]$ representando la asignación de tareas y $B = [p_0, p_1, p_1, p_0, p_1, p_1]$ representando los estados de ejecución y corresponden a la asignación

de tareas representada en la figura 4.1 y que se detalla a continuación:

- $r_0 = [(t_2, p_1), (t_3, p_0)]$
- $r_1 = [(t_4, p_1), (t_5, p_1)]$
- $r_2 = [(t_0, p_0), (t_1, p_1)]$

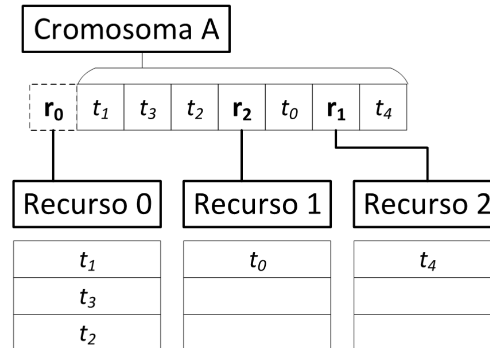


Figura 4.1: Representación de uno de los cromosomas y la asignación de tareas que representa

4.3.3. Metaheurística utilizada

Para la resolución de nuestro problema utilizamos el esqueleto de NSGA-II tomado del libro de Coello et al. (2007), debiendo utilizar algunos operadores específicos a nuestra representación mediante variables de tipo permutación y arreglo de enteros. Para mejorar los resultados obtenidos se implementó también un algoritmo de inicialización para crear el conjunto de soluciones iniciales.

Inicialización de la población Durante el desarrollo del algoritmo realizamos pruebas con dos tipos de inicialización, una en la cuál se asignan valores aleatorios a las variables que componen la solución, y otra que mediante múltiples ejecuciones de un algoritmo greedy mono objetivo, centrado en optimizar el makespan, genera un conjunto de soluciones. Un algoritmo greedy toma decisiones localmente óptimas, con la idea de llegar a un óptimo global del problema. El algoritmo propuesto para la inicialización, repite el siguiente proceso para cada solución que debe crear: toma las tareas de una en una en orden aleatorio, para cada tarea calcula el makespan que resultaría de la asignación a cada uno de los recursos, y elije le recurso que implique el menor aumento del makespan global. El procedimiento del algoritmo es similar al que una persona podría realizar de forma intuitiva. Se presenta el pseudocódigo en el Algoritmo 6.

Operador de Selección Para el operador de selección de NSGA-II utilizamos el algoritmo torneo binario, que se basa en el concepto de dominancia y distancia de crowding, presentado en la página 21.

Algoritmo 6 Algoritmo de inicialización de la población

```
function INICIALIZACIÓN(TamañoPoblación, Recursos, Tareas)
    Soluciones = CREARCONJUNTOVACIO(TamañoPoblación)
    for all Sol ∈ Soluciones do
        Tasignar = COPIAR(Tareas)
        while NoEsVacio(Tasignar) do
            t = SACARALAZAR(Tasignar)
            MejorMakespan = ∞
            for all r_i ∈ Recursos do
                m = CALCULARMAKESPAN(t, r_i)
                if m < MejorMakespan then
                    r = r_i
                    MejorMakespan = m
                end if
            end for
            Sol.ASIGNAR(t, r)
        end while
    end for
    return Conjunto de soluciones
end function
```

Operador de Mutación Implementamos un nuevo operador de mutación que sea compatible con nuestra solución. El operador implementado utiliza dos algoritmos de mutación, uno para cada tipo de variable. Utiliza el operador *BitFlipMutation* para la variable de tipo arreglo de enteros, y *SwapMutation* para la variable de tipo permutación. *BitFlipMutation* recorre el arreglo de valores y para un conjunto aleatorio de ellos cambia el valor por otro elegido al azar. *SwapMutation* elige dos posiciones al azar de la permutación e intercambia el valor de una por el de otra. Este operador recibe como parámetro la probabilidad de mutación, que es utilizada para determinar que valores de la variable serán cambiados. La solución mutada se conforma por las dos variables luego de haber sido mutadas.

Operador de Cruzamiento Para el cruzamiento debido a que los operadores provistos para los tipos de variables utilizados resultaban muy disruptivos, implementamos un operador específico a nuestra codificación, llamado HCTCrossover. El operador es similar a half uniform crossover (HUX) descrito en el artículo de Bäck et al. (2000).

Recibe como entrada dos soluciones padres, una probabilidad de cruzamiento, y devuelve dos soluciones hijas. En base a la probabilidad de cruzamiento se define si los hijos serán versiones sin modificar de sus padres, o se crearán nuevas soluciones utilizando el siguiente procedimiento: Para cada una de las tareas uno de los hijos tomará el recurso y el estado de ejecución al que se había asignado esa tarea en uno de los padres, y el otro tomará el recurso que se había asignado en el otro padre, la decisión se realiza mediante un sorteo aleatorio que da igual probabilidad a ambos padres. Se presenta el pseudocódigo en el Algoritmo 7.

Criterio de parada El criterio de parada determina cuando un algoritmo debe detener su ejecución. Nosotros utilizamos para esto la cantidad de evaluaciones realizadas, luego

Algoritmo 7 Pseudocódigo de HCTCrossover

```
function HCTCrossover(Probabilidad, Padre1, Padre2)
    hijo1 = Padre1
    hijo2 = Padre2
    if Random() < Probabilidad then
        for t_i ∈ tareas do
            r = Random()
            if r < 0,5 then
                (r1, e1) = obtener asignacion(Padre1, t_i)
                (r2, e2) = obtener asignacion(Padre2, t_i)
            else
                (r1, e1) = obtener asignacion(Padre2, t_i)
                (r2, e2) = obtener asignacion(Padre1, t_i)
            end if
            asignar(hijo1, r1, e1)
            asignar(hijo2, r2, e2)
        end for
    end if
    return hijo1, hijo2
end function
```

de haber evaluado 250.000 soluciones, el algoritmo se detendrá y devolverá el conjunto de soluciones que conformen la población en ese momento.

Parámetros y salida del algoritmo

El algoritmo implementado requiere para su ejecución la definición de las instancias del problema, así como los parámetros de los operadores evolutivos. Las instancias se conforman por el conjunto de tareas a ejecutar, y la definición de los recursos computacionales a los que se asignarán las tareas.

El archivo que define las tareas contiene una línea por cada tarea, indica el identificador de la tarea y la cantidad de operaciones que conlleva la tarea como números enteros. El archivo que define los recursos contiene una línea por cada recurso, se indica el identificador del recurso, el consumo en idle del mismo, y el consumo energético y la cantidad de operaciones por segundo, para cada uno de los posibles estados de ejecución definidos.
Además de la ruta de los archivos que definen la instancia, el algoritmo recibe la ruta dónde escribe los resultados y los parámetros a utilizar en los operadores evolutivos.

El algoritmo retorna un conjunto de soluciones, cada una de ellas indicando en qué recurso y en qué estado se debe ejecutar cada tarea, y los valores de las funciones objetivo para cada una de ellas.

4.4. Evaluación experimental

En esta sección se presenta la definición de las instancias del problema de la asignación de tareas en un cluster heterogéneo y los estudios realizados sobre nuestro algoritmo. Se realizó un estudio de ajuste paramétrico para determinar el conjunto óptimo

de parámetros para nuestro algoritmo. Se estudió el efecto de un algoritmo de inicialización específico para nuestra solución, y finalmente se comparó nuestro algoritmo con una heurística basada en MinMin. Para realizar los estudios se definieron cuatro instancias del problema de diferentes dimensiones.

4.4.1. Definición de las instancias

Con el objetivo de realizar la evaluación experimental de nuestro algoritmo, construimos un conjunto de instancias del problema HCTS, compuestas por dos conjuntos de datos, uno de las tareas a ejecutar y el otro con los recursos dónde se ejecutarán. Se generaron 2 conjuntos de recursos, uno con 10 y otro con 20, y 3 conjuntos de tareas con 500, 1000 y 2000 tareas cada uno. Estos conjuntos se combinaron de forma de obtener 4 instancias, con diferente cantidad de recursos y tareas (recursos$\times$tareas) 10×500, 10×1000, 20×1000 y 20×2000.

Datos de recursos

Para definir este conjunto de datos nos basamos en los recursos informáticos presentes en el Cluster FING, la plataforma de Computación Científica de Alto Desempeño de la Universidad de la República, Uruguay, detalles disponibles en www.fing.edu.uy/cluster (Nesmachnow, 2010).

Para construir los datos, primero recabamos información de los servidores que componen el Cluster FING, de cada uno se buscó sus componentes, en particular: cantidad de sockets, procesadores, cantidad de memoria, tipos de discos, etc. Las configuraciones varían entre servidores de 2 y 4 procesadores Intel Xeon E5 y AMD Opteron serie 6000, con 24 o 48 GB de memoria RAM y discos rígidos.

Con esta información buscamos servidores similares en la web SPEC (2015), cuyo objetivo es el de proveer y mantener un conjunto de pruebas de benchmark aplicables a sistemas de alto rendimiento, donde la web mantiene los resultados de estas pruebas provistos por miembros de la organización. Estos resultados contienen valores detallados de rendimiento y consumo energético de sistemas con distintas arquitecturas y componentes. En particular detalla para distintos niveles de performance la cantidad de operaciones, de un programa Java, que el sistema puede ejecutar por segundo junto con el consumo energético del sistema con esa carga.

Utilizando estos datos armamos un conjunto de datos, conteniendo la información del consumo energético en Idle, y el consumo energético y la cantidad de operaciones para distintos niveles de performance, para un conjunto de 10 recursos similares a los que se encuentran en el Cluster FING. Definimos 4 estados de ejecución correspondientes al 100 %, 80 %, 60 % y 40 % del nivel de rendimiento máximo del sistema. La instancia de 20 recursos se definió como un caso en el que hubiese dos de cada uno de los recursos de la primer instancia.

Con el objetivo de validar la información extraída, utilizamos un medidor de consumo conectado a la fuente de uno de los servidores del Cluster FING. Simultáneamente ejecutamos una rutina que calcula la transformación discreta de Fourier, operación orientada al consumo de CPU, con bajos requerimientos de disco y memoria. Esto nos permitió utilizar el 100 % de la CPU sin vernos restringidos por otros componentes de hardware, utilizando la implementación de la biblioteca FFTW disponible en la web de FFTW Org (2015). Realizamos la ejecución de forma que utilice progresivamente más núcleos de

CPU, y utilizando las medidas de consumo durante la ejecución de la rutina generamos los datos de relación entre uso de CPU y consumo energético.

Datos de tareas

Para la creación de los conjuntos de tareas se implementó un generador de tareas ficticias, cada tarea es definida con un identificador y la cantidad de operaciones de CPU que conlleva. El generador toma como parámetros la cantidad de tareas que se debe generar, un valor de referencia que será el promedio de la de cantidad de operaciones por tarea, y los porcentajes de tareas que tendrán más y menos operaciones que el valor de referencia. En base a estos parámetros, se genera un conjunto de tareas y escribe los resultados en un archivo.

Para la generación de las instancias se tomó en cuenta el promedio de operaciones por segundo de los recursos, y el tiempo de ejecución deseado de las tareas. En base a esto y buscando mantener los números en la solución en el rango de precisión de double, se definió que cada tarea tuviera una cantidad de operaciones equivalentes a un tiempo de ejecución de 5 minutos, con un 10 % de tareas con valores superiores, y un 10 % inferiores.

4.4.2. Metodología de la evaluación experimental

En esta sección se presentan los lineamientos seguidos en la evaluación experimental del algoritmo implementado. Los estudios fueron ejecutados utilizando de forma compartida los PC de las salas de máquinas de la Facultad de Ingeniería, Universidad de la República.

Debido a la cualidad estocástica de los algoritmos evolutivos, es posible obtener distintos resultados para diferentes ejecuciones de la misma instancia de un problema. Por lo tanto para poder obtener resultados estadísticamente significativos, es necesario realizar varias ejecuciones del algoritmo para cada instancia. En los estudios realizamos 30 ejecuciones independientes de cada algoritmo e instancia del problema. Para lograr la independencia de las ejecuciones se inicializó el generador de números aleatorios con diferentes semillas cada vez.

Para la comparación con algoritmos deterministas, los resultados se reportan tomando: el valor promedio y la desviación estándar de los valores alcanzados en las 30 ejecuciones independientes realizadas, de las soluciones que se comporten mejor para cada uno de los objetivos, y la solución de compromiso.

Para la comparación con variantes del algoritmo evolutivo, se tomaron en cuenta varias métricas propias de problemas multiobjetivo, en particular consideramos: Hypervolumen (HV), Spread, Epsilon, y la cantidad de soluciones en el frente de Pareto. Para comparar los valores obtenidos en estas métricas se utilizó el test de Friedman, test estadístico no paramétrico presentado en el artículo de Friedman (1937), con el cuál se elaboró un ranking para las métricas consideradas.

Para poder utilizar la media de los valores obtenidos para cada uno de los objetivos, se utilizaron tests estadísticos sobre las distribuciones de resultados obtenidos en las ejecuciones independientes de cada instancia. Se aplica el test de Shapiro–Wilk, presentado por Shapiro and Wilk (1965), sobre cada conjunto de resultados, para determinar si dicha muestra sigue o no una distribución normal. El test de Shapiro–Wilk plantea como hipótesis nula que el conjunto de valores estudiados proviene de una distribución normal. Si el p–valor obtenido es menor al nivel de confianza, se puede rechazar la hipótesis nula

y concluir que no sigue una distribución normal. A lo largo del trabajo se utiliza un nivel de confianza del 95 % ($\alpha = 0,05$) para el test de Shapiro–Wilk.

4.4.3. Comparación de diferentes métodos de inicilización

Durante el desarrollo del algoritmo realizamos un estudio para determinar si la utilización de un algoritmo específico a nuestro problema resultaría en mejores soluciones, frente a utilizar una inicialización aleatoria. Implementamos el algoritmo greedy mono objetivo descrito anteriormente, que tiene en cuenta únicamente el makespan de la solución. En el estudio definimos dos variaciones del algoritmo, uno que utilizó este algoritmo para generar los individuos de la población inicial, y otro que los generó de forma aleatoria.

Para realizar la comparación utilizamos una instancia de 500 tareas y 10 recursos. Para determinar cual de los algoritmos de inicialización permitía obtener los mejores resultados, se consideraron las métricas Hypervolumen, Spread, Epsilon y cantidad de puntos en el frente de Pareto. Se presentan además los valores promedios obtenidos para los objetivos de las soluciones: con mejor valor de makespan, con mejor valor de energía, y la solución de compromiso, aquella que se encuentre más cercana a la solución ideal tal como propusieron Rodríguez Villalobos and Coello (2012), definida como la solución ficticia conformada por los valores mínimos obtenidos para cada uno de los objetivos.

En el cuadro 4.1 se presentan los valores para las métricas estudiadas, y se puede observar como el algoritmo con la inicialización greedy obtiene mejores valores en todos los casos. Comparando los resultados obtenidos para los dos objetivos del algoritmo, se pudo observar que los valores obtenidos utilizando la inicialización aleatoria, son entre un 100 % y un 150 % peores respecto a la inicialización greedy.

A partir de los resultados obtenidos en este análisis se decidió utilizar la inicialización mediante el algoritmo greedy en las siguientes evaluaciones experimentales.

Métrica	**Tipo de Inicialización**	
	Greedy	**Aleatoria**
Cant. sol en FP	27	0
Hypervolumen	0.398 ± 0.0425	0.0 ± 0
Spread	0.710 ± 0.0828	0.930 ± 0.0404
Epsilon	1078 ± 309	9772 ± 2355

Cuadro 4.1: Valores para las métricas consideradas en la comparación del algoritmo de inicialización greedy y aleatorio, para la instancia de 500 tareas y 10 recursos, considerando 30 ejecuciones independientes. Para cada metrica se presenta el mejor valor resaltado, y en caso de que corresponda la desviación estandar.

4.4.4. Ajuste paramétrico

Para obtener los mejores resultados con nuestro algoritmo realizamos un estudio de ajuste paramétrico previo a la ejecución de los estudios comparativos. El estudio de ajuste fue realizado sobre una instancia diferente para evitar obtener resultados sesgados en los próximos estudios.

El análisis se realizó sobre los siguientes parámetros: el tamaño de la población ($\#P$), la probabilidad de cruzamiento (p_C) y la probabilidad de mutación (p_M). Los valores

considerados para cada parámetro fueron los siguientes: $\#P \in (50, 100, 150)$, $p_C \in (0.4, 0.6\ 0.8)$ y $p_M \in (0.01, 0.05, 0.1)$. Se definieron 27 algoritmos distintos, uno por cada combinación posible de los parámetros considerados, para cada uno de los algoritmos se realizaron 20 ejecuciones independientes.

Al tratarse de un problema multiobjetivo, para seleccionar la mejor combinación de parámetros no basta con elegir el que reporte el mejor resultado para uno de los objetivos, sino que se deben considerar métricas que representen la variedad de las soluciones reportadas y que tanto se acerquen al frente de Pareto. Para la comparación se utilizarán las métricas: Epsilon, Hypervolumen y Spread. Para cada una de las métricas se utilizó el test de Friedman para realizar un ranking de los algoritmos. Finalmente se calculó la posición promedio de cada algoritmo en los rankings de las distintas métricas, y se eligió la combinación de parámetros que produjo el algoritmo con un ranking promedio más bajo.

En la tabla 4.2 se presentan los resultados del estudio paramétrico, a partir del análisis de estos datos se eligió la combinación del algoritmo **NSGAII 13** con valores: probabilidad de cruzamiento **0.6**, probabilidad de mutación **0.1** y tamaño de población **50**.

4.4.5. Comparación con heurística MinMin

Se desarrolló una heurística greedy basada en el algoritmo MinMin presentado en el trabajo de Luo et al. (2007), que considera los objetivos de makespan y consumo energético. El algoritmo recibe como parámetros un conjunto de tareas y la lista de recursos disponibles para la asignación de las mismas. Para calcular el makespan y el consumo energético realiza el mismo procedimiento que el algoritmo evolutivo presentado, utilizando para ello la cantidad de operaciones de CPU que conlleva cada tarea, y la cantidad de operaciones que puede ejecutar en un segundo cada uno de los recursos, así como el consumo energético asociado a cada uno de los recursos. Para simplificar el desarrollo de la heurística sta considera únicamente un estado de ejecución por recurso, tomando aquel que permita ejecutar la mayor cantidad de operaciones por segundo.

El algoritmo repite el siguiente procedimiento mientras haya tareas que no tengan un recurso asignado: para cada tarea que aún no haya sido asignada determina el recurso que minimiza el makespan global, y realiza la asignación del conjunto de tarea y recurso que implique el menor aumento en el consumo energético. Se presenta el pseudocódigo en el Algoritmo 8

Para realizar la comparación de los dos algoritmos ejecutamos ambos sobre las cuatro instancias definidas previamente. Debido a la naturaleza estocástica de nuestro algoritmo, se realizaron 30 ejecuciones independientes para cada una de las instancias, y se realizó el test de Shapiro-Wilk, comprobando la hipótesis de normalidad para todos los casos. Por lo que se tomó la media y la desviación estándar de los valores considerados para comparar los resultados.

Para cada una de las instancias comparamos la solución obtenida al ejecutar el algoritmo MinMin, con las soluciones más destacadas obtenidas del algoritmo evolutivo. En particular comparamos: el promedio de las mejores soluciones en cada ejecución para ambos objetivos, y el promedio de la solución de compromiso de cada ejecución.

#p	p_C	p_M	**Algoritmo**	**HV**	**Spread**	**Epsilon**
50	0.6	0.05	NSGAII 0	10 %	0 %	5 %
		0.1	NSGAII 1	7 %	3 %	0 %
		0.01	NSGAII 2	5 %	4 %	2 %
	0.8	0.05	NSGAII 3	8 %	3 %	3 %
		0.1	NSGAII 4	5 %	3 %	2 %
		0.01	NSGAII 5	8 %	3 %	6 %
	0.4	0.05	NSGAII 6	8 %	1 %	4 %
		0.1	NSGAII 7	7 %	2 %	1 %
		0.01	NSGAII 8	8 %	7 %	5 %
100	0.6	0.05	NSGAII 9	3 %	4 %	6 %
		0.1	NSGAII 10	4 %	2 %	4 %
		0.01	NSGAII 11	0 %	3 %	2 %
	0.8	0.05	NSGAII 12	8 %	4 %	8 %
		0.1	NSGAII 13	6 %	0 %	1 %
		0.01	NSGAII 14	6 %	6 %	4 %
	0.4	0.05	NSGAII 15	10 %	2 %	0 %
		0.1	NSGAII 16	8 %	4 %	8 %
		0.01	NSGAII 17	9 %	0 %	3 %
150	0.6	0.05	NSGAII 18	1 %	6 %	2 %
		0.1	NSGAII 19	3 %	6 %	5 %
		0.01	NSGAII 20	10 %	4 %	1 %
	0.8	0.05	NSGAII 21	9 %	1 %	9 %
		0.1	NSGAII 22	10 %	2 %	1 %
		0.01	NSGAII 23	2 %	2 %	4 %
	0.4	0.05	NSGAII 24	5 %	1 %	3 %
		0.1	NSGAII 25	4 %	3 %	4 %
		0.01	NSGAII 26	8 %	6 %	4 %

Cuadro 4.2: Resultados del análisis paramétrico. Se presentan los parámetros evaluados, y para cada métrica la diferencia relativa al mejor valor de la misma. Se destacaron los valores tomados como referencia y el algoritmo elegido luego del análisis. Se consideró una instancia de 500 tareas y 10 recursos, realizando 20 ejecuciones independientes de cada algoritmo.

Algoritmo 8 Algoritmo MinMin

```
function MinMin(Tasks, Resources)
    U = Tasks                                   ▷ Set de tareas sin asignar
    P = Resources
    while U ≠ null do
        for all taskt_k ∈ U do
            for all machinem_h ∈ P do
                evaluate makespan(t_k, m_h)
            end for
            Store best pair (t_k, m_h)
        end for
        Select pair (t_k, m_h) with minimum Energy
        Assign t_k to m_h
        Remove t_k from U
    end while
    return task assignment
end function
```

4.5. Resultados

El cuadro 4.3 presenta los principales resultados obtenidos de comparar el algoritmo propuesto con la heurística MinMin, resolviendo las cuatro instancias del problema presentadas anteriormente. A partir de los resultados, se puede observar como el algoritmo evolutivo permite obtener soluciones que sean mejores para un objetivo específico, sin embargo, no logró superar al algoritmo MinMin en ambos objetivos a la vez.

Las soluciones del AE que se enfocaron en mejorar el Makespan obtuvieron mejores resultados que MinMin, implicando un aumento del consumo energético ligeramente mayor, menos de un 1 % por encima, y obteniendo en cambio una mejora del makespan de entre un 1 % y 2 %.

Las soluciones del AE que se enfocaron en mejorar el consumo energético, obtuvieron mejoras de entre un 5 % y 10 % frente a MinMin, pero estas mejoras implicaron un aumento del makespan de entre 25 % y 30 %. Las mejoras más importantes se dieron para las instancias de 500 y 1000 tareas, para 10 recursos.

Para la evaluación del algoritmo evolutivo se construyó un frente de Pareto empírico, conformado por el conjunto de todas las soluciones no dominadas obtenidas en todas las ejecuciones del algoritmo para una misma instancia del problema. En la figura 4.2 se presenta el frente de Pareto para la instancia de 500 tareas y 10 recursos, junto con la solución obtenida por el algoritmo MinMin. En el gráfico se puede ver claramente el abanico de soluciones ofrecidas por nuestro algoritmo, permitiendo a un tomador de decisiones elegir entre ellas según las necesidades del momento.

