Facultad de Ingeniería - Universidad de La República

Montevideo, Uruguay

Proyecto de Grado

Algoritmos de asignación de actividades a unidades y dimensionamiento de empresas de alto porte.

Integrantes:

Martin Delafuente y Patricia Bevilacqua

Tutor: Pablo Romero
Co-Tutor: Franco Robledo

Tabla de contenido

Resumen

La gestión de la fuerza laboral (en inglés workforce management – WFM) busca optimizar la eficiencia operacional y el manejo efectivo de los recursos humanos disponibles en una organización, lo cual se puede traducir en una mejora de la productividad y en una reducción de costos por contratación de personal.

En este sentido, esta tesis estudia el problema de determinar la cantidad mínima de personal necesario para cubrir una serie de actividades organizadas en unidades de trabajo en empresas de alto porte. Para ello presentamos una resolución de un problema de programación matemática que incluye aspectos de optimización combinatoria, procesos estocásticos y estadística.

Basándonos en las características del modelo podemos dividir el problema en tres fases bien diferenciadas. La primera de ellas hace referencia a la asignación de actividades en unidades, problema combinatorio identificado con la coloración de grafos. Luego de explorar distintas técnicas de coloreo, LRLF y EXTRACOL_CSR_ATS resultaron ser las más prometedoras y por tanto fueron las desarrolladas en este proyecto. La segunda fase tiene como objeto modelar la conducta de clientes, su arribo y atención, mediante un modelo estocástico tratable y simulable por computadora, capaz de imitar el comportamiento de una unidad, cuyo funcionamiento se basa fuertemente en teoría de colas y métodos de Monte Carlo. Para ello, como primera aproximación estudiamos la eficiencia de una unidad con una actividad a partir de la comparación de los resultados estadísticos obtenidos con la fórmula de Erlang C. Esta comparación tiene lugar porque el modelo considera que los clientes solicitan servicios ofrecidos por la empresa siguiendo la ley de Poisson, mientras que el personal atiende las solicitudes iniciadas por los clientes en un tiempo aleatorio que sigue la ley Exponencial. El objetivo de esta etapa es evaluar su correcto funcionamiento bajo este escenario y dar lugar a la calibración de la herramienta de simulación. Finalmente, en la tercera etapa, se construye la metaheurística que utiliza el resultado de la coloración y el simulador. De esta forma se busca obtener el número de personal óptimo (mínimo) para que las unidades puedan operar correctamente pero esta vez ejecutando varias actividades en cada una de ellas. Para validar el resultado obtenido utilizamos estimación de esperanzas mediante promedios empíricos y verificamos que se cumplan las restricciones del modelo.

A partir de las pruebas finales, se observó que para coloraciones con igual cantidad de colores, para una misma instancia (o grafo), es posible obtener diferentes asignaciones de personal. Además la propiedad más importante que pudimos verificar es la mejora en la asignación de personal a partir de una mejor coloración, o sea a menor cantidad de colores, menor cantidad de personal asignado. Lo anterior deja en evidencia la importancia de invertir tiempo y esfuerzo en encontrar buenos algoritmos de coloreo, buscando el balance adecuado entre los tiempos de ejecución y la calidad de la solución.

Palabras Clave: Problema de Optimización Combinatoria, Programación Matemática, Coloración de Grafos, Metaheurísticas, Erlang C.

Introducción

En la actualidad las empresas de alto porte se enfrentan a grandes desafíos en lo que tiene que ver con la gestión de la cantidad de personal que contratan, el manejo de las capacidades de dicho personal y al mismo tiempo lograr minimizar los gastos de la contratación. Esta problemática es abordada por el área de gestión de la fuerza laboral (en inglés Workforce Management - WFM), cuyo objetivo principal es la mejora en la eficacia y la eficiencia de la fuerza de trabajo.

Es de gran interés para la mayoría de las organizaciones gestionar la fuerza laboral debido a que constituye el setenta u ochenta por ciento de los costos. La experiencia ha demostrado que la correcta implementación de los principios de gestión de fuerza de trabajo puede reducir los costos laborales en alrededor de un diez por ciento, lo cual no es despreciable [1]. Estos beneficios son por lo general alcanzados a través de un riguroso análisis de la demanda y la adopción de formas flexibles de trabajo para satisfacer dicha demanda. En muchas organizaciones la variación de la demanda puede provocar falta de personal o exceso de capacidad, con las correspondientes deficiencias que esto conlleva.

La gestión de la fuerza de trabajo proporciona los métodos y sistemas necesarios para mejorar la eficiencia de la utilización de los trabajadores y al mismo tiempo mejorar las condiciones de trabajo.

Debido a que el mercado de la gestión de la fuerza laboral es todavía muy inmaduro también lo es el mercado de software en esta área. Sin embargo los analistas del sector prevén que dicho mercado experimentará un crecimiento dinámico.

Mientras que software especial es utilizado comúnmente en numerosas áreas tales como ERP (Enterprise Resource Planning), SLM (Service Lifecycle Management), CRM (Customer Relationship Management), la gestión de la fuerza laboral sigue siendo a menudo manejada a través de hojas de cálculo. Esto puede traducirse en tiempos de inactividad o no productivos, en altas tasas de fluctuación, mal servicio al cliente y costos de oportunidad en los que se incurre [2].

Por el contrario, mediante el uso de una herramienta de optimización integral para la gestión de la fuerza de trabajo orientado a la demanda, los planificadores pueden optimizar la dotación de personal mediante la creación de planes que en todo momento puedan ajustarse a las necesidades reales de la organización. Al mismo tiempo una solución WFM ayuda a los usuarios a observar la legislación relevante, los acuerdos locales y los contratos del personal.

Muchos sistemas de gestión de fuerza de trabajo también ofrecen capacidades de ajuste manual. Los valores obtenidos a partir de planificaciones previas se convierten luego en requerimientos reales de personal a través de un algoritmo que es ajustado a cada caso en

particular. El algoritmo en sí mismo es basado en la fórmula de Erlang[1] aunque la mayoría de las adaptaciones modernas de WFM se han orientado a un estado más rico de gestión y a optimizaciones de los algoritmos originales [3].

Este proyecto tiene como objetivo estudiar un problema perteneciente al área de la gestión de la fuerza de trabajo. Específicamente lo que investigaremos es la asignación óptima de diferentes actividades atómicas en unidades organizacionales y a partir de este resultado establecer la asignación mínima de personal necesaria para que dichas unidades puedan operar.

Este documento se estructura de la siguiente manera. En el capítulo 1 se muestra la formulación matemática que sustenta la solución al problema. En el capítulo 2 se explica el problema de la coloración de grafos que permite resolver la primera fase del problema general, a saber, la asignación óptima de diferentes actividades atómicas en unidades organizacionales. En el capítulo 3 se describe la fórmula de Erlang C para el caso en el cual se utiliza una actividad y una unidad. Posteriormente se realiza un relevamiento de literatura sobre sistemas de colas multi-clase, multi-servidor y finalmente se describen los fundamentos detrás de los métodos de Monte Carlo. En el capítulo 4 se detalla la algoritmia del simulador y el diseño de una metaheurística que nos permitirá obtener la asignación mínima de personal necesaria para que las unidades puedan operar. En el capítulo 5 se presentan los resultados obtenidos a partir de la ejecución de una serie de pruebas y finalmente en el capítulo 6 se exponen las conclusiones del trabajo realizado.

[1] La fórmula de Erlang, o una de sus generalizaciones es el punto de partida de la gestión de la fuerza laboral (ver [67])

Capítulo 1 – Marco Teórico

1.1 Introducción

Las empresas de mediano y alto porte ofrecen una cartera de servicios constituidas por un número importante de actividades que dan cumplimiento a las diferentes necesidades de los clientes. Debido a la cantidad de recursos humanos involucrados y servicios ofrecidos en este tipo de empresas, es necesario adoptar una estrategia eficiente que permita organizar de manera óptima el trabajo. Esto se puede lograr mediante el agrupamiento inteligente de tareas compatibles en departamentos o unidades especializadas, permitiendo de esta forma aumentar la productividad de los empleados, disminuir los tiempos de espera de los clientes, minimizar la cantidad de personal contratado y por ende mejorar los costos y la calidad del servicio. Esto motiva la necesidad de investigar y desarrollar un modelo que permita aproximar de la manera más exacta esta realidad y a su vez implementar un algoritmo basado en este modelo que resuelva eficientemente la agrupación de actividades en unidades y la asignación de personal.

Dada las características del problema, éste puede clasificarse como un problema matemático cuyos componentes principales incluyen procesos estocásticos, estadística y optimización combinatoria.

En este capítulo nos centramos en describir los conceptos del modelo matemático y explicaremos porqué nos permite aproximar a la solución del problema planteado.

1.2 Problema

Se desea resolver el problema de agrupar una lista de actividades en unidades de trabajo y de minimizar el personal contratado para cada unidad de manera de cumplir con todas las funciones de la organización, agrupando actividades compatibles, y respetando el tiempo de tolerancia de los clientes.

Para introducirnos en la temática, definimos a continuación una serie de conceptos:

Actividades: Las actividades $A = \{a_1, a_2, \dots \dots a_n\}$ están definidas como el conjunto de tareas atómicas o indivisibles, no paralelizables, las cuales pueden agruparse siempre que estas sean compatibles entre sí.

Proceso: Un proceso es una secuencia ordenada de actividades que se realizan para obtener un fin. En este caso los procesos son atómicos, es decir que están compuestos por una sola actividad. Por ello para nuestra realidad simplificada es equivalente hablar de procesos o actividades.

Matriz de compatibilidad: Es una matriz binaria simétrica que denotamos $C_{nxn},\ n = \#|A|$ que indica si dos actividades son compatibles o no entre sí, es decir:

$$c_{j,k}\begin{cases}1\ sii\ a_j\ a_k\ son\ actividades\ compatibles\\ 0\ si\ no\ lo\ son\end{cases}$$

Unidades: Una unidad es el mínimo grupo funcional que atiende actividades, al que puede asignarse personal. Sea $U = \{u_1, u_2, \dots \dots u_m\}$ el conjunto de unidades de la organización, llamaremos A_i al subconjunto de actividades que ejecuta la unidad u_i y que cumple las siguientes propiedades:

- Las actividades A_i que conforman la unidad u_i deben ser compatibles, es decir:

$$c_{j,k} = 1\ \forall\ a_j\ a_k\ \in\ A_i\ con\ 1 \le j,k \le n, 1 \le i \le m$$

- Cada actividad debe ser ejecutada en una única unidad, es decir:

$$A_i \cap\ A_j =\ \emptyset\ con\ i \ne j\ \ y\ 1 \le i,j \le m$$

- Todas las actividades están asignadas a alguna unidad, es decir:

$$\cup_{i=1}^{m}\ A_i = A$$

Matriz de asignaciones: Es una matriz binaria que denotamos $S_{mxn},\ con\ m = \#|U|\ y\ n = \#|A|$ que indica a que unidad pertenece cada actividad, es decir:

$$s_{j,k}\begin{cases}1\ sii\ la\ unidad\ u_j\ atiende\ la\ actividad\ a_k\\ 0\ en\ otro\ caso\end{cases}$$

A partir de la matriz de asignaciones, el subconjunto de actividades que atiende una unidad se puede definir formalmente como:

$$A_i = \{a_j \in A\ /\ s_{i,j} = 1\}$$

Personal: Son los recursos humanos con los que cuenta la organización para ejecutar las actividades de cada unidad. Sea $R = \{r_1,\ r_2, \dots \dots, r_t\}$ el conjunto total de recursos, llamaremos X_i al subconjunto de personal asignado a la unidad u_i, donde:

- Todo el personal está asignado a alguna unidad, es decir:

$$\exists\, j \,/\, \{r_i\} \cap\ X_j \neq\ \emptyset, \forall\, i, con\ 1 \leq i \leq t, 1 \leq j \leq m$$

- Cada persona puede pertenecer a una única unidad, es decir:

$$X_i \cap\ X_j =\ \emptyset, con\ i \neq j\ \ y\ 1 \leq i, j \leq m$$

- Cada unidad tiene personal asignado, es decir:

$$\#|X_i| = x_i\ > 0, con\ 1 \leq i \leq m$$

- La totalidad del personal posee el mismo nivel de conocimientos, por lo que pueden ejecutar cualquier actividad, los hace indistinguibles y por lo tanto asignable a cualquier unidad.

- Las personas no despachan más de una actividad simultáneamente.

Clientes: Son personas que demandan el cumplimiento de un servicio por parte de la organización. Dichos clientes serán clasificados dependiendo del servicio solicitado. Denotaremos $T_c = \{t_{c_1},\ t_{c_2}, \ldots t_{c_n}\}$ al conjunto de tipos de clientes donde t_{c_i} representa a los clientes que requieren la realización de la actividad $i,\ con\ 1 \leq i \leq n$.

Tasa de arribos (λ): Número promedio de arribos de clientes por unidad de tiempo (clientes/segundo). Además $1/\lambda$ es el tiempo medio entre llegadas (segundos/clientes).

Denotaremos $\lambda = \{\ \lambda_1, \lambda_2 \ldots \lambda_n\}$ al conjunto de tasas de arribo, donde $\lambda_i,\ con\ 1 \leq i \leq n$ representa la tasa de arribos para la actividad i de los clientes de tipo $i\ \ (t_{c_i})$. Los arribos de los clientes responden a un proceso de Poisson y el tiempo entre arribos consecutivos es exponencial [4].

Tasa de servicio (μ): Número promedio de clientes atendidos por unidad de tiempo (clientes/segundo). Además $1/\mu$ es el tiempo medio de servicio (segundos/clientes).

Denotaremos $\mu = \{\mu_1,\ \mu_2, \ldots,\ \mu_n\}$ al conjunto de tasas exponenciales de servicio, donde $\mu_i, con\ 1 \leq i \leq n$ representa la tasa de servicio para la actividad i a la cual es capaz de atender cualquier recurso del personal.

Tolerancia de un cliente: Es el tiempo que está dispuesto a esperar un cliente antes de retirarse. Incluye el tiempo que permanece en la cola y el tiempo en que se ejecuta la actividad a partir de la solicitud del servicio. Sea $T = \{t_1\ , t_2\ \ldots t_n\ \}$ el conjunto de tolerancias, donde $t_i\ \ (t_i\ > 0, con\ 1 \leq i \leq n)$ es el tiempo máximo que está dispuesto a esperar un cliente de tipo $i\ \ (t_{c_i})$, llamaremos T_i al subconjunto de tolerancias asociadas a todos los tipos de clientes que acuden a la unidad u_i.

Un cliente es atendido de manera exitosa si el tiempo de espera en cola (t_q) más el tiempo de ejecución de la actividad (t_e) por la que esperó es menor o igual a su tolerancia, es decir:

$$t_{q_{j,i}} + t_{e_{j,i}} \leq t_i, \forall\, j \in t_{c_i}, con\ 1 \leq i \leq n$$

Donde $t_{q_{j,i}}$ y $t_{e_{j,i}}$ denotan el tiempo de espera en la cola y el tiempo de ejecución para el $j - ésimo$ cliente de tipo i, respectivamente.

1.3 Formulación Matemática

Dados los conceptos del apartado anterior, nuestro problema consiste en construir un conjunto de unidades $U = \{u_1, u_2, \dots\dots, u_m\}$, asignando un subconjunto de actividades compatibles A_i, con $1 \leq i \leq m$ a cada una de ellas a partir de una lista de actividades $A = \{a_1, a_2, \dots\dots, a_n\}$. Para esto último utilizaremos la matriz de asignaciones S. Además es necesario asignar la menor cantidad de personal posible x_i con $1 \leq i \leq m$ a cada unidad de manera de poder atender con éxito las necesidades de los clientes respetando sus tiempos promedios de espera t_i con $1 \leq i \leq n$.

Este problema de optimización combinatoria se puede expresar mediante el siguiente modelo matemático:

$$mín \sum_{i=1}^{m} x_i \tag{1.1}$$

$$s.a. \begin{cases} \sum_{i=1}^{m} s_{i,j} = 1\,, \forall\, j \in \{1, \dots, n\} & (1.2) \\ s_{i,j} * s_{i,k} \leq c_{j,k}\,, \forall\, i \in \{1, \dots, m\}, \forall\, j,k \in \{1, \dots, n\} & (1.3) \\ E(F_i) \leq t_i\,, \forall\, i \in \{1, \dots, n\} & (1.4) \end{cases}$$

La función objetivo 1.1 busca minimizar la cantidad neta de personal contratado, respetando un conjunto de restricciones que enumeramos a continuación:

- La restricción 1.2 establece que se debe asignar cada actividad a una única unidad.
- La restricción 1.3 establece que todas las actividades agrupadas bajo una misma unidad deben ser compatibles.
- La restricción 1.4 establece que el valor esperado de los tiempos de permanencia en el sistema (tiempo de espera en la cola + tiempo de servicio) para cada tipo de cliente (F_i) no debe superar la tolerancia media de los clientes que requieren ese tipo de servicio (t_i).

1.4 Identificación del problema

La formulación del problema presenta características que se pueden relacionar con otros problemas matemáticos bien conocidos. En esta sección identificamos tales problemas y dejamos constancia de su vinculación con las restricciones planteadas en el modelo matemático.

A modo de introducción, todas las posibles agrupaciones de actividades en unidades de trabajo que cumplan con las restricciones 1.2 y 1.3 se pueden obtener mediante la resolución del problema de coloración de grafos. Luego de conformadas las unidades, es posible estudiar el comportamiento de cada una de ellas a partir de un sistema de filas de espera y analizar el cumplimiento de la restricción 1.4.

1.4.1 Restricciones de asignación

Como se menciona en la introducción, existe una relación entre las restricciones vinculadas a la agrupación de actividades en unidades (1.2 y 1.3) con un problema bien conocido en la teoría de grafos, más concretamente con el problema de coloreo.

Definición 1.4.1.1 - Un coloreo de un grafo $G(V, E)$ es una asignación c entre vértices de V y colores de $\{1..n\}$ / todo par de vértices adyacentes reciben distintos colores, es decir:

$$c: V \to \{1..n\} \,/\, c(u) \neq c(v), \forall\, (u, v) \in E$$

Definición 1.4.1.2 – Se define el grafo de incompatibilidades $G(A, E_{C^c})$ tal que sus vértices representan las actividades y sus relaciones (o aristas) están dadas por la matriz de compatibilidad complementada C^c (E_{C^c}: $\{(u, v) /\ C^c_{u,v} = 1\}$).

Teorema 1.4.1.3 – Sea S_{mxn} la matriz de asignaciones de actividades en unidades, S_{mxn} cumple con las restricciones 1.2 y 1.3 $\Leftrightarrow$ $\exists$ una coloración del grafo de actividades incompatibles $G(A, E_{C^c})$ que agrupa de forma equivalente las actividades en clases de color.

Demostración: ($\Longrightarrow$)

Dada una matriz de asignación S_{mxn} que respeta las restricciones 1.2 y 1.3, definimos una función $c: A \to \{1..m\} \,/\, c(a_i) = k \; sii \; s_{k,i} = 1, \forall\, i \in \{1, .., n\}\; y$

$k \in \{1, .., m\}$.

Por 1.2, sabemos que $\exists\, k \in \{1, .., m\} \,/\, s_{k,i} = 1, \forall\, i \in \{1, .., n\} \Longrightarrow \exists\, c(a_i)\; \forall\, a_i \in A$.

$$\left.\begin{array}{l} \text{Si } a_i \wedge a_j \in u_k \overset{Def\ S}{\Rightarrow} s_{k,i} = s_{k,j} = 1 \overset{1.3}{\Rightarrow} c_{i,j} = 1\,, \\ \forall a_i, a_j \in A, con\ a_i \neq a_j. \end{array}\right\} \Longrightarrow \quad c(a_i) = c(a_j)\; sii\; c_{i,j} = 1$$

Además si $s_{k,i} = s_{k,j} = 1 \overset{Def\ c}{\Rightarrow} c(a_i) = c(a_j) = k$

De lo anterior se desprende que la función c asigna el mismo valor a dos actividades diferentes solamente si estas son compatibles. También se puede observar que si dos actividades están agrupadas bajo la misma unidad de trabajo, la función c les asigna el mismo valor funcional (o color).

Por definición de matriz complemento tenemos que $c_{i,j} = 0 \iff c^c_{i,j} = 1, \forall\, i,j \in \{1,..,n\}$ y por lo tanto $(a_i, a_j) \in E_{C^c}$.

Como c sólo asigna el mismo valor funcional a actividades compatibles bajo la misma unidad y $c_{i,j} = 0\ si\ (a_i,\ a_j) \in E_{C^c}, \forall\, i,j \in \{1,\dots,n\}$, deja en evidencia que $c(a_i) \neq c(a_j)$ $\forall\, (a_i, a_j) \in E_{C^c}$ (definición 1.4.1.1).

(⟸)

Sea c una coloración de $G(A, E_{C^c})$, definimos la función c^{-1}: $\{1..m\} \to A^n$, que dado un color, devuelve el subconjunto de actividades agrupadas bajo esa clase color.

Por definición de coloreo se cumple que $\forall a_i, a_j \in A$, $c(a_i) \neq c(a_j) \iff (a_i, a_j) \in E_{C^c}$ y además $(a_i, a_j) \in E_{C^c} \iff a_i$ no es compatible con a_j por definición del grafo $G(A, E_{C^c}) \Longrightarrow c(a_i)$ puede ser igual a $c(a_j)$ sólo si $(a_i, a_j) \notin E_{C^c}$, lo que equivale a decir $c^c_{i,j} = 0$ o $c_{i,j} = 1$ (si dos actividades están bajo la misma clase color, las mismas deben ser compatibles).

$\Longrightarrow$ $c^{-1}(i)$ devuelve una agrupación de actividades compatibles bajo una misma clase color $\forall\, i \in \{1,..,m\}$. Además como cada actividad pertenece a una sola clase color $\Longrightarrow$ c^{-1} devuelve conjuntos disjuntos de actividades compatibles, es decir $c^{-1}(i) \cap c^{-1}(j) = \emptyset, \forall\, i,j \in \{1,..,m\}, con\ i \neq j$.

Por definición, una unidad de trabajo está conformada por una agrupación de actividades compatibles (A_i).

$\Longrightarrow$

$\Longrightarrow$ podemos redefinir c^{-1}: $\{1..m\} \to \{A_1,..A_m\}$ / $c^{-1}(i) = A_i$, donde se cumple que a_j pertenece a una sola A_i ya que c^{-1} devuelve conjuntos disjuntos y toda actividad está asignada a alguna agrupación A_i ya que $\nexists$ actividades sin colorear.

Sea S_{mxn} la matriz de asignaciones correspondiente a las unidades conformadas por las A_i devueltas por c^{-1}, como cada actividad pertenece a una sola A_i es trivial observar que se cumple 1.2.

Por último como c^{-1} devuelve agrupaciones de actividades compatibles pueden ocurrir dos situaciones:

$$\Longrightarrow \left\{ \begin{array}{l} a_j \; y \; a_k \in A_i \Longrightarrow c_{j,k} = 1 \; y \; s_{i,j} = s_{i,k} = 1, por\ lo\ tanto\ s_{i,j} * s_{i,k} = \; c_{j,k} \\ \\ a_j \; y/o \; a_k \notin A_i \Longrightarrow s_{i,j} = 0 \; y/o \; s_{i,k} = 0, por\ lo\ tanto\ s_{i,j} * s_{i,k} = 0 \; \leq \; c_{j,k} \end{array} \right\} \Longrightarrow \text{Se cumple 1.3}$$

Q.E.D.

El teorema 1.4.1.3 demuestra que el problema de construcción de unidades a partir de la agrupación de actividades respetando las restricciones planteadas por el modelo se puede abordar de forma análoga hallando una coloración del grafo de incompatibilidades. Esto último sucede ya que la matriz de asignaciones S_{mxn} se puede interpretar como un coloreo factible sobre el grafo $G(A, E_{C^c})$, donde cada unidad se puede representar como un color.