El consumo energético de cada uno de los nodos se puede dividir en: consumo útil y consumo idle, el primero corresponde a la energía utilizada por el recurso mientras ejecuta tareas y el segundo a la energía consumida cuando el recurso no realizaba ninguna tarea. En nuestro modelo, al contarse con un grupo de tareas inicial y no aceptarse nuevas tareas hasta haber terminado de ejecutar la última del conjunto, el principal causante del consumo idle corresponde a la diferencia de los makespan individuales de los nodos.

Al analizar los makespan individuales de cada uno de los nodos, así como el consumo

Recursos×Tareas	Algoritmo	Solución	Makespan	Energía
10×500	MinMin	-	1,6 %	9,5 %
	AE	Compromiso	15,7 %	2,4 %
		Mejor Makespan	0,0 %	10,2 %
		Mejor Energy	26,1 %	0,0 %
10×1000	MinMin	-	0,8 %	8,3 %
	AE	Compromiso	16,4 %	2,8 %
		Mejor Makespan	0,0 %	8,9 %
		Mejor Energy	29,2 %	0,0 %
20×1000	MinMin	-	0,9 %	5,7 %
	AE	Compromiso	25,5 %	1,0 %
		Mejor Makespan	0,0 %	5,9 %
		Mejor Energy	35,4 %	0,0 %
20×2000	MinMin	-	0,5 %	4,2 %
	AE	Compromiso	22,0 %	1,2 %
		Mejor Makespan	0,0 %	4,4 %
		Mejor Energy	32,0 %	0,0 %

Cuadro 4.3: Se presentan las diferencias relativas respecto al mejor valor obtenido para cada instancia. Para el algoritmo evolutivo se tomaron los valores promedios de las soluciones consideradas para realizar la comparación. Para cada instancia del AE se realizaron 30 ejecuciones independientes. Se resalta el mejor valor para cada instancia del problema.

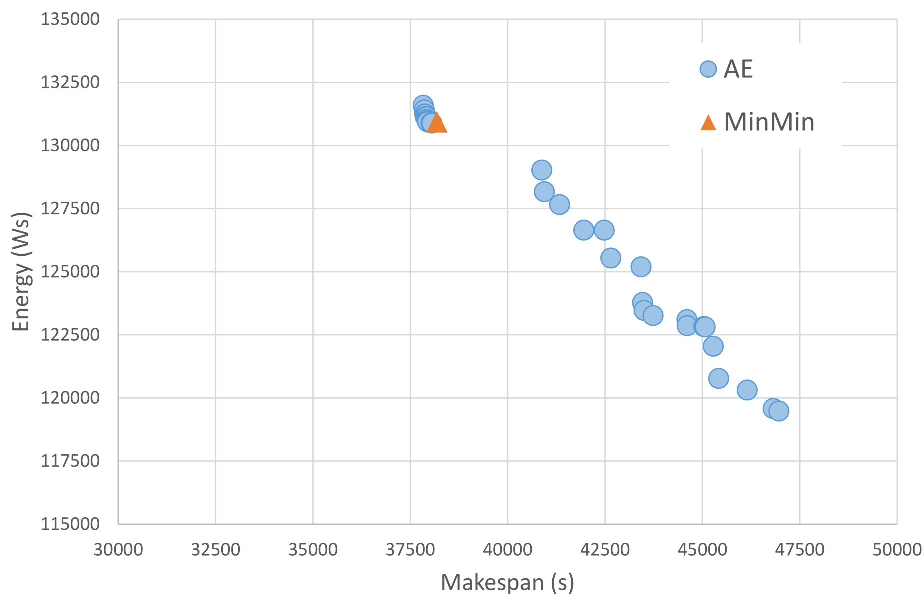


Figura 4.2: Frente de Pareto calculado por nuestro algoritmo junto con la solución del algoritmo MinMin, para la instancia de 1.000 tareas y 10 recursos

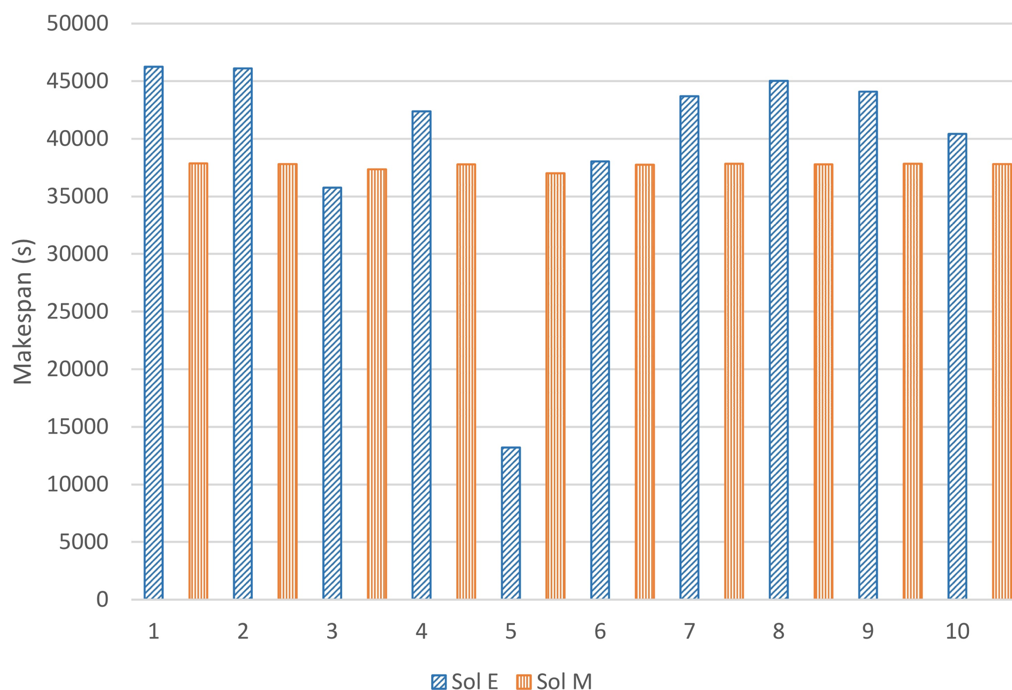


Figura 4.3: Valor de makespan individual de cada nodo para dos soluciones del Algoritmo Evolutivo en la instancia de 1.000 tareas y 10 recursos. *Sol E* corresponde a la solución con mejor valor para el objetivo energía, y *Sol M* para el objetivo makespan.

energético que implicaba cada uno de los nodos, hayamos que si la diferencia en el tiempo que le lleva a los nodos terminar de ejecutar sus tareas es muy grande, hay una porción del consumo energético que corresponderá al nodo en estado idle, o energía que se consumió sin realizar ninguna tarea útil. En nuestro modelo, que se basa en una bolsa de tareas inicial, no se realizan nuevas asignaciones hasta no haber terminado de ejecutar todas las tareas asignadas. Por lo que los elementos que tienen una mayor influencia sobre el consumo idle son: la diferencia en los makespan individuales, y la cantidad de energía por unidad de tiempo que consume cada recurso en estado idle. En otro posible enfoque, una vez terminada la asignación de las tareas en la bolsa, se podrían asignar nuevas tareas a los nodos que hayan terminado primero su ejecución, disminuyendo así el tiempo que estos permanecen ociosos, o eventualmente pasando los nodos sin utilizar a un estado de suspensión o apagado para ahorrar energía.

Para comparar el comportamiento de distintas soluciones se presenta la Figura 4.3, donde se puede observar la diferencia entre dos soluciones del algoritmo evolutivo, una con el mejor valor de makespan (*Sol M*) y otra con el mejor valor de consumo energético (*Sol E*). En *Sol M* el algoritmo buscó minimizar el makespan, sin tener en cuenta el posible beneficio que podría traer dejar un nodo en estado idle durante mayor tiempo para ejecutar tareas en un nodo que sea más eficiente, por lo que las tareas se repartieron equitativamente entre todos los nodos.

Analizando el comportamiento del consumo energético de las soluciones de ambos

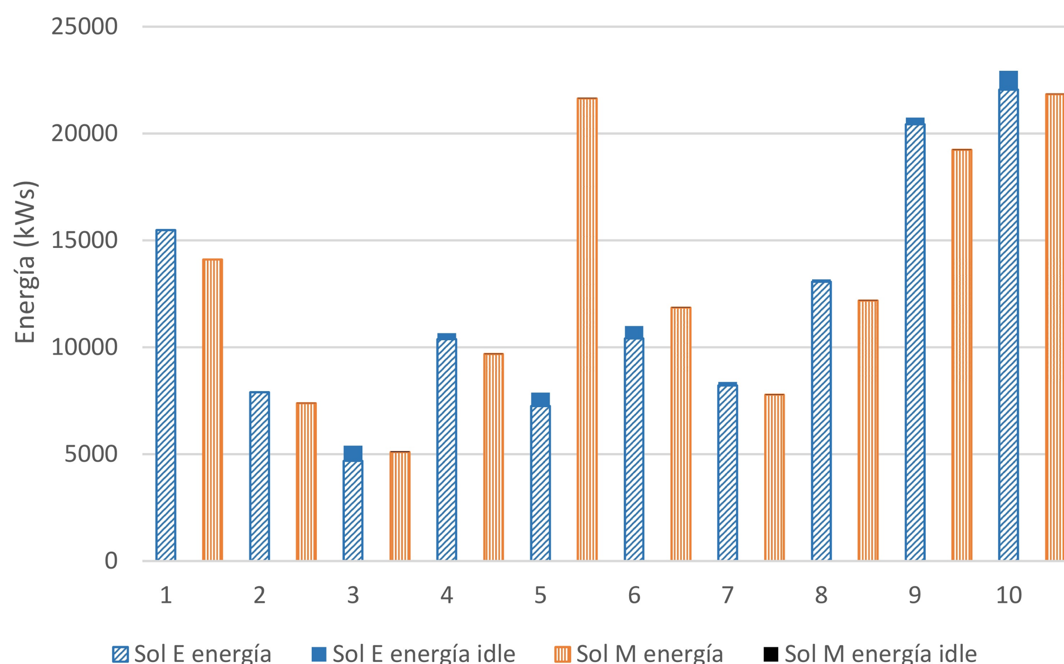


Figura 4.4: Valor del consumo energético de cada nodo para dos soluciones del Algoritmo Evolutivo en la instancia de 1000 tareas y 10 recursos. *Sol E* corresponde a la solución con mejor valor para el objetivo energía, y *Sol M* para el objetivo makespan. Se fracciona el consumo en idle y activo.

algoritmos, pudimos observar como el algoritmo evolutivo detecta aquellos nodos que son menos eficientes, y asigna menos tareas a estos como mecanismo para disminuir el consumo enrgético. En la Figura 4.4 se presenta una comparación de los consumos energéticos de cada nodo para la misma solución que las figuras anteriores, discriminados en consumo útil y consumo idle. En esta comparación en particular, la solución *Sol E* al utilizar menos el nodo 5, logra disminuir el consumo energético total del sistema en un 5.6 %, a pesar de que el consumo individual de otros nodos sea mayor.

4.6. Resumen

En este capítulo se presentó el trabajo realizado sobre la asignación de tareas en un cluster heterogéneo, buscando minimizar el makespan de ejecución de las tareas y el consumo energético del cluster. Para realizar la tarea se desarrolló un algoritmo evolutivo basado en NSGA-II, se definieron 4 instancias del problema basadas en datos obtenidos del Cluster FING, de la Facultad de Ingeniería, UDELAR, y se comparó el desempeño del algoritmo contra una heurística basada en MinMin. El algoritmo demostró ser competitivo con la heurística MinMin, permitiendo obtener mejores soluciones para cada uno de los objetivos individuales, y soluciones con valores similares al considerar ambos objetivos a la vez.

Se observó además, que nuestro algoritmo es capaz de identificar aquellos nodos que son menos eficientes, y evitar asignar tareas a éstos. En un escenario realista esta detección junto con mecanismos que apaguen o duerman nodos que se encuentren ociosos permitiría ahorrar aún más energía.

Durante el análisis del problema y con las soluciones obtenidas, se pudo observar que los dos objetivos considerados se encuentran relacionados, y que las soluciones obtenidas conforman un frente de Pareto, donde elegir una solución que mejora uno de los objetivos implica obtener peores resultados para el otro. Una de las ventajas de nuestro algoritmo radica en el hecho de brindar un abanico de opciones, en contraste con otros que brindan una solución única, este conjunto de soluciones presenta diferentes niveles de compromiso entre el consumo energético y el makespan. Al contar con múltiples soluciones es posible y necesario desarrollar un algoritmo que permita elegir la solución que más se adecue a las necesidades del momento. Por ejemplo, podría elegir una opción con un menor consumo energético cuando el coste de la energía sea mayor, y utilizar soluciones que favorezcan la performance cuando su coste sea menor. También sería posible incorporar la refrigeración del cluster como otro factor a tener en cuenta, y considerarlo un objetivo más del algoritmo.

Capítulo 5

Asignación de usuarios en un salón de informática

Este capítulo presenta el trabajo realizado sobre la asignación de usuarios en salones de informática. Para el problema se consideraron dos objetivos, minimizar el consumo energético del conjunto de recursos del salón, y minimizar la sobreasignación de los recursos, valor relacionado a la calidad de servicio.

Para resolver el problema se realizó un análisis de un caso real, en base al cual se modelaron las instancias utilizadas en la evaluación experimental de las soluciones propuestas. Este análisis consistió en la recopilación de datos de uno de los salones de informática de la Facultad de Ingeniería, UDELAR. Luego de la definición de las instancias, se implementó una heurística basada en las ideas intuitivas de una persona para realizar la asignación, dos heurísticas basadas en el algoritmo minmin (Luo et al., 2007), y un algoritmo evolutivo multiobjetivo basado en la metaheurística NSGA-II (Deb et al., 2002). Además, como solución de referencia se implementó un algoritmo con una política round robin (Hahne, 1991) y otro que realiza la asignación basándose en los recursos elegidos por los usuarios observados en la primer etapa del estudio. Se implementó también un algoritmo de backtracking, pero el mismo demostró no ser eficiente para resolver instancias de mayor dimensión, impidiendo utilizar modelos reales del problema.

El capítulo se organiza de la siguiente forma: primero se presenta el problema de HCSP, luego se presenta el modelo del problema junto con la formulación matemática del mismo. A continuación se presentan los algoritmos diseñados para su resolución, seguidos de detalles de la implementación de la solución, donde se describen los componentes de la solución y el sistema para la recopilación de datos. Luego se presenta la evaluación experimental, incluyendo un análisis de los datos recopilados del salón de informática, la construcción de la instancias y el estudio paramétrico. Finalmente se presentan los resultados obtenidos y un resumen del capítulo.

5.1. Presentación del problema

En este capítulo presentamos una solución a un caso particular del problema de HCSP, la asignación de usuarios en un salón de informática buscando minimizar el consumo energético de los recursos informáticos, a la vez que se pretende mantener la calidad del servicio ofrecida. La calidad de servicio se mide mediante una función de sobreasignación, basada en los requerimientos de los usuarios y la capacidad del recurso asignado

de satisfacer los mismos.

La realidad modelada en nuestra solución son los salones de informática de la Facultad de Ingeniería, UDELAR. Los usuarios se dividen en dos tipos según la forma en la que acceden a los recursos: los remotos y los presenciales. Los primeros, acceden a los recursos mediante una conexión remota que les permite elegir cuales recursos utilizarán. Los usuarios presenciales, en cambio, se presentan en los salones de informática y utilizan cualquiera de los recursos que no esté ocupado por otro usuario presencial. Nuestro modelo contempla que los usuarios tengan predilección por un recurso, y al acceder al sistema de forma presencial intentaran utilizar este recurso o el más cercano al mismo en caso de estar ocupado por otro usuario presencial. El sistema permite que varios usuarios accedan al mismo recurso a la vez, siempre que la cantidad de usuarios presenciales no sea mayor a uno.

En los salones de informática, en la actualidad, cualquier usuario puede utilizar cualquier recurso y los recursos que no son utilizados permanecen encendidos. Por lo tanto, múltiples usuarios remotos pueden utilizar el mismo recurso aunque exista en la red otro encendido y con menor carga. Esto probablemente genere un consumo energético innecesario al mantener recursos ociosos encendidos. Además, se presentará un deterioro en la calidad de servicio ofrecida por aquellos usuarios que comparten un recurso superando su capacidad, y provocando que el sistema vea afectada su performance.

Existen otros factores que influyen en el uso de los recursos, como ser el tiempo y la ubicación física del recurso. La fecha y el horario afectan la demanda por recurso; algunos horarios del día presentan una mayor cantidad de usuarios, y a su vez dependiendo del día de la semana o el período de estudio (por ej. período de parciales, receso, o días feriados) puede disminuir o aumentar la demanda. Es conveniente estudiar estas tendencias y construir un algoritmo que permita tener en cuenta la demanda variable al momento de determinar la cantidad de recursos que deberán encontrarse disponibles. Además del factor temporal, la ubicación física de los recursos influye en la cantidad de usuarios que buscan acceder al mismo.

Con el objetivo de atacar estos problemas se diseñó un sistema de asignación de usuarios que designa el recurso que debe utilizar cada usuario y los recursos que se encuentran disponibles, buscando minimizar el consumo energético y maximizar la calidad de servicio percibida por el usuario. Si bien son dos objetivos a tratar, no se desea ponderar uno sobre el otro, por lo tanto la solución propuesta se basa en un algoritmo multiobjetivo, que brinda un abanico de opciones de las cuáles se puede elegir aquella que se ajuste más a las necesidades del momento.

5.2. Modelo del problema

Se construyó un modelo que nos permite desarrollar y simular la utilización de distintas soluciones, sin la necesidad de utilizar la infraestructura real. Para ello creamos una simulación donde el modelo de la misma representa a un conjunto de usuarios que desean utilizar los recursos de un salón de informática. Los principales componentes del modelo son: el sistema, los recursos de cómputo y las sesiones de los usuarios. En nuestro modelo se requiere conocer de antemano todas las sesiones que se iniciarán en el sistema.

El salón, modelado por el concepto de sistema, está compuesto por un grupo de PCs

cada una representada mediante un recurso. De cada recurso se mantiene la información de su capacidad máxima y su nivel de uso actual, medido como el porcentaje de uso de su CPU y su memoria RAM. El uso de un recurso por parte de un usuario se denomina sesión, y es definida al momento que el usuario arriba al sistema. De la sesión de un usuario se conoce: el momento de inicio, su duración y los requerimientos máximos y promedios de CPU y memoria RAM. Cada sesión se modela en el sistema mediante dos eventos, uno representando la entrada del usuario en el sistema y el otro la salida. El sistema mantiene la lista de recursos disponibles, los usuarios actualmente conectados a un recurso, y el estado y nivel de uso de cada recurso.

En el modelado del problema se definen dos métricas a minimizar, el consumo energético y la sobreasignación de los recursos. La primera representa al consumo energético del sistema durante el período de tiempo entre la llegada del primer usuario y la salida del último. La sobreasignación es una métrica de la calidad de servicio percibida por los usuarios, es una función de la diferencia relativa entre los requerimientos de los usuarios en un recurso y la capacidad máxima del mismo, multiplicado por la cantidad de usuarios que se ven afectados por esta sobrecarga y el tiempo de la misma.

A continuación se describen de forma breve los principales componentes del modelo:

Sistema Representa el salón de informática manteniendo información sobre los recursos y usuarios, los requerimientos de cada usuario en el sistema, y la capacidad máxima y estado actual de cada recurso.

Recurso Representa individualmente a los recursos informáticos del sistema. Cada recurso tiene la información de su capacidad máxima de CPU, determinada por la cantidad de cores del procesador, y la capacidad máxima de memoria RAM medida como la cantidad de MB de memoria disponibles. Cada recurso puede encontrarse apagado o encendido. A su vez, un mismo recurso puede ser utilizado por varios usuarios a la vez.

Sesión Representa el uso de cada recurso por parte de un usuario. Se define al momento que el usuario solicita un recurso, y se mantiene hasta que lo abandona. Durante la sesión se registra información sobre el uso promedio y máximo de CPU y memoria RAM. Nuestro sistema modela dos tipos de sesiones, las presenciales y las remotas. Las sesiones presenciales corresponden a los usuarios que concurren al salón de informática con una PC de preferencia a la que desean conectarse. En la solución propuesta en caso de que un usuario no pueda utilizar su PC de preferencia por estar ocupada por otro usuario presencial, se le buscará el recurso más próximo a la que deseaba conectarse. Una sesión remota se define cuando el usuario se conecta de forma remota al recurso que le sea asignado, en lugar de acceder de forma presencial.

Evento Corresponden al inicio o fin de la sesión de un usuario, indicando el momento en el que el usuario ingresa o sale del sistema.

5.2.1. Formulación matemática del problema

Se presenta a continuación la formulación matemática del problema utilizada en nuestro trabajo.

Se definen las siguientes variables:

- Se cuenta con M recursos informáticos $R = r_1 \cdots r_M$
- Se cuenta con N sesiones $S = s_1 \cdots s_N$
- Se cuenta con $N \times 2$ eventos $E = e_1 \cdots e_{2N}$ ordenados cronológicamente

Una solución se expresa mediante un arreglo de tuplas de asignación $A = [a_i] = [(s_i, r_j)]$, que representa la asignación de la sesión s_i al recurso r_j.

El algoritmo debe mantener la información que le permita calcular las siguientes funciones:

- $tiempoinicio : (s_i) \longrightarrow \bar{t}$
- $tiempofin : (s_i) \longrightarrow \bar{t}$
- $sesiones\ activas : (r_j, t) \longrightarrow [s_{i1} \ldots s_{i2}]$
- $uso\ recursos : ([s_{i1} \ldots s_{i2}]) \longrightarrow uso = (cpu_{avg}, cpu_{max}, ram_{avg}, ram_{max})$
- $uso\ maximo : (r_i) \longrightarrow \overline{uso} = (\overline{cpu}_{avg}, \overline{cpu}_{max}, \overline{ram}_{avg}, \overline{ram}_{max})$

La función *tiempo inicio* y *tiempo fin* indican el momento en el que el usuario inicia y finaliza la sesión respectivamente, ambos corresponden a los tiempos de los eventos asociados a la sesión. La función *sesiones activas* retorna la cantidad de sesiones activas en el recurso r_j en el momento t. Una sesión se considera activa en un momento dado si este es mayor a su tiempo de inicio y menor al de fin. La función *uso recursos* retorna la sumatoria de los requerimientos de las sesiones pasadas consideradas, mientras que la función *uso maximo* retorna la capacidad máxima del recurso, ambos considerando los valores del uso medio y máximo de memoria RAM y CPU.

- $consumo : (r_j, [s_{i1} \ldots s_{i2}]) \longrightarrow \bar{c}$
- $sobreasignacion : (r_j, [s_{i1} \ldots s_{i2}]) \longrightarrow \overline{sa} = |[s_{i1} \ldots s_{i2}]| \times dif(uso, \overline{uso})$

Dados un recurso y un conjunto de sesiones, las funciones *consumo* y *sobreasignación* retornan el consumo energético del recurso por unidad de tiempo, y la sobreasignación del mismo. La sobreasignación del recurso se calcula como, la cantidad de sesiones activas en el recurso multiplicado por la suma de las diferencias relativas de cada uno de los valores de uso considerados $(cpu_{avg}, cpu_{max}, ram_{avg}, ram_{max})$ para el valor de *uso recursos* y *uso maximo*. La diferencia relativa se considera 0 cuando el uso es menor a la capacidad máxima, y en caso de que el recurso se encuentre apagado, tanto el consumo como la sobreasignación serán 0.

Objetivos del problema

El problema se basa en minimizar el consumo energético del sistema (C) y la sobreasignación del mismo (SA), como sumatoria de la sobreasignación de todos los recursos. Para definir el consumo energético del sistema, definiremos la función auxiliar $C_j(t)$, ver la ecuación 5.1, que para cada recurso r_j calcula el consumo energético del mismo en el instante t. El consumo energético del sistema representado en la ecuación 5.2, se calcula como la sumatoria del consumo energético de todos los recursos por la diferencia de

tiempo entre un evento y el siguiente. Se considera que el tiempo de e_1 es 0, y el del evento ficticio e_{2N+1} es igual al tiempo del último evento e_{2N}.

$$C_j(t) = consumo(r_j, sesiones_activas(t)) \tag{5.1}$$

$$C = \sum_{e_i \in E} \left((tiempo(e_{i+1}) - tiempo(e_i)) * \left(\sum_{r_j \in R} C_j(tiempo(e_i) \right) \right) \tag{5.2}$$

La sobreasignación del sistema se calcula de forma similar al consumo energético, utilizando las funciones auxiliares $SA_j(t)$ que se observan en la ecuación 5.3, que representan la sobreasignación del recurso r_j en el momento t. El valor de SA del sistema, tal como se muestra en la ecuación 5.4, se calcula como la sumatoria de los valores de sobreasignación de cada recurso, multiplicados por la diferencia de tiempo entre un evento y el siguiente, para todos los eventos considerados.

$$SA_j(t) = sobreasignacion(r_j, sesiones_activas(t)) \tag{5.3}$$

$$SA = \sum_{e_i \in E} \left((tiempo(e_{i+1}) - tiempo(e_i)) * \left(\sum_{r_j \in R} SA_j(tiempo(e_i) \right) \right) \tag{5.4}$$

Los funciones mencionadas anteriormente fueron implementados de forma independiente a los algoritmos, de forma que todos utilicen el mismo mecanismo para calcular los valores de los objetivos del problema. El pseudocódigo se presenta en el algoritmo 9.

Modelo del consumo de energía y la sobreasignación

En nuestro problema decidimos modelar el consumo de los recursos en función del porcentaje de uso de la CPU. Este modelo también es utilizado por Minas and Ellison (2009), donde define la ecuación de consumo para un porcentaje de uso de CPU n como: $P_n = (P_{max} - P_{idle}) \times \frac{n}{100} + P_{idle}$.

En Barroso and Hölzle (2007) se muestra una relación lineal entre el consumo energético de un servidor y el porcentaje de uso de su CPU, marcando que cuando el uso es 0 % y el servidor está activo, su consumo es casi un 50 % del consumo máximo.
Nosotros optamos por modelar el consumo energético como una función lineal escalonada del uso de CPU. En la instancia del problema se define el valor del consumo energético de cada recurso en intervalos de 10 % de uso de CPU, los valores intermedios se calculan como una función lineal entre los valores más cercanos. Para calcular el consumo energético de un recurso en un momento dado, se suma el valor de consumo de CPU promedio de todas las sesiones asignadas a este recurso, y se calcula el valor relativo a la CPU específica de este recurso, esto se efectúa dividiendo el porcentaje total por la cantidad de cores del procesador del recurso en cuestión. Una vez que se tiene la suma de consumo de CPU de los usuarios, se determinan los valores más cercanos definidos en la instancia, y asumiendo la linealidad de la función en el intervalo se determina el resultado para el valor de uso específico. Por ejemplo: si una CPU tiene 4 cores y la suma del uso de CPU de los usuarios corresponde a 350 %, este uso representa el 87.5 % de uso del procesador, y el valor de consumo específico se calcularía como $(87{,}5 - 80) \times pendiente(90, 80)$, siendo pendiente una función que retorna el valor de la derivada entre dos puntos de la función.

Algoritmo 9 Algoritmo para el calculo de los objetivos del problema

```
Recursos                                          ▷ Variable global con los recursos
Sistema                         ▷ Variable global con una implementación de sistema
procedure CalcularObjetivos(tiempo)
    ▷ calcula la energía para el estado actual del sistema
    for all r ∈ Recursos do
        consumo_sistema = consumo_sistema + Consumo(r)
        sobreasig_sistema = sobreasig_sistema + Sobreasignacion(r)
    end for
    return (tiempo × consumo_sistema),tiempo × sobreasig_sistema)
end procedure

procedure Consumo(recurso)
    if (recurso.estaEncendido()) then
        if (sistema.usuarios_asignados(recurso) = 0) then
            return ConsumoIdle(recurso)
        else
            max = Sistema.obtener_capacidad_maxima(recurso)
            uso = Sistema.obtener_uso_recursos(recurso)
            ▷ Uso de CPU debe ser menor a 100 %
            return GetConsumoPorCPU(min(uso.cpu_avg, max.cpu_avg), recurso)
        end if
    else
        return 0
    end if
end procedure

procedure Sobreasignacion(recurso)
    if recurso.estaEncendido() then
        cant_usuarios_recurso = cantidad_usuarios_asignados(recurso)
        max = Sistema.obtener_capacidad_maxima(recurso)
        uso = Sistema.obtener_uso_recursos(recurso)
        sobreasignacion_recurso = calcular_sobreasignacion(uso,max)
        sobre_asignacion += sobreasignacion_recurso * cant_usuarios_recurso
    end if
    return sobre_asignacion
end procedure
```

El consumo de todo el sistema se calcula como la suma de todos los recursos encendidos, multiplicado por el intervalo de tiempo considerado.

La sobreasignación la definimos como una función de la diferencia relativa entre los requerimientos de uso de recursos de los usuarios y la capacidad máxima del recurso. El valor de sobreasignación en un momento dado es igual a la suma de las diferencias relativas entre los requerimientos de todos los usuarios asignados, considerando uso promedio y máximo de memoria RAM y CPU, y los valores máximos del recurso definidos en la instancia. La suma de las diferencias relativas se multiplica por la cantidad de usuarios asignados, ya que todos ellos se ven afectados. Para calcular el valor del sistema se debe sumar la sobreasignación de todos los recursos y multiplicar por el intervalo de tiempo considerado.

5.3. Diseño de los algoritmos propuestos

Como parte de nuestro trabajo implementamos varios algoritmos para dar solución al problema de la asignación de usuarios a recursos informáticos. Nuestro problema pertenece a la familia de problemas de optimización de planificación que tal como demostró Ullman (1975) son NP-Hard, lo que implica son difíciles de resolver por tener espacios de solución muy grandes, funciones complejas o porque utilizan grandes volúmenes de datos. Por lo tanto las técnicas exactas no suelen ser útiles para casos prácticos.

Es por estos motivos que Nesmachnow (2015) sugiere utilizar heurísticas y metaheurísticas como técnicas para la resolución de estos problemas. Estas técnicas normalmente permiten encontrar soluciones de buena calidad en tiempos de ejecución razonables para problemas difíciles. Dado que nuestro problema consta de dos objetivos, el consumo energético del sistema y la sobreasignación de los recursos, los algoritmos planteados deben buscar soluciones que optimicen ambos objetivos.

El primer algoritmo desarrollado al cual llamamos *DetParam*, fue una heurística greedy determinista enfocada en minimizar el consumo energético. La heurística busca realizar una asignación de usuarios óptima utilizando criterios que nos parecieron seguiría una persona al realizar la asignación, como no asignar un usuario a un recurso que está apagado si hay otros disponibles, o asignar un usuario al recurso que pueda cubrir sus requerimientos.

También implementamos una heurística greedy multiobjetivo inspirada en el algoritmo minmin. Esta heurística tiene dos variantes, una que considera como primer objetivo el consumo energético y como segundo la sobreasignación, y otra que lo hace a la inversa. El algoritmo calcula en cada sesión para todos los recursos el valor del primer objetivo que resultaría de la asignación. Toma el 5 % de las soluciones que presenten mejor valor para el primer objetivo, y con esta lista de candidatos calcula los valores para el segundo objetivo, quedándose con el recurso que presente el menor valor de este.

También implementamos una metaheurística para solucionar el problema de HCSP, optando por un algoritmo evolutivo multiobjetivo que nos permite obtener un conjunto de soluciones que representan distintos puntos de equilibrio entre los objetivos considerados. A diferencia de las heurísticas greedy implementadas, este algoritmo no requiere ponderar un objetivo sobre el otro.

A pesar de que los algoritmos exactos no suelen ser prácticos para solucionar este tipo de problemas, implementamos un algoritmo de *backtracking* con distintas variantes,

una para encontrar la mejor solución considerando el objetivo del consumo energético, una para el objetivo de sobreasignación, y otra que busca determinar todas las soluciones no dominadas del problema. El objetivo de implementar este algoritmo es encontrar la mejor solución posible, y poder determinar que tan alejadas de estas soluciones óptimas se encuentran las resueltas por los otros algoritmos.

Para tener un valor de referencia al momento de comparar soluciones, se desarrollaron dos algoritmos de asignación simples. El primero asigna la sesión al recurso que se especifica en la instancia, este valor se determinó originalmente por el recurso donde se recopiló la información de la sesión. Dado que no utiliza ninguna lógica para realizar la asignación, sino que utiliza el valor original de la sesión, lo denominamos *asignación predeterminada*. El segundo algoritmo utiliza una política de *round robin* para asignar los recursos a las sesiones de forma secuencial, a la primer sesión se le asignará el recurso 1, a la segunda el recurso 2, y así sucesivamente.

5.4. Algoritmo DetParam

El primer algoritmo propuesto para resolver el problema de HCSP es DetParam, una heurística greedy que implementa la lógica de asignación de usuarios tal como la realizaría una persona. En esta sección se explica el desarrollo del algoritmo DetParam explicando el procedimiento que sigue para realizar la asignación. El algoritmo tiene en cuenta tanto el consumo energético como la sobreasignación para determinar las soluciones, priorizando aquellas soluciones con menor consumo energético. El objetivo de sobreasignación lo optimiza de forma indirecta, ya que solo tiene en cuenta el consumo promedio de CPU en el proceso de selección de las soluciones.

Diseño e invocación del algoritmo

Creamos un método principal que ejecuta el algoritmo determinista para cada sesión que debemos asignar. Luego de todas las asignaciones calcula e imprime el consumo energético final y la sobreasignación de recursos. Este método recibe como parámetros la dirección de los archivos que definen la instancia, y el nombre de la carpeta donde se deben escribir los datos de salida. A partir de los datos de los recursos se crean las siguientes variables globales:

- matrizRam: arreglo que contiene la memoria RAM de cada recurso
- matrizCores: arreglo que contiene la cantidad de cores de cada recurso
- matrizEnergia: matriz que contiene el consumo energético en cada estado para cada recurso
- matrizEnergiaSleep: arreglo que contiene el consumo energético en estado sleep para cada recurso
- probabilidadOcurrencia: arreglo con la probabilidad de ocurrencia de un usuario presencial por cada recurso

Para inicializar el algoritmo determinista le pasamos los recursos y su constructor carga los datos que va a utilizar el algoritmo de forma global: matriz con consumo de memoria RAM, matriz con cantidad de cores, cantidad de recursos, matriz de consumo energético, matriz de consumo energético en estado idle y matriz con probabilidad

de ocurrencia de un usuario presencial. Además, crea listas candidatas de recursos por memoria RAM y consumo de CPU. Lo que planteamos fue crear cuatro listas, dos por memoria RAM y dos por consumo de CPU. Una lista contiene los recursos mayores a una cierta cota definida (constante), y la otra contiene los recursos que tienen valor menor a la cota definida. Este par de listas se definen tanto para consumo de CPU como para memoria RAM con sus respectivas cotas. Las cotas las definimos estudiando los datos reales recabados de los salones de informática. Para la memoria RAM determinamos como cota 3,800 MB, y para el consumo de CPU determinamos que la cota sea 4 $cores$. Estas listas candidatas nos servirán más adelante para un mejor ajuste a los requerimientos de cómputo de cada sesión.

Llegado a este punto, con todas las variables del problema definidas, se comienza a iterar sobre los eventos de entrada del sistema para asignarle un recurso.

Por el formato de la lista de eventos de las sesiones, existen dos tipos: los eventos correspondientes al inicio de una sesión y los eventos que indican el fin de una sesión. Es decir, o bien una sesión requiere entrar al sistema, o requiere salir. Cuando una sesión inicia en el sistema se le debe asignar un recurso, y cuando finaliza se debe liberar el recurso que tenía asignado. Ambos eventos están relacionados para poder luego liberar el recurso.

Obtener candidatos

Un evento de entrada corresponde a un usuario que ingresa al sistema, el cual puede ser: un usuario presencial que acude físicamente al salón de informática en busca de un recurso específico, o también un usuario remoto que requiere conectarse a uno de los recursos del salón de forma remota, aunque sin requerimientos específicos respecto a que recurso utilizar.

Si el usuario es de tipo presencial, el recurso al que desea conectarse viene dado como parte de la información de la sesión. Si el recurso elegido está libre, el mismo es seleccionado para ser asignado a la sesión, por el contrario si está ocupado por otro usuario presencial, se selecciona el recurso más cercano que se encuentre libre.

Si el usuario es de tipo remoto, invocamos al algoritmo determinista DetParam para obtener los mejores candidatos que se considerarán primero en base al menor consumo energético, y luego según el menor uso de recurso.

El procedimiento de obtener los candidatos devuelve los identificadores de los recursos que mejor se ajustan a los requerimientos de la sesión. Primero se obtienen los recursos que son mejores candidatos según el consumo energético. Esta operación se divide en tres partes. Primero se crea una lista de candidatos de memoria RAM según el uso de la sesión. Esta lista fue creada previamente (ver subsección 5.4), aquí solo se obtienen los candidatos según el consumo de la sesión. Luego se crea la lista de candidatos de consumo de CPU de la misma forma que la lista de memoria. Y por último, se crea una sola lista candidata con los recursos que están en ambas, formando una única lista de recursos candidatos que mejor se adaptan según el consumo de memoria RAM y consumo de CPU de la sesión. Esto es, los candidatos que cumplen con ambos requerimientos.

Luego de obtener los recursos candidatos para asignarle a la sesión, se aplica un filtro necesario para el buen funcionamiento del sistema. Así, se implementa una función que

descarta los recursos que al asignarle la sesión, éste supera el 100 % de consumo de CPU.

Una vez obtenidos los recursos candidatos según los requerimientos de memoria RAM y consumo de CPU, y descartados los que se sobreasignarían, se debe calcular el consumo energético estimado para cada recurso en caso de asignarle la sesión. El consumo energético se debe calcular para el tiempo total de permanencia del usuario en el sistema.

Entonces, para cada recurso calculamos el delta consumo: variación de consumo energético de agregar la sesión en este recurso. Para realizar estos cálculos se utiliza el modelo del sistema. Luego de calculados los delta consumo por recurso, selecciono los candidatos con menor consumo energético.

A este algoritmo se le implementó una variación para un funcionamiento más efectivo del sistema modelando un escenario posible de la realidad. Era necesario modelar un sistema que tomara en cuenta un evento de entrada de un usuario presencial en el mismo recurso al que voy a asignar la sesión, en caso de no estar ocupado por otra sesión presencial. Es aquí que el algoritmo determinista utiliza las probabilidades de ocurrencia de una sesión presencial en cada recurso. La forma de implementarlo fue escoger los candidatos de mejor delta consumo (mínimo) pero que no superen el 50 % de probabilidad de ocurrencia de un evento de entrada de un usuario presencial. Estudiando los datos recopilados de los salones de informática, se observó que una sesión presencial consumía un porcentaje elevado de la memoria RAM y consumo de CPU del recurso informático, por lo que resultaba conveniente prever el evento de entrada de un usuario de este tipo antes de asignar un usuario remoto a este recurso.

Luego de obtener los mejores candidatos según el primer objetivo, se deben escoger los candidatos que mejor se adapten al uso del recurso. Por lo tanto, luego de este procedimiento se obtendrán los mejores candidatos a asignarle a la sesión según minimizan los dos objetivos.

Una vez obtenidos los mejores candidatos por el algoritmo determinista, en la clase principal se asignará la sesión al primer recurso de la lista de candidatos que no tenga una sesión presencial, con el objetivo de disminuir la sobreasignación.

Cuando el evento de la sesión es de salida, corresponde a la finalización de una sesión, indicando que el usuario terminó todas sus tareas y abandona el recurso que tenía asignado. En este caso el sistema utilizando el identificador de la sesión obtiene el recurso que tenía asignado, y finaliza la misma.

Luego de iterar todos los eventos de entrada de una sesión y asignarle un recurso a cada una, se debe calcular el consumo energético y la sobreasignación, utilizando los métodos implementados por el modelo del sistema.

Pseudocódigo

En el algoritmo 10 se presenta el pseudocódigo de las principales operaciones del algoritmo determinista que retorna los mejores candidatos, y en el anexo B.1 presentamos el detalle de implementación del algoritmo determinista para una mejor comprensión de sus tareas.

Algoritmo 10 Principales operaciones del algoritmo DetParam

```
procedure OBTENERCANDIDATO(arribo, sistema) ▷ Método que realiza la asignación
    obtenerRecursosCandidatos(arribo)
    ▷ arma e intersecta las listas candidatas por consumo de memoria RAM y CPU
    descartarRecursosSobrecargados()
    ▷ filtra candidatos que al agregarle un usuario superan el 100 % de uso de cpu
    obtenerConsumoEnergeticoEstimado(arribo)
    obtenerMejorCandidato()
end procedure

procedure OBTENERCONSUMOENERGETICOESTIMADO(arribo)
    ▷ Calcula el consumo energético del arribo por recurso para el tiempo estimado de
permanencia
    for all Recurso r ∈ recursos do
        consumoRecurso = obtenerConsumo(arribo, r)
        consumoAgregado = obtenerConsumoPostAsignacion(arribo, r)
        deltaConsumo = consumoAgregado − consumoRecurso
        ▷ el consumo energético se calcula como la variación de agregar arribo en r
    end for
end procedure

procedure OBTENERMEJORCANDIDATO
    Obtener los candidatos de menor consumo energético pero que no superen el 50 %
de probabilidad de ocurrencia de un usuario presencial
    ▷ Selecciono los candidatos cuyo valor sea igual al menor consumo energético
    Obtener el candidato de menor uso de recurso dentro de los mejores de consumo
energético
end procedure
```

5.5. Algoritmo minmin

Se desarrollaron dos heurísticas basadas en el algoritmo minmin, con el objetivo de resolver el problema estudiado en este capítulo. Las heurísticas son dos variaciones de un algoritmo greedy multiobjetivo, similar a la utilizada para resolver el problema de HCTS en el capítulo 3.

Las dos variantes del algoritmo se diferencian por el orden que le dan a los objetivos en su ejecución. Una de las variantes selecciona los recursos para hacer la asignación considerando primero el consumo energético, y luego de seleccionar un grupo de candidatos, elige entre éstos el mejor según el valor de sobreasignación. La otra variante lo hace a la inversa.

Para implementar la heurística, creamos una clase principal de manera similar a como se desarrolló el algoritmo determinista. Esta clase se encarga de leer los argumentos, inicializar los datos de los recursos, leer todas las sesiones a asignar y definir la carpeta donde se almacenarán los resultados obtenidos de consumo energético, sobreasignación y utilización de recursos. En esta clase principal se lee el algoritmo que se desea utilizar para resolver el problema y se invoca a la función que resuelve las asignaciones de sesiones a recursos. La lógica de este procedimiento es muy similar a la del algoritmo determinista descrito en el algoritmo DetParam.

La función de asignación itera todas las sesiones, si es un evento de entrada de la sesión se discrimina según si su tipo de usuario es presencial o remoto. Si una sesión es presencial entonces el recurso al que se desea conectar viene dado. Si ese recurso está libre se le asigna, y sino se debe buscar el recurso más cercano que esté libre. En el caso particular que ningún recurso esté libre de sesiones presenciales, entonces se le asigna el recurso que tenga menor cantidad de sesiones conectadas al momento.

Cuando la sesión del usuario es remota, se invoca la rutina que obtiene el mejor candidato según la versión del algoritmo minmin que se haya seleccionado. Esta rutina minimiza por el primer objetivo quedándose con el 5 % de los candidatos con mejor valor para el primer objetivo, y luego utilizando estos recursos candidatos, calcula los valores para el segundo objetivo eligiendo el recurso que reporte el menor valor. Originalmente se elegían los candidatos cuyo valor del primer objetivo se encontraban en un margen de 5 % del mejor valor, pero este generaba un problema cuando el mejor valor era 0 o muy cercano a 0, ya que solamente se seleccionaba el candidato con mejor valor, anulando la optimización por el segundo objetivo. Una vez detectado este comportamiento se realizó el cambio a la selección del 5 % de los mejores candidatos.

Se definió un caso particular cuando el primer objetivo corresponde al consumo energético, debido a que una vez que un recurso se encuentra funcionando al 100 % su consumo energético no puede aumentar, se definió que si el menor valor retornado de consumo energético para el primer objetivo es 0, se seguirán incorporando candidatos hasta encontrar alguno cuya delta consumo sea mayor. De esta forma se evita asignar todas las sesiones a un recurso que se encuentra colmado..

Una vez obtenido el mejor candidato, se asigna a la sesión el recurso en el sistema.

Cuando el evento de la sesión es de salida, se debe quitar a la sesión del sistema.

Por último, luego de realizadas todas las asignaciones, se debe calcular el consumo energético final del sistema y la sobreasignación de los recursos. En este algoritmo, el consumo energético se calcula únicamente en base al porcentaje de uso de CPU y el

tiempo que dura la sesión en el recurso. Luego de terminados los cálculos, se salvan estos valores en la carpeta especificada en los parámetros de entrada del algoritmo principal.

A continuación se presenta el pseudocódigo del algoritmo minmin implementado como una heurística greedy:

Algoritmo 11 Algoritmo minmin

```
function AsignarArribosARecursos(Arribos, Sistema)
    for all arribo ∈ Arribos do
        if esEntrada(arribo) then
            if esUsuarioPresencial(arribo) then
                recurso_deseado = obtenerRecurso(arribo)
                ▷ Obtiene el recurso sin otro usuario presencial más cercano
                recurso = obtenerRecursoCercano(Sistema, recurso_deseado)
            else
                ▷ El orden de Obj1 y Obj2 cambia según la versión de MinMin
                valores_obj1 = calcularObj1(arribo, Sistema)
                ▷ Retorna el 5 % de los recursos con mejor valor
                candidatos = mejoresValores(Sistema, valores_obj1, 0.05)
                valores_obj2 = calcularObj2(arribo, Sistema,candidatos)
                recurso = recursoConMejorValor(valores_obj2, candidatos)
            end if
            asignarRecurso(recurso, arribo)          ▷ Guarda el recurso elegido
            iniciarSesion(Sistema, arribo, recurso)
        else
            terminarSesion(Sistema, arribo, arribo.getRecurso())
        end if
    end for
    return obtenerAsignaciones(Sistema)
end function
```

5.6. Backtracking

Backtracking es un algoritmo general que permite encontrar una o todas las soluciones a un problema computacional. Opera construyendo candidatos a soluciones de forma incremental, abandonando las que previamente se puedan determinar que no permitirán construir una solución factible, ya sea por no cumplir las condiciones necesarias o por ser peores que otras soluciones ya determinadas. (Rossi et al., 2006)

Nuestra implementación construye la asignación de recursos a las sesiones, buscando obtener las mejores soluciones posibles, con el objetivo de construir un frente de Pareto real conformado por todas las soluciones no dominadas de una instancia del problema de HCSP. Para disminuir el tiempo de ejecución del algoritmo de backtracking, éste recibe un conjunto de soluciones previamente determinadas por la ejecución de los otros algoritmos, y descarta todas aquellas soluciones intermedias que sean peores a las ya determinadas.

Formulación del backtracking

La soluciones se conforman por una Tupla A de largo k de la forma $A = [a_1 \dots a_k]$, donde k es la cantidad de sesiones que arriban al sistema, y cada variable $a_i \in R = [r_1 \dots r_M]$ representa la asignación de la sesión s_i en el recurso a_i.

La única restricción del problema es que todos los valores de a_i deben corresponder a uno de los recursos definidos en el conjunto R. Las funciones objetivos corresponden a los objetivos del problema, consumo energético y sobreasignación de recursos. Como predicados de poda se utilizará que si un candidato a solución presenta valores de la función objetivo que sean dominados por alguna solución del frente de Pareto real, se abandonará la construcción de esa solución candidato.