1.4.2 Restricción tiempos de tolerancia

Si bien encontrar coloraciones factibles del grafo asegura las dos primeras restricciones, todavía falta evaluar cómo se desempeñan las unidades bajo distintas configuraciones para determinar si brindan el servicio de manera adecuada. Esta forma de crear unidades determina una independencia operativa, ya que ninguna de ellas tiene actividades en común. Esta característica permite enfocar el análisis de la restricción 1.4 a nivel de unidad, es decir que dado un conjunto de agrupaciones de las actividades de A determinados por una coloración factible de $G(A, E_{C^c})$, la restricción 1.4 se puede expresar como:

$$E(F_i) \leq \; t_i \, , \forall \; i \; / \; a_i \; \in A_j \, , \;\; i \in \{1, \dots, n\}, j \in \{1, \dots, m\} \qquad (1.5)$$

Definición 1.4.2.1 – A partir de la nueva restricción diremos que una unidad es eficiente cuando se cumple 1.5 para todas sus actividades.

Para poder cumplir con este parámetro de calidad, es necesario asignar la cantidad adecuada de personal de forma que la frecuencia de arribos de clientes que ingresan al sistema no supere la capacidad del personal de atender con éxito a los clientes (el sistema se encuentra estable y para cada actividad a_i que atiende la unidad $\lambda_i < \; x_i \mu_i$), asimismo se debe evitar que el personal disponga de tiempo ocioso buscando contratar la mínima cantidad

de personal que mantenga un nivel de servicio aceptable. Es intuitivo pensar que a menor cantidad de unidades, o lo que es análogo unidades con la máxima cantidad de actividades posibles, se maximice la productividad de los empleados y minimice la cantidad de personal necesario. Esto último se puede lograr encontrando una coloración de $G(A, E_{C^c})$ tal que la cantidad de colores encontrados coincida con el número cromático del grafo.

Encontrar la mínima asignación de personal x_i de cada unidad, que cumpla con las restricciones de tiempos de atención establecidos, determina una solución factible al problema, es por eso que además de buscar una partición óptima de las actividades en unidades, es necesario evaluar la eficiencia de cada una de ellas buscando la mínima cantidad de personal. Para lograrlo, una posible forma de modelar el funcionamiento de las unidades es mediante un proceso estocástico, más concretamente utilizando la teoría de colas y métodos de Monte Carlo.

Capítulo 2 – Coloración de Grafos

2.1 Introducción

El inicio de la teoría de grafos tiene un origen preciso, y es a partir de un artículo publicado en 1736 por el matemático suizo Leonhard Euler [5]. La idea principal de su trabajo surgió de un problema inspirador y que es conocido como los siete puentes de Königsberg. A partir de la solución a este problema (considerado el primer teorema de la teoría de grafos y de grafos planos) Euler desarrolló algunos de los conceptos y propiedades fundamentales de la teoría de grafos. Sin embargo, el término grafo es utilizado por primera vez en 1878 por James Sylvester, en un artículo de la revista Nature [6], para definir estructuras compuestas por un número finito de vértices y una familia de pares de vértices a los que se llamó aristas.

Un problema clásico de esta rama de las matemáticas es la coloración de grafos (Vertex Coloring Problem - VCP - en inglés). Dado un grafo $G = (V, E)$ no dirigido se requiere asignar un color a cada vértice de tal manera que colores en vértices adyacentes son diferentes y el número de colores es mínimo. Sus orígenes se remontan a 1852 cuando Francis Guthrie, intentando colorear el mapa de las distintas regiones de Inglaterra, se formuló una pregunta que originó el problema conocido como el Problema de los Cuatro Colores, el cual enunciamos a continuación:

"Dado cualquier mapa geográfico con regiones continuas, éste puede ser coloreado con cuatro colores diferentes, de forma que no queden regiones adyacentes (es decir, regiones que compartan no sólo un punto, sino todo un segmento de borde en común) con el mismo color".

Sin embargo el mismo no fue resuelto hasta el año 1976 por Kenneth Appel y Wolfgang Haken, haciendo uso de las nociones desarrolladas por Heesch [7]. Para la demostración de este problema se utilizó un complicado análisis computacional de 1936 configuraciones (reducibles). Esta prueba no fue aceptada totalmente por la comunidad científica debido a la complejidad manual de verificación de resultados.

En 1997, Neil Robertson, Daniel Sanders, Paul Seymour y Robin Thomas reducen la cantidad de configuraciones a 633 [8].

2.2 Aplicaciones

El VCP es conocido como un problema de optimización combinatoria que surge como una generalización del Problema de los Cuatro Colores. Su popularidad en la literatura proviene de sus aspectos teóricos, de su dificultad desde el punto de vista computacional y principalmente de sus múltiples aplicaciones en la resolución de problemas en el mundo real. Entre sus diversas aplicaciones las más conocidas son:

- **Planificación de tareas** [9]**:** Son problemas que generalmente involucran restricciones sobre la incompatibilidad de ejecutar dos actividades de forma simultánea.

  Por ejemplo, en la planificación de cursos de cualquier institución educativa, existen cursos que no pueden ubicarse dentro del mismo bloque horario. Se puede representar esta información mediante un grafo de incompatibilidades donde los vértices son los cursos y las aristas representan aquellos cursos que no pueden ubicarse dentro del mismo bloque. Una coloración de este grafo provee una planificación factible de los cursos. Si además se requiere minimizar la cantidad de bloques a utilizar (suponiendo que todos cursos duran lo mismo), basta con encontrar el número cromático del grafo.

  Otros problemas similares involucran la planificación de tareas que comparten recursos [10], planificación de vuelos [11], el diseño de trayectos con semáforos [12], planificación de la recolección municipal de residuos [13], asignación mensual de trabajo [14], asignación de rango satelital [15], etc.

- **Asignación de registros** [16]**:** El asignador de registros es un subproceso dentro del proceso de compilación de software el cual asigna las variables contenidas en el código fuente a registros de hardware, tratando de minimizar las referencias a memoria principal.

  Para resolver este problema, se construye un grafo de incompatibilidad donde los vértices representan las variables utilizadas en el código y las aristas modelan las variables en conflicto, por ejemplo se crea una arista en el caso de que se utilice una variable antes y después de la utilización de otra diferente en un período corto de tiempo. Una coloración de este grafo produce una asignación sin conflictos. Además si el número cromático del grafo de incompatibilidades es menor o igual a la cantidad de registros, entonces es posible obtener coloraciones libres de conflicto sin utilizar referencias a memoria principal. En caso contrario, es necesario volcar a memoria algunas de ellas. Para el segundo caso, se procede a eliminar las variables volcadas a memoria una a una hasta que el número cromático del grafo resultado no supere la cantidad de registros. Esto último produce una asignación parcial de variables a registros.

- **Asignación de frecuencias** [17]**:** En la industria de las telecomunicaciones la creciente demanda en la utilización de frecuencias o canales no fue acompasada por el incremento de las frecuencias utilizables. Al establecer diferentes enlaces de comunicación en frecuencias idénticas o similares, pueden ocurrir interferencias. El objetivo de este problema es adjudicar diferentes frecuencias a diferentes usuarios de forma que las comunicaciones a través de los enlaces tengan muy poca o ninguna interferencia.

  Para asignar frecuencias a usuarios dos transmisores no pueden utilizar la misma frecuencia durante el mismo período de tiempo si no se encuentran separados por una cierta distancia.

  Para resolverlo se construye un grafo donde los vértices representan los transmisores, y las aristas indican que las señales emitidas por las antenas generan interferencias con al

menos una combinación de frecuencias. El objetivo de la coloración, es generar una asignación de frecuencias a transmisores de forma tal que la interferencia total involucrada en dicha asignación sea mínima.

- **Testeo de tarjetas de circuitos impresos** [18]**:** Una prueba que se aplica a las tarjetas de circuitos impresos es la búsqueda de pequeños circuitos no deseados. Bajo ciertas suposiciones acerca de posibles tipos de pequeños circuitos, se puede transformar esta búsqueda exhaustiva de minimización en un problema de coloración de vértices sobre un cierto grupo de grafos especiales, llamados grafos line-of-sight (línea de visión). De hecho, según el grupo de suposiciones tomadas se muestra que estos grafos se pueden colorear con 5, 8 o 12 colores, independientemente del número de vértices.

2.3 Definiciones básicas y notación

Definimos un **grafo** G por un par $\big(V(G), E(G)\big)$, donde $V(G)$ representa un conjunto finito de vértices (o nodos), y $E(G)$ representa un multiconjunto de pares no ordenados de vértices de G llamados aristas. Definimos $n = |V(G)|$ y $m = |E(G)|$, donde $|C|$ indica la cantidad de elementos del conjunto C. De aquí en más, llamaremos a $V(G)$ y $E(G)$ simplemente V y E.

Un grafo $G(V, E)$ es **simple** si $\forall\, v \in V, (v, v) \notin E$ y a lo sumo existe una sola arista entre todo par de vértices.

Un grafo $G(V, E)$ **no dirigido** es aquel cuyas aristas no tiene un sentido u orientación.

En este trabajo utilizaremos solamente grafos simples no dirigidos.

Un vértice v es **adyacente** a otro vértice w en G, si $(v, w) \in E$. Diremos que v y w son los extremos de la arista.

El **vecindario abierto** de un vértice v es el conjunto $N(v)$ que consiste de todos los vértices adyacentes a v. El **vecindario cerrado** $N[v]$ de un vértice v es el vecindario abierto al que se le agrega el nodo v ($N[v] = N(v) \cup v$).

El **grado** de un vértice $v \in V$, al que denotaremos $d(v)$, es el número de vecinos de v, es decir $d(v) = |N(v)|$.

El **soporte** de un vértice $v \in V$, al que denotaremos $s(v)$, se define como la suma de los grados de los vértices adyacentes a v, es decir $s(v) = \sum_{u \in N(v)} d(u)$.

El **complemento** de un grafo G, denotado por G^c, es el grafo que tiene el mismo conjunto de vértices de G tal que dos vértices distintos son adyacentes en $G^c \iff$ no son adyacentes en G.

Un grafo H es un **subgrafo** de un grafo G si $V(H) \subseteq V(G)$ y $E(H) \subseteq E(G)$.

Un grafo G es **completo** si cualquier par de vértices distintos de G son adyacentes. Llamaremos K_n al grafo completo con n vértices.

Un conjunto de vértices M de un grafo G es un **subgrafo completo** si el subgrafo inducido por M es completo.

Un **clique** es un subgrafo completo maximal de G. Se define $\boldsymbol{\omega(G)}$ como el tamaño del clique máximo de G. De aquí en más, llamaremos a $\omega(G)$ simplemente ω.

Un **coloreo** de un grafo $G(V, E)$ es una función $c: V \rightarrow \{1..n\} / c(u) \neq c(v), \forall (u, v) \in E$. Llamaremos a $c(v)$ color del vértice v.

El **número cromático** de G, denotado por $\chi(G)$, se define como la mínima cantidad de colores necesarios para colorear los vértices de G de modo que dos vértices adyacentes no tengan el mismo color.

El **problema de coloreo** combina las dos definiciones precedentes y consiste en hallar una coloración de vértices $c: V \rightarrow \{1..n\}$ / el número de colores utilizados es $\chi(G)$.

El **grado de saturación** de un vértice, definido como $gs(v)$, define la cantidad de colores diferentes que tienen los vértices adyacentes a v.

Un **conjunto independiente** (o conjunto estable) de un grafo G es un conjunto $I \subseteq V$ tal que para ningún par de vértices $v, w \in I$ ocurre que $(v, w) \in E$.

Un conjunto independiente es **maximal** si al agregar cualquier otro vértice de V que no esté en él, éste deja de ser un conjunto independiente.

El número de estabilidad, o $\boldsymbol{\alpha(G)}$, es el tamaño del máximo conjunto independiente de G. De aquí en más, llamaremos a $\alpha(G)$ simplemente α.

Un **conjunto de cubrimiento** de G es un subconjunto V_c de V tal que para cada $(v, w) \in E$, $v \in V_c$ y/o $w \in V_c$.

Un conjunto de cubrimiento es **minimal** si al quitar cualquier vértice de V_c, éste deja de ser un conjunto de cubrimiento.

Si V_c es un conjunto de cubrimiento minimal de G, se cumple que $I = V - V_c$ y $|I| = \alpha$.

2.4 Complejidad computacional

Un **problema** $\pi(I, Q)$ es un conjunto (posiblemente infinito) de instancias $\boldsymbol{I}$, y una pregunta $\boldsymbol{Q}$ sobre alguna propiedad de esas instancias [19].

Dentro de todas las familias de problemas podemos encontrar las siguientes subfamilias:

- **Problema de decisión:** Las únicas respuestas posibles son SI o NO. Estos problemas son los más importantes porque casi todo problema puede ser transformado en un problema de este tipo. Por ejemplo, determinar si existe un coloreo de un grafo con menos de k colores.

- **Problema de optimización:** Se busca dar una respuesta óptima a la pregunta del problema formulado. Por ejemplo, buscar el número cromático de un grafo.

- **Problema de enumeración:** Se busca encontrar la cantidad de soluciones para el problema formulado. Por ejemplo, determinar la cantidad de coloraciones posibles de un grafo utilizando solamente k colores.

Un **algoritmo** es un proceso formal para encontrar una respuesta a la pregunta de un problema para una cierta instancia dada del mismo [20]. Por lo general, para un problema dado existen varios algoritmos con diferente nivel de eficiencia.

Diremos que un algoritmo tiene **complejidad** $O(f(n))$, si para una entrada de tamaño n, el número de operaciones está acotado por la función $f(n)$.

A partir de lo anterior definimos que un algoritmo tiene complejidad **polinomial** cuando el número de operaciones efectuadas está acotado por una **función polinomial** en el tamaño de su entrada, es decir:

"Si n es el tamaño de la entrada de un algoritmo $a, \exists\ k$ constante / la complejidad de a es $O(n^k)$"

Diremos que el conjunto de problemas para los cuales existen algoritmos de resolución con esta complejidad de ejecución pertenecen a la ***Clase de Complejidad P***.

Por otro lado diremos que $\boldsymbol{NP}$ es el conjunto de problemas cuya solución se puede **comprobar** en tiempo polinomial. Más específicamente podemos decir que la ***Clase de Complejidad NP*** es el conjunto de lenguajes (es decir de respuestas SI/NO) que se pueden resolver con una máquina de Turing no determinista que termina su ejecución en un número de pasos acotado por una función polinomial $O(n^k)$ [21].

Se sabe que $P \subseteq NP$, sin embargo no se conoce si esta inclusión es estricta. De hecho determinar si $P = NP$ es el problema abierto más importante en Computación teórica y existe una recompensa de 1 millón de dólares para quién lo resuelva.

Un conjunto de problemas P es ***difícil*** con respecto a un conjunto de problemas L (con $L \in NP$) si $L \leq P$, es decir que L se puede escribir como un conjunto de soluciones de los problemas P, o L es "más sencillo" que P [21].

El conjunto difícil más importante es el ***NP-difícil*** y se define formalmente como el conjunto de lenguajes o problemas P / $\forall$ problema $l \in L$, (con $L \in NP$), existe una transformación polinomial tal que $L \propto P$ [21].

Una transformación polinomial de un lenguaje $L_1 \subseteq \Sigma_1^*$ a un lenguaje $L_2 \subseteq \Sigma_2^*$ es una función $f: \Sigma_1^* \to \Sigma_2^*$ que satisface las siguientes dos condiciones [21]:

1. Hay un programa MDT (Maquina determinista de Turing) de tiempo polinomial que computa f.
2. Para todos los $x \in \Sigma_1^*$, $x \in L_1$ si y sólo si $f(x) \in L_2$.

Si se cumple esta definición se escribe $L_1 \propto L_2$ y se dice que "L_1 transforma a L_2".

Un conjunto de problemas P es ***completo*** con respecto a un conjunto de problemas L, si es difícil para L y además es un subconjunto de L [21].

Finalmente decimos que un lenguaje L se define como ***NP-completo*** si $L \in NP$ y para todos los demás lenguajes $L' \in NP, L' \propto L$ [21].

De lo anterior se desprende que un problema que pertenece a la ***Clase de complejidad NP-difícil***, no necesariamente pertenece a la ***Clase de complejidad NP-completo***, si se cumple el recíproco. Además se puede deducir que los problemas en NP-completo son los más difíciles de NP y muy probablemente no formen parte de la Clase de Complejidad P, de hecho si se llegara a encontrar una solución en tiempo polinomial para algún miembro del conjunto NP-completo, se podría confeccionar una solución en tiempo polinomial para todos los problemas de este conjunto y se demostraría que $P = NP$.

En 1971, Cook introdujo la teoría de la NP-completitud [22] demostrando que el problema de satisfacibilidad de la lógica matemática es NP-completo. En 1972, Karp [23] mostró que el problema de coloración de grafos pertenece a este conjunto de problemas. Más aún, en 1974, Garey y Johnson [21] y luego junto a Stockmeyer [24] fortalecieron este resultado mostrando que una k-coloración de un grafo aún pertenece a esta clase de problemas con $k \geq 3$.

Aunque existen procedimientos que encuentran soluciones exactas, la complejidad del problema requiere el desarrollo de algoritmos heurísticos para resolver instancias grandes del mismo. Se puede encontrar evidencia de soluciones heurísticas en [25], [26], [27] y en la siguiente sección de esta tesis.

2.5 Algoritmos de Coloreo

Debido a su complejidad computacional (pertenece al conjunto de los problemas NP-completos) los métodos de resolución exactos propuestos se pueden aplicar sólo para instancias pequeñas, mientras que para resolver instancias de tamaño mayor (cientos a miles de vértices) es necesario la utilización de algoritmos heurísticos y metaheurísticos.

Dentro de la categoría de algoritmos heurísticos se destacan los algoritmos golosos (o greedy) muy utilizados en diversos problemas de optimización ya que se caracterizan por su buena performance y la obtención de resultados de buena calidad [28], aunque son sensibles a cambios en los parámetros de entrada (por ejemplo la ordenación de los vértices). Estos algoritmos colorean secuencialmente los vértices del grafo siguiendo criterios locales óptimos para seleccionar el siguiente vértice a colorear.

En cuanto a los algoritmos metaheurísticos utilizados, consideraremos aquellos basados en búsqueda local. Comenzando con una solución inicial, esta puede ser mejorada a través de un método de búsqueda local, que iterativamente transforme la solución actual en una solución mejorada. La búsqueda local requiere de la definición de tres conceptos:

- Un conjunto de soluciones candidatas (espacio de búsqueda), que pueden corresponder a una coloración parcial o a una coloración completa.
- Una relación de vecindad.
- Una función de evaluación de la solución.

Con respecto al número de colores utilizados (llamémosle k), este puede ser fijo o variable. Para el caso de k fijo el algoritmo resuelve la versión de decisión y para k variable la versión de optimización.

A raíz de lo anterior, podemos particionar los algoritmos de búsqueda local para VCP en tres familias:

- k fijo, coloraciones completas
- k fijo, coloraciones parciales
- k variable, coloraciones completas

Los algoritmos implementados en esta tesis se clasifican bajo la última categoría.

2.5.1 Algoritmos Heurísticos

A continuación enumeramos los algoritmos heurísticos más conocidos:

- **Algoritmo secuencial o SEQ** [28]**:** Dados n vértices, el primero es etiquetado con la menor clase de color y para cada vértice restante se asigna a la menor clase de color que no contenga vértices adyacentes al vértice que se quiere procesar.

- **DSATUR (versión golosa)** [25]**:** Dados n vértices, el paso inicial consiste en elegir $v \,/\, d(v) = Max\{d(v), v \in V\}$, él cual es etiquetado con la menor clase de color. Luego para el resto de los vértices en el subgrafo sin colorear V', se selecciona $w \,/\, gs(w) = Max\{gs(u), u \in V'\}$, o sea el que tiene la mayor cantidad de vecinos con colores diferentes y se colorea con la menor clase de color posible. En caso de haber empate, se escoge aquel vértice de mayor grado en V'. El algoritmo repite el procedimiento (sin el paso inicial) mientras existan vértices por colorear.

- **RLF (Recursive Largest First)** [29]**:** A diferencia de los anteriores, este algoritmo construye secuencialmente conjuntos de vértices que pueden tomar el mismo color. Resultados computacionales muestran que éste último mejora claramente los otros dos en términos de calidad en una amplia gama de tipos de grafos. Sin embargo tiene un alto costo computacional si tenemos en cuenta que es de orden $O(n^3)$ para el peor caso a diferencia de las heurísticas anteriores que tienen un orden $O(n^2)$. En las sub-secciones 2.5.3 y 2.5.4 entraremos en más detalles sobre éste algoritmo en su versión estándar y optimizada.

2.5.2 Algoritmos Metaheurísticos

Basándonos en las tres familias de algoritmos definidas mencionaremos a continuación algunos algoritmos metaheurísticos utilizados para la resolución del VCP [28]:

- **TABUCOL:** Este algoritmo está basado en la metaheurística de búsqueda tabú [30]. Dado un número *k* fijo de colores disponibles, este algoritmo evoluciona construyendo coloraciones completas. Una solución es representada por una *k* coloración del grafo, en donde algunas aristas pueden ser conflictivas (los vértices que forman dicha arista comparten el mismo color). Un movimiento consiste en cambiar el color de uno de los vértices y la función de evaluación mide el número de conflictos en la coloración actual. El algoritmo sólo considera movimientos de vértices críticos, es decir, de aquellos vértices cuyo color se encuentra actualmente en conflicto con algún vértice adyacente. Una lista tabú almacena la asignación de colores a los vértices y prohíbe que esa asignación vuelva a realizarse durante un determinado número de iteraciones (tenencia tabú).

- **HCA (Hybrid Coloring Algorithm)** [31]**:** Es un algoritmo evolutivo que trabaja con k fijo y que combina una versión mejorada de **TABUCOL** con un operador de cruzamiento especializado para el VCP llamado **Greedy Partitioning Crossover**. Este operador considera dos soluciones que representan los padres (particiones de vértices en k conjuntos, no necesariamente independientes), y alternativamente elige la clase de color de cardinalidad máxima de uno de los padres para generar la próxima clase de color del descendiente. Todos los vértices de esta clase de color se eliminan de los padres. Luego de ejecutar k pasos, puede suceder que algunos vértices permanezcan sin asignar, por lo que cada uno de ellos es asignado a una clase elegida en forma aleatoria. El poder de este operador reside en su capacidad de transmitir a los descendientes la estructura de los padres, es decir la partición en conjuntos.

- **Impasse Class Neighborhood (ICN)** [32]**:** Es una estructura utilizada para tratar de convertir una k coloración parcial en una k coloración completa. Una solución *ICN* es una partición de V en $k+1$ clases de color $(V_1, \dots, V_k, V_{k+1})$, en donde todas las clases excepto la V_{k+1} son conjuntos independientes. V_{k+1}, denominada **impasse class**, contiene todos los vértices sin colorear que no se pueden incluir en las k primeras clases. El método que utiliza esta estructura requiere inicialmente un valor objetivo k de colores a ser utilizados para formar una solución con las características antes descriptas. El objetivo es vaciar el conjunto V_{k+1} para obtener una coloración completa. A tales efectos se distribuyen sus vértices en los restantes k conjuntos preservando la propiedad de conjuntos independientes. Para ello en cada paso el algoritmo toma un vértice de forma aleatoria de V_{k+1} y escoge un conjunto $h\,(1 \leq h \leq k)$ de tal forma que insertar ese vértice allí minimice el grado global de los vértices en V_{k+1}.

- **MIPS-CLR** [33]**:** Este algoritmo combina una técnica de búsqueda tabú basada en **Impasse Class Neighborhood** con diferentes procedimientos heurísticos, coloración fija y recombinación de soluciones con el objetivo de expandir una posible coloración parcial a una coloración completa. Este algoritmo trabaja con k variable.

- **VNS (Variable Neighborhood Search)** [34]**:** Es una técnica que combina búsqueda local con diferentes vecindades: cuando el algoritmo es atrapado en un óptimo local, la vecindad es cambiada y de esta forma la búsqueda puede continuar. Un algoritmo para VCP basado en VNS fue propuesto por Avanthay (2003) [35].