Implementación del backtracking

El backtracking se implementó como una función recursiva que construye gradualmente el arreglo de asignaciones, iterando en cada recursión sobre todos los valores posibles para una asignación. La función recibe como parámetro una variable que contiene el valor acumulado de los dos objetivos de la solución parcial, y un índice indicando que valor del arreglo de asignaciones se debe completar. Luego itera sobre todos los recursos, asignando la sesión al recurso, calculando el valor de ambos objetivos para esa asignación, almacenando estos valores en la variable de acumulado e invocando nuevamente a la función con el valor de índice incrementado. La recursión se detiene cuando se asignó un recurso a todas las sesiones. El pseudocódigo en detalle se puede ver en el algoritmo 12.

Se desarrollaron tres variantes del algoritmo, dos que consideran minimizar únicamente un objetivo a la vez, y otra que tiene en cuenta ambos objetivos y que aplica el concepto de dominancia. Estas variantes se desarrollaron con la intención de reducir el tiempo de ejecución del algoritmo, y que además nos permitiera encontrar la mejor solución para cada uno de los objetivos, determinando así los extremos del frente de Pareto. Se puede utilizar la versión multiobjetivo en caso de querer hallar el resto de las soluciones que conforman el frente.

A pesar de las múltiples mejoras que se realizaron al algoritmo con el fin de optimizar su performance, no se pudo obtener resultados para instancias más grandes que unas pocas sesiones y 4 recursos, dado que los tiempos de ejecución del mismo superaban las dos semanas. La razón principal del elevado tiempo de ejecución es la gran cantidad de soluciones que debe evaluar, creciendo de forma exponencial con cada sesión a considerar, y la poca capacidad de aplicar el algoritmo de poda tempranamente, ya que difícilmente una solución candidata que aún no haya considerado todas las sesiones sea peor que una solución final.

Algoritmo 12 Backtracking

```
Arribos, Recursos                                          ▷ Variables globales
FrentePareto                          ▷ Contiene las soluciones no dominadas

function EncontrarSoluciones(candidata, indice, sistema)
    ▷ La primer invocación se realiza con Indice = 0
    if solucionesDominanACandidata(FrentePareto, candidata) then
        return
    else if indice = obtenerCantidad(Arribos) then
        agregarSolucion(FrentePareto, candidata)
        return
    end if
    a = obtenerArribo(Arribos, indice)    ▷ Puede corresponder a inicio o fin de sesión
    if a.EsInicio then
        for all r ∈ Recursos do
            asignarRecurso(candidata, r, indice)
            asignarRecurso(a, r)
            agregarIngresoASistema(Sistema, a, r)
            consumo = calcularConsumo(Sistema, obtenerTiempo(a))
            sobreasignacion = calcularSobreasignacion(Sistema, obtenerTiempo(a))
            asignarObjetivos(candidata, consumo, sobreasignacion)
            EncontrarSoluciones(candidata, indice + 1, Sistema)        ▷ Recursión
            quitarIngresoASistema(Sistema, a, r)
        end for
    else
        asignarRecurso(obtenerArriboIngreso(a), r)
        agregarSalidaASistema(Sistema, a, r)
        consumo = calcularConsumo(Sistema, obtenerTiempo(a))
        sobreasignacion = calcularSobreasignacion(Sistema, obtenerTiempo(a))
        asignarObjetivos(candidata, consumo, sobreasignacion)
        EncontrarSoluciones(candidata, indice +1, Sistema)             ▷ Recursión
        quitarSalidaASistema(Sistema, a, r)
    end if
end function
```

5.7. Algoritmo evolutivo

Una de las soluciones propuestas para resolver el problema de HCSP fue implementar un algoritmo evolutivo. Optamos por utilizar un algoritmo evolutivo multiobjetivo, basado en el esqueleto de NSGA-II. En esta sección se presenta la definición del mismo.

Codificación Las soluciones del algoritmo fueron codificadas en un único cromosoma, conteniendo la asignación a un recurso para todas las sesiones de usuarios remotos, la asignación de usuarios presenciales no forma parte de las soluciones construidas por el algoritmo evolutivo, sino que se determinan en base al recurso de preferencia del usuario.

Sea $R = [r_1 \dots r_M]$ el conjunto de recursos, $S = [s_1 \dots s_N]$ el conjunto de sesiones, y $SR = [sr_1 \dots sr_P]$ el subconjunto de las sesiones remotas de S. El cromosoma es entonces un arreglo de enteros $A = [a_1 \dots a_P]$ de largo P, donde Si r_j es el valor de la variable a_i del arreglo, entonces a la sesión remota sr_i se le asignará el recurso r_j. Los valores del arreglo A se encuentran comprendidos entre 1 y M, dado que únicamente pueden corresponder a uno de los recursos de la instancia. La codificación no admite la creación de soluciones no válidas, por lo que no es necesario implementar algoritmos de corrección para los operadores de mutación y cruzamiento.

Presentamos a continuación un ejemplo de codificación: sea un conjunto de recursos $R = [r_0, r_1, r_2]$, un conjunto de sesiones $S = [s_0, s_1, s_2, s_3, s_4, s_5, s_6]$ todas con un tiempo de inicio igual a 0, y donde s_2 y s_6 corresponden a sesiones presenciales con preferencia por el recurso r_2. Por lo que el subconjunto de sesiones remotas sería $SR = [s_0, s_1, s_3, s_4, s_5]$, y una posible solución correspondería al cromosoma $A = [r_2, r_0, r_2, r_0, r_1]$ representando la siguiente asignación de recursos:

- $r_0 = [s_4, s_1]$
- $r_1 = [s_5, s_6]$
- $r_2 = [s_2, s_3, s_0]$

Dado que la sesión s_6 encuentra el recurso r_2 ocupado por otro usuario presencial, procede a ocupar el recurso más cercano.

Metaheurística utilizada

Para resolver nuestro problema utilizamos el esqueleto de la metaheurística *NSGA-II*, con los mismos operadores de cruzamiento, mutación y selección que los utilizados en el libro de Coello et al. (2007).

Operador de Selección Como operador de selección utilizamos el algoritmo de torneo binario descrito en la página 21. La implementación del algoritmo utilizada corresponde a la presentada en Deb et al. (2002), y utiliza la comparación por distancia de crowding en el caso que al comparar dos soluciones ninguna domine a la otra.

Operador de Mutación Para realizar la mutación de cada solución utilizamos el operador *BitFlipMutation*. Este operador recorre el arreglo de enteros y para un conjunto aleatorio de ellos cambia el valor por otro elegido al azar.

Operador de Cruzamiento Como operador de cruzamiento utilizamos el operador de *cruzamiento en un punto*, descrito previamente en la página 11. Este algoritmo toma dos soluciones padres, elige un punto o índice de referencia, y construye dos hijos donde cada hijo toma las variables que se encuentren antes de ese punto de uno de los padres y las variables posteriores del otro.

Criterio de parada El criterio de parada determina cuando el algoritmo evolutivo debe detener su ejecución. En esta implementación realizamos un estudio para determinar la cantidad de evaluaciones de soluciones que permitía obtener los mejores resultados, sin incrementar demasiado el tiempo de ejecución, y finalmente optamos por detener la ejecución luego de evaluar 25.000 soluciones.

Parámetros y salida del algoritmo Para poder ejecutar el algoritmo evolutivo se debe proveer al mismo un conjunto de parámetros, que le permitan a este definir la instancia del problema, parametrizar los operadores evolutivos y escribir los resultados de la ejecución.

Para poder definir las instancias se le brinda al algoritmo las rutas donde se encuentran los archivos que definen las sesiones y los recursos. El archivo de las sesiones contiene toda la información necesaria para poder determinar qué usuarios ingresarán al sistema, en qué momento y cuáles serán sus requerimientos. El archivo que define los recursos contiene una lista de los recursos disponibles en el sistema, y las características de éstos. Ambos archivos son descritos en mayor detalle en la sección 5.9.2.

El algoritmo recibe como parámetros el tamaño de la población que utilizará, las probabilidades de cruzamiento y mutación, y la cantidad máxima de evaluaciones como criterio de parada. Se definieron dos parámetros opcionales que permiten modificar las instancias utilizadas. Uno determina el tiempo máximo a considerar para las sesiones, asignar un valor a este parámetro ocasiona que todos las sesiones que se inicien en el sistema luego del tiempo especificado serán descartadas. El segundo parámetro determina la cantidad máxima de recursos a ser utilizados, de la misma forma, los recursos que queden por fuera de esta cantidad serán descartados y no se utilizarán.

Para que el algoritmo pueda retornar los resultados, se le indica la ruta dónde debe escribir los mismos. En esta ruta el algoritmo escribe para cada solución un arreglo conteniendo la asignación de recursos para todas las sesiones consideradas, y los valores de ambos objetivos para la solución.

5.8. Implementación de la solución

Para solucionar el problema presentado se procedió en etapas. En una primera instancia se implementó una herramienta que recaba datos del uso actual de alguno de los salones de informática de la Facultad de Ingeniería, UDELAR. Los datos recabados incluyen información del uso de los recursos, los tipos de usuarios de los salones, la duración de las sesiones de los usuarios, y los procesos y programas que se ejecutan en una sesión (e.g. editor de texto, navegador web, entorno gráfico).

En una segunda etapa, se analizaron estos datos determinando algunas características claves del uso del salón. Se construyó un modelo informático de uno de los salones de informática, permitiéndonos probar y comparar las distintas soluciones y resultados de los algoritmos de asignación, y generar un conjunto de instancias del problema. Para estudiar

esto se varió la cantidad de recursos del salón, la cantidad de sesiones consideradas y el tiempo entre la primera y última sesión.

Finalmente se desarrollaron distintos algoritmos que realizan la asignación de usuarios a recursos, y se compararon las distintas soluciones obtenidas.

5.8.1. Componentes de la solución

Dada la complejidad de un sistema para la asignación de usuarios en un salón de informática, su solución se dividió en cuatro componentes: un sistema que recopila la información de los recursos informáticos, compuesto por un cliente y un servidor, los algoritmos de asignación presentados en este capítulo, una interfaz para los usuarios y un planificador que conecta la interfaz de usuario con los algoritmos de asignación. En nuestro trabajo se desarrollaron tres de ellos, siendo parte del trabajo a futuro la definición e implementación del planificador. Para realizar la evaluación experimental se implementó además un modelo que simula el salón de informática, manteniendo el estado de los usuarios y recursos. Se realizaron también algunos estudios a modo de prueba de concepto para una futura implementación de la solución (ver Sección 6.2). A continuación se presentan los componentes de la solución.

Recopilación de datos El primer componente permite recabar información del uso actual de los salones de informática. Se desarrolló una plataforma cliente/servidor en la que los clientes fueron instalados en las PC del salón, y reportan datos a un servidor central que recaba esta información y la almacena en una base de datos. La información recabada contiene el estado actual de cada PC, el uso que los usuarios le dan a los recursos que utilizan, información de los procesos que ejecutan, el uso de memoria RAM y CPU de cada usuario, y el tiempo de su ingreso y egreso al sistema.

Algoritmos de asignación El segundo componente consta de los algoritmos que realizan la planificación de la asignación de usuarios. Todos los algoritmos reciben un conjunto de recursos y una lista de sesiones, y retorna la asignación de cada sesión a uno de los recursos, utilizando el mismo modelo de salón para calcular los valores de los objetivos correspondientes a cada solución. Los algoritmos resuelven la asignación de los usuarios de tipo remoto, y determinan el recurso a utilizar de los usuarios presenciales, siguiendo la lógica de encontrar el recurso más cercano al de preferencia del usuario que no esté ocupado por otro usuario presencial. Las instancias del problema se construyeron de forma que en el sistema nunca existan más usuarios presenciales que recursos.

Interfaz web El tercer componente del sistema es la interfaz de usuario. Esta interfaz se desarrolló como una aplicación web que permite el registro de usuarios. En la misma, se identifican al menos dos roles de usuarios, uno corresponde al administrador de la plataforma y el otro a los usuarios del salón. Los administradores de la plataforma pueden ver todos los recursos disponibles, consultar el estado actual e histórico de cada uno de ellos, y realizar algunas tareas básicas sobre los mismos, como apagar, encender o reiniciar un equipo. Los usuarios de la plataforma pueden solicitar un recurso informático, especificando si el mismo será utilizado de forma presencial o remota, el tiempo que estiman durará su sesión y el tipo de tareas que realizarán (programación, ejecución de tareas intensivas, navegación web, etc.). Además, un usuario puede elegir ingresar una tarea para que el sistema la ejecute cuando resulte más conveniente para los objetivos de

consumo energético y sobreasignación. En el Apéndice C.1 se presenta más información sobre la aplicación web y sus funciones.

Planificador El cuarto componente, el planificador, no fue implementado. Su tarea consiste en procesar el pedido de un recurso de parte de un usuario, ejecutar los algoritmos de planificación, y retornar al usuario el recurso asignado. Además, debe determinar el mejor recurso y momento para ejecutar las tareas ingresadas por los usuarios.

Modelo del salón Para realizar la evaluación experimental se simuló el comportamiento del salón de informática mediante un modelo que denominamos Sistema. Este modelo mantiene información del estado de cada uno de los recursos y sus sesiones activas, y en base a esta información calcula su consumo energético y sobreasignación. El modelo permite tomar la información de los recursos que lo componen de un archivo, lo que facilita la utilización de múltiples instancias durante las pruebas.

Las principales funcionalidades brindadas por el modelo son:

- Inicio de sesión en el sistema: inicia la sesión de un usuario en un recurso.
- Fin de sesión en el sistema: da por finalizada la sesión del usuario en el recurso que tenía asignado.
- Cantidad usuarios asignados: retorna la cantidad de usuarios asignados en el Sistema del recurso indicado.
- Existe usuario presencial: comprueba si existe un usuario de tipo presencial asignado del recurso indicado.
- Obtener capacidad máxima: retorna el Uso de Recurso máximo que puede tener el recurso indicado
- Obtener recurso cercano: busca el recurso sin usuarios presenciales más cercano del recurso indicado.
- Obtener usuarios actuales: retorna la lista actual de usuarios asignados para cada uno de los recursos.
- Quitar usuario: permite deshacer una asignación de un usuario.
- Obtener cantidad de recursos: retorna la cantidad de recursos en el Sistema.

En el Anexo A.1 se presenta un detalle de implementación del Sistema.

5.8.2. Sistema para la recopilación de datos

Para recabar información del uso de los recursos de los salones de informática se desarrolló un sistema con arquitectura cliente-servidor, que obtiene la información de cada recurso, la envía a un servidor central y éste la almacena para que posteriormente pueda ser utilizada.

Para desarrollar el sistema se contempló que los clientes fueran multiplataforma para poder instalarlos en recursos con diferentes sistemas operativos. Además, los clientes deben ser muy livianos para que el consumo en recursos de su ejecución altere lo mínimo posible la lectura de datos. Para facilitar el despliegue de los clientes, éstos deben tener un proceso de instalación sencillo y no requerir librerías o frameworks que no estuviesen instalados en los recursos a estudiar.

Clientes Los clientes del sistema de recopilación de datos cumplen el rol de recabar información de los recursos y enviarla al servidor. La información obtenida se divide en: información general del recurso e información de los usuarios que se encuentren en el mismo.

Cada recurso se registra en el servidor la primera vez que envía información, en este registro inicial envía el nombre de la PC, su dirección física (dirección MAC por su sigla en inglés que significa control de acceso al medio), la cantidad total de memoria RAM, la cantidad de cores y arquitectura del procesador, y el nombre y versión del sistema operativo que ejecuta. Al registrarse obtiene un identificador único que enviará en los siguientes informes y le permite al sistema identificar de dónde proviene la información. Una vez registrado, el sistema reportará periódicamente una actualización de su estado general incluyendo: dirección IP, tiempo que lleva encendido, cantidad de usuarios activos, cantidad de procesos activos, cantidad de procesos total, uso de CPU y uso de memoria RAM.

Además de la información general de la PC, el cliente también reporta información específica de los procesos ejecutados por usuarios, incluyendo: el usuario que ejecutó el proceso, el nombre e identificador del proceso, la fecha y hora de inicio y fin de ejecución, los consumos mínimos, promedios y máximos de CPU y memoria RAM.

El script desarrollado no permanece en ejecución, sino que es invocado como una tarea programada cada un intervalo de tiempo parametrizable, y cuando termina de procesar la información finaliza su ejecución. Para poder informar de procesos de larga ejecución, almacena en disco una lista de los procesos y su estado actual, identificador y uso de recursos, y cuando detecta que un proceso terminó su ejecución reporta un resumen de la información del mismo y lo borra del registro en disco.

Servidor El servidor del sistema cumple el rol de recibir la información enviada por los clientes, y almacenarla de forma ordenada para facilitar su procesamiento y posterior análisis. Es deseable que el mismo sea rápido, compatible con distintas versiones de los clientes, y que permita escalar la cantidad de clientes que envían la información.

Este servidor recibe información de dos tipos, una concerniente a la información general de la PC, y otra con información específica de los procesos que se ejecutan en la misma. Cuando recibe información de una PC por primera vez debe registrarla. Para ello, busca el primer identificador libre en su base de datos y lo devuelve al cliente que lo invocó. Una vez registrada la PC puede comenzar a recibir información general de la misma, además de la información de sus procesos en ejecución. Todos estos datos los almacena en una base de datos para su posterior análisis.

Debido a que este servidor recibirá información constantemente por parte de diversos clientes, incluso de distintos sistemas operativos, deberá permanecer corriendo durante todo el tiempo. Los instantes en que el servidor esté caído o su conexión de red esté fallando, no se almacenará información en la base de datos correspondientes a los usuarios y procesos de cada PC.

Implementación del cliente y servidor

Para la implementación del cliente optamos por el lenguaje de scripting multiplataforma Python en su versión 2.7, por ser la que se encontraba instalada. Los detalles del lenguaje se pueden consultar en la web oficial de Python Software Foundation (2016b). Dado que el cliente debía poder ejecutarse en plataformas Linux y Windows, se dividió

el mismo en varios componentes, separando aquellas partes generales de las específicas de cada plataforma.

En el cliente se desarrolló de forma común a ambas plataformas las funcionalidades de empaquetar la información en objetos JSON, consumir la API REST y almacenar la información entre ejecuciones del script. Sin embargo, debido a que una parte de la información recabada se debió implementar mediante comandos específicos de cada sistema operativo, se desarrollaron componentes independientes para Linux y para Windows.

Para la recopilación de información independiente del sistema operativo se utilizó el módulo estándar de Python *platform*, detalles en la web de Python Software Foundation (2016a). Este módulo brinda información del sistema operativo del recurso, el nombre de la PC, y la arquitectura y cantidad de cores del procesador.

En Windows utilizamos un módulo de Python llamado *psutil*, desarrollado por Rodola (2016), que nos permite acceder a información de los usuarios activos en el recurso, y el uso de CPU y memoria RAM global del mismo. Para obtener información específica de los procesos en ejecución analizamos la salida del comando *Tasklist*, que reporta todos los procesos del sistema junto a su uso de memoria RAM y procesador. Detalles del comando en Microsoft TechNet (2016).

En el cliente Linux el módulo *psutil* no se encontraba instalado y decidimos no utilizarlo. Como alternativa se utilizó el comando *who* (IEEE and The Open Group, 2016b) para obtener información de los usuarios que iniciaron sesión en una PC, y el comando *top* (LeFebvre, 2016), que brinda información de todos los procesos en ejecución, y del uso de memoria RAM y CPU de cada proceso y del total del sistema. Para extraer esta información analizamos la salida en texto del comando.

Mecanismos para la ejecución de tareas programadas se encuentran disponibles tanto en Windows como en Linux. En Windows se puede utilizar el sistema de tareas programadas integrado en el sistema operativo, y en Linux optamos por utilizar *cron* (IEEE and The Open Group, 2016a), demonio incluido en los sistemas Unix que permite la ejecución de procesos en segundo plano de forma desatendida.

Para recibir la información de los clientes, se implementó un servidor en Python que expone una interfaz basada en web services REST, utilizando JSON como formato para el intercambio de datos. Esta información recibida es almacenada en una base de datos MySql, una base de datos relacional de alta velocidad, de código abierto y multiplataforma (MySQL, 2001).

JSON (notación de objeto de JavaScript) es un formato ligero basado en texto para el intercambio de datos, que consta de un conjunto de claves-valor con la posibilidad de anidar un objeto JSON dentro de otro (ECMA, 2016). La flexibilidad de este formato nos permitió enviar información adicional desde los clientes sin necesidad de modificar la interfaz, simplemente actualizando el servidor para que la procese, y actualizando los clientes de forma paulatina.

REST (por su sigla en ingles que significa transferencia de estado representacional) es un estilo de arquitectura de alto nivel, compatible con múltiples tecnologías. REST incluye los conceptos de recursos y una interfaz uniforme, esto se puede ver como que cada recurso debe responder a los mismos métodos, sin embargo REST no especifica cuáles o cuántos son estos métodos. Normalmente se asocia un Web services REST con una arquitectura de servicios que permite utilizar los verbos del protocolo de comunicación estándar de la web HTTP (en español protocolo de transferencias de hypertexto) Get, Post, Put y Delete para realizar operaciones sobre los recursos de una aplicación (Tilkov,

2007).

La interfaz REST define los siguientes elementos: salones, PCs, datos, usuarios y procesos, cada uno de ellos representado en la base de datos mediante una tabla.

5.9. Evaluación experimental

A continuación presentaremos los lineamientos generales que seguimos en la evaluación experimental, y los estudios realizados sobre los algoritmos desarrollados para la resolución del problema de HCSP. Se realizó un estudio de ajuste paramétrico para determinar el conjunto óptimo de parámetros para el algoritmo evolutivo, también se realizó un estudio comparativo de las soluciones obtenidas por todos los algoritmos para un conjunto de siete instancias del problema.

De los algoritmos implementados, las heurísticas *minmin*, *DetParam*, y los algoritmos de *backtracking*, *round robin* y *asignación predeterminada*, son algoritmos deterministas, por lo que dado una instancia del problema siempre reportarán las mismas soluciones. Por lo tanto, se requiere únicamente una ejecución de cada algoritmo por instancia del problema. Por otra parte, debido a que el algoritmo evolutivo es un procedimiento estocástico, requiere realizar múltiples ejecuciones independientes para obtener resultados estadísticamente significativos. Para lograr la independencia de las ejecuciones se debe inicializar el generador de números aleatorios con una semilla diferente cada vez.

En el estudio de ajuste paramétrico se realizaron 20 ejecuciones independientes de cada variante del algoritmo, y se reportaron las siguientes métricas específicas de AE multiobjetivo: *Hypervolumen*, *Spread*, *Epsilon* y la cantidad de soluciones no dominadas ($\#ND$). Para comparar los valores de las métricas utilizamos el test de Friedman, con el cual elaboramos un ranking de los algoritmos para cada una de ellas. Además del ranking por métrica, se calcula la diferencia relativa de los valores de cada métrica.

Para la comparación de los algoritmos, reportamos para cada uno el valor de consumo energético y sobreasignación de cada solución, y en particular para el AE, realizamos 30 ejecuciones independientes para cada instancia. Luego seleccionamos de las soluciones devueltas en cada ejecución el mejor valor para cada uno de los objetivos, además de la mediana de la soluciones ordenadas por consumo energético. Para poder utilizar la media de los valores obtenidos para cada uno de los objetivos, validamos la distribución normal de los resultados con el test de Shapiro-Wilk con un intervalo de confianza del $95\,\%(\alpha = 0{,}05)$,

Todas las ejecuciones fueron realizadas en una laptop con un procesador Intel Core i5-5200U a 2.2 GHz, con 12GB de memoria RAM y disco de estado sólido, con sistema operativo Windows 10.

5.9.1. Análisis de los datos recopilados

Utilizando el sistema de recopilación de datos descrito en la sección 5.8.2, se recabó información con una frecuencia de 5 minutos de un total de 44 computadores, todas del salón 114 de Facultad de Ingeniería, UDELAR, durante un período de 52 días entre el 27/07/2015 y el 15/08/2015.

La frecuencia de muestreo elegida nos permitió recopilar datos con suficiente precisión sin impactar significativamente el PC dónde ejecuta. Durante la ejecución del cliente se observó el consumo de recursos del mismo para descartar que afectara las lecturas obtenidas de los recursos informáticos dónde estaba instalado. La observación se realizó

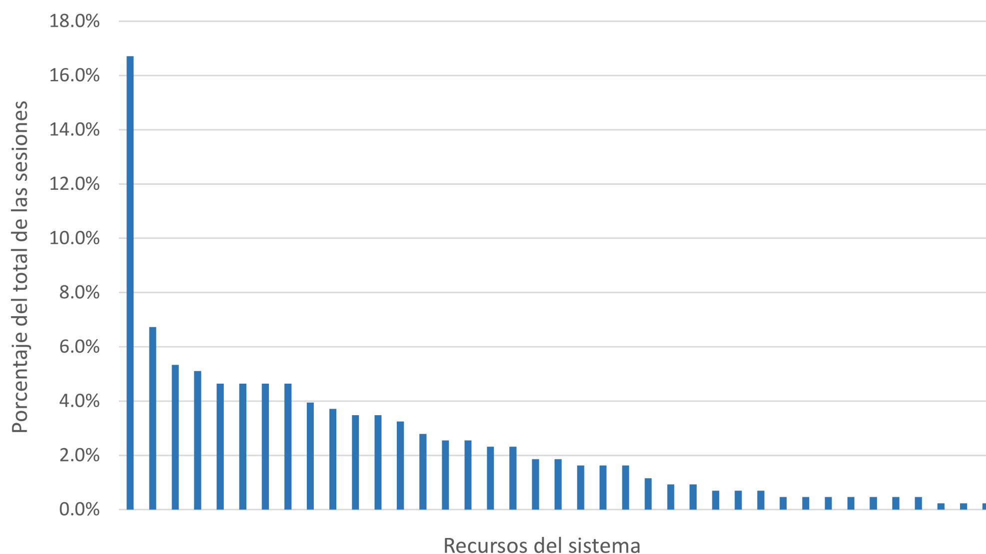


Figura 5.1: Porcentaje del total de sesiones que corresponde a cada recurso, en base a los datos recabados en el salón de informática 114 de la Facultad de Ingeniería, UDELAR.

observando las salidas del comando top mientras se ejecutaba la herramienta, y el consumo de recursos demostró ser mínimo siendo menor a 4 MB de memoria RAM y un uso de CPU cercano al 0 %.

En este período se registraron más de 1.2 millones de datos sobre el estado de las PCs, más de 20 mil registros de procesos individuales y casi 500 sesiones de usuarios, 289 presenciales y 208 remotas, correspondiendo a 145 usuarios diferentes y con una media de 3.4 sesiones por usuario. Los datos no incluyen a usuarios propios del sistema operativo como *root*, *dbus* o *ntp*. Para diferenciar a los usuarios remotos de los usuarios presenciales, se eligió un conjunto de procesos que son representativos de un usuario presencial, incluyendo el navegador Firefox y los procesos de la interfaz gráfica KDE, y se marcaron aquellas sesiones de usuarios que ejecutaron estos procesos como sesiones presenciales.

A partir de la información recabada se pueden extraer algunos datos sobre el uso de los salones de informática. Como muestra la gráfica 5.2, la mayor parte del tiempo los recursos se encuentran ociosos, siendo únicamente el 1.3 % de los recursos cuyo porcentaje de registros con al menos un usuario supera el 50 %, siendo el valor promedio 15 %. Se pudo corroborar que la distribución de los usuarios no se realiza de forma uniforme entre los recursos, sino que uno de los recursos acumula el 25 % de las sesiones de usuarios remotos y el 17 % del total de las sesiones. Esto se observa con claridad en la gráfica 5.1.

El tiempo máximo de duración de una sesión remota fue de 9 horas, y 8 horas y media para una sesión principal, siendo el tiempo mínimo registrado de 30 segundos, y promedio de 1 hora y 30 minutos de duración. En los resultados salvadas se notó una diferencia entre la cantidad de procesos ejecutados en una sesión, siendo el promedio 71 para las sesiones presenciales y tan solo 7 procesos para las remotas.

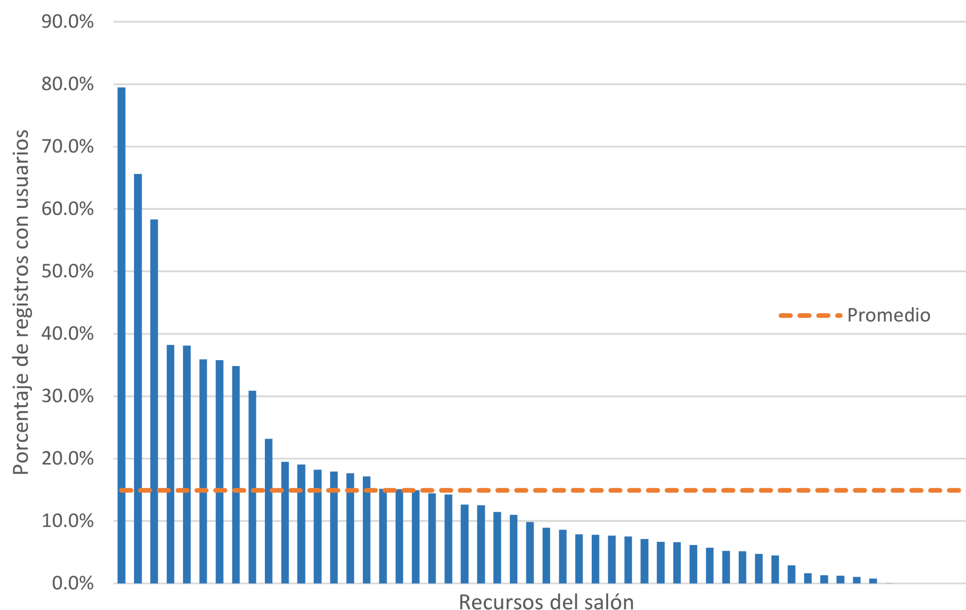


Figura 5.2: Representa el nivel de uso de los recursos, medido como el porcentaje del total de los registros que contienen al menos un usuario activo para cada recurso. Datos recabados en el salón de informática 114 de la Facultad de Ingeniería, UDELAR.

5.9.2. Definición de las instancias

Para evaluar los algoritmos propuestos se definieron un conjunto de instancias basadas en los datos analizados en la sección anterior. Luego de procesar la información contenida en la base de datos, se generaron dos archivos, uno conteniendo todos los datos de los recursos que reportaron datos, y otro conteniendo información de todas las sesiones de usuarios en el sistema.

El archivo de recursos contiene en cada línea la información de un recurso, incluyendo: el identificador del recurso en el sistema, el nombre, la cantidad de memoria RAM, la cantidad de cores de la CPU, el coeficiente de uso de usuarios presenciales, calculado como la cantidad que utilizó este recurso sobre el total del sistema, y un detalle del consumo energético en idle y para distintos porcentajes de uso de la CPU (intervalos del 10 %).

Las sesiones son definidas en otro archivo donde cada linea representa una sesión, conteniendo la información del momento de inicio de sesión, su duración, consumo promedio y máximo de CPU y memoria RAM del usuario durante la misma, el tipo de usuario pudiendo ser presencial o remoto, y el recurso de preferencia del usuario.

Para generar los conjuntos de sesiones que conforman las instancias se implementó un algoritmo que genera el conjunto de sesiones, tomando como parámetros la cantidad de sesiones a generar, el lapso de tiempo en el que ingresarán al sistema todos los usuarios, la cantidad de recursos informáticos considerados y el porcentaje de usuarios presenciales. Las instancias generadas por este algoritmo respetan el límite establecido que la cantidad de usuarios presenciales concurrentes no supere la cantidad de recursos, ésto se implementó así ya que en la realidad no sería posible que dos usuarios presencia-

les compartan un mismo recurso, sino que el usuario que llega al salón y no encuentra ningún recurso libre se retira sin haber ingresado al sistema. El algoritmo también considera un tiempo máximo de 1 hora desde el inicio de la última sesión para que todos los usuarios hayan abandonado el sistema. Ésto se definió así para disminuir el tiempo en la instancia donde no ingresen usuarios al sistema y únicamente permanezcan las sesiones ya iniciadas, disminuyendo progresivamente su número. En el algoritmo 13 se presenta el pseudocódigo del procedimiento detallado para generar sesiones.

Algoritmo 13 Pseudocódigo del algoritmo para generar sesiones de usuario

```
Recursos = Obtener recursos disponibles desde archivo            ▷ Variables globales
SesionesBase = Obtener sesiones presenciales y remotas desde archivo

function GenerarSesiones(CantSesiones, HorasMax)
    probPresencial = 0.25
    DistSesiones = HorasMax ÷ CantSesiones
    Tiempo = 0
    Sesiones = {}                                     ▷ Contabiliza sesiones activas
    Remotas = Obtener sesiones remotas de SesionesBase
    Presenciales = Obtener sesiones presenciales de SesionesBase
    for indice ∈ (1 . . . CantSesiones) do
        Avanzar tiempo según intervalo entre sesiones (DistSesiones)   ▷ Modifica cant
sesiones activas
        if random() <= ProbPresencial then        ▷ Si tocó presencial o hay esperando
            Agregar sesion a la cola
        end if
        if hay sesion en cola y cantidad de presenciales es menor a la cantidad de
recursos then
            Quitar sesion de la cola
            sesion = Sacar una sesion al azar de Presenciales
        else
            sesion = Sacar una sesion al azar de Remotas
        end if
        Setearle tiempo de arribo a la sesion con la variable Tiempo
        Setearle la duracion a la sesion con Min(HorasMax, Tiempo + sesion.Duracion)
▷ Ningún usuario se queda después de 1 hora del fin de ejecución
        Agregar sesion a Sesiones
        Tiempo = Tiempo + DistSesiones
        // Si se vacía Remotas o Presenciales se vuelven a copiar de SesionesBase
    end for
end function
```

De los datos observados del salón de informática se observó que aproximadamente la mitad de los sesiones fueron presenciales, sin embargo dado que los algoritmos diseñados actúan sobre los usuario remotos, se decidió que la proporción entre usuarios presenciales y remotos fuera diferente.

Utilizando el algoritmo descrito, y en base a los datos recopilados de la Facultad de Ingeniería se definieron en total 9 instancias: una para el estudio paramétrico, una reducida para el algoritmo de backtracking, y 7 para el estudio comparativo de los al-

goritmos implementados. La instancia del estudio paramétrico con 25 recursos y 500 sesiones distribuidas en un lapso de tiempo de 10 horas, conteniendo un 25 % de usuarios presenciales. Por otro lado, la instancia reducida se compone de 4 recursos y 23 sesiones en un lapso de 2 horas. Las instancias definidas para el estudio comparativo se detallan en la tabla 5.1.

Propiedad	Valor en la instancia						
Cant. sesiones	100	100	100	500	500	1000	2000
Cant. recursos	10	10	25	25	50	50	50
Lapso tiempo (h)	6	24	24	24	24	24	24

Cuadro 5.1: Detalle de las instancias definidas para el estudio comparativo, todas utilizan un valor de 25 % de arribos presenciales

Datos del consumo energético

Para calcular el objetivo del consumo energético se utiliza el modelo descrito en la sección 5.2.1, donde el consumo de un recurso se define como una función lineal escalonada del porcentaje de uso de CPU en intervalos del 10 %. Los valores específicos de la instancia definida fueron obtenidos de la web del consorcio de evaluación de rendimiento estándar (SPEC, por su sigla en inglés), una corporación sin fines de lucro formada para establecer, mantener y aprobar un conjunto estandarizado de pruebas de *benchmark* relevantes que pueden aplicarse a la nueva generación de equipos de alto rendimiento (SPEC, 2015). SPEC revisa y publica los resultados de las pruebas enviados por miembros de la organización y otras entidades. Este conjunto de resultados incluye información detallada del consumo energético de diferentes recursos informáticos ante pruebas de benchmark estandarizadas, incluyendo su variación ante distintos niveles de carga.

En las instancias definidas contamos con dos tipos de recursos, unos son PCs con procesador de dos núcleos de la familia Intel Core i3, con 4GB de memoria RAM. Y los otros son procesadores de cuatro núcleos de la familia Intel Core i5, también con 4GB de RAM [1]. En la figura 5.3 se ven los valores de consumo energético relativos al consumo máximo, en relación al porcentaje de uso de CPU.

[1]Datos de consumo disponibles en:
Core i3: http://www.spec.org/power_ssj2008/results/res2011q1/power_ssj2008-20110206-00346.html
Core i5: http://www.spec.org/power_ssj2008/results/res2014q4/power_ssj2008-20141023-00677.html

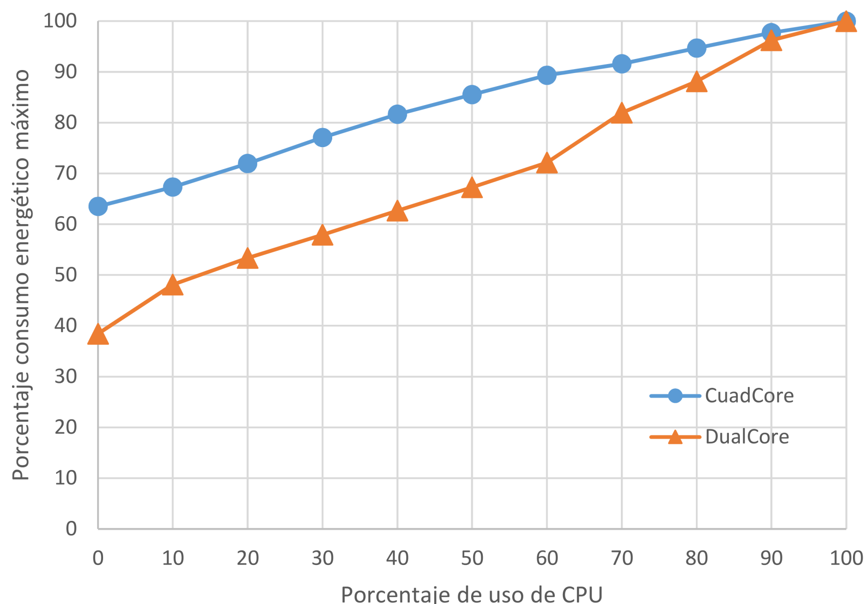


Figura 5.3: Porcentaje del consumo energético máximo en relación al porcentaje de uso de CPU para los dos tipos de recursos modelados en las instancias del problema de HCSP.

Ajuste paramétrico

Con el objetivo de obtener los mejores resultados posibles del algoritmo evolutivo, se realizó un test de ajuste paramétrico. En esta prueba se evaluaron variantes del algoritmo, utilizando distintos valores para los parámetros de los operadores evolutivos. La prueba se dividió en dos etapas, en una primera se estudió el efecto de distintos parámetros para el tamaño de la población ($\#P$), la probabilidad de mutación (p_M) y la probabilidad de cruzamiento (p_C). En una segunda etapa se estudiaron distintos valores para el criterio de parada del algoritmo.

Primera etapa estudio paramétrico

El estudio se realizó sobre los siguientes parámetros: el tamaño de la población ($\#P$), la probabilidad de mutación (p_M) y la probabilidad de cruzamiento (p_C). Los valores que consideramos para estos parámetros fueron #P $\in$ (50, 100, 150), $p_M \in$ (0.01, 0.05, 0.1), y $p_C \in$ (0.4, 0.6, 0.8). En total definimos 27 algoritmos diferentes, uno por cada combinación posible de los parámetros considerados.

En la tabla 5.2 se presentan los resultados del estudio paramétrico.

Analizando los resultados de la media porcentual y observando el ranking del HV, elegimos utilizar la siguiente combinación de parámetros en el resto del estudio: tamaño de población **50**, probabilidad de mutación **0.05** y probabilidad de cruzamiento **0.8**.

#p	p_C	p_M	HV	Spread	Epsilon	#ND	Diferencia media
50	0.4	0.01	1 %	3 %	24 %	0 %	10 %
		0.05	1 %	5 %	18 %	1 %	8 %
		0.10	1 %	1 %	22 %	8 %	8 %
	0.6	0.01	1 %	4 %	16 %	0 %	7 %
		0.05	1 %	4 %	13 %	2 %	6 %
		0.10	1 %	2 %	7 %	5 %	3 %
	0.8	0.01	1 %	2 %	17 %	2 %	7 %
		0.05	0 %	3 %	0 %	3 %	1 %
		0.10	1 %	1 %	5 %	1 %	2 %
100	0.4	0.01	0 %	4 %	10 %	2 %	5 %
		0.05	1 %	2 %	16 %	2 %	6 %
		0.10	0 %	2 %	4 %	4 %	2 %
	0.6	0.01	1 %	3 %	10 %	5 %	4 %
		0.05	0 %	2 %	1 %	10 %	1 %
		0.10	1 %	5 %	12 %	5 %	6 %
	0.8	0.01	1 %	3 %	10 %	2 %	5 %
		0.05	1 %	2 %	20 %	2 %	8 %
		0.10	1 %	2 %	9 %	1 %	4 %
150	0.4	0.01	1 %	3 %	20 %	5 %	8 %
		0.05	2 %	3 %	34 %	2 %	13 %
		0.10	1 %	2 %	24 %	3 %	9 %
	0.6	0.01	1 %	2 %	12 %	1 %	5 %
		0.05	1 %	1 %	17 %	3 %	7 %
		0.10	1 %	3 %	21 %	5 %	8 %
	0.8	0.01	1 %	4 %	8 %	6 %	4 %
		0.05	1 %	0 %	6 %	6 %	2 %
		0.10	1 %	5 %	15 %	2 %	7 %

Cuadro 5.2: Resultados del análisis paramétrico. Se presentan los parámetros evaluados, y para cada métrica la diferencia relativa al mejor valor de la misma. Se destacaron los valores tomados como referencia y el algoritmo elegido luego del análisis. Se consideró una instancia de 500 sesiones en 25 recursos, durante un lapso de 10 horas. Por cada algoritmo se realizaron 20 ejecuciones independientes.

Segunda etapa estudio paramétrico

En esta segunda etapa, se realizó un estudio con 3 variantes del AE con distintos valores en la cantidad de soluciones a evaluar antes de detener la ejecución del algoritmo. Se probaron los siguientes valores: 25.000, 75.000 y 150.000.

La segunda etapa del estudio se realizó de igual forma que la primera, adicionando a las métricas reportadas el tiempo de ejecución de cada algoritmo.

En todas las métricas consideradas, exceptuando *Spread*, las soluciones obtenidas evaluando 25.000 fueron mejores, resultando primeras en el ranking de *HV* y *Epsilon*. Los valores medios de las mejor solución para cada uno de los objetivos fue mejor, y su

tiempo de ejecución fue de solo 3 segundos contra 10 y 15 segundos evaluando 75.000 y 15.000 soluciones respectivamente.

Por lo tanto, el valor utilizado para el criterio de detención en el resto del trabajo fue de **25.000** evaluaciones.

5.10. Resultados

En esta sección se presentan los resultados del estudio comparativo entre todos los algoritmos implementados. La comparación se realizó sobre las 7 instancias definidas en la sección 5.9.2. Las distintas instancias del problema representan distintos escenarios, variando la cantidad máxima de usuarios concurrentes, la frecuencia de inicio de las sesiones, y la cantidad de recursos disponibles. Por ejemplo: la instancia de 100 sesiones en 25 recursos, tiene un máximo de 11 usuarios concurrentes en el sistema, con un requerimiento máximo de memoria RAM de 15 GB. Mientras que la instancia de 500 sesiones para los mismos recursos, tiene un máximo de 38 usuarios concurrentes y 47 GB de memoria RAM requeridos. La variedad de la instancias nos permite probar el comportamiento de los algoritmos bajo distintos escenarios, con distintos niveles de carga del sistema. Los principales resultados de esta comparación se muestran en el cuadro 5.3. Si bien no fue uno de los principales objetivos planteados, el tiempo de ejecución de un algoritmo de asignación es importante, el AE demoró en promedio 3 segundos y medio en completar su ejecución, mientras que los demás algoritmos demoraron un promedio de 30 milisegundos. Si bien la diferencia relativa es sustancial, 3 segundos de ejecución continúa siendo un tiempo razonable que permitiría ejecutar el algoritmo mientras un usuario espera el resultado.

Al comparar los resultados se observa que el AE obtuvo mejores resultados de sobreasignación en toda las instancias, y de consumo en las instancias más pequeñas. El algoritmo *DetParam* en general obtuvo buenos resultados de consumo energético, pero a costas de valores mucho mayores de sobreasignación, de entre un 200 % y 3.800 % respecto al mejor valor de cada instancia. Esto se debió a que el algoritmo optimiza en primera instancia el consumo energético, tomando luego las soluciones candidatas cuyo valor de consumo se encuentra en un margen de 5 % respecto a la menor y eligiendo de entre ellas la que tenga menor sobreasignación. En los momentos en que un recurso se encuentra con más del 100 % de uso promedio de CPU, su consumo energético no puede aumentar más, por lo que el valor de consumo energético de agregar otro usuario al recurso es 0. Esto ocasiona que cuando algún recurso se encuentra saturado se continúen asignándole usuarios, generando soluciones con un bajo valor de consumo energético pero muy alto valor de sobreasignación.

Las heurísticas minmin tuvieron resultados variables. Exceptuando en la primer instancia, la variante que optimiza primero el consumo energético obtuvo mejores valores para este objetivo, aunque sus soluciones tienen valores de sobreasignación muy superiores a las soluciones del AE de mejor consumo. En cambio la variante que optimiza primero la sobreasignación obtuvo soluciones más equilibradas, presentando buenos valores para este objetivo, sin incrementar tanto el consumo energético.

Los algoritmos de round robin y asignación predeterminada encontraron soluciones cercanas a los valores promedio de la mediana del AE, aunque con peores resultados para ambos objetivos. Ambos algoritmos presentaron de forma consistente peores resultados

para el objetivo de consumo energético que los demás algoritmos, y resultados intermedios para la sobreasignación. Los resultados variaron entre un 10 % y un 60 % peores para el objetivo de consumo respecto al mejor valor, y entre 60 % y 200 % peores valores de sobreasignación.

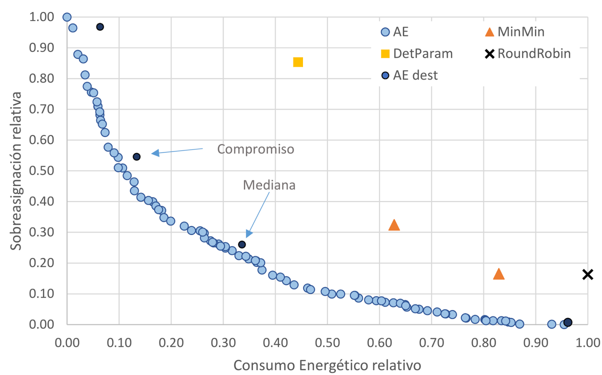
Los algoritmos basados en minmin se comportaron peor para aquellas instancias que presentaban una sobrecarga alta del sistema, mientras que el algoritmo evolutivo pudo obtener buenos resultados para todas las instancias consideradas. En la medida que la cantidad de sesiones fue aumentando, las soluciones encontradas por minmin se acercaron cada más al extremo de sobreasignación del frente de Pareto. Para las instancias más grandes el algoritmo minmin que prioriza el consumo resultó demasiado ávido, obteniendo valores de sobreasignación hasta un 800 % peor que el mejor valor encontrado. Esto se debió a que el consumo de un recurso tiene un límite cuando se encuentra al 100 % de su capacidad, el algoritmo minmin explota esta característica para mantener el consumo bajo, pero ocasionado los elevados valores de sobreasignación que se mencionaron.