- **EXTRACOL** [36]**:** Este algoritmo está basado en el principio general de **reducir y solucionar**, trabaja con k variable y genera coloraciones completas. Este enfoque consiste en aplicar primero una etapa de extracción de conjuntos independientes de mayor tamaño posible de manera que el grafo residual resultante sea lo suficientemente pequeño para ser coloreado con cualquier algoritmo de coloración. El algoritmo utilizado originalmente para colorear el grafo residual es un algoritmo memético reciente denominado **MACOL** [37].

Vimos anteriormente en la sección 1.4.1 que el problema de asignación de actividades a unidades puede ser resuelto a partir de una coloración factible del grafo de incompatibilidades. En particular, es de interés encontrar la mejor agrupación de actividades de forma de minimizar la cantidad de unidades necesarias para el correcto funcionamiento de la empresa. Intuitivamente esto equivale a buscar una coloración tal que el número de colores se corresponda con el número cromático del grafo (ver definición de número cromático en 2.3). Para resolver el problema de coloración, decidimos utilizar algoritmos basados en la formulación de Conjuntos de Cubrimiento, propuesto por Mehrotra y Trick [38], debido a la existencia de muchos algoritmos heurísticos y metaheurísticos, algunos de ellos muy recientes, que se basan en la extracción de conjuntos independientes (ver en [28], [36] y [39]).

La formulación de Conjuntos de Cubrimiento enuncia lo siguiente:

Dado F_I la familia de todos los conjuntos independientes de G, se asocia una variable binaria x_I a cada conjunto independiente $I \in F_I \ / \ x_I = 1 \Leftrightarrow \forall\, v, w \in I, color(v) = color(w)$

$$mín \sum_{I \in F_I} x_I \qquad (2.5.1)$$

$$s.a. \begin{cases} \sum_{I \in F_I} x_I \geq 1, \forall\, w \in V & (2.5.2) \\ x_I \in \{0,1\}, I \in F_I & (2.5.3) \end{cases}$$

La función objetivo 2.5.1 busca minimizar la cantidad de conjuntos independientes, respetando un conjunto de restricciones que enumeramos a continuación:

- La restricción 2.5.2 establece que todo vértice *w* del grafo debe pertenecer al menos a un conjunto independiente.
- La restricción 2.5.3 establece que x_I debe ser una variable binaria.

Para resolver este modelo matemático decidimos utilizar, luego de la realización de pruebas, el algoritmo **LRLF** (Lazy RLF) para encontrar soluciones de buena calidad rápidamente sobre instancias pequeñas y medianas (número de vértices $\leq$ 1000). Para instancias de mayor tamaño utilizamos una adaptación del **EXTRACOL**, en el cual combinamos la técnica de obtención de conjuntos independientes de este algoritmo, sustituyendo el **MACOL** por un algoritmo heurístico para colorear el grafo residual con la finalidad de mejorar las soluciones obtenidas y de esa manera poder visualizar si mejora aún más la asignación de personal. En las siguientes secciones describimos en detalle cada uno de

los algoritmos propuestos así como mejoras y resultados obtenidos producto de la ejecución en diferentes instancias de DIMACS y del grupo de módulo.

2.5.3 Algoritmo goloso de coloreo – RLF (Recursive Largest First)

A continuación definimos el algoritmo "clásico" de RLF para resolver el problema de coloración de grafos. En la sección que sigue describimos las modificaciones realizadas al algoritmo clásico para mejorar el tiempo de ejecución del mismo.

Sea $\boldsymbol{P}$ el conjunto de los vértices sin colorear que pueden ser coloreados con un cierto color $\boldsymbol{k}$, y $\boldsymbol{U}$ el conjunto de los vértices sin colorear que no pueden ser coloreados con $\boldsymbol{k}$, el algoritmo RLF colorea los vértices tomando en cuenta una clase de color a la vez de la siguiente manera:

- Inicializar $\boldsymbol{P}$ con todos los vértices del grafo y dejar $\boldsymbol{U}$ vacío.
- Elegir el primer vértice $\boldsymbol{v_0}$ perteneciente a $\boldsymbol{P}$ que tiene el máximo número de vértices adyacentes en $\boldsymbol{P}$ (vértice con mayor grado). Colorear $\boldsymbol{v_0}$ con $\boldsymbol{k}$ y mover todos los vértices $\boldsymbol{u}$ pertenecientes a $\boldsymbol{P}$ que son adyacentes a $\boldsymbol{v_0}$ desde $\boldsymbol{P}$ a $\boldsymbol{U}$.
- Mientras $\boldsymbol{P}$ no sea vacía hacer lo siguiente: elegir el primer vértice $\boldsymbol{v}$ perteneciente a $\boldsymbol{P}$ que tiene el máximo número de vértices adyacentes en $\boldsymbol{U}$ (vértice de mayor grado inducido por el conjunto $\boldsymbol{U}$); colorear $\boldsymbol{v}$ con $\boldsymbol{k}$ y mover todos los vértices $\boldsymbol{u}$ pertenecientes a $\boldsymbol{P}$ que son adyacentes a $\boldsymbol{v}$ desde $\boldsymbol{P}$ a $\boldsymbol{U}$.
- Para la próxima iteración, incrementar el valor de $\boldsymbol{k}$, inicializar nuevamente el conjunto $\boldsymbol{P}$ con los vértices aún sin colorear que hay en $\boldsymbol{U}$ y vaciar $\boldsymbol{U}$.

Antes de exponer el pseudocódigo del algoritmo, definimos la siguiente notación:

- $G[X]$: Subgrafo de G inducido por el conjunto de vértices X, es decir:

$$G[X] = (X, E[X]), \text{ donde } E[X] = \{(u, v) \in E \mid u, v \in X\}$$

- $\delta_X(v)$: Conjunto que contiene los vértices adyacentes a v en $G[X \cup v]$.
- $d_X(v)$: Grado de v inducido por X, es decir:

$$d_X(v) = |\delta_X(v)|$$

2.5.3.1 Pseudocódigo del Algoritmo RLF

RLF(G)

In $G(V, E)$: (Grafo de entrada)
Out k: cantidad de colores de la coloración devuelta
Out c: una coloración $c: V \rightarrow k$ de G
Var P: conjunto de vértices sin colorear que pueden ser coloreados con el color k
Var U: conjunto de vértices sin colorear y que no pueden ser coloreados con el color actual

```
1: k ← 0
2: while |V| > 0 do
3:       k ← k + 1                                  // Agrego un nuevo color a la coloración
4:       P ← V, U ← ∅
5:       for all v ϵ P do
6:            d_U(v) ← 0
7:       endfor
8:       v ← w | w ϵ P y d_P(w) es máximo
9:       c(v) ← k                                   // El vértice v es coloreado con el color k
10:      for all w ϵ δ_P(v)do
11:           P ← P\{w}
12:           U ← U ∪ {w}                           //Muevo w de P a U
13:      endfor
7- 14:   while |P| > 0 do
8- 15:        v ← w | w ϵ P y d_U(w)es máximo        // Vértice con mayor grado inducido por U
9- 16:        c(v) ← k                              // El vértice v es coloreado con el color k
10- 17:       V ← V \ {v}                           //G = (V, E) se reduce en cada iteración
11-18:        for all w ϵ δ_V(v)do
12-19:             if w ϵ P
13-20:                  P ← P\{w}
14-21:                  U ← U ∪ {w}                 //Muevo w de P a U
15-22:             endif
16-23:             E ← E\ {v, w}
17-24:             for all u ϵ δ_P(w) do
18-25:                  d_U(u) ← d_U(u) + 1         //Actualizo el grado inducido por U
19-26:             endfor
20-27:        endfor
21-28: endwhile
22-29: P ← U, U ← ∅                                 //Actualizo P con los vértices de U
23-30: endwhile
24-31: return k
```

Al principio el conjunto P contiene todos los vértices del grafo y el grado inducido por U para todos los vértices es igual a 0.

El algoritmo en cada iteración selecciona un vértice v a ser coloreado (líneas 15 y 16), mueve sus vecinos $\delta_P(v)$ desde P a U (líneas 19 a 22), reduce el grafo G (líneas 17 y 23) y

actualiza el grado inducido por U de todos los vértices adyacentes a los vértices que fueron movidos de P a U.

Una aclaración importante sobre la reducción de sentencias debido a la utilización de notación es que cuando se menciona algo del estilo "for all w ϵ δ_X(v) do" en el pseudocódigo, donde $X \subseteq V$, por lo general, si no existe una función especializada que implemente lo expresado, en código esto se puede traducir en las siguientes 2 sentencias que devuelven un resultado equivalente:

```
for all w ϵ δV(v)do
        if w ϵ X
                ……
                ……
                ……
        endif
```

2.5.4 Algoritmo goloso de coloreo – LRLF (Lazy Recursive Largest First)

La mejora que propone este algoritmo es seleccionar el vértice v con máximo grado inducido por U mediante el cálculo de $d_U(v)$ de forma perezosa en vez de mantener un arreglo de grados y actualizarlo en cada selección de vértices [39]. A continuación mostramos la preposición que sustenta esta idea:

Definición 2.5.1: Se define $d_U^{max} = max\{d_U(v) | v \in P\}$ y $\bar{d}_U$ como el mayor grado inducido por U encontrado hasta el momento.

Teorema 2.5.2: Un vértice v con $d_V(v) < \bar{d}_U$ tiene $d_U(v) < d_U^{max}$

Demostración: En el grafo reducido $G(V, E)$ que es el grafo inducido por el conjunto de vértices sin colorear, tenemos que $\delta_V(v) = \delta_P(v) \cup \delta_U(v)$ y por lo tanto

$d_V(v) = d_P(v) + d_U(v)$. Por lo que, si $d_V(v) < \bar{d}_U$, entonces $d_U(v) \leq d_V(v) < \bar{d}_U \leq d_U^{max}$.

Como consecuencia, una vez que encontramos un vértice con $\bar{d}_U$, podemos omitir el cálculo de d_U para todos los vértices con $d_V(v) < \bar{d}_U$. Además para aprovechar más aun la propiedad, en vez de calcular $d_U(v)$ comenzando desde 0 e incrementando su valor cada vez que uno de sus vecinos pertenece a U (Método A) podemos comenzar desde $d_V(v)$ y decrementar su valor cada vez que un vecino pertenece a P (o sea que no está en U), como muestra el Método B. De esta manera, cada vez que $d_U(v) < \bar{d}_U$ llevamos a cabo $d_V(v) - \bar{d}_U$ operaciones en vez de $d_V(v)$. Cuanto mayor es el valor de $\bar{d}_U$ mayor es el ahorro. Además si comenzamos con el vértice de mayor grado en G, se tienen buenas posibilidades de que también tenga un valor alto de $\bar{d}_U(v)$.

Método A: Cálculo $d_U(v)$ tradicional

```
d_U(v) ← 0
for all w ϵ δ_U(v) do
        d_U(v) = d_U(v)+ 1
return d_U(v)
```

Método B: Cálculo $d_U(v)$ perezoso

```
d_U(v) ← d_V(v)
for all w ϵ δ_P(v) do
        d_U(v) = d_U(v) – 1
        if d_U(v) < d̄_U then
                return 0
return d_U(v)
```

A continuación presentamos el pseudocódigo del algoritmo LRLF con las modificaciones que implementan la mejora descrita anteriormente.

2.5.4.1 Pseudocódigo del Algoritmo LRLF

LRLF(G)

In $G(V, E)$: (Grafo de entrada)
Out k: cantidad de colores de la coloración devuelta
Out c: una coloración $c: V \rightarrow k$ de G
Var P: conjunto de vértices sin colorear que pueden ser coloreados con el color k
Var U: conjunto de vértices sin colorear y que no pueden ser coloreados con el color actual

```
1: k ← 0
2: while |V| > 0 do
3:      k ← k + 1                                  // Agrego un nuevo color a la coloración
4:      P ← V, U ← ∅
5:      while |P| > 0 do
6:              v ← w | w ϵ P y d_P(v) es máximo     // Vértice con mayor grado en P
7:              d̄_U ← 0                             // Calculamos d̄_U desde 0
8:              for all w ϵ δ_U(v) do
9:                      d̄_U = d̄_U + 1
10:             for all w ϵ P\{v} do                //Búsqueda de v | d_U(v) = d_U^max
11:                     if d_V(w) ≥ d̄_U             //Descartamos los vértices con d_V < d̄_U
12:                             d_U(w) ← d_V(w)
13:                             for all u ϵ δ_P(w)do
14:                                     d_U(w) = d_U(w) – 1
15:                                     if d_U(w) < d̄_U then
16:                                             continuar desde 10
17:                                     endif
```

```
18:                         endfor
19:                         if d_U(w) = d̄_U and d_V(w) ≥ d_V(v) then
20:                             continuar desde 10
21:                         endif
22:                         v ← w
23:                         d̄_U ← d_U(w)
24:                     endif
25:             endfor
26:             c(v) ← k                        // El vértice c es coloreado con el color k
27:             V ← V \ {v}                     //G = (V, E) se reduce en cada iteración
28:             for all w ϵ δ_V(v)do
29:                 if w ϵ P
30:                     P ← P\{w}
31:                     U ← U ∪ {w}             //Muevo w de P a U
32:                 endif
33:                 E ← E\ {v, w}
34:             endfor
35:     endwhile
36:     P ← U, U ← ∅
37: endwhile
38: return k
```

Se eliminan las líneas 5 y 6 del algoritmo original que inicializan el grado inducido por U en 0 para todos los vértices del grafo, ya que ahora se inicializan con el valor del grado inducido por V (es decir con el valor del grado de los vértices del grafo original - línea 12). También se eliminan las filas 24 a 26, que es donde se incrementa en uno el grado inducido por U de los vértices adyacentes a los vértices que han sido movidos de P a U. Estas líneas se eliminan ya que en el algoritmo mejorado el grado inducido por U se obtiene decrementando en uno cada vez que un vértice adyacente al vértice movido de P a U, pertenece a P y tiene un grado mayor o igual al mayor grado calculado hasta el momento, en otro caso no se actualiza.

2.5.5 Algoritmo heurístico – Coloreo Residual (CR)

Antes de implementar el Extracol, decidimos probar un algoritmo más simple aplicando de forma pura un enfoque de reducción que consiste en extraer en cada iteración un conjunto independiente de tamaño máximo del grafo residual, hasta que no queden vértices para procesar.

Se generaron varias versiones de este algoritmo donde cada una de ellas utiliza una solución diferente de búsqueda de máximo conjunto independiente (CR_MCIS, CR_VSA, CR_ATS).

Básicamente el algoritmo es un template donde la única diferencia entre las diferentes versiones está en la invocación del algoritmo que busca el máximo conjunto independiente (MCIS, VSA, ATS).

A continuación presentamos el pseudocódigo del algoritmo CR indicando las líneas que difieren entre versiones.

2.5.5.1 Pseudocódigo del Algoritmo CR

CR(G)

In $G(V, E)$: (Grafo de entrada)
Out k: cantidad de colores de la coloración devuelta
Out c: una coloración $c: V \rightarrow k$ de G

```
1: k ← 0
2: while |V| > 0 do
3:     k ← k + 1                                  // Agrego un nuevo color a la coloración
4:     I_M = Max_Conjunto_Independiente(G)        // Aquí aplicamos algún algoritmo de máximo c. i.
5:     V = V – I_M                                // Elimino vértices de G
6:     E = E – {(v,w), v ϵ I_M, w ϵ V, (v,w) ϵ E } // Elimino aristas de G
7:     for all v ϵ I_M do
8:         c(v) = k
9:     endfor
10: endwhile
11: return k
```

Mientras queden vértices por procesar (línea 2), se aplica algún algoritmo de los descriptos en 2.6 para encontrar un conjunto independiente maximal de G (línea 4).

Finalmente se remueven todos los vértices y aristas involucrados en I_M de G (líneas 5 y 6) y se colorean todos los vértices contenidos en I_M con el mismo color (líneas 7 a 9).

2.5.6 Algoritmo metaheurístico de coloreo – Extracol

Este algoritmo está basado en el enfoque de "reducir y resolver", que consiste en aplicar primero una **etapa de preprocesamiento** basada en extraer conjuntos independientes del mayor tamaño posible, hasta que el grafo residual sea lo sufucientemente pequeño para poder ser coloreado eficientemente utilizando cualquier algoritmo de coloreo (**etapa de coloreo**) [36]. Típicamente, los métodos convencionales en la primer etapa operan de forma golosa extrayendo de a un conjunto independiente por vez (por ejemplo ver [30] y [40]). En [41], se utiliza un refinamiento en la forma de selección de el conjunto independiente, escogiendo aquel que esté conectado a la mayor cantidad de vértices posibles en el grafo residual. Extracol propone un nuevo enfoque para la extracción de estos conjuntos, básicamente en vez de quitarlos de uno en uno, se intenta identificar la mayor cantidad de conjuntos independientes de mayor tamaño disjuntos entre sí, de forma de poder extraer más vértices en cada paso de preprocesamiento y de esta manera intentar obtener un grafo residual más fácil de procesar en la siguiente iteración.

La búsqueda del máximo conjunto independiente es un problema NP-completo en sí mismo [21], por lo tanto utilizar esta aproximación como medio de resolución para encontrar una coloración hace necesario optar por la aplicación de algoritmos heurísticos o metaheurísticos. Es por eso que para la búsqueda del máximo conjunto independiente y como parte de la construcción del máximo conjunto de conjuntos independientes disjuntos entre sí, Extracol propone la implementación de una búsqueda Tabú llamada Adaptative Tabú Search Algorithm (o ATS, ver sección 2.6.3).

El algoritmo original, combina esta técnica de pre procesamiento mejorada con un algoritmo memético para resolver la coloración del grafo residual. En este trabajo proponemos cambiar el paso de coloración sustituyendo el algoritmo memético por la heurística LRLF mencionada en 2.5.4.

A continuación presentamos el pseudocódigo del algoritmo Extracol con las modificaciones mencionadas.

2.5.6.1 Pseudocódigo del Algoritmo Extracol

EXTRACOL(G)

In $G(V, E)$: (Grafo de entrada)
In q: Tamaño del grafo residual (medido en cantidad de vértices)
In ici_{Max}: máxima cantidad de iteraciones sin éxito para encontrar nuevos conjuntos independientes.
In M_{Max}: tamaño máximo del conjunto de conjuntos independientes.
Out k: cantidad de colores de la coloración devuelta
Out c: una coloración $c: V \rightarrow k$ de G
Var I_{Max}: conjunto independiente inicial.
Var I: conjunto independiente / $|I| = |I_M|$.
Var z_{Max}: tamaño de I_M.
Var M: conjunto de conjuntos independientes.
Var $iter_{max}$: máxima cantidad de intentos para encontrar un conjunto independiente mejor.

```
1: k ← 0
2: while |V| > q do
3:     I_Max = ATS(G,_,Iter)            // Busco el máximo conjunto independiente en G
4:     z_Max = |I_Max|
5:     M = {I_Max}
6:     cir = 0                          //Cuenta los conjuntos independientes repetidos seguidos
7:     while (cir < ici_Max and |M| < M_Max) do
8:         I = ATS(G,z_Max,iter_Max)    // Busco un conjunto independiente de tamaño z_max
9:         if I ϵ M then
10:            cir = cir + 1
11:        else
12:            M = M ∪ I
13:            cir = 0
```

```
14:             endif
15:     endwhile
16:     {I1,…,It} = max{|S| : S ⊆ M, ∀ Ix, Iy ϵ S, Ix ∩ Iy = ∅}
17:     if |V| – |I1∪……∪ It| > q then
18:             V = V – {I1∪……∪ It}                          // Elimino vértices de G
19:             E = E – {(v,w), v,w ϵ {I1∪……∪ It}, (v,w) ϵ E }   // Elimino aristas de G
20:             for all I ϵ {I1,…,It} do
21:                     k ← k + 1                            // Agrego un nuevo color a la coloración
22:                     for all v ϵ I do
23:                             c(v) = k
24:                     endfor
25:             endfor
26:     else
27:             {I'1,…,I'p} = random{R ⊆ {I1∪……∪ It}, |R| = p = ⌈|V|−q / zMax⌉}
28:             V = V – {I'1∪……∪ I'p}                        // Elimino vértices de G
29:             E = E – {(v,w), v,w ϵ {I'1∪……∪ I'p}, (v,w) ϵ E }   // Elimino aristas de G
30:             for all I ϵ {I'1,…,I'p} do
31:                     k = k + 1
32:                     for all v ϵ I do
33:                             c(v) = k
34:                     endfor
35:             endfor
36:     endif
37: endwhile
38: k = k + Heurística(G, c)        // Termino de colorear el grafo residual aplicando un algoritmo heurístico
39: return k
```

Inicialmente se aplica ATS para encontrar un conjunto independiente maximal I_{Max} en G y en consecuencia su tamaño ($z_{Max} = |I_{Max}|$). Luego se utiliza I_{Max} para inicializar M, que es un conjunto cuya intención es almacenar conjuntos independientes de igual tamaño (líneas 3 a 5).

Se aplica repetidamente ATS para generar la mayor cantidad de conjuntos independientes de tamaño z_{Max} , los cuales son insertados en M mientras no se supere el tope de conjuntos repetidos ici_{max} o el máximo M_{Max} de conjuntos que se pueden almacenar en este conjunto (líneas 7 a 15).

Se busca el subconjunto de M que contenga la mayor cantidad de conjuntos independientes disjuntos $\{I_1, \dots, I_t\}$ (línea 16, ver 2.5.6.2).

Para finalizar la etapa de preprocesamiento, se remueven todos los vértices y aristas involucrados en $\{I_1 \cup \dots \cup I_t\}$ de G si el grafo residual tiene a los sumo q vértices y se colorean todos los vértices de cada $I_i, con\ 1 \leq i \leq t$ con el mismo color, variando el color entre conjuntos diferentes (líneas 18 a 25), de otra forma se escogen randómcamente conjuntos independientes de $\{I_1, \dots, I_t\}$, llamemos a este nuevo subconjunto $\{I'_1, \dots, I'_p\}$, de forma tal que $|\{I'_1 \cup \dots \cup I'_p\}| \geq \left\lceil \frac{|V|-q}{z_{Max}} \right\rceil$, con el cual procedemos a remover y colorear de

forma idéntica tomando en cuenta solamente los elementos de G(vértices y aristas) involucrados en este nuevo subconjunto (líneas 27 a 35).

Finalmente se aplica un algoritmo heurísitico, que especificaremos en la sección de pruebas de algoritmos de coloreo 2.7.5) sobre el grafo residual resultado de la etapa anterior para completar la coloración del grafo original (línea 38).

2.5.6.2 Conjunto Maximal de Conjuntos Independientes disjuntos

Como se mencionó en el item anterior, para la etapa de preprocesamiento, es necesario encontrar el mayor subconjunto de conjuntos independientes disjuntos de M. Éste problema lo podemos relacionar con el problema de máximo empaquetamiento (MSPP – maximum set packing problem), o lo que equivale a encontrar el máximo conjunto independiente.

El enfoque aplicado para resolver este problema saca provecho de la relación de equivalencia que existe entre el MSPP y la búsqueda del máximo conjunto independiente. De esta forma se formula un problema análogo donde dados n conjuntos independientes $\{I_1, \dots, I_n\}$, se define un grafo nuevo $G'(V', E')$ de la siguiente manera:

$$V' = \{1, \dots, n\}$$

$$e_{i,j} = \begin{cases} 0, si\ I_i \cap I_j = \emptyset, i, j \in \{1, \dots, n\} \\ 1, de\ otra\ manera \end{cases}$$

Notar que la cantidad de vértices de G' está acotado por M_{max} (ver 2.5.6.1).

A partir de lo anterior, es secillo visualizar que si $\{i_1, \dots, i_t\}$ es un conjunto independiente de G', entonces $\{I_{i1}, \dots, I_{it}\}$ es un conjunto de conjuntos independientes disjuntos. Entonces para resolver el problema basta con encontrar el máximo conjunto independiente en G', para lo cual decidimos aplicar ATS sin especificar un tamaño de conjunto objetivo (ATS(G,_,Iter)) sobre este grafo (ver 2.6.3).

2.6 Algoritmos de Máximo Conjunto Independiente

2.6.1 Algoritmo heurístico Máximo Conjunto Independiente Simple (MCIS)

Es un algoritmo simple que en cada paso toma el vértice de menor grado del grafo residual, sin tener en cuenta ninguna otra propiedad, para luego eliminar el vértice escogido (que se incluye en el conjunto independiente) y sus adyacentes del grafo (reducción del grafo). De esta forma lo que se busca es descartar la menor cantidad de vértices posibles en cada iteración.

Fue concebido en 3 versiones diferentes, cada una de ellas aportando un grado mayor de complejidad, si bien en términos generales todas siguen la misma estrategia de resolución.

Las versiones implementadas son:

- MCIS: Es el algoritmo más performante, pues solamente realiza una ordenación inicial de los vértices según el grado de menor a mayor. Si bien la ventaja es la reducción de operaciones, como desventaja tiene que a medida que el algoritmo avanza la ordenación inicial de vértices probablemente deje de ser válida, ya que los grados de los vértices cambian dinámicamente en cada iteración.

- MCISM (M = Mejorado): Para reflejar el cambio dinámico de los grados de los vértices en el criterio de selección, éste algoritmo busca el vértice de menor grado en cada iteración, ya sea realizando una comparación exhaustiva entre todos los vértices, u ordenando los mismos de forma creciente. De esta forma siempre se escoge realmente el vértice de menor grado, a expensas de un aumento en el tiempo de ejecución (debido a las sucesivas ordenaciones o búsquedas).

- MCISMR (R = Randómico): Esta última versión introduce aleatoriedad al momento de producirse un empate debido a la existencia de más de un vértice con grado mínimo, de esta manera se incorpora la capacidad de poder producir diferentes resultados.

A continuación presentamos el pseudocódigo del algoritmo MCISMR indicando las variantes que se deberían realizar para generar las versiones más simples.

2.6.1.1 Pseudocódigo del Algoritmo MCISMR

MICSMR(G)

In $G(V, E)$: (Grafo de entrada)
Out I_{Max}: Conjunto independiente maximal de G.

1: **while** |V| > 0 **do**
2 $V_{Min} = \{v \in V, \nexists\, w\ d(w) < d(v), w \in V\}$ // Conjunto de vértices con menor grado
3: v = Random(V_{Min}) // Retorna cualquier vértice de V_{Min}
4: V = V – N[v] // N(v) es el vecindario cerrado de v
5: $I_{Max} = I_{Max} \cup \{v\}$
6: **endwhile**
7: **return** I_{Max}

Mientras queden nodos por procesar, se realiza una búsqueda de todos los vértices de menor grado (línea 2). Una vez obtenido dicho subconjunto de V se escoge uno de ellos (cualquiera) y se elimina, junto a todos sus vecinos, del conjunto de vértices original (líneas 3 y 4). Finalmente se agrega el vértice seleccionado al conjunto independiente en construcción.

Para implementar el algoritmo MCISM, se debe cambiar la línea 3 por First(V_{Min}), función que retorna el primer elemento del conjunto.

Para implementar MCIS, la línea 2 se elimina y se agrega antes del while de la línea 1 la sentencia Sort(V) que ordena el conjunto de vértices según el grado de menor a mayor. Luego la línea 3 se modifica por First(V), función que tiene el mismo efecto que en MCISM pero aplicado sobre V.

2.6.2 Algoritmo heurístico Vertex Support Algorithm (VSA)

El objetivo del algoritmo que presentamos a continuación es el de encontrar el conjunto independiente maximal o máximo conjunto independiente (MIS por sus siglas en ingles) de un grafo dado, que se calcula a través de la resolución del problema de conjunto de cubrimiento minimal o cobertura mínima de vértices (MVC) utilizando el soporte de vértices.

El problema del máximo conjunto independiente (MIS) es un problema NP-hard clásico de optimización en grafos, que posee muchas aplicaciones en el mundo real. Dado un grafo $G = (V, E)$, el problema del conjunto independiente es el de encontrar un subconjunto S de V de cardinalidad máxima de tal forma que no haya dos vértices adyacentes en S. Por otro lado el problema del conjunto de cubrimiento minimal de un grafo trata de encontrar un conjunto de cubrimiento V_C de G de cardinalidad mínima. Ambos problemas se encuentran relacionados dado que el conjunto independiente maximal S de G contiene todos los vértices del grafo G que no forman parte del conjunto de cubrimiento minimal V_C de G, es decir $S = V - V_C$.

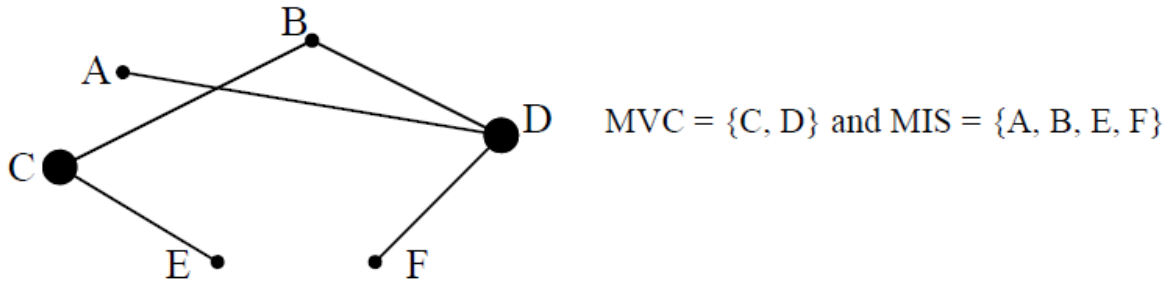

Figura 1 – Conjuntos independiente maximal y cubrimiento minimal

La figura anterior describe claramente la relación entre los conjuntos independiente maximal y de cubrimiento minimal.

Para el problema del conjunto de cubrimiento existen dos versiones, la de decisión y la de optimización. Para la versión de decisión la tarea consiste en verificar para un grafo G dado si existe un conjunto de cubrimiento de un tamaño especificado, pero en la versión de optimización la tarea es la de encontrar un conjunto de cubrimiento de tamaño mínimo. Para este caso se consideró la opción utilizada por el autor [42] es decir la versión de optimización del conjunto de cubrimiento minimal con el objetivo de obtener una solución óptima. Éste problema puede ser formulado como un problema de programación entera usando las siguientes condiciones:

- Se definen las variables binarias a_{ij} $(i = 1,2,3, \ldots., n; j = 1,2,3, \ldots\ldots, n)$ que forman la matriz de adyacencia del grafo G.
- Cada variable tiene dos valores (1 o 0) dependiendo de si la arista existe o no, es decir si una arista (v_i, v_j) está en E entonces $a_{ij} = 1$ sino $a_{ij} = 0$.
- La salida del algoritmo muestra si el vértice v_i está en el conjunto independiente o no, si $v_i = 1$ está en el conjunto independiente en otro caso $v_i = 0$.
- El número de vértices del conjunto independiente se expresa como $Z = \sum v_i$
- Se cumple que sólo uno o ninguno de los vértices de la arista (v_i, v_j) está incluido en el conjunto independiente, por lo que se puede escribir la siguiente restricción para el MIS:

$$v_i + v_j \leq 1$$

Finalmente el problema puede ser transformado en el siguiente problema de optimización de programación entera:

$$Max\ Z = \sum v_i$$

$$s.a. \begin{cases} v_i + v_j \leq 1 \ \ \forall (v_i, v_j) \in E \\ v_i \in \{0,1\} \ \forall \ v_i \in V \end{cases}$$

2.6.2.1 Pseudocódigo del Algoritmo VSA

Como se mencionó anteriormente este algoritmo fue diseñado para encontrar el conjunto independiente maximal de un grafo G dado. La matriz de adyacencia $A = (a_{ij})$ de G, es un parámetro de entrada del algoritmo mientras que en la salida se devuelve el conjunto independiente maximal S de G. El grado $d(v_i)$ y soporte $s(v_i)$ de cada vértice $v_i \in V$ de G son calculados mediante las siguientes relaciones:

- $d(v_i) = \sum_j a_{ij}$
- $s(v_i) = \sum_{u \in N(v)} d(u)$

El algoritmo en cada paso elije el vértice v_i con máximo soporte y lo agrega al conjunto de cubrimiento V_C. En caso de que haya empate en cuanto al número máximo de soporte se elije aquel vértice que tenga mayor grado de los dos y se agrega al conjunto de cubrimiento V_C. Luego se actualiza la matriz de adyacencia de G poniendo un cero en las entradas correspondientes al vértice v que fue agregado al conjunto de cubrimiento V_C. El algoritmo continúa mientras G tenga aristas, es decir culmina cuando $a_{ij} = 0 \ \forall \ i, j$. Finalmente el conjunto independiente maximal S de G es obtenido a partir del conjunto de cubrimiento minimal V_C de G por $S = V - V_C$

Para el pseudocódigo del algoritmo VSA fueron utilizadas las definiciones de vecindario abierto, vecindario cerrado, grado y soporte de un vértice descriptas en la sección 2.3 de este mismo trabajo.

VSA(G)

In $G(V, E)$: (Grafo de entrada)
Out S: Conjunto independiente maximal de G, $S(G) = V - V_C$, donde V_C es el conjunto de cubrimiento minimal de G

1: $V_C = \emptyset$; selection = 0;
2: $\forall i \in V$
3: **do**
4: $d(v_i) = \sum_j a_{ij}$; $s(v_i) = \sum_{v_j \in N(v_i)} d(v_j)$;
5: Max = $s(v_1)$, $k = 1$
6: **for** i ← 2 to n
7: **if** max < $s(v_i)$ **then**

8: $t = i$;
9: **else if** max = $s(v_i)$ and $d(v_i - k) <= d(v_i)$ **then**
10: $t = i$;
11: **else** max = $s(v_i)$ and $d(v_i - k) > d(v_i)$ then
12: $t = i - k$;
13: $k = k + 1$;
14: **end for**
15: $V_C = V_C \cup \{v_t\}$;
16: selection = selection + 1;
17: Asignar el valor 0 a la t^{ma} fila y t^{ma} columna de la matriz $A = (a_{ij})$;
18: **while** $A \neq (0)$
19: $V_C =$ selection;
20: **return** $S = V - V_C$

2.6.3 Algoritmo metaheurístico Adaptative Tabu Search (ATS)

Éste algoritmo está diseñado para buscar un conjunto independiente de un tamaño especificado k en un grafo G(V, E). En el caso de no proporcionar un tamaño objetivo, el algoritmo trata de encontrar el conjunto independiente de mayor tamaño posible a partir de la aplicación de otro método de búsqueda.

El espacio de búsqueda del ATS está compuesto por subconjuntos de V de tamaño k, formalmente definido como $\Omega = \{I \subset V: |I| = k\}$.

Se define la función $f(I)$, $I \in \Omega$ que cuenta el número de aristas de $G[I]$ (ver definición de subgrafo inducido en 2.5.3), o sea que $f(I) = |E[I]|$. Como corolario de lo anterior tenemos que si $f(I) = 0$, todos los vértices contenidos en I no son adyacentes entre sí lo que convierte a I en un conjunto independiente, en caso contrario I no es un conjunto independiente. Definido $f(I)$, podemos decir que el objetivo de ATS es encontrar un I_{Max} / $f(I_{Max}) = 0, |I_{Max}|$ es maximal.

2.6.3.1 Pseudocódigo del Algoritmo ATS

Dado $I \in \Omega$, una solución candidata, $d_I(v)$ el grado inducido de v en I (ver 2.5.3), definimos:

- $Lista_{Tabu}$ es una lista tabú que contiene los vértices prohibidos para realizar cualquier acción.
- $MaxInI = \max\{d_I(u), u \in I \; y \; u \notin Lista_{Tabu}\}$: devuelve el máximo grado inducido por I solamente considerando los vértices de I.
- $MinOutI = \min\{d_I(v), v \in V - I \; y \; v \notin Lista_{Tabu}\}$: devuelve el mínimo grado inducido por I solamente considerando los vértices que no estén en I.

- $A = \{u \in I, u \notin Lista_{Tabu}\ y\ d_I(u) = MaxInI\}$: conjunto que contiene los vértices de I cuyo grado inducido es $MaxInI$.
- $B = \{v \in V - I, v \notin Lista_{Tabu}\ y\ d_I(v) = MinOutI\}$: conjunto que contiene vértices fuera de I cuyo grado inducido es $MinOutI$.
- $T = \{(u, v), u \in A, v \in B, (u, v) \in E\}$: conjunto de aristas entre vértices de A y B.
- $T_A = f(I) + Random(4)$: define la tenencia tabú de los vértices de A que se ingresan en $Lista_{Tabu}$, donde $1 \leq Random(4) \leq 4$.
- $T_B = 0.6 * T_A$: define la tenencia tabú de los vértices de B que se ingresan en $Lista_{Tabu}$.

Utilizando las definiciones anteriores, podemos establecer una restricción de vecindad que permita realizar una búsqueda eficiente de nuevas soluciones en el espacio de búsqueda. Para generar una nueva solución I' a partir de I definimos el siguiente movimiento:

$$I' = I - \{u\} \cup v, u \in A, v \in B.$$

A partir de este movimiento podemos establecer que $f(I') = f(I) - d_I(u) + d_I(v) - e_{u,v}$, donde $e_{u,v} = 1$ si $(u, v) \in E$ y 0 en caso contrario. Entonces para minimizar el cálculo anterior, va a ser conveniente escoger vértices de A y B que tengan aristas en T.

En definitiva para escoger un par de vértices u y v a intercambiar, se define la siguiente estrategia de búsqueda de un nuevo vértice candidato:

- El primer paso será buscar cualquier par de vértices en T. En el caso de que T sea vacío, se escogerán aleatoriamente un vértice de $u \in A$ y otro de $v \in B$.
- Una vez elegidos los vértices, se procede a realizar el movimiento para generar I y se ingresan u y v en $Lista_{Tabu}$ por $T_A\ y\ T_B$ iteraciones respectivamente (se establecen las tenencias de ambos vértices).

A continuación presentamos el pseudocódigo del algoritmo ATS utilizando los conceptos definidos previamente.

ATS(G)

In $G(V, E)$: (Grafo de entrada)
In k: Tamaño objetivo del conjunto independiente
In $iter_{Max}$: máxima cantidad de intentos para encontrar un conjunto independiente mejor
Out I_{Max}: Conjunto independiente de tamaño k o maximal de G
Var I_{Best}: mejor conjunto independiente encontrado hasta el momento.
Var I: conjunto independiente encontrado en cada iteración
Var z_{Best}: tamaño de I_{Best}.

1: $I_{Max} = \emptyset$

```
2: cinm = 0                                   // Cuenta las veces que no mejora un conjunto independiente
3: if k no está especificado then
4:     I_Best = MCISMR(G)                     // Calculo un conjunto independiente inicial
5:     z_Best = |I_Best|
6:     do
7:         I = ATS(G,z_Best + 1,iter_Max)
8:         cinm = cinm + 1
9:         if |I| > z_Best then               // Si encuentro un conjunto independiente de mayor tamaño
10:            I_Best = I
11:            z_Best = |I_Best|
12:            cinm = 0
13:        endif
14:    while cinm < 2                         // Se busca como máximo 2 veces un conjunto mejor
15:    I_Max = I_Best
16: else
17:    I_Best = Generar_Conjunto_Aleatorio(k) // Genera un conjunto aleatorio de tamaño k
18:    I = I_Best
19:    while cinm < iter_Max and f(I) > 0
20:        Buscar_Mejor_Movimiento(u,v)       // u ∈ A y v ∈ B
21:        I = I – {u} ∪ {v}                  // Realizo movimiento para buscar un nuevo I
22:        Lista_Tabu.Insertar(u,T_A)
23:        Lista_Tabu.Insertar(u,T_B)
24:        cinm = cinm + 1
25:        if f(I) < f(I_Best) then
26:            I_Best = I
27:            cinm = 0
28:        endif
29:    endwhile
30:    if f(I_Best) = 0 then
31:        I_Max = I_Best
32:    endif
33: endif
34: return I_Max
```

El algoritmo básicamente tiene 2 comportamientos diferentes: uno cuando no se indica un *k* objetivo (a partir de la línea 4) y otro cuando si es especificado (a partir de la línea 17). En el primer caso, no se tiene un tamaño de conjunto como objetivo, por lo tanto se busca un conjunto independiente inicial (línea 4) aplicando el algoritmo heurístico MCISMR (ver 2.6.1). Luego se intenta encontrar sistemáticamente un conjunto independiente mejor aplicando ATS pasando como tamaño objetivo el tamaño del conjunto actual +1 (línea 7). Si se encuentra un conjunto de mayor tamaño, se actualiza el conjunto de referencia actual. En caso contrario, se incrementa la cantidad de intentos para buscar nuevos resultados (líneas 8 a 14).

Cuando existe un *k* de referencia, se crea un conjunto inicial de forma aleatoria de tamaño *k* que no es necesariamente independiente (línea 17). Mientras el conjunto considerado no sea independiente y no se acaben los intentos de encontrar este conjunto, en

cada iteración se busca un movimiento válido siguiendo la estrategia de búsqueda de un nuevo vértice candidato (como se especificó anteriormente), se realiza el intercambio de los vértices para generar una nueva posible solución y se marcan los vértices intercambiados en lista tabú (líneas 20 a 23). Si la nueva solución tiene menor número de aristas, se actualiza la mejor solución encontrada hasta el momento y se reinicia el contador de intentos (líneas 25 a 28). Finalmente, si el algoritmo encuentra un conjunto independiente válido, lo devuelve en I_{Max}; en caso contrario, el conjunto se devuelve vacío.

2.7 Análisis comparativo de algoritmos de coloreo

En esta sección explicaremos las diferentes decisiones que tomamos para el diseño de nuestros algoritmos, su parametrización y la elección de algunos de ellos. Como criterios generales, tomamos en cuenta el tiempo de ejecución y la calidad de la solución obtenida de cada algoritmo. También haremos mención sobre los resultados obtenidos por el grupo del módulo de taller sobre algunas de sus instancias de prueba [43].

2.7.1 Infraestructura utilizada (hardware y software)

Para la implementación de los algoritmos y la ejecución de pruebas se utilizó un PC de doble núcleo, con una frecuencia máxima de 2.8 GHz y arquitectura de 64 bits, con 4GB de memoria RAM, sistema operativo Windows 7 de 64 bits, Visual Studio 2010 como entorno de desarrollo y Visual C++ como lenguaje de programación.

2.7.2 Instancias de prueba

Antes de comenzar con el análisis, haremos una breve introducción sobre los diferentes tipos de instancias de prueba que fueron utilizadas. Para los experimentos se consideraron 2 subconjuntos de instancias, el primero corresponde a un subconjunto de a instancias estándar para la prueba de algoritmos (DIMACS (ver [44])) y el segundo son algunas de las instancias generadas por el grupo del módulo de taller. A continuación daremos una descripción de cada una de ellas:

- **Grafos aleatorios:** Son grafos generados de forma aleatoria, utilizados en [45] para los cuales no se conoce su número cromático. Los archivos correspondientes a estos tipos de grafos respetan el formato de nombre "**DSJCn.d.col**", donde n representa el número de vértices, d indica la densidad de aristas del grafo y col es la extensión (pe: DSCJ1000.1.col, corresponde a un grafo de 1000 vértices y densidad 10%).

- **Grafos planos:** Son grafos estructurados utilizados por Joseph Culberson en la resolución del problema de coloreo Quasi aleatorio (ver [46]), cuyo número cromático es conocido. Los archivos correspondientes a estos tipos de grafos respetan el formato de nombre "**flatn_k_0.col**", donde n representa el número de vértices, k el número cromático y col es la extensión (pe: flat1000_50_0.col, corresponde a un grafo de 1000 vértices y número cromático 50).

- **Grafos aleatorios geométricos:** Son grafos creados mediante la selección de vértices randómicos dentro de un plano, creando aristas entre vértices cuya separación se menor o igual a una distancia geométrica. El número cromático es conocido para algunas instancias. Los archivos correspondientes a estos tipos de grafos respetan el formato de nombre "**Rn.d[c].col**", donde n representa el número de vértices, d la densidad de aristas del grafo, [c] es opcional e indica que es un grafo complementado a partir de otra de las instancias proporcionadas y col es la extensión (pe: R1000.1.col, corresponde a un grafo de 1000 vértices y densidad 10%, R1000.1c, es el grafo complemento del R1000.1).

- **Grafos aleatorios grandes:** Son grafos con las mismas características que los grafos aleatorios, con la particularidad de que no son muy utilizados en experimentos computacionales debido a su gran tamaño. Los archivos correspondientes a estos tipos de grafos respetan el formato de nombre "**Cn.d.col**", donde n representa el número de vértices, d indica la densidad de aristas del grafo y col es la extensión (pe: C4000.5.col, corresponde a un grafo de 4000 vértices y densidad 50%).

- **Grafos grupo módulo de taller:** Son grafos generados de forma aleatoria de los cuales no se conoce su número cromático. Los archivos correspondientes a estos tipos de grafos respetan el formato de nombre "**instancia_prueba_an._ut_pd.txt**", donde n representa el número de vértices, t indica el umbral de tolerancia (utilizado en la medición de la eficiencia de una unidad, concepto que veremos en el capítulo siguiente), d indica la densidad de aristas del grafo y txt es la extensión (pe: instancia_prueba_a100._u15_p05.txt, corresponde a un grafo de 100 vértices, umbral de tolerancia de un 15% y densidad 50%).

El formato de archivo del primer subconjunto de instancias (DIMACS) cumple con la siguiente estructura:

- **Región de comentarios:** Esta sección **puede** estar **solamente** al principio del archivo y puede contener cero o más líneas de comentario. Una línea de comentario debe comenzar con el carácter **"c"** y un **"espacio en blanco"** seguido de cualquier cadena de caracteres (por ejemplo: c FILE: DSJC125.1). La información contenida en esta sección del archivo se utiliza **sólo** con fines informativos.

- **Información general del grafo:** Esta sección contiene los parámetros generales del grafo y se especifica con una línea a continuación de la región de comentarios. Esta línea de parámetros debe comenzar con la cadena **"p edge"** seguido del **número de vértices** y el **número de aristas** (por ejemplo: p edge 125 1472).

- **Aristas:** En esta sección del archivo se determinan las aristas que contiene el grafo. Como los grafos contienen aristas sin dirección, es indiferente cuál de ellas se debe especificar. De igual forma como criterio de especificación se indican las aristas que parten del nodo con mayor identificador numérico hacia los de menor identificador. Existe una línea por cada arista cuyo formato es el caracter **"e"** seguido por el id del primer vértice y el id del segundo vértice (por ejemplo: e 5 1). Un pequeño detalle a tener

en cuenta es que la cantidad de aristas especificadas en la sección anterior es exactamente el doble de las detalladas aquí debido a que se especifican todas las aristas necesarias para configurar el grafo no dirigido.

Por otro lado, el formato de los archivos del Grupo de Taller respeta la siguiente estructura:

- **Umbral de tolerancia** (por ejemplo: 0.15).
- **Cantidad de vértices del grafo** (por ejemplo: 200).
- **Matriz de compatibilidad:** Por cada vértice (o actividad) se genera una línea de ceros (0 = no arista) y unos (1 = arista) para determinar las aristas que contiene con otros vértices del grafo (por ejemplo. 0 1 0 0 1……).
- **Tasas y tolerancias:** Por cada vértice se genera una línea que contiene el id del vértice o actividad, la tasa de arribo, la tasa de servicio y la tolerancia del cliente (por ejemplo: 0 1.554033 3.130481 1.281902). Esta información no es considerada para la etapa de coloreo, aunque si es útil para la simulación, de igual manera la misma no es obtenida de aquí, sino de un archivo de configuración utilizado por el simulador (como puede verse en el Apéndice D).