Para poder visualizar mejor como se comparan las soluciones obtenidas por los distintos algoritmos, se presentan el conjunto de las soluciones para cuatro instancias del problema en los gráficos 5.4. Entre las distintas instancias varía la frecuencia en la que llegan los usuarios, recordando que el arribo de usuarios es equiespaciado, y la cantidad máximo de usuarios concurrentes en el sistema. Al comparar los conjuntos de soluciones se observa el efecto mencionado del desplazamiento del algoritmo minmin hacia el extremo de sobreasignación, efecto que también ocurre con el algoritmo DetParam. También se observa que las soluciones de round robin producen resultados buenos para el valor de sobreasignación, aunque obtiene los peores valores de energía. Sin embargo a medida que las instancias crecen, el AE es capaz de encontrar soluciones que mejoran ampliamente en ambos objetivos.

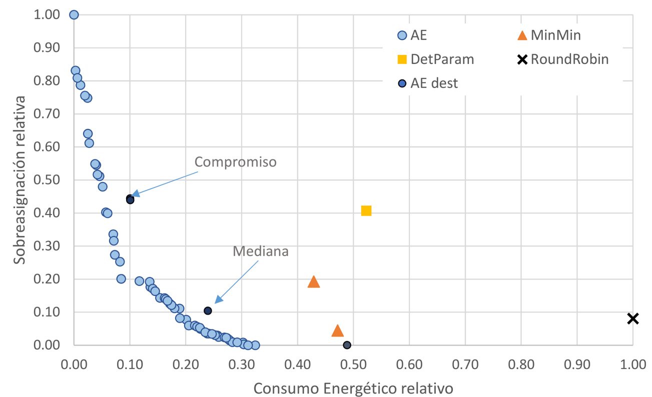
La gráfica 5.5 muestra el número máximo de usuarios concurrentes en cada uno de los recursos para la instancia más pequeña del problema. En esta figura se puede observar también, como el AE puede no asignar usuarios a un recurso, permitiendo que el sistema apague el mismo y su consumo baje a 0. Aún en la instancia más pequeña ya se puede observar como las soluciones de las heurísticas minmin, y sobretodo DetParam, tienden a sobrecargar un recurso con múltiples usuarios concurrentes, degradando la performance que perciben los usuarios que utilizan ese recurso. En la gráfica correspondiente a la instancia de 500 sesiones y 50 recursos, para poder comparar con claridad las demás soluciones la solución de DetParam no fue graficada, ya que era dominada por otras soluciones y tenía un valor de sobreasignación muy grande, 3.800 % respecto al menor.

Teniendo en cuenta todo lo discutido, el AE resulta el algoritmo que propone las mejores soluciones, siendo la versión de minmin que prioriza la sobreasignación el segundo mejor algoritmo. Esto se debe a que permite obtener valores buenos de sobreasignación, mejorando el consumo respecto a round robin, y generando una distribución pareja de los usuarios.

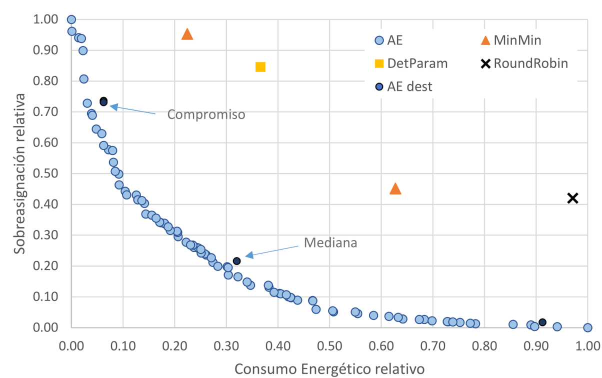
A pesar de que el algoritmo evolutivo permite obtener mejores resultados para ambos objetivos, y de que también ofrece un rango de soluciones intermedias, tiene el problema que para funcionar necesita conocer el total de las sesiones que ingresarán al sistema. En contraste con los otros algoritmos que pueden resolver la asignación de cada sesión



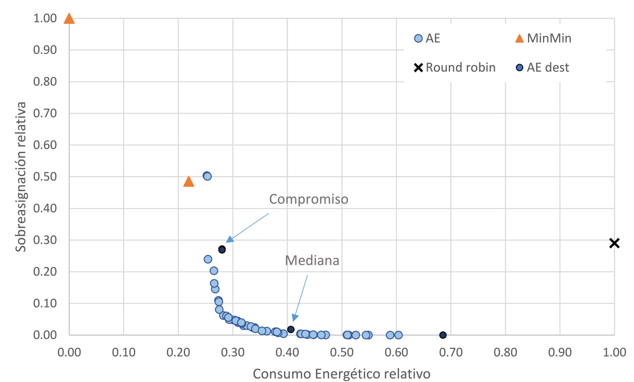
(a) 100 sesiones, 10 recursos, 6 horas



(b) 100 sesiones, 10 recursos, 24 horas



(c) 500 sesiones, 25 recursos, 24 horas



(d) 500 sesiones, 50 recursos, 24 horas

Figura 5.4: Soluciones de todos los algoritmos para cuatro instancias del problema de HCSP. Los valores se encuentran representados como valores relativos entre el menor y el mayor valor de cada objetivo. Del AE se presentan todas las soluciones no dominadas encontradas, y se destacan la media de las mejores soluciones de cada objetivo, la mediana y la solución de compromiso.

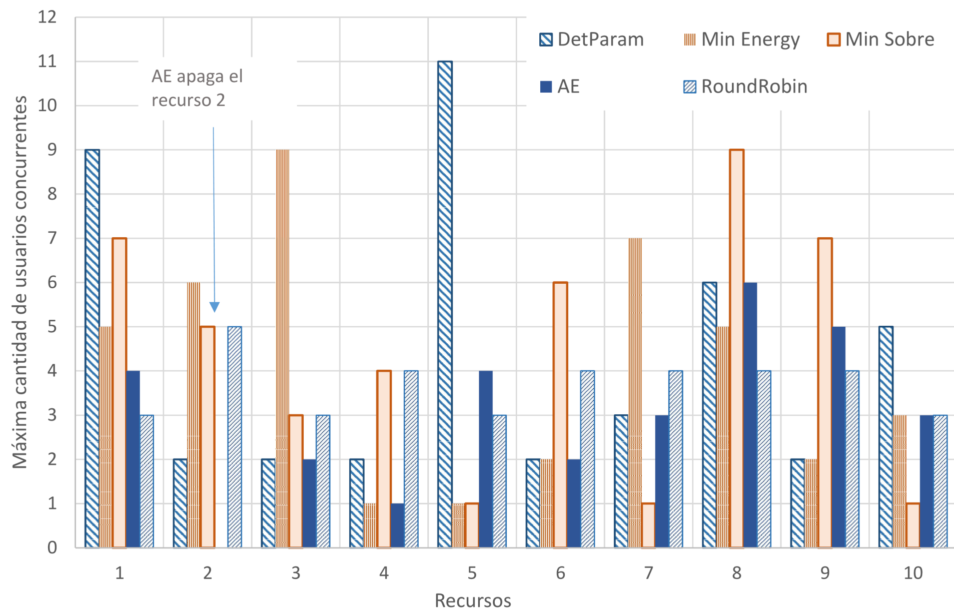


Figura 5.5: Cantidad máxima de usuarios concurrentes para las soluciones de todos los algoritmos de la instancia con 100 sesiones en 10 recurso en 6 horas.

a medida que arriban al sistema. Una posible solución a este problema, es utilizar el algoritmo evolutivo de forma periódica con una predicción de los usuarios que ingresarán al sistema en ese período. Y en base a las soluciones retornadas determinar la cantidad de recursos que deben permanecer encendidos. Luego a medida que arriben usuarios, se les puede asignar un recurso tomando el recurso asignado a la sesión más similar de la predicción, o ejecutando alguna de las otras heurísticas desarrolladas.

Instancia R×S×H	Algoritmo	Solución	Consumo	Sobreasignación
10×100×6	Minmin	Consumo	18.5 %	162.5 %
		Sobreasignación	13.6 %	326.0 %
	AE	Consumo	0.0 %	989.1 %
		Sobreasignación	21.7 %	0.0 %
		Mediana	6.6 %	260.3 %
	DetParam		9.2 %	871.1 %
	Predeterminado		20.1 %	297.9 %
	Round robin		22.6 %	160.6 %
10×100×24	Minmin	Consumo	16.8 %	191.3 %
		Sobreasignación	19.0 %	43.9 %
	AE	Consumo	0.0 %	441.1 %
		Sobreasignación	19.9 %	0.0 %
		Mediana	7.1 %	103.5 %
	DetParam		21.6 %	404.4 %
	Predeterminado		44.9 %	103.4 %
	Round robin		46.0 %	80.0 %
25×100×24	Minmin	Consumo	8.0 %	80.1 %
		Sobreasignación	18.9 %	39.8 %
	AE	Consumo	0.0 %	364.7 %
		Sobreasignación	6.5 %	0.0 %
		Mediana	2.6 %	110.5 %
	DetParam		44.2 %	202.6 %
	Predeterminado		57.7 %	22.0 %
	Round robin		56.8 %	59.0 %
25×500×24	Minmin	Consumo	3.0 %	291.6 %
		Sobreasignación	10.4 %	135.4 %
	AE	Consumo	0.0 %	224.1 %
		Sobreasignación	15.7 %	0.0 %
		Mediana	4.8 %	62.0 %
	DetParam		5.6 %	258.3 %
	Predeterminado		16.0 %	181.0 %
	Round robin		16.8 %	125.8 %
50×500×24	Minmin	Consumo	0.0 %	857.3 %
		Sobreasignación	15.9 %	416.0 %
	AE	Consumo	20.3 %	232.7 %
		Sobreasignación	49.6 %	0.0 %
		Mediana	29.4 %	15.4 %
	DetParam		6.9	3819.7 %
	Predeterminado		67.0 %	189.6 %
	Round robin		72.4 %	249.0 %
50×1000×24	Minmin	Consumo	0.0 %	593.3 %
		Sobreasignación	13.3	195.4 %
	AE	Consumo	14.5 %	209.6 %
		Sobreasignación	35.8 %	0.0 %
		Mediana	22.4 %	52.2 %
	DetParam		3.3 %	2186.1 %
	Predeterminado		32.3 %	259.3 %
	Round robin		42.0 %	174.0 %
50×2000×24	Minmin	Consumo	1.0 %	228.6 %
		Sobreasignación	7.6 %	73.6 %
	AE	Consumo	5.4 %	89.0 %
		Sobreasignación	12.5 %	0.0 %
		Mediana	8.1 %	29.2 %
	DetParam		0.0 %	609.0 %
	Predeterminado		12.0 %	71.6 %
	Round robin		13.4 %	66.0 %

Cuadro 5.3: Se presentan las diferencias relativas respecto al mejor valor obtenido de los objetivos para cada instancia (resaltado). Para el algoritmo evolutivo se realizaron 30 ejecuciones independientes para cada instancia y se tomó el valor promedio de las mejores soluciones para cada objetivo.

5.11. Resumen

En este capítulo se presentó el trabajo realizado sobre un caso particular del problema de HCSP, la asignación de usuarios en un salón de informática. El problema consta de minimizar dos objetivos, el consumo energético del conjunto de los recursos informáticos del salón, y minimizar el valor de sobreasignación de los recursos, valor que representa la insatisfacción de los usuarios del sistema.

El problema modeló el caso real de algunos de los salones de informática de la Facultad de Ingeniería, UDELAR. Para esto implementamos una herramienta que permitió monitorear el uso de estos salones, capturando información sobre el uso de los recursos por parte de los usuarios, incluyendo duración de una sesión y el uso de CPU y memoria RAM durante la misma. Fue en base a estos datos que se construyeron las instancias del problema. Otro de los resultados de este trabajo fue la creación de un modelo computacional del salón, que facilitó el desarrollo y las pruebas de los algoritmos de asignación.

Para resolver el problema se implementaron seis algoritmos, tres de ellos heurísticas, dos basadas en el algoritmo minmin, y la tercera utilizando criterios de sentido común para realizar la asignación. Además se implementó un algoritmo evolutivo multiobjetivo basado en la metaheurística NSGA-II, y como referencia se implementó un algoritmo de round robin y la asignación predeterminada tomando los recursos utilizados en las sesiones capturadas. También fue implementado un algoritmo de backtracking, pero el mismo solamente pudo ejecutarse para instancias muy pequeñas del problema, con únicamente 3 recursos y 23 sesiones, no siendo posible completar su ejecución para instancias más grandes.

En el análisis de los resultados se pudo determinar que el algoritmo evolutivo generó las mejores soluciones teniendo en cuenta la distribución de usuarios, la variedad, y los mejores valores para ambos objetivos. Los tiempos de ejecución del AE fueron mayores que los demás algoritmos, pero se mantuvieron dentro de valores razonables con un tiempo promedio de 3 segundos. Obtuvo resultados mejores o similares para todos los objetivos, además de un conjunto de soluciones intermedias que ofrecen distintos niveles de compromiso entre los objetivos del problema. También pudimos observar que el AE fue capaz de prescindir de algunos de los recursos para realizar la asignación, permitiendo que el sistema los apague y reducir así su consumo a 0.

El principal problema del AE es la necesidad de contar con todas las sesiones que ingresarán al sistema previamente a calcular la asignación. Este problema se mitigaría realizando un modelo de los usuarios esperados en base a los datos históricos que se poseen. Con estas simulaciones se puede determinar la cantidad de recursos que deberían permanecer encendidos, apagando el resto. La asignación de los usuarios que finalmente ingresen al sistema se realizaría utilizando alguno de los otros algoritmos, o tomando el valor de la asignación de la sesión prevista que sea más parecida al usuario real.

Capítulo 6

Conclusiones y trabajo futuro

Este capítulo presenta las conclusiones de la aplicación de AEs para resolver el problema de asignación de usuarios y tareas en un cluster heterogéneo. También se introducen las principales líneas del trabajo futuro.

6.1. Conclusiones

Este trabajo presentó la resolución a dos variantes del problema de HCSP utilizando algoritmos evolutivos. Se resolvieron dos variantes del problema, una enfocada a la planificación de tareas en un cluster de computadoras, y otra a la asignación de usuarios en un salón de informática. Se consideraron dos objetivos para el problema, minimizar el consumo energético de los recursos informáticos y mantener la calidad de servicio ofrecida. Se estudió el problema, se relevaron los principales trabajos de la literatura relacionada y los métodos utilizados para controlar el consumo energético en sistemas computacionales, y se realizó la formulación matemática de cada una de las variantes del problema.

Para el problema de la planificación de tareas en el cluster la calidad de servicio se baso en minimizar el makespan, el tiempo que toma ejecutar todas las tareas. Para resolver el problema se implementó NSGA-II, un AE multiobjetivo explícito, y una variante multiobjetivo de la heurística minmin. El análisis experimental se realizó sobre tres instancias sintéticas de diferentes dimensiones, donde el AE obtuvo mejores resultados para ambos objetivos, además de brindar un conjunto amplio de soluciones ofreciendo distintos niveles de compromiso entre los dos objetivos considerados.

Para el problema de la asignación de usuarios en un salón de informática, se consideró la calidad de servicio buscando minimizar la sobreasignación, una función que compara los requerimientos de los usuarios de un recurso y la capacidad máxima de éste, evaluando el nivel en el que no se puede cumplir con los requerimientos de estos usuarios (considerando el consumo de CPU y memoria RAM).

Como parte del estudio del problema se implementó una herramienta que permitió capturar información del salón de informática 114 de la Facultad de Ingeniería, UDELAR. Esta herramienta recopiló datos de 44 computadoras durante un período de 52 días. A partir de esta información se pudo determinar que el 98.7 % de las computadoras están inactivas más del 50 % del tiempo, siendo el promedio de utilización apenas el 15 %. A partir de la información se pudo obtener métricas del uso de los recursos de 500 sesiones de los usuarios del salón, que incluyen uso mínimo, promedio y máximo de memoria RAM y CPU, nombre de los procesos ejecutados y la duración de los mismos, y fecha y

hora de inicio y fin de la sesión.

Para resolver el problema se implementaron seis algoritmos, una heurística determinista basada en las ideas intuitivas de una persona para realizar la asignación, dos variantes de la heurística minmin, un algoritmo que implementa la política de round robin, un algoritmo que simula la asignación realizada por los usuarios cuando no existe ningún planificador, el algoritmo evolutivo multiobjetivo NSGA-II, y un algoritmo de backtracking.

La evaluación experimental se realizó sobre un conjunto de siete instancias realistas creadas a partir de los datos recopilados en el salón, simulando las asignaciones con un modelo desarrollado en el marco de este trabajo. En el análisis de los resultados se pudo determinar que el AE generó las mejores soluciones teniendo en cuenta la distribución de usuarios, la variedad de soluciones ofrecidas y los mejores valores para ambos objetivos. En comparación con la asignación realizada por los usuarios, el AE permitió obtener mejoras entre un **15 %** y un **55 %** en el consumo energético y entre un **60 %** y **300 %** en la sobreasignación. Comparando con una política de round robin, las mejoras fueron entre un **3 %** y un **20 %** en el consumo energético, y entre un **15 %** y **160 %** en la sobreasignación. En algunas de las instancias el AE fue capaz de realizar la asignación de los usuarios prescindiendo de algunos recursos, lo que permite apagar los mismos y reducir su consumo. Además los tiempos de ejecución del algoritmo fueron menores a tres segundos, lo que resulta muy conveniente para poder utilizarlo en una planificación *online*.

Como parte del trabajo se implementó también un sitio web que permite a los usuarios registrarse y solicitar un recurso, que es seleccionado mediante los algoritmos presentados en este trabajo respetando los objetivos de consumo energético y sobreasignación.

En base al trabajo realizado consideramos que los AEs son una técnica adecuada para resolver problemas de planificación de recursos informáticos, ya sea en la asignación de usuarios o de tareas. Permitió obtener iguales o mejores resultados que otros algoritmos deterministas, contando además con la ventaja de brindar un abanico de soluciones entre las que se puede elegir la que mejor se adapte a las circunstancias.

6.2. Trabajo futuro

A continuación se presentan las principales líneas de trabajo futuro que surgen a partir de este proyecto.

Se propone la implantación del sistema de planificación de usuarios, presentado en el capítulo 5, en un entorno real. En este sentido, para implantar el sistema presentado es necesario desarrollar el planificador, componente que realiza la comunicación entre la aplicación web a la que acceden los usuarios y los algoritmos de planificación desarrollados.

Además de la asignación de recursos para sesiones *online* de los usuarios, sería deseable que la plataforma permita aceptar tareas a ejecutar de forma asincrónica. Esto le da mayor flexibilidad a los algoritmos de asignación, ya que pueden determinar el mejor momento para ejecutar las tareas teniendo en cuenta los objetivos de consumo energético y sobreasignación. Como parte de nuestro trabajo estudiamos los mecanismos posibles para ejecutar tareas remotamente en nombre de un usuario. Estudiamos dos posibilidades para ejecutar comandos de forma remota, la librería de Python *Paramiko*, y para sistemas operativos Windows la herramienta *Plink*. Se presentan algunas pruebas de con-

cepto en el Apéndice D.1.

Otra línea de trabajo futuro consiste en considerar el costo de la energía consumida como objetivo a minimizar. Para lograr esto se deben incorporar otros factores asociados al costo del consumo energético del salón, como la variación del precio de la energía eléctrica según la hora, la época del año, la disponibilidad de fuentes alternativas de energía (e.g. solar o eólica), entre otros factores. Con estas consideraciones el sistema tendría más herramientas para determinar el horario óptimo para ejecutar las tareas diferidas, ayudando a minimizar el costo de operación del sistema.

Bibliografía

ACPI. Advanced configuration and power interface specification, 2015. URL `http://acpi.info/`. Visitado en 2015-12-22.

Y. Agarwal, S. Hodges, R. Chandra, J. Scott, P. Bahl, and R. Gupta. Somniloquy: Augmenting network interfaces to reduce pc energy usage. *NSDI*, 9:365–380, April 2009.

T. Bäck. *Evolutionary Algorithms in Theory and Practice: Evolution Strategies, Evolutionary Programming, Genetic Algorithms.* Oxford University Press, 1996. ISBN 9780195099713.

T. Bäck, D. B. Fogel, and Z. Michalewicz. *Evolutionary computation 1: Basic algorithms and operators*, volume 1 of *Basic algorithms and operators.* Taylor & Francis, 2000. ISBN 9780750306645.

J. Balladini, E. Grosclaude, M. Hanzich, R. Suppi, D. Rexachs del Rosario, and E. Luque Fadón. Incidencia de los modelos de programación paralela y escalado de frecuencia de cpus en el consumo energético de los sistemas de hpc. In *XVI Congreso Argentino de Ciencias de la Computación*, 2010.

R. S. Barr, B. L. Golden, J. P. Kelly, M. G. Resende, and W. R. Stewart Jr. Designing and reporting on computational experiments with heuristic methods. *Journal of heuristics*, 1(1):9–32, 1995.

L. A. Barroso and U. Hölzle. The case for energy-proportional computing. *IEEE Computer*, 40, 2007. URL `http://www.computer.org/portal/site/computer/index.jsp?pageID=computer_level1&path=computer/homepage/Dec07&file=feature.xml&xsl=article.xsl`.

A. Beloglazov, R. Buyya, Y. C. Lee, A. Zomaya, et al. A taxonomy and survey of energy-efficient data centers and cloud computing systems. *Advances in computers*, 82 (2):47–111, 2011.

N. Bila, E. J. Wright, E. D. Lara, J. Joshi, H. Andrés Lagar-Cavilla, E. Park, A. Goel, M. Hiltunen, and M. Satyanarayanan. Energy-oriented partial desktop virtual machine migration. *ACM Transactions on Computer Systems*, 33(1):2:1 – 2:51, 2015. ISSN 07342071.

C. Blum and A. Roli. Metaheuristics in combinatorial optimization: Overview and conceptual comparison. *ACM Computing Surveys (CSUR)*, 35(3):268–308, 2003.

T. Bridi, A. Bartolini, M. Lombardi, M. Milano, and L. Benini. A constraint programming scheduler for heterogeneous high-performance computing machines. *IEEE Transactions on Parallel and Distributed Systems*, 27(10):2781 – 2794, 2016. ISSN 10459219.

R. Buyya. Economic-based distributed resource management and scheduling for grid computing. *arXiv preprint cs/0204048*, 2002.

B. Casanova, J. Balladini, A. E. De Giusti, R. Suppi, D. Rexachs del Rosario, and E. Luque Fadón. Mejora de la eficiencia energética en sistemas de computación de altas prestaciones. In *XVIII Congreso Argentino de Ciencias de la Computación*, 2012.

J.-H. Chen. *Theory and applications of efficient multi-objective evolutionary algorithms.* PhD thesis, Doctoral dissertation, Feng Chia University, Taichung, Taiwan, 2004.

C. A. C. Coello. A comprehensive survey of evolutionary-based multiobjective optimization techniques. *Knowledge and Information systems*, 1(3):269–308, 1999.

C. A. C. Coello. Theoretical and numerical constraint-handling techniques used with evolutionary algorithms: a survey of the state of the art. *Computer methods in applied mechanics and engineering*, 191(11):1245–1287, 2002.

C. A. C. Coello, G. B. Lamont, and D. A. Van Veldhuizen. *Evolutionary algorithms for solving multi-objective problems.* Springer Science & Business Media, 2007.

O. Cordon, F. Herrera, and T. Stützle. A review on the ant colony optimization metaheuristic: Basis, models and new trends. In *Mathware & Soft Computing*, pages 2–3. Citeseer, 2002.