Para este proyecto se utilizó la siguiente lista de instancias, discriminadas por etapa de experimento y agrupadas en las categorías descriptas al principio de la sección:

- **Etapa Diseño y Calibración:** Para tomar decisiones de diseño como por ejemplo la selección de estructuras de datos, o algoritmos embebidos y para encontrar la configuración óptima de los algoritmos metaheurísticos se ejecutaron 3 veces instancias de tamaño <= 1000 vértices.
  - DSJC125.1.col, DSJC250.5.col, DSJC500.5.col, DSJC500.9.col, DSJC1000.1.col, DSJC1000.5.col.
  - flat300_26_0.col.
  - R125.1c.col, R1000.5.col.
  - instancia_prueba_a200_u015_p025.txt, instancia_prueba_a200_u015_p075.txt.
- **Etapa Pruebas de Coloreo:** Para realizar los experimentos se ejecutaron 3 veces instancias de tamaño <= 4000 vértices.
  - DSJC125.1.col, DSJC125.9.col, DSJC250.5.col, DSJC500.9.col, DSJC1000.1.col, DSJC1000.5.col, DSJC1000.9.col.
  - flat300_26_0.col, flat1000_50_0.col, flat1000_76_0.col.
  - R125.1c.col, R250.5.col, R1000.5.col.

- C2000.5.col, C2000.9.col, C4000.5.col.
- instancia_prueba_a200_u015_p025.txt, instancia_prueba_a200_u015_p075.txt.

2.7.3 Estructuras de datos utilizadas

Se implementaron 2 tipos de grafos, uno basado en matriz de adyacencia y otro en listas de adyacencia. Utilizando estas construcciones de software, criterio de selección para escoger la mejor estructura se comparó la eficiencia en términos de tiempo de ejecución para cada una de ellas a partir de la ejecución del mismo conjunto de algoritmos.

2.7.3.1 Matriz de Adyacencia Binaria

Como en nuestra realidad no utilizamos grafos ponderados, para las estructuras basadas en matriz de adyacencia, basta para representar la existencia de una arista entre un par de vértices el valor 1 y con 0 el caso contrario. A partir de lo anterior, como primera alternativa, se decidió realizar una implementación de grafo utilizando una matriz de adyacencia binaria, con la particularidad de que esta matriz internamente está representada por un vector de bits de tamaño filas x columnas, donde cada bit representa un elemento de la matriz y si un bit es positivo determina la existencia de una arista entre un par de vértices. Por ser una matriz de adyacencia, es cuadrada de tamaño $|V| * |V|$, además como trabajamos con grafos no dirigidos y no existen lazos hacia un mismo vértice, estas matrices son simétricas con su diagonal en 0.

Gráficamente la implementación de esta estructura para una matriz genérica (sin considerar propiedades particulares) se podría representar como:

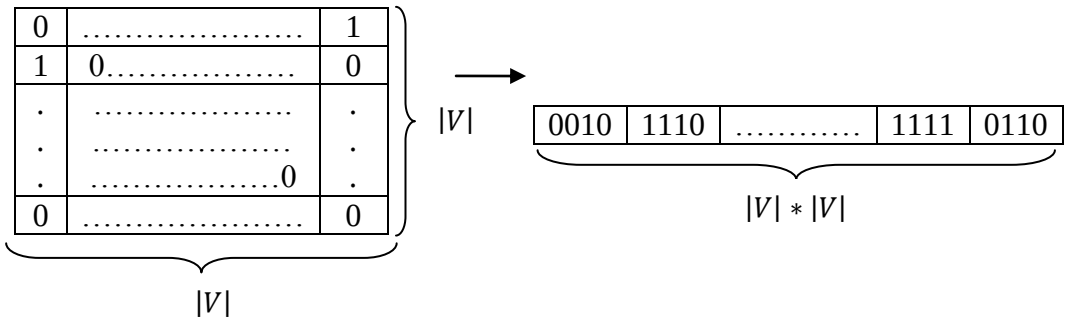

Figura 2 – Representación de estructura para una matriz genérica

Donde el gráfico de la izquierda de la Figura 2 muestra la idea conceptual, y el de la derecha muestra efectivamente la estructura implementada.

Adicionalmente con el objetivo de optimizar la obtención del grado de un vértice, se almacena la suma de elementos positivos para cada fila y cada columna. De esta forma se obtiene esta propiedad con un costo $O(1)$.

2.7.3.2 Listas de adyacencia

El conjunto de vértices es almacenado en una lista doblemente encadenada donde cada elemento tiene un puntero a una lista de adyacencia. Cada lista de adyacencia, es una lista doblemente encadenada que contiene los vértices adyacentes a un nodo del grafo v, donde cada elemento de esta lista tiene básicamente un puntero al vértice adyacente w de v y un puntero a la posición que ocupa v en la lista de adyacentes de w. Los dos punteros de la lista de adyacencia permiten la eliminación de aristas en tiempo contante. Para ambas listas se utilizó una implementación basadas en arreglos de forma de poder minimizar los fallos de cache (cache – misses [47]) ya que los arreglos son almacenados en lugares contiguos de memoria.

Gráficamente la implementación de esta estructura se podría representar como:

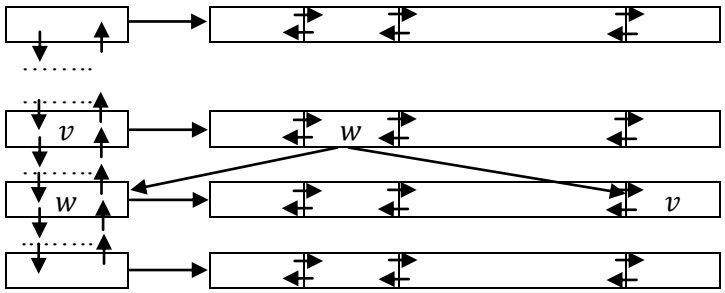

Figura 3 – Listas de Adyacencia

2.7.3.3 Análisis de estructuras

Una bondad de las matrices binarias es que permite almacenar un grafo de forma comprimida, reduciendo el consumo de memoria. Como el tipo de dato más pequeño representable en C++ es el byte, el tamaño aproximado para grafo de n vértices está dado por:

$$\left\lceil \frac{\sum_{i=1}^{n} i}{8} \right\rceil + (4 * \text{n}) \text{ bytes} = \left\lceil \frac{\frac{n * (n+1)}{2}}{8} \right\rceil + (4 * \text{n}) \text{ bytes} = \left\lceil \frac{n^2 + n}{16} \right\rceil + (4 * \text{n}) \text{ bytes}$$

Dado que para una matriz simétrica se cumple que $m_{i.j} = m_{j.i}, \forall i, j$, no es necesario representar la totalidad de los elementos en memoria, es suficiente con almacenar los valores de la diagonal junto con los elementos ubicados por encima o por debajo de ella. Tomando en cuenta el caso del almacenamiento de los elementos por debajo de la diagonal, se cumple que la fila i contiene i elementos. Si generalizamos lo anterior, para una matriz de n filas necesitamos almacenar $\sum_{i=1}^{n} i = \frac{n*(n+1)}{2}$ elementos. Como la representación más pequeña para

un tipo de datos en C++ ocupa un byte (8 bits), y cada elemento de la matriz se puede representar con un bit, podemos almacenar hasta un máximo de 8 elementos. Considerando un vector de bytes como estructura de almacenamiento, el tamaño necesario para guardar en una matriz simétrica de tamaño n es $\left\lceil \frac{elementos}{cantidad\ de\ elementos\ x\ indice\ de\ vector} \right\rceil = \left\lceil \frac{\frac{n*(n+1)}{2}}{8} \right\rceil$ espacios de memoria ya que la división puede no ser exacta y la granularidad en la representación no permite ajustar este valor (el desperdicio de memoria es, a lo sumo, de 7 bits si se utiliza un único bit del último elemento del vector para representar el último elemento de la matriz). Como el grafo considerado tiene n vértices y para almacenar cada grado utilizamos un entero de 32 bits (4 bytes), se necesitan $4 * n$ bytes para implementar la optimización sobre la obtención del grado.

Otra ventaja que demostró esta estructura fue la velocidad superior de carga de las instancias de grafos en memoria. Esto se debe a que su tamaño es reducido, la asignación total de memoria se realiza al inicio del proceso de carga y las operaciones de inserción aristas involucran un pequeño conjunto de operaciones matemáticas.

Por último la operación de complemento de la matriz se lleva a cabo simplemente negando todos bits almacenados en cada elemento del vector, no es necesario buscar el conjunto de vértices no adyacentes a un nodo del grafo para representar las aristas del nodo en el grafo complemento.

A pesar de las ventajas mencionadas, las matrices binarias no son la mejor estructura para representar grafos con pocas aristas, debido a que se almacenan muchos 0 para indicar que no existe una arista, cuando en realidad lo relevante es saber solamente cuando existe. Además las operaciones que involucran vértices adyacentes son ineficientes, debido a que para obtener el conjunto de vértices adyacentes a un vértice dado se requiere de una recorrida completa de una fila de la matriz $(O(n))$.

Para el caso de las listas de adyacencia, si bien no economizan en espacio, mejoran considerablemente las operaciones relacionadas con vértices adyacentes, ya que esta estructura registra exclusivamente las relaciones existentes en un grafo. De esta forma la obtención del conjunto de vértices adyacentes es directa, por lo cual se eliminan muchas recorridas innecesarias. Además la eliminación de vértices y relaciones se realiza mediante la manipulación directa de punteros (eliminación perezosa), a diferencia de las matrices binarias donde es necesario realizar una serie de cálculos para poder eliminar una simple arista y no es posible eliminar un vértice sin tener que modificar toda la estructura. Debido a que todos los algoritmos hacen un uso exhaustivo de operaciones que involucran vértices adyacentes, con esta estructura se mejora considerablemente los tiempos de ejecución de los mismos.

A continuación se presentan los resultados obtenidos producto de la ejecución de los algoritmos de coloreo RLF y LRLF y de cálculo del máximo conjunto independiente MCISMR y VSA sobre diferentes tipos de instancias:

Instancia	Tiempo(s) RLF GB	Tiempo(s) RLF GC	Tiempo(s) LRLF GB	Tiempo(s) LRLF GC
instancia_prueba_a200_u015_p025	0,018	0,001	0,008	0,001
instancia_prueba_a200_u015_p075	0,083	0,005	0,025	0,001
DSJC125.1	0,118	0,001	0,001	0
DSJC250.5	0,198	0,003	0,028	0,002
flat300_26_0	0,35	0,005	0,041	0,003
DSJC500.9	0,768	0,229	0,432	0,018
R1000.5	7,173	0,716	2,578	0,06
DSJC1000.5	6,674	0,358	1,256	0,14

Tabla 1 – Resultados algoritmos RLF y LRLF

La tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 y 3 muestran el tiempo de ejecución del algoritmo RLF para el grafo basado en matriz binaria de adyacencia (de aquí en más lo llamaremos grafo binario o GB) y el grafo basado en listas de adyacencia (de aquí en más lo llamaremos grafo común o GC) y las columnas 4 y 5 muestran el tiempo de ejecución del algoritmo LRLF para el grafo binario y común.

Instancia	Tiempo(s) MCISMR GB	Tiempo(s) MCISMR GC	Tiempo(s) VSA GB	Tiempo(s) VSA GC
instancia_prueba_a200_u015_p025	0,002	**0**	0,046	**0,002**
instancia_prueba_a200_u015_p075	0,083	**0,005**	0,025	**0,001**
DSJC125.1	0	**0**	0,004	**0**
DSJC250.5	0,004	**0**	0,253	**0,009**
flat300_26_0	0,007	**0,001**	0,391	**0,014**
DSJC500.9	0,024	**0,003**	5,727	**0,319**
R1000.5	0,068	**0,011**	14,333	**0,013**
DSJC1000.5	0,073	**0,014**	15,691	**0,445**

Tabla 2 – Resultados MCISMR y VSA

La tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 y 3 muestran el tiempo de ejecución del algoritmo MCISMR para el grafo basado en matriz binaria de adyacencia (de aquí en más lo llamaremos grafo binario o GB) y el grafo basado en listas de adyacencia (de aquí en más lo llamaremos grafo común o GC) y las columnas 4 y 5 muestran el tiempo de ejecución del algoritmo VSA para el grafo binario y común.

Como se puede observar, los tiempos de ejecución de los algoritmos que utilizan grafos comunes son muy inferiores a los tiempos de los algoritmos que utilizan grafos

binarios. Como ya comentamos anteriormente, la razón principal es la ineficiencia que presentan los grafos binarios para realizar algunas operaciones que involucran vértices adyacentes.

A partir de estos resultados, decidimos utilizar los grafos comunes para la implementación de todos los algoritmos de coloreo y máximo conjunto independiente.

2.7.4 Parametrización y diseño EXTRACOL

Para diseñar el algoritmo EXTRACOL, la etapa de pre procesamiento requiere de la parametrización del algoritmo ATS, donde a su vez es necesario para el diseño de este último especificar un algoritmo de máximo conjunto independiente para el caso en que no está determinado el parámetro de tamaño de conjunto independiente objetivo y para su ejecución se necesita especificar la cantidad máxima de intentos infructuosos soportado. Para la etapa de coloreo del grafo residual resultante de la etapa anterior, es necesario incorporar un algoritmo de coloreo para encontrar una coloración factible de mínima cardinalidad. Además para la ejecución de este algoritmo es necesario especificar el tamaño del grafo residual (parámetro condicionado por la elección del algoritmo de coloreo de la segunda etapa), la cantidad máxima de iteraciones infructuosas sin encontrar nuevos conjuntos independientes y el tamaño máximo del conjunto de conjuntos independientes.

2.7.4.1 Parametrización y diseño ATS

Para implementar la búsqueda de un conjunto independiente de tamaño máximo para el caso en que no hay un tamaño objetivo especificado, fue necesario incorporar un algoritmo heurístico para encontrar una solución inicial que permitiera establecer un tamaño objetivo de partida para intentar mejorar. Para realizar esta selección, se consideraron los algoritmos MCIS en sus tres versiones y el algoritmo VSA, los cuales fueron aplicados 3 veces sobre una serie de instancias de diferentes tamaños. Como criterios de selección se consideró el tiempo de ejecución y la calidad de la solución obtenida. A continuación se presentan los mejores resultados obtenidos producto de la ejecución de los algoritmos mencionados sobre el grafo binario:

Instancia	k - MCIS	k - MCISM	k - MCISMR	k - VSA	Tiempo(s) MCIS	Tiempo(s) MCISM	Tiempo(s) MCISMR	Tiempo(s) VSA
R1000.5	6	**7**	**7**	5	**0,001**	0,069	0,068	14,333
DSJC250.5	7	9	**11**	8	**0,001**	0,005	0,004	0,253
DSJC1000.9	4	4	**5**	4	**0,001**	0,102	0,101	42,904

Tabla 3 – Mejores resultados grafo binario

La tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran el tamaño del conjunto independiente obtenido por cada algoritmo y las columnas 6 a 9 muestran el tiempo de ejecución del algoritmo necesario para la obtención de cada resultado.

En todos los casos, los mejores resultados fueron obtenidos por el algoritmo MCISMR, si bien el tiempo de ejecución más bajo fue para el MCIS. El peor desempeño considerando los 2 criterios fue para el VSA quién no superó a ninguno de los algoritmos, salvo para la ejecución de la instancia DSJC250.5, donde superó sólo al algoritmo MCIS, y alcanzó el tiempo de ejecución más elevado en todos los casos. Si bien **MCISMR** no obtiene los mejores tiempos de ejecución, tampoco ejecuta durante períodos de tiempo prolongados, por lo cual lo hace el candidato más promisorio para ser utilizado en el ATS.

Otro aspecto a considerar, es la especificación del parámetro cantidad máxima de iteraciones infructuosas. Para ello se probó ejecutar el algoritmo sobre un pequeño conjunto de instancias específicamente seleccionadas, cuya dificultad reside en encontrar conjuntos independientes de máximo tamaño sin determinar un tamaño objetivo y variando la cantidad de iteraciones entre 50, 100, 200 y 500. Los resultados obtenidos fueron:

Instancia	|S| - ATS(50)	|S| - ATS(100)	|S| - ATS(200)	|S| - ATS(500)
R125.1c	6	**7**	**7**	**7**
DSJC125.1	**34**	**34**	**34**	**34**
DSJC250.5	10	11	**12**	11
DSJC500.5	12	12	**13**	**13**
DSJC1000.1	62	63	62	**65**

Instancia	%Éxito ATS(50)	% Éxito ATS(100)	%Éxito ATS(200)	%Éxito ATS(500)
R125.1c	100	**66**	**66**	**66**
DSJC125.1	33	**100**	**100**	**100**
DSJC250.5	100	33	**33**	100
DSJC500.5	100	100	33	**66**
DSJC1000.1	66	33	100	**33**

Instancia	Tiempo(s) ATS(50)	Tiempo(s) ATS(100)	Tiempo(s) ATS(200)	Tiempo(s) ATS(500)
R125.1c	0	0	0,016	0,036
DSJC125.1	0,011	0,016	0,035	0,049
DSJC250.5	0,024	0,04	0,101	0,135
DSJC500.5	0,048	0,123	0,246	0,665
DSJC1000.1	0,232	0,625	0,473	1,769

Tabla 4 – Resultados parametrización ATS

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran el tamaño del conjunto independiente obtenido por cada parametrización diferente. La segunda tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el porcentaje de éxito para la obtención resultado de la primera tabla (para 3 ejecuciones consecutivas). La tercera tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el mejor tiempo de ejecución del algoritmo necesario para la obtención del resultado de la primera tabla.

A partir de estos resultados, se puede observar que para 200 y 500 el ATS obtiene los mejores resultados para 4 de las 5 instancias. Tomando en cuenta las 3 instancias en común (R125.1c, DSJC125.1, DSJC500.5) donde ambas configuraciones ejecutan mejor, la parametrización de 500 obtiene un porcentaje mayor de éxito que la de 200 para la instancia DSJC500.5, sin embargo, el tiempo de ejecución es más del doble, propiedad que se repite también para las instancias R125.1c y DSJC1000.1.

En conclusión, tomando en cuenta la calidad de la solución, el porcentaje de acierto y los tiempos de ejecución se decide utilizar **200** como tope de iteraciones infructuosas.

2.7.4.2 Decisiones de diseño EXTRACOL

Luego de llevarse a acabo la etapa de pre procesamiento, el siguiente paso del algoritmo consiste en aplicar un algoritmo de coloreo sobre un grafo reducido para encontrar una solución factible al problema. Para realizar esta selección, se consideraron los algoritmos CSR en sus tres versiones y los algoritmos RLF y LRLF, donde cada uno de ellos fue aplicado 3 veces sobre una serie de instancias de diferentes tamaños. Como criterios de selección se consideró el tiempo de ejecución y la calidad de la solución obtenida. A continuación se presentan los mejores resultados obtenidos producto de la ejecución de los algoritmos mencionados sobre el grafo común:

Instancia	k - RLF	k - LRLF	k - CSR_MCISMR	k - CSR_VSA	k - CSR_ATS
R125.1c	**46**	**46**	49	51	49
DSJC125.1	**6**	**6**	8	8	7
DSJC250.5	**34**	**34**	36	41	**34**
DSJC500.5	59	59	62	71	**56**
DSJC1000.1	**24**	**24**	28	32	25

Instancia	%Éxito - RLF	%Éxito - LRLF	%Éxito - CSR_MCISMR	%Éxito - CSR_VSA	%Éxito - CSR_ATS
R125.1c	**33**	**33**	33	100	66
DSJC125.1	**100**	**100**	66	66	33
DSJC250.5	66	**100**	66	33	66
DSJC500.5	66	100	66	33	33
DSJC1000.1	**100**	**100**	100	33	66

Instancia	Tiempo(s) - RLF	Tiempo(s) - LRLF	Tiempo(s) - CSR_MCISMR	Tiempo(s) - CSR_VSA	Tiempo(s) - CSR_ATS
R125.1c	0,003	**0**	0,024	0,051	0,333
DSJC125.1	**0**	**0**	0,001	0,001	0,078
DSJC250.5	0,004	**0,001**	0,047	0,13	1,015
DSJC500.5	0,041	**0,013**	0,38	1,533	4,808
DSJC1000.1	**0,014**	0,028	14	0,367	4,953

Tabla 5 – Mejores resultados grafo común

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran la cantidad de colores óptima obtenido por cada algoritmo diferente. La segunda tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el porcentaje de éxito para la obtención resultado de la primera tabla (para 3 ejecuciones consecutivas). La tercera tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el mejor tiempo de ejecución del algoritmo necesario para la obtención del resultado de la primera tabla.

Es trivial observar que el algoritmo LRLF es el que obtiene los mejores resultados, porcentajes de éxito y tiempos de ejecución en 4 de las 5 pruebas, si bien RLF, que es un algoritmo equivalente, tiene un porcentaje de acierto un poco inferior en 2 de los 5 casos. Esta característica difiere entre estos dos algoritmos debido a algunas decisiones aleatorias, que determinan casuísticas diferentes, como la selección de un nodo a procesar en un paso cuando se produce un empate en el criterio de selección (básicamente se tira una moneda). CSR_ATS logra un mejor valor óptimo para la instancia DSJC500.5, pero a una baja tasa de éxito (33%) y con un tiempo de ejecución elevado (4,808 s) en comparación con RLF (0,041 s) y LRLF (0,013 s). Debido la superioridad general en los resultados se decide utilizar **LRLF** para encontrar una coloración del grafo residual como paso final del algoritmo EXTRACOL.

2.7.4.3 Parametrización EXTRACOL

Además de calibrar el algoritmo ATS embebido, es necesario especificar el tamaño del grafo residual (q), la cantidad máxima de iteraciones infructuosas (maxIter) y el tamaño máximo del conjunto de conjuntos independientes (maxPool), cuyos elementos disjuntos son utilizados para generar el grafo residual de la siguiente iteración.

Para la configuración de éstos parámetros se realizaron 3 ejecuciones de prueba con cada instancia tomando q = 65% sobre la cantidad de vértices del grafo original (el autor recomienda utilizar un tope de 800 vértices para grafos de 1000 vértices o superiores [36]), variando maxIter y maxPool en los siguientes valores:

- maxIter = 5, maxPool = 20
- maxIter = 10, maxPool = 40

- maxIter = 10, maxPool = 80
- maxIter = 20, maxPool = 80

Para este conjunto de casuísticas los resultados obtenidos fueron los siguientes:

Instancia	k –Extracol (%65,5,20)	k –Extracol (%65,10,40)	k –Extracol (%65,10,80)	k –Extracol (%65,20,80)
R125.1c	49	**47**	**47**	48
DSJC125.1	**6**	**6**	**6**	**6**
DSJC250.5	**33**	**33**	**33**	**33**
DSJC500.5	57	57	**56**	58
DSJC1000.1	24	24	**23**	**23**

Instancia	%Exito-Extracol (%65,5,20)	%Exito-Extracol (%65,10,40)	%Exito-Extracol (%65,10,80)	%Exito-Extracol (%65,20,80)
R125.1c	66	66	33	66
DSJC125.1	100	100	100	100
DSJC250.5	66	100	33	33
DSJC500.5	33	66	33	66
DSJC1000.1	100	100	33	33

Instancia	Tiempo(s)-Extracol (%65,5,20)	Tiempo(s)-Extracol (%65,10,40)	Tiempo(s)-Extracol (%65,10,80)	Tiempo(s)-Extracol (%65,20,80)
R125.1c	10,34	3,972	5,007	6,04
DSJC125.1	0,064	0,08	0,079	0,111
DSJC250.5	0,965	2,273	1,517	2,829
DSJC500.5	9,2	4,118	9,639	9,853
DSJC1000.1	5,289	13,756	15,087	63,055

Tabla 6 – Resultados parametrización EXTRACOL

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran la cantidad de colores óptima obtenido por cada parametrización diferente. La segunda tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el porcentaje de éxito para la obtención resultado de la primera tabla (para 3 ejecuciones consecutivas). La tercera tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 5 muestran el mejor tiempo de ejecución del algoritmo necesario para la obtención del resultado de la primera tabla.

A partir de estos resultados, se puede observar que tomando maxIter = 10, maxPool = 80 se obtienen los mejores resultados para todos los casos, sin embargo el porcentaje de éxito para 4 de los 5 casos son de un 33% (1/3 de probabilidad de obtener el resultado). Por otro lado tomando maxIter = 10, maxPool = 40, 3 de los 5 casos son óptimos y los dos restantes son casi óptimos (diferencia de 1 con el resultado óptimo), además obtiene los mejores

porcentajes de éxito para todos los casos y los tiempos de ejecución en general son buenos. Para maxIter = 20, maxPool = 80 ocurre lo mismo que para el caso anterior, pero uno de los resultados inferiores tiene una diferencia de 2 con respecto al mejor resultado. Cuando maxIter = 5, maxPool = 20 sólo 2 de los 5 casos son óptimos.

En conclusión, tomando en cuenta la calidad de la solución, el porcentaje de acierto y los tiempos de ejecución se decide utilizar maxIter = 10 y maxPool = 40.

2.7.5 Selección de algoritmos de coloreo

Las pruebas de los algoritmos de coloreo fueron realizadas sobre un grupo extenso de instancias (ver 2.7.2), con el objetivo de visualizar cuáles son los algoritmos que mejores soluciones encuentran. Se ejecutaron 18 instancias un total de 3 veces cada una de ellas, a partir de lo cual se obtuvieron los siguientes resultados:

Instancia	k - RLF	k - LRLF	k - EXTRACOL_LRLF	k - CSR_MCISMR	k - CSR_VSA	k - CSR_ATS
instancia_prueba_a200_u015_p025	16	**15**	**15**	18	19	16
instancia_prueba_a200_u015_p075	**46**	**46**	**46**	51	57	49
R125.1c	**47**	**47**	48	48	51	48
R250.5	**70**	**70**	74	87	105	89
R1000.5	**249**	**249**	268	349	399	346
flat300_26_0	38	38	**37**	41	45	38
flat1000_50_0	105	104	97	104	122	**93**
flat1000_76_0	105	105	99	105	121	**96**
DSJC125.1	**6**	**6**	**6**	7	8	7
DSJC125.9	49	**48**	50	52	56	54
DSJC250.5	34	34	34	37	42	**33**
DSJC500.9	152	150	**148**	154	173	150
DSJC1000.1	**24**	**24**	**24**	27	31	25
DSJC1000.5	105	106	101	108	123	**96**
DSJC1000.9	280	274	265	280	315	**261**
C2000.5	194	193	175	188	221	**169**
C2000.9	510	511	469	501	568	**464**
C4000.5	355	354	307	335	395	**301**

Instancia	Tiempo(s) - RLF	Tiempo(s) - LRLF	Tiempo(s) EXTRACOL _LRLF	Tiempo(s) - CSR_MCISMR	Tiempo(s) - CSR_VSA	Tiempo(s) - CSR_ATS
instancia_prueba_a200_u015_p025	0,001	**0,001**	0,362	0,006	0,013	0,324
instancia_prueba_a200_u015_p075	0,005	**0,001**	0,816	0,052	0,166	0,986
R125.1c	0,003	**0**	3,997	0,05	0,271	0,354
R250.5	0,006	**0,001**	3,938	0,086	0,272	1,491
R1000.5	0,716	**0,06**	56,668	7,06	67,882	52,388
flat300_26_0	0,005	**0,003**	1,582	0,068	0,226	1,444
flat1000_50_0	0,337	0,145	116,961	1,451	22,835	28,246
flat1000_76_0	0,348	0,042	193,996	2,567	24,265	27,849
DSJC125.1	0,001	**0**	0,095	0,001	0,002	0,055
DSJC125.9	0,002	**0**	0,295	0,02	0,055	0,477
DSJC250.5	0,003	**0,002**	1,242	0,044	0,12	1,054
DSJC500.9	0,229	**0,018**	15,378	1,507	15,305	11,171
DSJC1000.1	0,014	0,031	8,603	0,17	0,369	4,445
DSJC1000.5	0,358	0,14	116,266	1,512	32,639	30,52
DSJC1000.9	1,675	0,119	76,36	13,113	202,135	75,07
C2000.5	3,274	1,318	480,419	21,667	376,988	281,865
C2000.9	13,79	0,828	1282,17	106,87	3160,6	656,963
C4000.5	24,666	9,39	2395,2	207,454	6200,73	1740

Tabla 7 – Resultados selección de algoritmos

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 7 muestran la cantidad de colores óptima obtenido por cada parametrización diferente. La segunda tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a 7 muestran el mejor tiempo de ejecución del algoritmo necesario para la obtención del resultado de la primera tabla. Para ambas tablas se muestra en rojo los mejores resultados y la casilla en amarillo muestra el siguiente mejor resultado.

Para grafos cuya cantidad de vértices es inferior a 1000, el algoritmo LRLF obtiene los mejores resultados para todas las instancias salvo para flat300_26_0, DSJC250.5 y DSJC500.9 (esto da un total de 6 de 9 instancias), donde obtiene un valores subóptimo muy cercanos a los mejores resultados (con diferencias entre 1 y 2 colores), donde el tiempo de ejecución para todos ellos es el menor.

Los algoritmos CSR en sus versiones MCISMR y VSA, no superaron a ninguno de los otros algoritmos, en cuanto a la calidad de la coloración, siendo el VSA el peor de todos incluyendo los tiempos de ejecución.

Para instancias de 1000 vértices o superiores, se destacaron los algoritmos EXTRACOL (con LRLF para coloreo del grafo residual) y CSR_ATS, no pos sus tiempos de

ejecución, ya que el principio de búsqueda de buenas soluciones de estos algoritmos está ligado fuertemente al hecho de aumentar la cantidad de iteraciones (entre otros parámetros) y en consecuencia el tiempo final de ejecución, sino diferencia entre las soluciones encontradas por los restantes algoritmos. Sin embargo fue una sorpresa ver que el algoritmo CSR_ATS obtuvo los mejores resultados para 7 de las 9 instancias, si bien EXTRACOL obtiene para todas ellas resultados sub-óptimos muy similares (la diferencia más grande es de 6 colores). Con respecto a los grafos geométricos, ninguno obtiene resultados óptimos siendo el CSR_ATS el peor de los 2, en particular sobre la instancia R1000.5 (donde LRLF obtiene un resultado óptimo), de hecho todos los algoritmos CSR muestran una mala performance, por lo cual se puede deducir que la estrategia de extraer de a un conjunto independiente sobe un grafo residual no es buena. Que EXTRACOL no funcione bien sobre grafos geométricos no es una novedad ya que el autor de advierte de la mala performance sobre este tipo de grafos en [36]. Desde un punto de vista un poco más general, tomando en cuenta la totalidad de las pruebas y los algoritmos que producen mayor cantidad de buenos resultados, vemos que CSR_ATS obtiene 8 coloraciones óptimas y 5 sub-óptimas , LRLF obtiene 8 óptimas y 3 sub-óptimas y EXTRACOL obtiene 6 resultados óptimos y 11 sub-óptimos, donde para éste último los mejores resultados se producen sobre algunas instancias pequeñas, una mediana (DSJC500.9), una grande (DSJC1000.1), y los resultados sub-óptimos se producen para instancias de todos los tamaños. Del análisis anterior, podemos concluir que EXTRACOL es el algoritmo que mejor se comporta en la generalidad de los casos ya que obtiene buenos resultados para la mayoría de las pruebas.

Debido a la inferioridad en los resultados obtenidos por EXTRACOL para grandes instancias (DSJC1000.5, DSJC1000.9, C2000.5, C2000.9, C4000.5), con respecto al algoritmo CSR_ATS, decidimos crear otra versión donde el grafo residual es coloreado utilizando éste último. El objetivo de esta prueba consiste en visualizar si la combinación de la etapa de pre procesamiento sofisticado de EXTRACOL con un algoritmo de coloreo de mejor rendimiento para grandes instancias (CSR_ATS), mejoran los valores finales del EXTRACOL (sin considerar tiempo de ejecución). Los resultados obtenidos producto de esta prueba son:

Instancia	k - EXTRACOL_LRLF	k - EXTRACOL_CSR_ATS	k - CSR_ATS
R1000.5	**249**	342	346
flat1000_50_0	105	**93**	**93**
flat1000_76_0	105	**94**	96
DSJC1000.1	24	**24**	25
DSJC1000.5	105	97	**96**
DSJC1000.9	280	**259**	261
C2000.5	194	**168**	169
C2000.9	510	**456**	464
C4000.5	355	**298**	301

Tabla 8 – Resultados EXTRACOL_CSR_ATS

La tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 4 muestran la cantidad de colores óptima obtenido por cada algoritmo.

Como se esperaba, aplicar un algoritmo que encuentra mejores coloraciones sobre el grafo residual, mejora la solución global del algoritmo. EXTRACOL combinado con CSR_ATS mejora casi todos los resultados donde CSR_ATS superaba en el experimento principal a EXTRACOL combinado con LRLF. Además el único caso donde no genera el mejor resultado (DSJC1000.5), la diferencia con la mejor coloración es solamente de un color.

Como conclusión final podemos decir que para encontrar coloraciones óptimas utilizaremos LRLF para instancias con menos de 1000 vértices, con la excepción de los grafos aleatorios geométricos (instancias R125.1.c, R250.5 y R1000.5), donde el LRLF encuentra buenas soluciones, y EXTRACOL en su versión EXTRACOL_CSR_ATS para procesar instancias de tamaño superior. Los restantes algoritmos, que producen resultados de calidad inferior, se utilizarán para probar si peores agrupaciones de actividades en unidades, producen peores asignaciones de personal.

2.7.6 Resultados grupo módulo de taller

A raíz de los resultados obtenidos por el grupo de taller, mostramos a continuación una tabla comparativa que muestra los mejores resultados generados por los algoritmos del grupo (ver [43]) y nuestros resultados generados a partir de la aplicación de nuestro algoritmo aplicando LRLF 3 veces sobre las 2 instancias de pruebas más grandes (complementadas) consideradas por el grupo de taller (instancia_prueba_a200_u015_p025 y instancia_prueba_a200_u015_p075). A continuación se muestran los resultados obtenidos:

Instancia	k - LRLF	Estable - 2	Estable - 3
instancia_prueba_a200_u015_p025	**47**	56	56
instancia_prueba_a200_u015_p075	**16**	18	18

Instancia	Tiempo(s) - LRLF	Tiempo(s) - Estable - 2	Tiempo(s) - Estable - 3
instancia_prueba_a200_u015_p025	0,001	0,03	0,03
instancia_prueba_a200_u015_p075	0	0,08	0,24

Tabla 9 – Resultados comparativos de coloreo Grupo Taller

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 4 muestran la cantidad de colores óptima obtenido por cada algoritmo. La segunda tabla muestra el nombre de las instancias utilizada en la columna 1, las columnas 2 a

4 muestran el mejor tiempo de ejecución de cada algoritmo necesario para la obtención del resultado de la primera tabla.

A modo de comentario, vemos que nuestros resultados son mejores tanto en calidad de la solución como en tiempos de ejecución. En cuanto a los tiempos de ejecución un factor que puede influir en el desempeño, es el lenguaje de programación utilizado (C++ vs. Python), adicionalmente a la propia codificación del algoritmo. Según el trabajo del grupo, el número cromático calculado por una aplicación descargada de internet dio como resultado 52 colores para instancia_prueba_a200_u015_p025 (de aquí en más la llamaremos **a**) y 16 para instancia_prueba_a200_u015_p075 (de aquí en más la llamaremos **b**). Bajo el supuesto anterior vemos que en realidad el número cromático de **a** no es 52, sino que puede ser menor o igual a 47 y para **b** el algoritmo encontró la solución óptima.

Capítulo 3 – Eficiencia de unidades

3.1 Introducción

En este capítulo abordaremos el tema de la eficiencia de unidades a través de un sistema de múltiples clases con múltiples servidores (modelo $M/M/x$, donde x es el número de trabajadores (o servidores) por unidad). Se tienen n tipos o clases diferentes de clientes que arriban a la unidad siguiendo un proceso de Poisson con tasas $\lambda_1, \dots\dots\dots, \lambda_n$ y los servidores son idénticos con tasas de servicio $\mu_1, \dots\dots\dots, \mu_n$. Por otra parte aquellos clientes que no encuentran un servidor disponible se ubican en una fila de espera de capacidad infinita.

Dada las características del modelo, dividimos el capítulo en tres partes. Primero relevamos las principales características de las teorías de colas y algunas definiciones y términos que utilizaremos a largo del capítulo. Luego desarrollaremos la solución propuesta por Erlang para el caso particular en el que una unidad atiende sólo una actividad lo cual nos permitirá analizar el rendimiento del sistema. Finalmente haremos una breve reseña sobre algunos trabajos de investigación que hacen referencia al caso general antes mencionado de múltiples servidores que atienden múltiples tipos de clientes, y además daremos algunas definiciones teóricas importantes que sustentan los métodos de Montecarlo, enfoque que decidimos utilizar para modelar el funcionamiento de las diferentes unidades.

3.2 Teoría de colas

3.2.1 Introducción

El origen de la denominación de Teoría de Colas surge en 1909 de la mano del danés Agner Kraup Erlang, trabajador de la Copenhagen Telephone Exchange, quien analizó la congestión del tráfico telefónico a través del dimensionamiento de líneas y de centrales de conmutación telefónica, con la finalidad de cumplir con la demanda variable de servicios en el sistema telefónico de Copenhague. Como consecuencia de sus investigaciones surge una nueva teoría denominada teoría de colas o de líneas de espera [48].

Los modelos de líneas de espera son de relevante importancia dentro de la simulación y modelado de sistemas. Esto se debe fundamentalmente a que pueden ser utilizados para entender aspectos estocásticos tanto para redes de comunicación y cómputo, como para simular el comportamiento de servidores de diversos sistemas tales como:

- Sistemas bancarios
- Sistemas de producción
- Simulación de vuelos
- Simulación de oficinas de atención al público

Habitualmente, los profesionales que se enfrentan a este tipo de problemas son los ingenieros industriales y de sistemas; sin embargo, estos problemas afectan todos los sectores productivos, de manera que las líneas de espera tienen un gran potencial de aplicación [49].

Hoy en día la teoría de colas es considerada una herramienta de gran valor, ya que un gran número de problemas de la vida real pueden caracterizarse como problemas de congestión de llegada-salida. Sin embargo, la teoría de formación de una cola es a menudo demasiado restrictiva matemáticamente para ser capaz de modelar todo tipo de situaciones. A modo de ejemplo los modelos matemáticos frecuentemente asumen el número de clientes, o la capacidad de la cola infinitos, cuando en realidad no lo son. Por esta razón existen medios alternativos para el análisis de la teoría de colas que consisten en simulaciones por computadora (mediante la implementación de un modelo de simulación de eventos discretos) y/o el análisis de datos experimentales.

A pesar de las restricciones mencionadas, los modelos matemáticos presentan una serie de ventajas que los hacen muy atractivos [49]:

- Son más rápidos (en costo computacional) y fáciles de implementar, que los modelos de simulación.
- Pueden ser utilizados para validar sistemas de simulación de Montecarlo, a partir de la verificación de algunas condiciones consideradas en las líneas de espera.

Sin embargo, a menudo resulta difícil abordar la resolución de un problema aplicando el primer enfoque para el caso en que se debe partir del desarrollo de un nuevo modelo matemático, aspecto no menor a la hora de elegir qué método de resolución utilizar. Por otra parte, si bien la opción de simular tiene un gasto computacional más elevado, resulta más natural y sencillo de aplicar en ausencia de un modelo matemático, opción que decidimos utilizar para este trabajo.

Independientemente del método de resolución que se elija, los objetivos perseguidos por esta disciplina son:

- Identificar el nivel óptimo de capacidad del sistema que minimiza el costo del mismo.
- Evaluar el impacto en el costo total a partir de las posibles alternativas de modificación en la capacidad del sistema.
- Establecer un balance equilibrado (“óptimo”) entre las consideraciones cuantitativas de costos y las cualitativas de servicio.
- Tener en cuenta el tiempo de permanencia en el sistema o en la cola de espera, para poder medir la eficiencia del sistema y la calidad del servicio.

3.2.2 Definiciones y notación

La **teoría de colas** a través de un conjunto de modelos matemáticos describe los sistemas de líneas de espera y la causa de la formación de las mismas, las cuales se deben a la existencia de momentos en los que es mayor la demanda de un servicio que la capacidad del sistema para suministrarlo. El objetivo de la teoría de colas es encontrar el estado estable del sistema y determinar una capacidad de servicio apropiada.

Una **cola** es una línea de espera para un determinado servicio el cual es provisto por uno o varios servidores.

El **mecanismo de servicio** consiste en una o más instalaciones de servicio, cada una de ellas con uno o más canales paralelos de servicio, llamados servidores.

El modelo básico de los **sistemas de colas** está conformado por dos componentes, la cola propiamente dicha y el mecanismo de servicio. A continuación enumeramos las características más relevantes de los sistemas de colas:

- **Modelo de arribo de los clientes**: El índice de arribos será el número medio de arribos por unidad de tiempo.
- **Modelo de servicio:** Viene dado por el tiempo de servicio o por el número de clientes atendidos por unidad de tiempo.
- **Disciplina de la cola:** Establece el orden en que son atendidos los clientes. Puede ser de varios tipos:
  - **FIFO (First in first out):** Primero en llegar, primero en salir, es decir primero se atiende el cliente que haya llegado primero.
  - **LIFO (Last in first out):** Último en llegar, primero en salir, es decir primero se atiende el cliente que haya llegado de último.
  - **RSS (Random selection of service):** Selecciona los clientes de manera aleatoria, de acuerdo a algún procedimiento de prioridad o a algún otro orden.
  - **Processor Sharing:** Sirve a los clientes igualmente. La capacidad de la red se comparte entre los clientes y todos experimentan con eficacia el mismo retraso.
- **Capacidad de la cola:** Es el máximo número de clientes que pueden estar haciendo cola (antes de comenzar a ser servidos). La cola puede ser finita o infinita.
- **Número de canales de servicio:** Es el número de servidores que denotaremos x. Puede haber una cola para cada servidor o una sola cola global. Más adelante, en 3.2.4, mostraremos que siempre que se pueda es mejor utilizar una sola cola que varias de ellas.

El parámetro $\boldsymbol{\rho}$ es el factor de utilización del sistema (o intensidad de tráfico) y se cumple que $\rho = \frac{\lambda}{\mu}$. Las variables λ y μ son la definidas en 1.2.

Condición de no saturación: Si $\lambda \geq \mu$ el sistema se satura, y por lo tanto el número de clientes en la cola crece indefinidamente con el tiempo. La condición de no saturación establece que $\rho < x$. Cuando una cola no se satura, o sea cuando la distribución de clientes se conserva a través del tiempo, se dice que el sistema alcanza el estado estacionario.

3.2.3 Notación de Kendall

La notación de Kendall permite expresar de manera resumida las características de los sistemas de colas que describimos anteriormente.

Un sistema de colas se notará como A | B | X | Y | Z donde:

- A es la distribución de tiempos entre arribos. Los valores posibles son:
  - M (De tipo markoviano): La tasa de arribos sigue una distribución de Poisson y los tiempos entre arribos son exponenciales.
  - D: Tiempos entre arribos es determinista
  - Er: Distribución Erlang – r (Gamma)
  - Hr: Distribución Hiperexponencial - R
  - G: Tiempos entre llegadas generales (cualquier distribución)
- B es el modelo de servicio y puede tomar los mismos valores de A.
- X es el número de servidores
- Y es la capacidad del sistema (número máximo de clientes en el sistema), se puede omitir si es infinita.
- Z es la disciplina de la cola, se puede omitir si es FIFO [50].

3.2.4 Algunas propiedades de las líneas de espera

Para interiorizarnos en la problemática particular de nuestra tesis, profundizaremos en algunas propiedades que presentan las líneas de espera en las que nos basaremos de ahora en más para desarrollar la solución del problema de eficiencia de unidades.

Supongamos que tenemos una cola $M/M/1$ con parámetros λ y μ que se sustituye por n colas $M/M/1$ independientes de parámetros λ/n y μ/n, es decir que se divide la carga de trabajo y la capacidad de proceso en n partes iguales [50]. Mostraremos el efecto del cambio usando como medidas de rendimiento el tiempo medio de respuesta (o de permanencia) W y el número medio de clientes en el sistema L .

- **Opción 1**: Una sola cola $\lambda_1 = \lambda$ y $\mu_1 = \mu$:

$$L_1 = \frac{\rho_1}{1-\rho_1} = \frac{\lambda}{\mu - \lambda}$$

$$W_1 = \frac{1}{\mu_1 - \lambda_1} = \frac{1}{\mu - \lambda}$$

- **Opción 2:** n colas independientes $\lambda_n = \lambda/n$ y $\mu_n = \mu/n$

$$L_n = \sum_{i=1}^{n} \frac{\rho_n}{1-\rho_n} = n\frac{\rho_n}{1-\rho_n} = n\frac{\frac{\lambda_n}{\mu_n}}{1-\frac{\lambda_n}{\mu_n}} = n\frac{\frac{\lambda/n}{\mu/n}}{1-\frac{\lambda/n}{\mu/n}} = n\frac{\lambda/\mu}{1-\lambda/\mu} = n\frac{\lambda}{\mu-\lambda} = nL_1$$

$$W_n = \frac{1}{\mu_n - \lambda_n} = \frac{1}{\mu/n - \lambda/n} = n\frac{1}{\mu - \lambda} = nW_1$$

Como se observa en los cálculos, la opción 1 tiene valores menores para ambas medidas de rendimiento por lo que resulta la mejor de las opciones.

Esto nos muestra que es mejor no dividir la capacidad de procesamiento, es decir tener una sola cola en vez de varias de ellas.

Basándonos en lo mostrado anteriormente y aplicándolo a nuestro problema de asignación de actividades en unidades, podemos ver que de forma equivalente es mejor agrupar actividades que dividir la capacidad de procesamiento para atender actividades específicas, por lo que es más conveniente agrupar la mayor cantidad de actividades posibles en cada unidad.

3.2.5 Aplicaciones

La teoría de colas tiene una amplia variedad de aplicaciones en el mundo real. A continuación describimos algunas de ellas:

- **Planificación de tareas:** Existe una amplia variedad de estrategias utilizadas para la planificación de tareas computacionales; en este ejemplo describimos una estrategia sencilla que no utiliza priorización de tareas, se trata del algoritmo de procesamiento por lotes. En este algoritmo cada proceso obtiene el tiempo que necesita para ejecutar completamente, existe una única cola para los procesos en espera de ser procesados y los nuevos procesos son ubicados al final de la cola.

  Este modelo puede ser considerado como un caso $M/G/1$.

  El tiempo medio de respuesta del sistema para un proceso que toma t segundos para completar ($W(t)$), será el tiempo medio de espera en la cola, más el tiempo para ejecutar el proceso:

  $$W(t) = \frac{W_q}{1 - \rho} + t$$

  Donde W_q es el tiempo promedio en la cola y ρ es el factor de utilización del sistema.

  Para una cola $M/G/1$:

  $$W_q = \frac{\lambda \overline{x^2}}{2}$$

  El tiempo de espera es independiente del largo de la tarea, esto tiene la desventaja de que trabajos cortos no reciben prioridad por lo tanto los sistemas de procesamiento por lotes no resultan adecuados para implementar sistemas de tiempo compartido [51].

- **Colas en un banco:** Esta aplicación se puede ver en detalle en el Apéndice A.

Otros ejemplos de aplicaciones que utilizan teoría de colas son:

- En aeropuertos el diseño de la pista, recolección de equipaje, control de pasaportes, etc.
- Supermercados.
- Restaurantes.
- Planificación del transporte colectivo.
- Colas de impresión.
- Reducción de fallos de página.