T. Das, P. Padala, V. Padmanabhan, R. Ramjee, and K. G. Shin. Litegreen: Saving energy in networked desktops using virtualization. In *USENIX Annual Technical Conference*. USENIX, June 2010.

K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE Transactions on Evolutionary Computation*, 6(2): 182–197, apr 2002. doi: 10.1109/4235.996017.

J. J. Durillo, A. J. Nebro, and E. Alba. The jMetal framework for multi-objective optimization: Design and architecture. In *IEEE Congress on Evolutionary Computation*. Institute of Electrical & Electronics Engineers (IEEE), jul 2010. doi: 10.1109/cec.2010.5586354. URL `http://dx.doi.org/10.1109/CEC.2010.5586354`.

J. J. Durillo, A. J. Nebro, F. Luna, B. Dorronsoro, and E. Alba. jMetal: A java framework for multi-objective optimization. *Advances in Engineering Software*, 42(10):760–771, oct 2011.

ECMA. Json, 2016. URL `http://www.json.org/json-es.html`. Visitado en 2016-11-29.

M. Ehrgott. *Multicriteria optimization.* Springer Science & Business Media, 2006.

T. A. Feo and M. G. Resende. Greedy randomized adaptive search procedures. *Journal of global optimization*, 6(2):109–133, 1995.

P. Festa and M. G. Resende. Grasp: An annotated bibliography. In *Essays and surveys in metaheuristics*, pages 325–367. Springer, 2002.

FFTW Org. Fftw home page, 2015. URL `http://www.fftw.org/`. Visitado en 2015-12-20.

C. M. Fonseca and P. J. Fleming. Multiobjective optimization and multiple constraint handling with evolutionary algorithms. i. a unified formulation. *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, 28(1):26–37, 1998.

I. Foster and C. Kesselman. *The Grid 2: Blueprint for a new computing infrastructure*. Elsevier, 2003.

M. Friedman. The use of ranks to avoid the assumption of normality implicit in the analysis of variance. *Journal of the American Statistical Association*, 32(200):675–701, 1937. doi: 10.1080/01621459.1937.10503522.

R. Gallego, A. Escobar, and E. Toro. Técnicas metaheurísticas de optimización. *Universidad Tecnológica de Pereira*, 92, 2008.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & operations research*, 13(5):533–549, 1986.

D. E. Goldberg. Genetic algorithms in search, optimization, and machine learning, addison-wesley professional. *Reading, Massachusetts, US*, 1989.

D. E. Goldberg and K. Deb. A comparative analysis of selection schemes used in genetic algorithms. *Foundations of genetic algorithms*, 1:69–93, 1991.

E. L. Hahne. Round-robin scheduling for max-min fairness in data networks. *IEEE Journal on Selected Areas in communications*, 9(7):1024–1039, 1991.

S. . . . Hartmann and . . Briskorn, D. (2. A survey of variants and extensions of the resource-constrained project scheduling problem. *European Journal of Operational Research*, 207(1):1–14, 2010. ISSN 03772217.

J. Horn, N. Nafpliotis, and D. E. Goldberg. Multiobjective optimization using the niched pareto genetic algorithm. *IlliGAL report*, 1(93005):61801–2296, 1993.

J. Horn, N. Nafpliotis, and D. E. Goldberg. A niched pareto genetic algorithm for multiobjective optimization. In *Evolutionary Computation, 1994. IEEE World Congress on Computational Intelligence., Proceedings of the First IEEE Conference on*, pages 82–87. Ieee, 1994.

T. IEEE and The Open Group. The open group base specifications issue 7, 2016a. URL `http://pubs.opengroup.org/onlinepubs/9699919799/`. Visitado en 2016-12-20.

T. IEEE and The Open Group. The open group base specifications issue 7, 2016b. URL `http://pubs.opengroup.org/onlinepubs/9699919799/`. Visitado en 2016-11-29.

S. Iturriaga, S. Nesmachnow, and C. Tutté. Metaheuristics for multiobjective energy-aware scheduling in heterogeneous computing systems. EU/Metaheuristics Meeting, Copenague, Dinamarca, 2012.

A. A. Khokhar, V. K. Prasanna, M. E. Shaaban, and C.-L. Wang. Heterogeneous computing: Challenges and opportunities. *Computer*, 26(6):18–27, jun 1993. doi: 10.1109/2.214439.

K. H. Kim, R. Buyya, and J. Kim. Power aware scheduling of bag-of-tasks applications with deadline constraints on dvs-enabled clusters. In *CCGRID*, volume 7, pages 541–548, 2007.

J. G. Koomey. Worldwide electricity used in data centers. *Environmental Research Letters*, 3(3):034008, 2008.

A. D. Kshemkalyani and M. Singhal. *Distributed computing: principles, algorithms, and systems*. Cambridge University Press, 2011.

W. LeFebvre. Unix top, 2016. URL `http://www.unixtop.org/`. Visitado en 2016-11-29.

Y. Li, Y. Liu, and D. Qian. A heuristic energy-aware scheduling algorithm for heterogeneous clusters. In *Parallel and Distributed Systems (ICPADS), 2009 15th International Conference on*, pages 407–413. IEEE, 2009.

P. Luo, K. Lü, and Z. Shi. A revisit of fast greedy heuristics for mapping a class of independent tasks onto heterogeneous computing systems. *Journal of Parallel and Distributed Computing*, 67:695 – 714, 2007. ISSN 0743-7315.

K. R. MacCrimmon. An overview of multiple objective decision making. *Multiple criteria decision making*, 3:18–44, 1973.

M. d. l. Maza and B. Tidor. An analysis of selection procedures with particular attention paid to proportional and boltzmann selection. In *Proceedings of the 5th International Conference on Genetic Algorithms*, pages 124–131. Morgan Kaufmann Publishers Inc., 1993.

L. D. Merkle and G. B. Lamont. A random function based framework for evolutionary algorithms. In *ICGA*, pages 105–112, 1997.

Microsoft TechNet. Technet tasklist, 2016. URL `https://technet.microsoft.com/es-es/library/bb491010.aspx`. Visitado en 2016-11-29.

L. Minas and B. Ellison. The problem of power consumption in servers. *Intel Developer Zone*, 2009. URL `https://software.intel.com/en-us/articles/the-problem-of-power-consumption-in-servers`. Visitado en 2016-11-30.

J. Murana, S. Iturriaga, and S. Nesmachnow. A multiobjective evolutionary algorithm for QoS-aware planning in heterogeneous computing systems. In *2014 XL Latin American Computing Conference (CLEI)*. Institute of Electrical & Electronics Engineers (IEEE), sep 2014.

A. Murias Rodríguez. Estudio de bloques constructivos en algoritmos genéticos. 2007.

A. MySQL. Mysql, 2001.

S. Nesmachnow. Computación científica de alto desempeño en la facultad de ingeniería, universidad de la república. *Revista de la Asociación de Ingenieros del Uruguay*, 61 (1):12–15, 2010.

S. Nesmachnow. An overview of metaheuristics: accurate and efficient methods for optimisation. *International Journal of Metaheuristics*, 3(4):320–347, 2014.

S. Nesmachnow. Using metaheuristics as soft computing techniques for efficient optimization. In *Encyclopedia of Information Science and Technology, Third Edition*, pages 7390–7399. IGI Global, 2015. doi: 10.4018/978-1-4666-5888-2.ch727. URL http://dx.doi.org/10.4018/978-1-4666-5888-2.ch727.

S. Nesmachnow, H. Cancela, and E. Alba. Heterogeneous computing scheduling with evolutionary algorithms. *Soft Computing*, 15(4):685–701, 2010.

B. Nordman and K. Christensen. Greener pcs for the enterprise. *IT Professional*, 11(4): 28–37, 2009. ISSN 15209202.

A. Orgerie, M. D. d. Assuncao, and L. Lefevre. A survey on techniques for improving the energy efficiency of large-scale distributed systems. *ACM Computer Survey*, 46(4): 47:1–47:31, Mar. 2014. ISSN 0360-0300.

I. H. Osman and J. P. Kelly. Meta-heuristics: an overview. In *Meta-Heuristics*, pages 1–21. Springer, 1996.

A. Osyczka. Multicriteria optimization for engineering design. *Design optimization*, 1: 193–227, 1985.

Paramiko. Welcome to paramiko!, 2016. URL http://www.paramiko.org/. Visitado en 2016-05-10.

J. Pearl. *Heuristics: Intelligent Search Strategies for Computer Problem Solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984. ISBN 0-201-05594-5.

Python Software Foundation. platform, 2016a. URL https://docs.python.org/2/library/platform.html. Visitado en 2016-11-29.

Python Software Foundation. Python org, 2016b. URL https://www.python.org/. Visitado en 2016-11-29.

J. M. Rabaey, A. P. Chandrakasan, and B. Nikolic. Digital integrated circuits, 2002.

J. T. Richardson, M. R. Palmer, G. E. Liepins, and M. Hilliard. Some guidelines for genetic algorithms with penalty functions. In *Proceedings of the third international conference on Genetic algorithms*, pages 191–197. Morgan Kaufmann Publishers Inc., 1989.

G. Rodola. psutil, 2016. URL https://github.com/giampaolo/psutil. Visitado en 2016-11-29.

C. A. Rodríguez Villalobos and C. A. C. Coello. A new multi-objective evolutionary algorithm based on a performance assessment indicator. In *Proceedings of the 14th annual conference on Genetic and evolutionary computation*, pages 505–512. ACM, 2012.

F. Rossi, P. Van Beek, and T. Walsh. *Handbook of constraint programming*. Elsevier, 2006.

N. Sadashiv and S. M. D. Kumar. Cluster, grid and cloud computing: A detailed comparison. In *Computer Science Education (ICCSE), 2011 6th International Conference on*, pages 477–482, Aug 2011.

B. Sareni and L. Krahenbuhl. Fitness sharing and niching methods revisited. *IEEE Transactions on Evolutionary computation*, 2(3):97–106, 1998.

F. Schlottmann and D. Seese. Financial applications of multi-objective evolutionary algorithms: Recent developments and future research directions. *Applications of multi-objective evolutionary algorithms*, pages 627–652, 2004.

S. S. Shapiro and M. B. Wilk. An analysis of variance test for normality (complete samples). *Biometrika*, 52(3/4):591–611, 1965.

D. C. Snowdon, S. Ruocco, and G. Heiser. Power management and dynamic voltage scaling: Myths and facts. In *Proceedings of the 2005 workshop on power aware real-time computing*, volume 12, 2005.

SPEC. All published spec power 2008 results, 2015. URL `http://www.spec.org/power_ssj2008/results/power_ssj2008.html`. Visitado en 2015-12-22.

N. Srinivas and K. Deb. Muiltiobjective optimization using nondominated sorting in genetic algorithms. *Evolutionary computation*, 2(3):221–248, 1994.

K. M. Tarplee, R. Friese, A. A. Maciejewski, H. J. Siegel, and E. K. P. Chong. Energy and makespan tradeoffs in heterogeneous computing systems using efficient linear programming techniques. *IEEE Transactions on Parallel and Distributed Systems*, 27(6): 1633 – 1646, 2016. ISSN 10459219.

S. Tilkov. A brief introduction to rest. *InfoQ, Dec*, 10, 2007.

J. Ullman. NP-complete scheduling problems. *Journal of Computer and System Sciences*, 10(3):384–393, jun 1975.

D. A. Van Veldhuizen. Multiobjective evolutionary algorithms: classifications, analyses, and new innovations. Technical report, DTIC Document, 1999.

D. A. Van Veldhuizen and G. B. Lamont. Multiobjective evolutionary algorithm research: A history and analysis. Technical report, Citeseer, 1998.

M. Weiser, B. Welch, A. Demers, and S. Shenker. Scheduling for reduced cpu energy. In *Mobile Computing*, pages 449–471. Springer, 1994.

X.-S. Yang. *Engineering optimization: an introduction with metaheuristic applications*. John Wiley & Sons, 2010a.

X.-S. Yang. *Nature-inspired metaheuristic algorithms*. Luniver press, 2010b.

Y. Zhao, I. Raicu, and I. Foster. Scientific workflow systems for 21st century, new bottle or new wine? In *2008 IEEE Congress on Services-Part I*, pages 467–471. IEEE, 2008.

E. Zitzler and L. Thiele. Multiobjective optimization using evolutionary algorithms — a comparative case study. In *International Conference on Parallel Problem Solving from Nature*, pages 292–301. Springer, 1998.

E. Zitzler and L. Thiele. Multiobjective evolutionary algorithms: a comparative case study and the strength pareto approach. *IEEE transactions on Evolutionary Computation*, 3(4):257–271, 1999.

E. Zitzler, M. Laumanns, L. Thiele, et al. Spea2: Improving the strength pareto evolutionary algorithm. In *Eurogen*, number 103, pages 95–100, 2001.

Apéndices

Apéndice A

Interfaz Sistema

A.1. Pseudocódigo de la interfaz del Sistema

Algoritmo 14 Interfaz Sistema

```
procedure AGREGAR_USUARIO(usuario, recurso)
    ▷ no se puede asignar un usuario a un recurso apagado
    if (el recurso está encendido) then
        if (el usuario entra al sistema) then
            Obtener el uso actual del recurso
            Asignar el usuario al recurso
            Aumentar el uso de recurso al recurso
        else
            Obtener el uso actual del recurso
            Desasignar el usuario del recurso
            Disminuir el uso de recurso del recurso
        end if
    end if
end procedure

procedure CANTIDAD_USUARIOS_ASIGNADOS(recurso)
    ▷ Retorna la cantidad actual de usuarios asignados en el Sistema al recurso
end procedure

procedure EXISTE_USUARIO_PRESENCIAL(recurso)
    ▷ comprueba si existe un arribo de tipo PRESENCIAL asignado al recurso
end procedure

procedure OBTENER_CAPACIDAD_MAXIMA(recurso)
    ▷ Retorna el Uso de Recurso máximo que puede tener el recurso
end procedure

procedure OBTENER_RECURSO_CERCANO(recurso)
    ▷ busca el recurso sin usuarios presenciales más cercano a recurso
end procedure

procedure OBTENER_USUARIO_ACTUALES
    ▷ Retorna la lista actual de arribos asignados para cada uno de los recursos
end procedure

procedure QUITAR_USUARIO(usuario, recurso)
    ▷ permite deshacer una asignación de un usuario
end procedure

procedure OBTENER_CANTIDAD_RECURSOS // Retorna la cantidad de recursos en el
Sistema actual
end procedure
```

A.2. Detalle de implementación del Sistema

A continuación se presenta un resumen del código de las dos clases sistemas mencionadas en la subsección 5.8.1, la clase *SistemaBacktracking* especialmente optimizada para el algoritmo backtracking, y la clase *Sistema* utilizada por el resto de los algoritmos. Estas clases extienden de una clase *SistemaBase* que implementa funciones en común para los dos algoritmos:

```
public abstract class SistemaBase implements ISistema {

   protected Datos_recursos data;
   protected boolean[] recurso_encendido;

   public SistemaBase(Datos_recursos p_data) {
      super();
      data = p_data;
   }

   @Override
   public int obtener_cantidad_cores(int recurso) {
      if (recurso >= 0 && recurso < data.getMatriz_cores().length)
         return data.getMatriz_cores()[recurso];
      else
         return -1;
   }

   @Override
   public Uso_Recursos obtener_capacidad_maxima(int recurso) {
      return data.getCapacidadMaximoRecurso(recurso);
   }

   @Override
   public int obtener_recurso_cercano(int partida) {
      int i, j = i = partida;
      while ((i < this.obtener_cantidad_recursos() || j >= 0)) {
         if (i < this.obtener_cantidad_recursos()) {
            if (!existe_usuario_presencial(i)) {
               return i;
            }
         }
         if (i != j && j >= 0) {
            if (!existe_usuario_presencial(j)) {
               return j;
            }
         }
         i++;
         j--;
      }
      return -1;
   }

   @Override
   public boolean recurso_encendido(int i) {
```

```
      return recurso_encendido[i];
   }

   @Override
   public int obtener_cantidad_recursos() {
      return data.getCantidad_recursos();
   }

}
```

```
public class Sistema extends SistemaBase implements ISistema {

    // Almacena el historico de todos los usuarios que se asignaron a un recurso
   private List<List<Arribo>> user_asignation;
   // Usuarios actualmente asignados a un recurso
   private List<List<Arribo>> current_users;
   private List<Uso_Recursos> resource_usage;
   private int[] max_users;

   public Sistema(Datos_recursos p_data) {
      super(p_data);
      user_asignation = new ArrayList<>();
      current_users = new ArrayList<>();
      resource_usage = new ArrayList<>();
      max_users = new int[p_data.getCantidad_recursos()];
      recurso_encendido = new boolean[p_data.getCantidad_recursos()];
      for (int i = 0; i < p_data.getCantidad_recursos(); i++) {
         user_asignation.add(new ArrayList<Arribo>());
         current_users.add(new ArrayList<Arribo>());
         resource_usage.add(new Uso_Recursos(0, 0, 0, 0));
         recurso_encendido[i] = true;
      }
   }

   @Override
   public void agregar_usuario(Arribo usuario, int recurso) throws Exception {
      if (!recurso_encendido[recurso])
         throw new Exception("No se puede asignar un usuario a un recurso
             apagado, recurso " + recurso + " usuario "
              + usuario.getId() + "_" + usuario.getTiempo());
      if (usuario.getEntrada()) {
         Uso_Recursos uso_actual = resource_usage.get(recurso);
         user_asignation.get(recurso).add(usuario);
         current_users.get(recurso).add(usuario);
         uso_actual.add(usuario.get_uso_recursos());
      } else {
         Uso_Recursos uso_actual = resource_usage.get(recurso);
         current_users.get(recurso).remove(usuario.getLlegada());
         uso_actual.remove(usuario.get_uso_recursos());
      }
      max_users[recurso] = Math.max(max_users[recurso],
           current_users.get(recurso).size());
   }
```

```java
    @Override
    public boolean apagar_recurso(int recurso) {
        if (recurso < 0 || recurso > obtener_cantidad_recursos())
            return false;
        if (current_users.get(recurso).size() > 0) {
            return false;
        } else {
            recurso_encendido[recurso] = false;
            return true;
        }
    }

    @Override
    public boolean encender_recurso(int recurso) {
        if (recurso < 0 || recurso > obtener_cantidad_recursos())
            return false;
        recurso_encendido[recurso] = true;
        return true;
    }

    @Override
    public void quitar_usuario(Arribo usuario, int recurso) throws Exception {
        if (usuario.getEntrada())
            throw new Exception("Usuario de entrada usado en quitar usuario:
                recurso " + recurso + " usuario "
                  + usuario.getId() + "_" + usuario.getTiempo());
        else {
            Uso_Recursos uso_actual = resource_usage.get(recurso);
            current_users.get(recurso).remove(usuario);
            while (user_asignation.get(recurso).remove(usuario)) {
            }
            ;
            uso_actual.remove(usuario.get_uso_recursos());
        }
        max_users[recurso] = Math.max(max_users[recurso],
            current_users.get(recurso).size());
    }

}
```

```java
public class SistemaBacktracking extends SistemaBase implements ISistema {

     // Usuarios actualmente asignados a un recurso
    private ArrayList<Uso_Recursos> resource_usage;
    // Contador de la cantidad de usuarios totales por recurso
    private int[] usuarios_recursos;
    private int[] usuarios_recursos_historicos;
    // Contador de los usuarios presenciales por recurso
    private int[] usuarios_presenciales_recursos;
    private int[] usuarios_presenciales_recursos_historicos;
    private int[] max_presenciales;
    private int[] max_total;
```

```
public SistemaBacktracking(Datos_recursos p_data) {
   super(p_data);
   resource_usage = new ArrayList<>();
   usuarios_recursos = new int[p_data.getCantidad_recursos()];
   usuarios_recursos_historicos = new int[p_data.getCantidad_recursos()];
   usuarios_presenciales_recursos = new int[p_data.getCantidad_recursos()];
   usuarios_presenciales_recursos_historicos = new
       int[p_data.getCantidad_recursos()];
   max_presenciales = new int[p_data.getCantidad_recursos()];
   max_total = new int[p_data.getCantidad_recursos()];
   recurso_encendido = new boolean[p_data.getCantidad_recursos()];
   for (int i = 0; i < p_data.getCantidad_recursos(); i++) {
      resource_usage.add(new Uso_Recursos(0, 0, 0, 0));
      recurso_encendido[i] = true;
   }
}

@Override
public void agregar_usuario(Arribo usuario, int recurso) throws Exception {
   if (!recurso_encendido[recurso])
      throw new Exception("No se puede asignar un usuario a un recurso
          apagado, recurso " + recurso + " usuario "
           + usuario.getId() + "_" + usuario.getTiempo());
   if (usuario.getEntrada()) {
      Uso_Recursos uso_actual = resource_usage.get(recurso);
      uso_actual.add(usuario.get_uso_recursos());
      usuarios_recursos[recurso] += 1;
      usuarios_recursos_historicos[recurso] += 1;
      if (usuario.tipo_usuario == UserTypes.PRESENCIAL) {
         usuarios_presenciales_recursos[recurso] += 1;
         usuarios_presenciales_recursos_historicos[recurso] += 1;
      }
   } else {
      Uso_Recursos uso_actual = resource_usage.get(recurso);
      uso_actual.remove(usuario.get_uso_recursos());
      usuarios_recursos[recurso] -= 1;
      if (usuario.tipo_usuario == UserTypes.PRESENCIAL)
         usuarios_presenciales_recursos[recurso] -= 1;
   }
   max_presenciales[recurso] = Math.max(max_presenciales[recurso],
       usuarios_presenciales_recursos[recurso]);
   max_total[recurso] = Math.max(max_total[recurso],
       usuarios_recursos[recurso]);
}

@Override
public boolean apagar_recurso(int recurso) {
   if (recurso < 0 || recurso > obtener_cantidad_recursos())
      return false;
   if (usuarios_recursos[recurso] > 0) {
      return false;
   } else {
      recurso_encendido[recurso] = false;
```

```
            return true;
        }
    }

    @Override
    public boolean encender_recurso(int recurso) {
        if (recurso < 0 || recurso > obtener_cantidad_recursos())
            return false;
        recurso_encendido[recurso] = true;
        return true;
    }

    @Override
    public void quitar_usuario(Arribo usuario, int recurso) throws Exception {
        if (!usuario.getEntrada())
            throw new Exception("No es un usuario de entrada y se usa como de
                entrada: recurso " + recurso + " usuario "
                  + usuario.getId() + "_" + usuario.getTiempo());
        else {
            Uso_Recursos uso_actual = resource_usage.get(recurso);
            uso_actual.remove(usuario.get_uso_recursos());
            usuarios_recursos[recurso] -= 1;
            if (usuarios_recursos[recurso] < 0)
                throw new Exception("No se puede tener menos de 0 usuarios");
            if (usuario.tipo_usuario == UserTypes.PRESENCIAL)
                usuarios_presenciales_recursos[recurso] -= 1;
        }
        max_presenciales[recurso] = Math.max(max_presenciales[recurso],
             usuarios_presenciales_recursos[recurso]);
        max_total[recurso] = Math.max(max_total[recurso],
             usuarios_recursos[recurso]);
    }

}
```