3.3 Eficiencia para unidades con un sólo tipo de actividad – Fórmula de Erlang C

Analizaremos aquí el caso particular de un sistema en el que se tienen s servidores idénticos por unidad que atienden con tasas de servicio exponenciales $\mu > 0$, los arribos son de Poisson de tasa exponencial $\lambda > 0$ y los clientes que no encuentran un servidor disponible deben esperar en una cola con cantidad infinita de posiciones de espera. Además se asume accesibilidad completa, lo que significa que no puede haber clientes en la cola cuando los servidores están ociosos. En este caso asumimos unidades con una sola clase de tarea, es decir que todos los clientes demandan el mismo tipo de actividad.

Sea $X(t)$ una variable aleatoria de tiempo continuo que cuenta el número de clientes en el sistema en el instante t, en un período de tiempo infinitesimal $(t, t+h)$ solamente se puede producir una llegada nueva o una salida de un cliente del sistema, pero no ambos sucesos a la vez. Esto es un proceso de nacimiento y muerte que se puede representar mediante el siguiente diagrama de transición de estados:

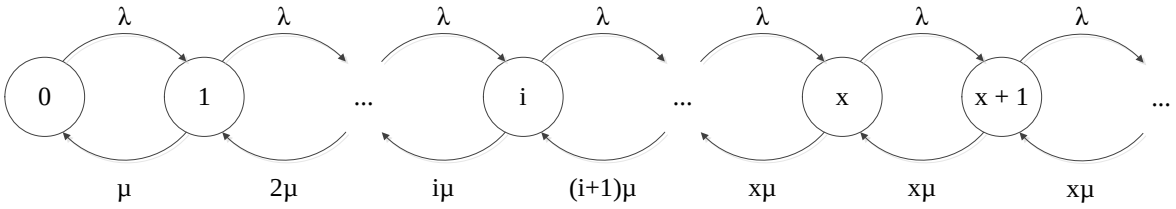

Figura 4 – Diagrama de transición de estados de un sistema de espera M/M/x con x servidores y un número ilimitado de posiciones de espera.

Para estudiar el sistema, asumiremos equilibrio estadístico, es decir $\lambda/\mu < x$, de esta manera nos aseguramos que la cola de espera no crezca indefinidamente.

Definimos $p(i) = P(X(t) = i)$ como la probabilidad de que el sistema esté en el estado i en el instante t.

A partir de la definición anterior y del diagrama de la Figura 4 se obtiene el siguiente sistema de ecuaciones de corte:

$$\lambda.p(0) = \mu.p(1),$$

$$\lambda.p(1) = 2\mu.p(2),$$

… …

$$\lambda.p(i) = (i+1)\mu.p(i+1),$$

… …

$$\lambda.p(x-1) = x\mu.p(x),$$

$$\lambda.p(x) = x\mu.p(x+1),$$

… …

$$\lambda.p(x+j) = x\mu.p(x+j+1)$$

Despejando $p(i)$ con $i \geq 1$ y sustituyendo $\frac{\lambda}{\mu} = \rho$ tenemos [52]:

$$p(i) = \begin{cases} p(i-1)\frac{\rho}{i}\ , & 1 \leq i < x \\ \\ p(i-1)\frac{\rho}{x}\ , & i \geq x \end{cases} \tag{3.3.1}$$

Resolviendo recursivamente cada $p(i)$ con $i \geq 1$, obtenemos las siguientes probabilidades en función de $p(0)$:

$$p(i) = \begin{cases} p(0)\frac{\rho^i}{i!}\ , & 1 \leq i < x \\ \\ p(x)\left(\frac{\rho}{x}\right)^{i-x} = p(0)\frac{\rho^i}{x!\,x^{i-x}}\ , & i \geq x \end{cases} \tag{3.3.2}$$

Para las cuales se impone la condición de equilibrio $\rho < x$. Utilizando 3.3.1 y normalizando las probabilidades de estado obtenemos:

$$1 = \sum_{i=0}^{\infty} p(i) \tag{3.3.3}$$

$$1 = p(0)\left\{1 + \frac{\rho}{1} + \frac{\rho^2}{2!} + \cdots + \frac{\rho^x}{x!}\left(1 + \frac{\rho}{x} + \frac{\rho^2}{x^2} + \cdots\right)\right\} \tag{3.3.4}$$

Donde la parte derecha es una serie geométrica de término no nulo $\frac{\rho^x}{x!}$ y razón $\left|\frac{\rho}{x}\right| < 1$, por lo tanto la serie es convergente y se puede rescribir de la siguiente manera:

$$\frac{\rho^x}{x!}\left(1 + \frac{\rho}{x} + \frac{\rho^2}{x^2} + \cdots\right) = \frac{\rho^x}{x!}\frac{x}{x - \rho} \tag{3.3.5}$$

Aplicando 3.3.5 en 3.3.4 y despejando $p(0)$ finalmente obtenemos la probabilidad del estado inicial:

$$p(0) = \frac{1}{\sum_{i=0}^{x-1}\frac{\rho^i}{i!} + \frac{\rho^x}{x!}\frac{x}{x - \rho}}, \qquad \rho < x \tag{3.3.6}$$

Luego de obtener la expresión analítica para $p(0)$, estamos en condiciones de introducir la fórmula de Erlang, denotada $E_{2,x}(\rho)$[2]. Sea $W(t)$ una variable aleatoria de tiempo continuo que representa el tiempo de espera de un cliente en estado estacionario. A partir de la propiedad **PASTA** (Poisson Arrival, See Time Averages), la cual enuncia que la probabilidad de que un cliente que arriba al sistema tenga que esperar en la cola, es igual a la proporción de tiempo en la que todos los servidores se encuentran ocupados, se deduce lo siguiente:

$$E_{2,x}(\rho) = P\{W > 0\} = P(X \geq x)$$

$$= \frac{\sum_{i=x}^{\infty}\lambda p(i)}{\sum_{i=0}^{\infty}\lambda p(i)} = \frac{\sum_{i=x}^{\infty} p(i)}{\sum_{i=0}^{\infty} p(i)} \overset{\text{def}}{=} \sum_{i=x}^{\infty} p(i) \tag{3.3.7}$$

Aplicando 3.3.2 y 3.3.5 llegamos a:

[2] Segunda fórmula de Erlang para x servidores de parámetro $\rho = \frac{\lambda}{\mu}$.

$$E_{2,x}(\rho) = p(x)\frac{x}{x-\rho} = p(0)\frac{\rho^x}{x!}\frac{x}{x-\rho} \tag{3.3.8}$$

Finalmente aplicando 3.3.6 se obtiene la fórmula de Erlang:

$$E_{2,x}(\rho) = \frac{\frac{\rho^x}{x!}\frac{x}{x-\rho}}{\sum_{i=0}^{x-1}\frac{\rho^i}{i!} + \frac{\rho^x}{x!}\frac{x}{x-\rho}}, \rho < x \tag{3.3.9}$$

Ahora que obtuvimos la fórmula de Erlang estamos en condiciones de calcular el tiempo medio de espera de un cliente (W_q) a partir del largo medio de cola (L_q) que definimos a continuación:

$$L_q = 0\sum_{i=0}^{x} p(i) + \sum_{i=x+1}^{\infty}(i-x)p(i) \tag{3.3.10}$$

$$= \sum_{i=x+1}^{\infty}(i-x)p(x)\left(\frac{\rho}{x}\right)^{i-x} \tag{3.3.11}$$

$$= p(x)\sum_{i=1}^{\infty} i\left(\frac{\rho}{x}\right)^{i} \tag{3.3.12}$$

$$= p(x)\frac{\rho}{x}\sum_{i=1}^{\infty}\frac{\partial}{\partial(\rho/x)}\left\{\left(\frac{\rho}{x}\right)^{i}\right\} \tag{3.3.13}$$

$$= p(x)\frac{\rho}{x}\frac{\partial}{\partial(\rho/x)}\sum_{i=1}^{\infty}\left(\frac{\rho}{x}\right)^{i} \tag{3.3.14}$$

$$\xRightarrow[por\ 3.3.5]{} p(x)\frac{\rho}{x}\frac{\partial}{\partial(\rho/x)}\frac{\frac{\rho}{x}}{1-\frac{\rho}{x}} \tag{3.3.15}$$

$$= p(x)\frac{\frac{\rho}{x}}{\left(1-\frac{\rho}{x}\right)^2} \tag{3.3.16}$$

$$= E_{2,x}(\rho)\frac{\frac{\rho}{x}}{1-\frac{\rho}{x}} \tag{3.3.17}$$

Como $\rho/x < 1$ dado que la condición de estabilidad del sistema así lo establece y como sabemos que la serie de la ecuación 3.3.13 es uniformemente convergente por 3.3.5, podemos aplicar la propiedad que establece que la serie de las derivadas es la derivada de una serie. Finalmente el largo medio de cola puede expresarse como:

$$L_q = E_{2,x}(\rho)\frac{\rho}{x-\rho} \tag{3.3.18}$$

Aplicando el teorema de Little y utilizando el resultado anterior obtenemos el tiempo medio de espera de un cliente en la cola. El teorema enuncia que el largo medio de cola es igual al tiempo medio de espera multiplicado por la intensidad media de arribos. Sea W_q el tiempo medio de espera en la cola, por el teorema tenemos que:

$$L_q = \lambda W_q \tag{3.3.19}$$

$$W_q = \frac{L_q}{\lambda} \tag{3.3.20}$$

Como sabemos que el tiempo medio de permanencia en el sistema (W) o, tiempo medio total de espera es la suma del tiempo medio de espera en la cola y el tiempo medio de servicio, y utilizando 3.3.20 tenemos que:

$$W = W_q + \frac{1}{\mu} \tag{3.3.21}$$

Sustituyendo W_q en 3.3.21 obtenemos:

$$W = \frac{\frac{\rho^x}{x!}\frac{x}{x-\rho}\frac{1}{\lambda}}{\sum_{i=0}^{x-1}\frac{\rho^i}{i!} + \frac{\rho^x}{x!}\frac{x}{x-\rho}}\frac{\rho}{x-\rho} + \frac{1}{\mu} \quad (3.3.22)$$

Para este caso particular, donde cada unidad atiende una sola actividad ($\#T_i = 1, \forall i \in \{1, .., m\}$), se puede ver que:

$$E(F_i) = W(\lambda_i, \mu_i, x_i), \forall i \in \{1, .., m\} \quad (3.3.23)$$

Donde λ_i, μ_i son las tasas medias de arribos y servicios respectivamente y x_i es la cantidad de trabajadores para la unidad i.

A partir de 3.3.23 podemos rescribir la restricción 1.5 de la siguiente manera:

$$W(\lambda_i, \mu_i, x_i) \leq t_i, \forall i \in \{1, .., m\}, con\ t_i \in T_i \quad (3.3.24)$$

3.4 Eficiencia para unidades con múltiples tipos de actividades

Ahora analizaremos el caso general en donde las unidades atienden varios tipos de tareas, es decir que los clientes demandan servicios de diferente tipo. Al igual que para el caso anterior se tienen s servidores idénticos por unidad, pero que esta vez atienden con tasas de servicio exponenciales $\mu_1, \mu_1, \ldots\ldots, \mu_n$, donde μ_i representa la tasa de servicio para la actividad i. Los arribos son de Poisson con tasas exponenciales $\lambda_1, \lambda_2 \ldots \lambda_n$ donde λ_i representa la tasa de arribos para la actividad i de los clientes de tipo i, es decir los clientes que demandan la actividad i. Al igual que para el caso de un sólo tipo de actividad, los clientes que no encuentran un servidor disponible deberán esperar en una cola con cantidad infinita de posiciones de espera. También se asume accesibilidad completa.

Este escenario se corresponde a un sistema de colas multi-clase multi-servidor. Estos problemas son una generalización de los procesos de arribo $M/M/K$ con N tipos de clientes que requieren un servicio de distribución exponencial con diferentes tasas de servicio.

En esta sección veremos un breve relevamiento de la literatura recomendada, luego mostraremos diferentes propuestas para resolver este tipo de problemas, en particular describimos un procedimiento desarrollado por Sleptchenko y van Harten que propone la construcción de soluciones exactas para las ecuaciones de estado estacionario mediante a un ejemplo análogo a nuestro caso de estudio y finalmente revisaremos algunas definiciones teóricas en las que se basan los métodos de Montecarlo, los que utilizamos para el modelado de las unidades.

3.4.1 Relevamiento de literatura

Como parte de nuestro trabajo de relevamiento de literatura referente a sistemas multi-clase, multi-servidor estudiamos los siguientes trabajos de investigación:

- El trabajo de Jay Sethuraman y Mark S. Squillante [53] considera el problema de la planificación de diferentes clases de clientes en múltiples servidores distribuidos para minimizar una función objetivo basada en tiempos medios de respuesta por clase.
- El trabajo de Jouini, Aksin y Dallery [54] considera dos modelos básicos de call centers multi-clase con y sin abandonos. En este trabajo se estudia el problema de anunciar retrasos a los clientes a su llegada.
- El trabajo de Jouini, Pot, Koole y Dallery [55] considera un call center con dos clases de clientes, clientes vip y clientes comunes. Este trabajo se enfoca en el desarrollo de políticas de planificación que satisfacen restricciones sobre la relación entre las probabilidades de abandono de clientes vip sobre los clientes comunes.
- El trabajo de Sihan Ding [56] analiza la performance en sistemas de cola fork-join multi-clase.
- El trabajo de A. Sleptchenko [57] analiza un sistema de cola multi-clase, multi-servidor con prioridades no expropiativo.

3.4.2 Ecuaciones de estado estacionario

3.4.2.1 Descripción del caso de estudio

El autor propone en su trabajo, la construcción de soluciones exactas para los modelos de colas multi-clase multi-servidor ilustrado a través de un caso de gestión de repuestos en donde se tienen M sistemas y cada uno de dichos sistemas consta de varios componentes propensos a fallas. Como ejemplo de componentes podemos citar el motor, la bomba y las válvulas de una red de tuberías de una embarcación. Cuando alguno de estos componentes son dañados se produce una entrada multi-fuente a los talleres de reparación. El objetivo es sustituir rápidamente un componente que ha fallado en alguno de los M sistemas. Existe una etapa de diagnóstico de fallo en la que se determina a que taller de reparación se enviará el componente para ser reparado. La pieza dañada se reemplaza de inmediato por una pieza de repuesto si esta se encuentra en stock, de lo contrario se debe esperar hasta que la reparación pueda realizarse en alguno de los talleres de reparación. Esta espera puede provocar una considerable caída del sistema que demanda la reparación, por lo que es de suma importancia la organización y capacidad de los talleres de reparación. Una forma posible de organizar los talleres de reparación es en “disciplinas” como ingeniería mecánica, ingeniería eléctrica, etc.

Cada una de las disciplinas de un taller de reparación no se dedica exclusivamente a un tipo de componente en particular (a diferencia de los "ítems dedicados" [58]), sino que, en general, el arribo a un taller de reparación seguirá siendo de tipo multi-fuente, pero ahora con diferentes disciplinas de trabajo para diferentes tipos de componentes. Los diferentes tipos de trabajos tendrán diferentes características de tiempo de reparación y la capacidad dentro de cada disciplina es modelada con servidores idénticos. De lo anterior se concluye que la disponibilidad total del sistema depende de la cantidad de piezas de repuesto y de la capacidad del taller de reparación entre otras.

Antes de seguir con el caso de estudio, haremos una breve analogía con nuestro trabajo. A continuación mostraremos como se relaciona cada uno de los objetos de la realidad planteada por Sleptchenko y van Harten con la realidad que se plantea en este trabajo.

- Disciplinas – Unidades: Las disciplinas de trabajo de los talleres de reparación son análogas a las unidades de trabajo.
- Sistemas – Clientes: Los M sistemas que solicitan la reparación de algún componente son análogos a los clientes que demandan un cierto tipo de actividad.
- Componentes – Actividades: Los componentes que requieren reparación son equivalentes a las actividades que son requeridas por los clientes.
- Tiempo de reparación – Tiempo de Servicio: El tiempo de reparación de los diferentes tipos de trabajo son análogos a los tiempos de servicios por tipo de actividad.
- Dentro de cada una de las disciplinas se tienen servidores idénticos de la misma forma que se tienen servidores idénticos para cada una de las unidades.

Cada componente que ha fallado da lugar a una orden de reparación. Puede suponerse que cada componente puede ser reparado en exactamente un taller de reparación, dependiendo de la disciplina que se requiera, de la misma forma que asumimos que un tipo de actividad es atendida por una sola unidad. Por lo tanto se tiene una asignación fija de tipos de componentes a talleres de reparación [59].

3.4.2.2 Solución

El análisis de problemas multi-clase (MC) multi-servidor (MS) comienza como una generalización de los procesos de arribo $M/M/K$. En el contexto más general de los sistemas multi-clase muti-servidor, la cola $M/M/K$ puede ser considerada como un caso especial, en donde las clases de clientes son indistinguibles con respecto a sus características de servicio. Entonces, la tasa de servicio $\mu(i)$ de clase i es igual a la tasa de servicio promedio μ. En el caso general con N > 1 clases, $\mu(i)$ tiene un valor diferente para cada clase y $\mu(i) = (1 + \delta(i)).\mu$ donde $\delta(i)$ mide la fuerza de la perturbación con respecto a la tasa promedio de servicio μ.

Los autores se enfocan en la distribución de probabilidad estacionaria sobre los estados de un sistema general MC, MS con $\delta(i) \neq 0$ [59]. La definición de estado tiene que tener en cuenta los tipos de trabajo en ejecución y el ordenamiento de los tipos de trabajo en la cola. Es fundamental en el caso general, que la distribución sobre el ordenamiento en la cola mantenga una estructura producto basada en las fracciones de arribos de las clases de cliente. Usando esta estructura para n = el número de clientes en el sistema $> k$, se obtiene una estructura más transparente para las ecuaciones de estado. Para $n > k$ se reduce la ecuación diferencial de segundo orden a un vector en un espacio lineal de dimensión $d(N,k) = (N + k + 1)!/\{k!\,(N-1)!\}$. Esta dimensión crece rápidamente con N y k. Ahora las ecuaciones diferenciales pueden ser resueltas en términos de valores propios y vectores propios de la matriz de iteración. Todos los valores propios y vectores propios pueden ser determinados explícitamente si $\delta(i) = 0$. Precisamente $d(N,k)$ de los valores propios son reales y > 1 en el caso imperturbable, que se corresponde con un decremento exponencial para $n \to \infty$. Esta es la clave para resolver las ecuaciones de estado, primero para $n > k$ con el vector inicial para $n = k$ como desconocido y después, con recursión hacia atrás, también para $n \leq k$, hasta que sólo quede una constante escalar, la probabilidad del estado vacío. La constante escalar surge del hecho de que las probabilidades de estado suman 1 [59].

3.4.2.2.1 Definiciones y notación

- Los trabajos de tipo i arriban según un proceso de Poisson de tasa $\lambda(i)$.
- Los trabajos de tipo i son atendidos con una tasa exponencial $\mu(i)$ y es la misma para todos los servidores.
- La disciplina de trabajo mediante la cual se asignan los trabajos es First Come First Serve (FCFS) y los k servidores disponibles tienen las misma probabilidad de obtener el próximo trabajo $(1/k)$.
- La tasa de arribos total está dada por $\Lambda = \sum \lambda(i)$.
- La fracción de arribos de la clase i se denota por $a(i) \stackrel{\text{def}}{=} \lambda(i)/\Lambda$
- ρ denota el rendimiento o utilización.
- La tasa de servicio promedio μ es definida por $1/\mu \stackrel{\text{def}}{=} \sum a(i)/\mu(i)$ de forma equivalente $\mu = \Lambda/k\rho$
- Se define el estado (w,s), donde:
    - w: es un p- vector de trabajos encolados con componentes $w(i)$ y $w(i)$ se refiere a la clase del i-ésimo trabajo en la cola.

- s: es un k- vector de trabajos siendo atendidos con componentes $s(i)$ y $s(i)$ se refiere a la clase del trabajo en ejecución por el servidor.
- $w(i), s(i) \in \{0, 1, \ldots \ldots N\}, 0$ significa que no hay trabajos.

- n es el número de clientes en el sistema
- $p = \max(0,\ n - k)$ es el número de clientes en espera
- $P(w, s)$ es la distribución de probabilidad estacionaria sobre los estados para $\rho < 1$.
- s^* ordenación lexicográfica de los trabajos asignados a los servidores
- w^* ordenación lexicográfica de los trabajos en la cola

Aquí el ordenamiento lexicográfico se define como el incremento en el índice en el tipo de trabajo con un índice incremental del servidor o del lugar en la cola respectivamente.

- $\chi[s^*]$ es el número de trabajos asignados s que equivale a s^* bajo permutación, y es igual a $k!/\{k(0; s^*)! \ldots k(N; s^*)!\}$ con $k(i; s^*)$ el número de servidores ocupados por trabajos de tipo i en estado s^*.
- Análogamente, el número de secuencias de cola, equivalentes a w^* está dada por $\chi[w^*] = p!/\{k'(0; w^*)! \ldots k'(N; w^*)!\}$ con p el largo de cola y $k'(i; w^*)$ el número de trabajos de tipo i en la cola.
- El número total de asignaciones lexicográficamente ordenadas con todos los servidores ocupados es: $d(N, k) = (N + k + 1)!\ /\{k!\ (N - 1)!\}$
- Bajo condición de estabilidad $\rho < 1$, existe una única distribución de probabilidad estacionaria $P(w, s)$ y $P(w^*, s^*)$ sobre los estados y lexo-estados, respectivamente.
- $\sum P(w, s) = \sum \chi[w^*]\, \chi[s^*] P[w^*,\ s^*] = 1$

3.4.2.2.2 Ecuaciones de estado estacionario para problemas MC, MS.

El cambio de probabilidad en un intervalo de tiempo infinitesimal de un estado dado con sus vecinos tiene que ser cero en una situación de equilibrio. Los vecinos de un estado (w, s) con n clientes son estados desde o hacia los cuales un paso de transición es posible tanto por un evento de arribo (A) como la completitud de un evento de servicio (C). Los estados vecinos tienen $n - 1$ o $n + 1$ clientes. Si $n > k$, las transiciones desde o hacia los vecinos (w', s') con $n + 1$ clientes son especificadas por uno de los siguientes eventos [59]:

(a) Transiciones desde (w, s) hacia (w', s'): Un trabajo arriba y el vector cola es expandido como $w' = (w, l)$ si el tipo de trabajo es l.

(b) Transiciones desde (w', s') hacia (w, s): Un servicio ha sido completado. Si el servicio se completó en el servidor j, el primer trabajo en la cola debería ser del tipo $l = s(j)$; por lo tanto el estado (w', s') es especificado por $w' = (s(j), w)$ y $s' = E_l^j s$, donde el operador E_l^j, cambia el contenido del componente j-ésimo del vector por el trabajo de tipo l.

De manera análoga, si $n > k$, las transiciones desde o hacia los vecinos con $n - 1$ clientes son especificadas por:

(c) Transiciones desde (w, s) hacia (w', s'): Un trabajo en el servidor j se ha completado y el primer trabajo en la cola se mueve al servidor j. El primer trabajo en la cola se denota por $l = w(1)$, tenemos que $s' = E_l^j s$; además todos los trabajos de la cola son desplazados una posición, por lo que $w'(i) = w(i + 1)$ para $i \leq p - 1$ y $w'(p) = 0$.

(d) Transiciones desde (w', s') hacia (w, s): Arriba un trabajo con tipo $w(p)$, el último trabajo en la cola en el estado (w, s); entonces $w' = E_0^p w$ (omitiendo el último trabajo en la cola) y $s' = s$.

De manera análoga pueden describirse los vecinos para los estados con $n \leq k$.