Apéndice B

Algoritmo Determinista

B.1. Detalle de implementación del algoritmo determinista

A continuación se presenta un resumen del código para las dos clases principales del algoritmo determinista mencionado en la subsección 5.4. La clase *main_scheduller* es quien invoca al algoritmo determinista *DetParam* que obtiene los candidatos a asignarle al arribo:

```
public class main_scheduller {

   // Ejecuta el Algoritmo Determinista para cada arribo y luego calcula la
   // Energia final consumida
   public static void main(String[] args) throws Exception {
      try {
         if (args != null) {
            if (args.length >= 1) {
               arribos_file = args[0];
            }
            if (args.length >= 2) {
               recursos_file = args[1];
            }
            if (args.length >= 3) {
               salida_folder = args[2];
            }
            if (args.length == 4) {
               tiempo = Long.parseLong(args[3]);
            }
         }
         File theDir = new File(salida_folder);
         if (!theDir.exists()) {
            theDir.mkdir();
         }
      } catch (Exception e) {
         System.exit(-1);
      }

      Datos_recursos recursos = new Datos_recursos(recursos_file);
      List<Arribo> arribos = Arribo.leer_arribos(arribos_file, tiempo);
      // Para cada arribo almacena el recurso asignado
```

```
int[] recurso_arribo = new int[arribos.size()];
// Se usa un array por separado porque el id de arribo puede no
// coincidir con el orden de llegada.
// Mantiene recursos asignados a cada arribo
ArrayInt asignaciones = new ArrayInt(arribos.size() / 2);
// Creo el Sistema inicial: le paso todos los datos de los recursos
Sistema sistema1 = new Sistema(recursos);
int indice_llegada = 0;
// Inicializo DetParam para que obtenga los recursos en su constructor
DetParam detParam = new DetParam(recursos);
for (Arribo arribo : arribos) {
   if (arribo.getEntrada()) {
      List<Integer> recursos_candidatos = new ArrayList<Integer>();
      int recurso_elegido = -1;
      switch (arribo.tipo_usuario) {
      case PRESENCIAL:
         // Si es presencial el recurso viene dado. Si ese recurso esta
         // ocupado por un usuario presencial se obtiene el mas cercano
         recurso_elegido =
             sistema1.obtener_recurso_cercano(arribo.getRecurso());
         if (recurso_elegido == -1)
            recurso_elegido =
                Utilidades.asignar_primer_recurso_libre(sistema1);
         // Se cambia el recurso original del arribo por otro, porque
         // calcular variables no lo considera de otra forma
         arribo.setRecurso(recurso_elegido);
         break;
      case REMOTO_COMANDO:
      case REMOTO_TEXTO:
      case REMOTO_PROGRAMA:
         recursos_candidatos = detParam.obtenerCandidatos(arribo,
             sistema1);
         int i = 0;
         boolean asignacion_invalida = true;
         while ((asignacion_invalida) && (i < recursos_candidatos.size()))
             {
            if
                (!sistema1.existe_usuario_presencial(recursos_candidatos.get(i)))
                {
               asignacion_invalida = false;
               recurso_elegido = recursos_candidatos.get(i);
               break;
            }
            i++;
         }
         // Caso particular en donde todos los candidatos a asignar
         // tienen un usuario presencial.
         // En este caso le asigno el primer candidato de la lista
         if (asignacion_invalida)
            recurso_elegido = recursos_candidatos.get(0);
         break;
      }
      sistema1.agregar_usuario(arribo, recurso_elegido);
      recurso_arribo[arribo.getId()] = recurso_elegido;
```

```
            asignaciones.setValue(indice_llegada, recurso_elegido);
            indice_llegada++;
         } else {
            sistema1.agregar_usuario(arribo,
                recurso_arribo[arribo.getLlegada().getId()]);
         }
      }
      CalculoEnergia calculoEnergia = new CalculoEnergia(recursos, new
          Sistema(recursos));
      retorno_calculo retornoCalculoDeterminista =
          calculoEnergia.calcular_variables_uso_sistema(asignaciones.array_,
           arribos, false, false);
   }

}
```

```
public class DetParam {

   private Arribo arribo;
   private ISistema sistema;
   private List<Pc> recursos;
   private List<Pc> recursosMayoresACotaMemoria;
   private List<Pc> recursosMenoresACotaMemoria;
   private List<Pc> recursosCandidatosMemoria;
   private List<Pc> recursosMayoresACotaCpu;
   private List<Pc> recursosMenoresACotaCpu;
   private List<Pc> recursosCandidatosCpu;

   // Lista de posibles candidatos para asignar
   private List<Pc> recursosCandidatos;
   // Candidatos posibles a asignar que consumen menos Energia
   private List<Integer> mejoresCandidatosPorConsumo;
   // Hash que contiene la probabilidad de ocurrencia de un usuario presencial
   // para cada recurso a asignar
   private HashMap<Integer, Double> mapaPc_Probabilidad = new HashMap<Integer,
       Double>();
   // Hash con el Consumo Energetico de cada candidato a asignar
   private HashMap<Integer, Double> mapaPc_ConsumoUsuario = new
       HashMap<Integer, Double>();
   // Lista final de recursos para asignar al arribo
   private List<Integer> recursosParaAsignar;
   // Matriz de Ram por recurso
   private double[] matriz_ram;
   // Matriz de Cpu por recurso
   private int[] matriz_cores;
   // Matriz con la probabilidad de ocurrencia de un usuario presencial
   private Double[] probabilidad_ocurrencia;
   private int cantidad_recursos = 0;
   private Datos_recursos datos_recursos;
   private CalculoEnergia calculoEnergia;

   public DetParam(Datos_recursos recurso) {
      // Inicializo variables globales
```

```
    this.matriz_ram = recurso.getMatriz_ram();
    this.matriz_cores = recurso.getMatriz_cores();
    this.cantidad_recursos = recurso.getCantidad_recursos();
    this.probabilidad_ocurrencia = recurso.getProbabilidad_ocurrencia();
    this.datos_recursos = recurso;

    int i = 0;
    this.recursos = new ArrayList<Pc>();
    while (i < cantidad_recursos) {
      Integer id = i;
      Double ramD = new Double(matriz_ram[i]);
      Integer ram = ramD.intValue();
      Double cpu = new Double(matriz_cores[i]);
      Double prob = probabilidad_ocurrencia[i];

      Pc pc = new Pc(id, "pcunix", ram, cpu, "up", prob);
      this.recursos.add(pc);
      i++;
    }

    this.calcularFuncionProbabilidadPorRecurso();
    this.obtenerRecursosPorMemoriaYCpu(Constantes.COTA_MEMORIA,
        Constantes.COTA_CPU);
  }

  public List<Integer> obtenerCandidatos(Arribo a, ISistema sistema) throws
      Exception {

    this.arribo = a;
    this.sistema = sistema;

    this.calculoEnergia = new CalculoEnergia(this.datos_recursos,
        this.sistema);

    this.obtenerRecursosCandidatos();

    this.descartarRecursosSobrecargados();

    this.obtenerConsumoEnergeticoEstimado();

    this.obtenerMejorCandidato();

    return recursosParaAsignar;

  }

  public void obtenerRecursosCandidatos() {
    // Determino con que lista de Memoria quedarme segun si su consumo es
    // mayor o menor a una cota determinada de memoria sobre un coeficiente
    Double consumoRam = arribo.get_uso_recursos().ram_avg;
    int cotaM = Integer.parseInt(Constantes.COTA_MEMORIA) /
        Constantes.COEFICIENTE;
    if (consumoRam.compareTo((double) cotaM) > 0)
      this.recursosCandidatosMemoria = this.recursosMayoresACotaMemoria;
```

```
    else
      this.recursosCandidatosMemoria = this.recursosMenoresACotaMemoria;

    // Determino con que lista de Cpu quedarme segun si su estimacion
    // promedio es mayor o menor a una cota de Cpu
    Double consumoCpu = new
        Double(arribo.get_uso_recursos().cpu_avg.doubleValue());
    if (consumoCpu.compareTo((double) (50 * 4)) > 0)
      this.recursosCandidatosCpu = this.recursosMayoresACotaCpu;
    else
      this.recursosCandidatosCpu = this.recursosMenoresACotaCpu;

    // Busco los recursos que cumplen ambas condiciones
    intersectarSoluciones(this.recursosCandidatosMemoria,
        this.recursosCandidatosCpu);
  }

  private void obtenerRecursosPorMemoriaYCpu(String cotaMemoria, String
      cotaCpu) {
    // Obtengo recursos con cpu mayor y menor a cotaCpu. Idem para cotaMemoria
    this.recursosMayoresACotaMemoria = new ArrayList<Pc>();
    this.recursosMenoresACotaMemoria = new ArrayList<Pc>();
    this.recursosCandidatosMemoria = new ArrayList<Pc>();
    this.recursosMayoresACotaCpu = new ArrayList<Pc>();
    this.recursosMenoresACotaCpu = new ArrayList<Pc>();
    this.recursosCandidatosCpu = new ArrayList<Pc>();

    for (Pc recurso : this.recursos) {
      // Filtro por cota de Memoria
      if (recurso.getRam().compareTo(Integer.parseInt(cotaMemoria) / 1000) >
          0)
        this.recursosMayoresACotaMemoria.add(recurso);
      else
        this.recursosMenoresACotaMemoria.add(recurso);
      // Filtro por cota de Cpu
      if (recurso.getCpu().compareTo(Double.parseDouble(cotaCpu)) >= 0)
        this.recursosMayoresACotaCpu.add(recurso);
      else
        this.recursosMenoresACotaCpu.add(recurso);
    }
  }

  private void intersectarSoluciones(List<Pc> recursosCandidatosMemoria2,
      List<Pc> recursosCandidatosCpu2) {
    this.recursosCandidatos = new ArrayList<Pc>();
    for (Pc recursoMem : recursosCandidatosMemoria2)
      for (Pc recursoCpu : recursosCandidatosCpu2)
        if (recursoMem.getId().intValue() == recursoCpu.getId().intValue())
          this.recursosCandidatos.add(recursoMem);

    if (this.recursosCandidatos.isEmpty()) {
      this.recursosCandidatos.addAll(this.recursosCandidatosMemoria);
      this.recursosCandidatos.addAll(this.recursosCandidatosCpu);
    }
```

```
}

// Funcion que filtra los candidatos que al agregarle un usuario superan el
// 100% de uso de cpu
public void descartarRecursosSobrecargados() {
   for (int i = 0; i < this.recursosCandidatos.size(); i++) {
      Pc candidato = this.recursosCandidatos.get(i);
      if
          ((sistema.obtener_uso_recursos(candidato.getId()).cpu_avg.doubleValue()
            + arribo.get_uso_recursos().cpu_avg.doubleValue()) > 400) {
         this.recursosCandidatos.remove(i);
         i--;
      }
   }
   if (recursosCandidatos.isEmpty())
      this.recursosCandidatos.addAll(this.recursos);
}

private void calcularFuncionProbabilidadPorRecurso() {
   int i = 0;
   while (i < recursos.size()) {
      this.mapaPc_Probabilidad.put(i,
          this.recursos.get(i).getProbabilidad_ocurrencia() / 100);
      i++;
   }
}

private void obtenerConsumoEnergeticoEstimado() {
   for (Pc recurso : this.recursosCandidatos) {
      double consumoRecurso =
          this.calculoEnergia.consumo_energia_recurso(recurso.getId().intValue());
      double consumoAgregado =
          this.calculoEnergia.consumo_energia_post_asignacion(recurso.getId().intValue(),
            arribo);
      double deltaConsumo = consumoAgregado - consumoRecurso;

      this.mapaPc_ConsumoUsuario.put(recurso.getId(),
          this.arribo.getTiempo() * deltaConsumo);
   }
}

private void obtenerMejorCandidato() {
   this.mejoresCandidatosPorConsumo = new ArrayList<Integer>();
   double menor = Double.MAX_VALUE;
   Iterator it = this.mapaPc_ConsumoUsuario.entrySet().iterator();
   while (it.hasNext()) {
      Map.Entry e = (Map.Entry) it.next();
      Double p = (Double) e.getValue();
      // Me quedo con el recurso que consuma menos y su probabilidad de
      // ocurrencia de un usuario presencial sea menor al 50%
      if ((p.doubleValue() < menor) &&
          (this.mapaPc_Probabilidad.get(e.getKey())
            .doubleValue() < Constantes.MAX_PROBABILIDAD_OCURRENCIA)) {
         menor = p.doubleValue();
```

```
            // Remuevo a todos de la lista porque ya no son mejores candidatos
            this.mejoresCandidatosPorConsumo.clear();
            // Agrego un candidato
            this.mejoresCandidatosPorConsumo.add((Integer) e.getKey());
         } else if ((p.doubleValue() == menor) &&
             (this.mapaPc_Probabilidad.get(e.getKey())
               .doubleValue() < Constantes.MAX_PROBABILIDAD_OCURRENCIA))
            // Agrego un candidato
            this.mejoresCandidatosPorConsumo.add((Integer) e.getKey());
      }
      this.recursosParaAsignar = new ArrayList<Integer>();
      if (menor == Double.MAX_VALUE) { // Sino encontre elijo el primero
         this.recursosParaAsignar.add(this.recursosCandidatos.get(0).getId());
      } else
         obtenerMenorUsoDeRecurso();
   }

   private void obtenerMenorUsoDeRecurso() {
      double menorUso = Double.MAX_VALUE;
      for (Integer id_pc : this.mejoresCandidatosPorConsumo) {
         Uso_Recursos uso = this.sistema.obtener_uso_recursos(id_pc.intValue());
         double uso_cpu = uso.cpu_avg.doubleValue();
         if (uso_cpu < menorUso) {
            menorUso = uso_cpu;
            // Remuevo a todos de la lista porque ya no son mejores candidatos
            this.recursosParaAsignar.clear();
            this.recursosParaAsignar.add(id_pc); // Agrego un candidato
         } else if (uso_cpu == menorUso)
            this.recursosParaAsignar.add(id_pc); // Agrego un candidato
      }
   }

}
```

Apéndice C

Aplicación web

C.1. Casos de uso

Se desarrolló una aplicación web en Django, framework de desarrollo web en lenguaje Python. La aplicación cuenta con dos roles de usuarios, un usuario administrador y un usuario estudiantil, ambos deben registrarse en nuestra plataforma para hacer uso de sus funciones (ver figura C.1). El rol del usuario determina las funcionalidades de la web a la que accederán. Luego de iniciar sesión un administrador, la página web le despliega todos los recursos que estén activos, permitiéndole escoger uno de ellos para ver su información. En la información se puede observar la utilización de CPU y memoria RAM de las últimas horas de cada recurso (en forma gráfica), como se ve en la figura C.2. Además un usuario administrador puede observar una gráfica con información de la cantidad de usuarios por recurso en las últimas horas.

Cuando un usuario estudiantil inicia sesión, la aplicación web le despliega todos los recursos disponibles del salón. En la figura C.3 se muestra la grilla de las pcs disponibles. Además, a un usuario que es estudiante se le permitirá ejecutar dos funcionalidades, subir archivos y solicitar un recurso. En la figura C.4 se observa el caso de uso donde el usuario puede ingresar en un formulario los datos principales de su sesión. Estos datos son los que utilizará el planificador para determinar cuál es el mejor recurso para asignarle, utilizando nuestro algoritmo evolutivo con los datos del formulario ingresado por el usuario. Luego de ingresar el formulario la aplicación web despliega la pantalla que se muestra en la imagen C.5. La otra funcionalidad que se le permite al estudiante es subir un archivo de texto (ver figura C.6) con las instrucciones de la tarea que desea ejecutar. Este archivo, junto con su usuario y contraseña, serán guardados en la base de datos del servidor para que posteriormente sea elegida y ejecutada la tarea por parte del planificador, en nombre del usuario que la subió.

Figura C.1: Pantalla inicial de registro de usuario de la aplicación web

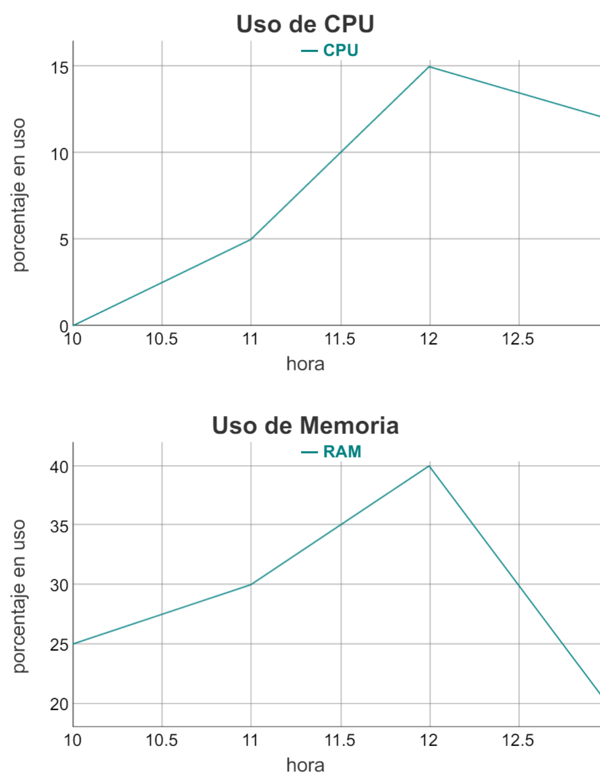


Figura C.2: Pantalla con gráficas de utilización de CPU y memoria RAM por recurso, en las últimas horas

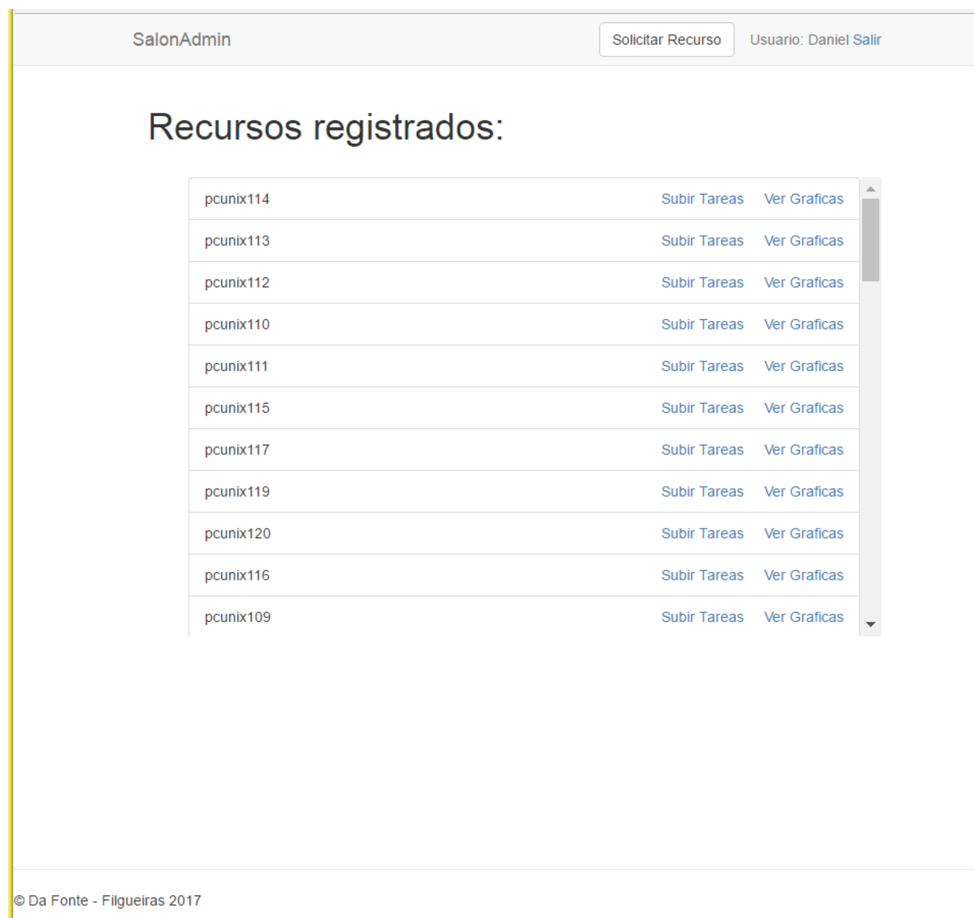


Figura C.3: Pantalla con las máquinas disponibles del salón una vez logueado el usuario

Figura C.4: Pantalla con formulario de ingreso para que un estudiante solicite un recurso

Figura C.5: Pantalla con el resultado del envío del formulario

Figura C.6: Pantalla con formulario para subir una tarea como archivo de texto conteniendo las instrucciones

Apéndice D

Estudio de comandos remotos

D.1. Pruebas de librerías para Windows y Linux

En una futura puesta en producción, el planificador deberá poder ejecutar comandos en las máquinas de los salones de informática de forma remota. Estos comandos pueden ser tanto para ejecutar las tareas de los usuarios en su nombre, como para eventualmente mandar a dormir un recurso que esté ocioso, y sea así más eficiente para el sistema que éste no esté encendido. Para realizar esto se estudiaron formas posibles y se encontraron dos librerías que permiten ejecutar comandos en forma remota con un usuario. Las librerías estudiadas fueron Paramiko para Linux, y Plink tanto para el sistema operativo Windows como para Linux. Como condición es necesario que el planificador tenga permisos de root (permisos de administrador) en caso de querer encender, apagar o dormir una máquina del salón de informática de forma remota.

D.1.1. Librería Paramiko

Paramiko es una implementación de Python (2.6+, 3.3+) del protocolo SSHv2, que provee funcionalidades tanto del lado del cliente como del servidor. Si bien aprovecha una extensión de Python C para la criptografía de bajo nivel, Paramiko es en si una interfaz pura Python alrededor de los conceptos de red SSH. (Paramiko, 2016)

A continuación se presenta el código que estudiamos y probamos en una máquina del salón de informática que tiene instalado el sistema operativo Linux en sus máquinas. Como comando ejemplo utilizamos uno de los más sencillos que nos presenta Linux, como es el de listar los archivos del directorio donde se ejecute la sentencia: ”ls”. La porción de código presentado es un bash de Python, lo que significa que obtendremos el resultado con tan solo ejecutar el archivo que contenga el código.

```
import paramiko

ssh = paramiko.SSHClient()
ssh.set_missing_host_key_policy(paramiko.AutoAddPolicy()) # Setea politica por
    defecto para hosts desconocidos
ssh.connect('lulu.fing.edu.uy', username='usuario.facultad',
    password='psw.usuario')
stdin, stdout, stderr = \ ssh.exec_command("ls")
```

```
type(stdin)
stdout.readlines()
```

D.1.2. Librería Plink

En el caso de esta librería, creamos un código que obtiene el sistema operativo dónde está ejecutando y adapta su código a éste, logrando de esta forma que funcione tanto para el sistema operativo Windows como para Linux. A continuación presentamos dos formas de utilizar la librería Plink, mediante el comando ”system”, y mediante el comando ”popen”, ambos de la librería .ºs”de Python. Ambas funciones las implementamos paramétricamente de forma de invocarlas con el usuario, contraseña y comando que se desee.

Tener en cuenta que el código presentado a continuación debe ser adaptado en caso se requiera sea multiplataforma; a pesar de adaptar su sintaxis los comandos entre los distintos sistemas operativos no son los mismos.

```
#Ejecuta comandos remoto mediante SSH
from platform import system
import os
def ssh(self,host,user,password,cmd):
   PLINK_STRING="plink"
   so = system()
   if so == 'Linux':
      PLINK_STRING = "plink"
   elif so == 'Windows':
      PLINK_STRING = "plink.exe"</p>
   comando = PLINK_STRING + " " + "-pw " + password + " " + user + "@" + host +
      " " + cmd
   print("debug: " + comando)
   try:
      os.system(comando)
   except Exception,err:
      print("Error: " + str(err))</p>


#Ejecuta en remoto el contenido del script indicado
def sshScript(self,host,user,password,script,fOut=None):
   PLINK_STRING="plink"
   so = system()
   if so == 'Linux':
      PLINK_STRING="plink"
   elif so == 'Windows':
      PLINK_STRING="plink.exe"</p>
   comando=PLINK_STRING+" "+"-pw "+password+" -m "+script+" "+user+"@"+host+" "
   print("debug: "+comando)
   try:</p>
      stdout = os.popen(comando)
      OUT=stdout.read()
      print (OUT)
```

```
    if (fOut!=None):
        f=open(fOut,"wb")
        f.write(OUT)
        f.close()</p>
except Exception,err:
    print("Error: "+str(err))</p>
```