Antes de pasar a las ecuaciones de estado estacionario, definimos la siguiente notación:

- Sea Tel operador de desplazamiento a la izquierda que cambia un p −vector en un $(p - 1)$ −vector con $Tw(i) = w(i + 1)$
- $Largo(v)$ es el número de entradas distintas de 0 en el vector v.
- La perturbación promedio se define como:

$$\bar{\delta}(s) \stackrel{\text{def}}{=} \frac{1}{k} \sum_{s(j) \neq 0} \delta(s(j))$$

Ahora es posible formular las ecuaciones de estado estacionario:

- $Largo(w) = p \geq 1$

$$\begin{aligned} &\left(1 + \rho + \bar{\delta}(s)\right) P(w, s) = \\ &\rho a\big(w(p)\big) P(Tw, s) + \frac{1}{k} \sum_{j=1}^{k} \sum_{l=1}^{N} (1 + \delta(l)) P((s(j), w), E_l^j s) \end{aligned} \tag{3.4.1}$$

- $0 < Largo(s) = n < k$[3]

$$\left(\frac{n}{k} + \rho + \bar{\delta}(s)\right) P(0,s) = (k-n+1)^{-1}\rho \sum_{j:s(j)\neq 0} a\big(s(j)\big) P(0, E_0^j s) + \frac{1}{k} \sum_{j:s(j)=0} \sum_{l=1}^{N} (1 + \delta(l)) P(0, E_l^j s) \tag{3.4.2}$$

- $Largo(s) = k, Largo(w) = 0$

$$\left(1 + \rho + \bar{\delta}(s)\right) P(0,s) = \rho \sum_{j=1}^{k} a\big(s(j)\big) P(0, E_0^j s) + \frac{1}{k} \sum_{j=1}^{k} \sum_{l=1}^{N} (1 + \delta(l)) P(s(j), E_l^j s) \tag{3.4.3}$$

- $Largo(s) = 0$

$$\rho P(0,0) = \frac{1}{k} \sum_{j=1}^{k} \sum_{l=1}^{N} (1 + \delta(l)) P(0, E_l^j 0) \tag{3.4.4}$$

La complejidad en la construcción de las ecuaciones de estado estacionario para los problemas MC, MS, y el desarrollo posterior para construir las soluciones exactas de dichas ecuaciones, propuesto por Sleptchenko y van Harten, decidimos optar por los métodos de Montecarlo para la resolución del problema de asignación óptima de personal en unidades, que detallamos en la siguiente sección.

[3] El factor $(k-n+1)^{-1}$ en el lado derecho de esta ecuación surge de la selección aleatoria de un trabajo que arriba entre $(k-n+1)$ servidores disponibles.

3.4.3 Métodos de Montecarlo

3.4.3.1 Introducción

El método de Monte Carlo es un método no determinístico que permite encontrar soluciones aproximadas a una gran variedad de problemas matemáticos complejos y costosos de evaluar con exactitud, mediante la simulación de variables aleatorias en una computadora.

El método es aplicable a cualquier tipo de problema, independientemente de que sea estocástico o determinista. A diferencia de los métodos numéricos que se basan en la evaluación de N puntos en un espacio M-dimensional para producir una solución aproximada que tiene un error que decrece con orden $n^{-1/m}$ en el mejor de los casos, los métodos de Monte Carlo obtienen estimaciones con un error absoluto que decrece como $\frac{1}{\sqrt{N}}$, de forma independiente de m. Esta es la principal ventaja del método, ya que en muchos casos permite que sea el único método aplicable [60].

El nombre del método surge en alusión al Casino de Monte Carlo en el Principado de Mónaco, por ser considerada "la capital del juego de azar", y por ser la ruleta un generador simple de números aleatorios.

Los métodos de Monte Carlo aparecen alrededor de 1940 y fueron mejorando con el desarrollo de la computadora. En esa época se utilizaron como herramienta de investigación en el desarrollo de la bomba atómica durante la Segunda Guerra Mundial en EE.UU. El trabajo involucraba la simulación directa de problemas probabilísticos de hidrodinámica. En la actualidad se utilizan en algoritmos de Raytracing para la obtención de imágenes en 3D, entre muchas otras aplicaciones.

3.4.3.2 Aplicaciones

Los métodos de Monte Carlo son utilizados en una amplia gama de aplicaciones. A continuación detallamos algunas de ellas:

Lanzamiento de una moneda: Si se desea reproducir, mediante números aleatorios, el lanzamiento sucesivo de una moneda, debemos previamente asignarle un intervalo de números aleatorios a CARA y otro intervalo a CRUZ, de manera de poder interpretar el resultado de la simulación. Tales intervalos se asignan en función de las probabilidades de ocurrencia de cada cara de la moneda. Se tiene:

- Probabilidad de que se obtenga CARA: 0,50 Números aleatorios: 0,000 al 0,499
- Probabilidad de que se obtenga CRUZ: 0,50 Números aleatorios: 0,500 al 0,999

Después, al generar un número aleatorio se obtiene el resultado simulado. Así, si se obtiene el número aleatorio 0,385, se observa que está incluido en el intervalo asignado a CARA [61].

Problema de Buffon: Una aplicación muy conocida de los métodos de Monte Carlo es el problema de Buffon que detallamos en el Apéndice B de este documento.

Otros ejemplos de la aplicación de los métodos de Monte Carlo a problemas de la vida real son los siguientes:

- La simulación de sistemas con varios grados de libertad, tales como fluidos y sólidos fuertemente acoplados,
- Para modelar fenómenos con una cantidad significativa de datos de entrada inciertos como por ejemplo el cálculo de riesgo en los negocios.
- Para evaluar integrales multidimensionales definidas con condiciones de borde complejas.
- Para estimar la confiabilidad (probabilidad de conexión de red con enlaces de Bernoulli) en redes confiables [62].

3.4.3.3 Esquema de un Método de Monte Carlo.

Si se desea calcular un cierto valor, digamos ϕ, y se conoce una variable aleatoria X con distribución F_X tal que $\phi = E(X)$.

El método de Monte Carlo consiste en:

1. Generar aleatoriamente, siguiendo la distribución de probabilidad F_X, valores para un conjunto $X^{(1)}$, $X^{(2)}$, ... , $X^{(n)}$, de variables aleatoria i.i.d (independientes e idénticamente distribuidas) a X.
2. Calcular la suma de los n valores generados, donde n es el tamaño de la muestra:

$$S_n = X^{(1)} + X^{(2)} + \ldots\ldots + X^{(n)}$$

3. Calcular $\widehat{X} = {S_n}/{n}$, donde $\widehat{X}$, es una estimador para ϕ
4. Calcular $\hat{V} = \sum_{i=1}^{n} (X^{(i)})^2/n(n-1) - \hat{X}^2/(n-1)$, donde $\hat{V}$ es un estimador para $Var(\hat{X})$

A continuación incluimos un pseudocódigo que reproduce los pasos del método básico descripto anteriormente:

Estimación Monte Carlo

In n: Tamaño de la muestra
Out $\hat{X}$: Estimador de ϕ
Out $\hat{V}$: Estimador de $Var(\hat{X})$

1: $\hat{X} = 0$; //Inicialización
2: $\hat{V} = 0$;
3: **for** i ← 1 to n
4: $Sortear\ un\ valor\ de\ la\ variable\ X^{(i)}$ con distribuición F_X;
5: $\hat{X} = \hat{X} + X^{(i)}$
6: $\hat{V} = \hat{V} + (X^{(i)})^2$
7: **endfor**
8: $\hat{X} = \hat{X}/n$
9: $\hat{V} = \hat{V}\ /\big(n * (n-1)\big) - \hat{X}^2/(n-1)$

3.4.3.4 Métodos de Monte Carlo para simulación de unidades.

Considerando el modelo básico de un método de Montecarlo visto en 3.4.3.3 podemos decir que para el problema de asignar de forma óptima personal a cada una de las unidades obtenidas a partir de una coloración utilizamos dicho modelo de la siguiente manera:

Sea X_i la variable aleatoria que denota el suceso de generar un cliente en el sistema y que demanda una actividad de tipo i. Además se definen las variables aleatorias Y_j y Z_k, que denotan el tiempo de espera en cola para un cliente que demanda una actividad de tipo j y el tiempo de permanencia en el sistema de un cliente que demanda una actividad de tipo k, respectivamente.

1. Generamos aleatoriamente valores para los conjuntos:

$$X_1^{(1)}, X_1^{(2)}, \dots\dots, X_1^{(p)},$$

$$X_2^{(1)}, X_2^{(2)}, \dots\dots, X_2^{(q)},$$

$$\dots\dots$$

$$\dots\dots$$

$$X_n^{(1)}, X_n^{(2)}, \dots\dots, X_n^{(t)},$$

de variables i.i.d a cada X_i. Luego de que cada variable de tipo i es generada por la herramienta OmneT++ (ver Apéndice C), se generan aleatoriamente los valores para Y_j y Z_k dependiendo de varios factores:

- La disponibilidad de servidores.

- La existencia de clientes encolados.
- Las tasas de servicio.

2. Como nos interesa calcular el valor esperado $E(Y_j)$ que se corresponde con el tiempo promedio de espera en cola para cada tipo de cliente y el valor esperado $E(Z_k)$ que se corresponde con el tiempo de permanencia en el sistema para cada tipo de cliente, debemos obtener los siguientes valores:

$$SY_1^{(p)} = Y_1^{(1)} + Y_1^{(2)} + \dots\dots + Y_1^{(p)}, \qquad SZ_{1p} = Z_1^{(1)} + Z_1^{(2)} + \dots\dots + Z_1^{(p)},$$

$$SY_{2q} = Y_2^{(1)} + Y_2^{(2)} + \dots\dots + Y_2^{(q)} \qquad SZ_{2q} = Z_2^{(1)} + Z_2^{(2)} + \dots\dots + Z_2^{(q)}$$

$$\dots\dots \qquad\qquad \dots\dots$$

$$\dots\dots \qquad\qquad \dots\dots$$

$$SY_{nt} = Y_n^{(1)} + Y_n^{(2)} + \dots\dots + Y_n^{(t)}, \qquad SZ_{nt} = Z_n^{(1)} + Z_n^{(2)} + \dots\dots + Z_n^{(t)},$$

Correspondientes a la suma de valores sorteados por cada tipo de actividad.

3. Finalmente obtenemos:

$$E(Y_1) = {SY_1^{(p)}}/{p}, \qquad E(Z_1) = {SZ_1^{(p)}}/{p},$$

$$E(Y_2) = {SY_2^{(q)}}/{q}, \qquad E(Z_2) = {SZ_2^{(q)}}/{q},$$

$$\dots\dots \qquad\qquad \dots\dots$$

$$\dots\dots \qquad\qquad \dots\dots$$

$$E(Y_n) = {SY_n^{(t)}}/{t}, \qquad E(Z_n) = {SZ_n^{(t)}}/{t},$$

De esta forma obtenemos el tiempo promedio de espera en cola por tipo de cliente así como también el tiempo de permanencia en el sistema por tipo de cliente.

Los tiempos promedio de espera en cola y de permanencia en el sistema fueron comparados con las fórmulas de Little para el caso de unidades con un sólo tipo de actividad para verificar la cercanía entre los resultados empíricos y los teóricos. También se incluyó el cálculo de probabilidad de que un cliente de cierto tipo deba esperar en la cola. Para ello aproximamos esta probabilidad de la siguiente manera:

$$P\{Y_i > 0\} \cong \frac{\#\{clientes_{encolados_i}\}}{\#\{clientes_i\}}$$

Este resultado fue comparado con el obtenido aplicando la fórmula de Erlang C. Estas comparaciones pueden verse en Apéndice E.

Capítulo 4 – Implementación de la solución

4.1 Introducción

En la sección 1.4, se demostró que el problema de dimensionamiento de empresas de alto porte se puede resolver a partir de la aplicación secuencial de dos problemas matemáticos bien conocidos. El primer paso consiste en generar una coloración óptima utilizando alguno de los algoritmos de coloración seleccionados en la sección 2.7.5 del capítulo 2 y el segundo paso se resuelve a partir de la implementación de un simulador que imita el funcionamiento de cada unidad como parte de una función de aptitud que evalúa la factibilidad de asignar determinada cantidad de personal a partir del cumplimiento de las restricción 1.5 del modelo presentado en la sección 1.3. La función de aptitud en combinación con una estrategia de bipartición (para la asignación de personal) constituye en su conjunto la metaheurística que busca dar solución a la función objetivo 1.1 de la formulación matemática.

En este capítulo utilizaremos un enfoque top-down de análisis de la solución completa. Inicialmente mostraremos el esquema general de la solución (sección 4.2.2), para luego profundizar en los pasos más importantes del algoritmo implementado (secciones 4.2.3, 4.2.4 y 4.2.5).

Si el lector desea profundizar sobre otros temas relacionados con la implementación de la solución, podrá encontrar una breve explicación sobre el formato del archivo de configuración utilizado por el simulador en el Apéndice D, además se brinda un análisis de calibración del tiempo de simulación y de la asignación inicial de personal en el Apéndice E y finalmente en el Apéndice F, encontrará información detallada acerca del funcionamiento del generador de archivos de configuración de la herramienta.

4.2 Implementación de la solución

4.2.1 Parámetros de línea de comando

Antes de entrar en materia, vale la pena brindar una breve explicación de los diferentes parámetros que puede recibir la aplicación y qué efecto tiene cada uno de ellos en su ejecución.

La sintaxis general para invocar al programa es la siguiente:

metaheuristica.exe **nombre_experimento [tipo_archivo] [algoritmo] [estrategia]**

Donde **nombre_experimento** es el nombre del "conjunto de archivos" (sin extensión y puede contener una ruta como prefijo) que contienen los metadatos que especifican toda la información necesaria para poder ejecutar el programa. Para funcionar, la aplicación necesita básicamente 2 grupos de información. El primero es la matriz de compatibilidad (archivo de extensión **.net**) que determina, como sugiere su nombre, las compatibilidades entre las diferentes actividades de la organización y el segundo grupo que contiene toda la información necesaria para correr una simulación (archivo de extensión **.sim**, los cuales se ven en detalle en el Apéndice D). Algo importante a tener en cuenta es que, para configurar un experimento, ambos archivos deben estar contenidos dentro de un mismo directorio y deben llamarse igual, es por eso que el parámetro **nombre_experimento** no debe hacer referencia a archivos en particular.

El parámetro **tipo_archivo**, es un parámetro opcional que indica el tipo de instancia, dado que existen diferencias entre las instancias de DIMACS y las del grupo de taller, de esta forma podemos indicar que formato de especificación de grafo debe interpretar la aplicación (ver formatos en 2.7.2). Las opciones disponibles son DIMACS y GRUPO.

El parámetro **algoritmo**, es otro parámetro opcional que indica el algoritmo de coloreo que se debe utilizar. Las opciones disponibles son LRLF y EXTRACOL (ver algoritmos escogidos en 2.7.5).

Finalmente el parámetro **estrategia**, también opcional, determina la estrategia utilizada para crear una asignación inicial de personas para cada unidad. El cálculo de la asignación lo veremos más adelante en 4.2.3.
Las opciones disponibles son:

- PROMEDIO:
  - Establece que para el cálculo del personal se utilicen los tiempos promedio de arribo y servicio.
- RO_PROMEDIO:
  - Establece que para el cálculo del personal se utilice el rendimiento promedio (ρ) para realizar el cálculo.
- PEOR_CASO:
  - Se utiliza la mejor tasa de arribos y la peor tasa de servicios para realizar el cálculo.
- ADICIONAL:
  - Se utilizan las mismas tasas que el caso anterior, y se duplica el resultado final obtenido.

A continuación resumimos en una tabla los diferentes valores que pueden adoptar estos parámetros y daremos algunos ejemplos de invocación que podrá utilizar el lector:

Parámetro	Tipo	Valores	Defecto
nombre_experimento	Obligatorio	../../nombre_experimento	N/A
tipo_archivo	Opcional	DIMACS, GRUPO	DIMACS
algoritmo	Opcional	LRLF, EXTRACOL	LRLF
estrategia	Opcional	PROMEDIO, RO_PROMEDIO, PEOR_CASO, ADICIONAL	PROMEDIO

Tabla 10 – Parámetros de invocación para la metaheurística

Ejemplos:

1. metaheuristica.exe ../instancias/ DSJC125.1 DIMACS LRLF PROMEDIO
2. metaheuristica.exe ../instancias/ DSJC125.1 DIMACS
3. metaheuristica.exe ../instancias/ DSJC125.1 LRLF
4. metaheuristica.exe ../instancias/ DSJC125.1 PROMEDIO
5. metaheuristica.exe ../instancias/ DSJC125.1
6. metaheuristica.exe ../instancias/ DSJC125.1 EXTRACOL
7. metaheuristica.exe instancia_prueba_a200_u015_p025 GRUPO
8. metaheuristica.exe instancia_prueba_a200_u015_p025 GRUPO EXTRACOL
9. metaheuristica.exe ../instancias/ DSJC125.1 EXTRACOL PEOR_CASO

Las opciones de la 1 a la 5 son equivalentes porque los 3 valores ingresados son los que toma el sistema por defecto. En el caso 6 se ejecuta EXTRACOL_CSR_ATS sobre una instancia de DIMACS con una estrategia PROMEDIO. Entre el caso 7 y 8 la única diferencia radica en que para el 7 se ejecuta LRLF. El noveno caso ejecuta EXTRACOL_CSR_ATS sobre una instancia de DIMACS utilizando una estrategia de inicialización de personal de PEOR_CASO.

4.2.2 Esquema general de la solución

Para resolver el problema de asignación y dimensionamiento, se implementó una solución integral aplicando una estrategia de subdivisión del problema en etapas, como se sugiere en la sección 1.4.
La estructura general del algoritmo es la siguiente:

a) Procesamiento de parámetros:
   - Básicamente en este paso se realiza todo el procesamiento de los parámetros de entrada para configurar las diferentes opciones de funcionamiento ofrecidas por la solución (ver sección anterior 4.2.1).

b) Búsqueda de una coloración factible:
   - Este paso recibe como parámetros de entrada el archivo con la especificación del grafo a cargar (*.net), el formato de dicho archivo y el tipo de algoritmo a aplicar (información obtenida a partir del paso a)) y devuelve una coloración posible, o sea una asignación de actividades en unidades factible aplicando el algoritmo de

coloreo indicado en el parámetro de entrada. De esta forma se estarían cumpliendo las restricciones 1.2 y 1.3 de la formulación matemática como se establece en 1.4.1.

c) Búsqueda del número óptimo de servidores:
   - Una vez obtenida una coloración factible en b) se procede a buscar la mejor asignación de personal para cada unidad departamental. Este paso recibe el archivo de configuración a utilizar por el simulador (*.sim), la coloración del paso anterior y la estrategia de asignación inicial de personal para las unidades y devuelve la cantidad de personal necesario para cumplir con las restricciones de tiempo establecidas por la restricción 1.5 de la formulación matemática en 1.4.1. El algoritmo de este paso se verá en detalle en 4.2.3.

A continuación presentamos un pseudocódigo que permite visualizar de forma gráfica la estructura detallada de la solución:

Solución_Integral

In $nombre_{exp}$: Nombre del experimento
In $tipo_{arch}$: Tipo de Archivo (DIMACS, GRUPO)
In $algoritmo$: Nombre del algoritmo de coloreo (LRLF, EXTRACOL)
In $estrategia$: Estrategia utilizada en la asignación inicial (PROMEDIO, RO_PROMEDIO, PEOR_CASO, ADICIONAL)
Out n: Número óptimo de personal
Var $instancia_{coloreo}$: Archivo que contiene la especificación del grafo
Var $param_{sim}$: Archivo que contiene los parámetros del simulador
Var $coloreo$: Contiene una coloración factible obtenida a partir de aplicar LRLF o EXTRACOL sobre $instancia_{coloreo}$

```
1: instancia_coloreo = Concatenar(nombre_exp, ".net"); // Concatenar = función estándar de strings
2: param_sim = Concatenar(nombre_exp, ".sim");
3: coloreo = Colorear(instancia_coloreo, tipo_arch, algoritmo);
4: n = Asignacion_Optima_Personal(param_sim, coloreo, estrategia);
5: return n;
```

Donde Colorear tiene la siguiente lógica:

Colorear

In $instancia_{coloreo}$: Archivo que contiene la especificación del grafo
In $tipo_{arch}$: Tipo de Archivo (DIMACS, GRUPO)
In $algoritmo$: Nombre del algoritmo de coloreo (LRLF, EXTRACOL)
Out $coloreo$: Coloración factible de $instancia_{coloreo}$
Var $grafo$: Contiene el grafo especificado en $instancia_{coloreo}$

```
1: if tipo_arch = DIMACS then
2:       grafo = Leer_DIMACS(instancia_coloreo);
3: else
4:       grafo = Leer_GRUPO(instancia_coloreo);
5: endif
6: grafo_c = grafo.Complemento(); // Se complementa el grafo, para construir la matriz de
7:                                // incompatibildades.
8: if algoritmo = LRLF then
```

```
9:      coloreo = grafo_c.LRLF();
10: else
11:     if grafo_c.size() <= 1000
12:             residuo = grafo_c.size() * 0.65;
13:     else
14:             residuo = 800;          // Dato sugerido por el autor de EXTRACOL
15:     endif
16:     coloreo = grafo_c.EXTRACOL_CSR_ATS(residuo, 10, 40);
17: endif
18: return coloreo;
```

Notar que la invocación del algoritmo EXTRACOL_CSR_ATS recibe 3 parámetros (ver pseudocódigo en 2.5.6.1), cuyos valores óptimos fueron investigados en la sección de parametrización del algoritmo en 2.7.4.3.

4.2.3 Número óptimo de servidores

El paso final de la solución busca determinar cuál es la menor asignación de personal necesaria para poder cumplir con las restricciones del modelo. Para llevar a cabo este cometido se implementó un algoritmo metaheurístico que utiliza una estrategia de búsqueda de bipartición (ver algoritmos de divide y vencerás en [63]), donde el criterio de verificación de éxito (determinación de una solución factible) es una función de aptitud que combina la simulación de una unidad particular con cierta asignación de personal, con una función de análisis de los resultados de la simulación que compara los tiempos de permanencia promedio en el sistema de cada tipo de cliente con su tolerancia.

Antes de ejecutar el algoritmo solución, se deben configurar algunos parámetros de ambiente necesarios para la simulación, como la escala de tiempo utilizada por las variables temporales, la cantidad de repeticiones por experimento, la carga de tasas de arribos y servicios, etc. (ver Apéndice D).

A continuación presentamos el pseudocódigo de configuración de ambiente y de ejecución del algoritmo solución:

Asignacion_Optima_Personal

In $param_{sim}$: Archivo que contiene los parámetros del simulador
In $coloreo$: Contiene una asignación de actividades en unidades (coloración factible)
In $estrategia$: Estrategia utilizada en la asignación inicial (PROMEDIO, RO_PROMEDIO, PEOR_CASO, ADICIONAL)
Out n: Número óptimo de personal
Var $config$: Contiene los parámetros de configuración del simulación especificados en $param_{sim}$
Var sim: Motor de simulación
Var $n_{inicial}$: Número inicial de personal para cada unidad
Var $tArribo$: Vector que contiene las tasas de arribo para todas las actividades
Var $tArribo_c$: Vector que contiene las tasas de arribo de las actividades asignadas a

una unidad
Var $tServicio$: Vector que contiene las tasas de servicio para todas las actividades
Var $tServicio_c$: Vector que contiene las tasas de servicio de las actividades asignadas a una unidad
Var $tolerancias$: Vector que contiene las tolerancias para todas las actividades
Var $tolerancias_c$: Vector que contiene las tolerancias de las actividades asignadas a una unidad

```
1: config = Cargar_Configuracion(param_sim);
2: sim.Escala_Tiempo(config.ObtenerParametro("Escala_Tiempo"));
3: sim.Límite_Tiempo(config.ObtenerParametro("Limite_Tiempo"));
4: sim.Repeticiones(config.Obtener_Parametro("Repeticiones"));
5: tArribo = config.Obtener_Parametro("Tasas_Arribo");
6: tServicio = config.Obtener_Parametro("Tasas_Servicio");
7: tolerancias = config.Obtener_Parametro("Tolerancias");
8: foreach c in coloreo do     // c contiene las actividades para una unidad
9:     tArribo_c = Obtener_Elementos(c, tArribo);
10:    tServicio_c = Obtener_Elementos(c, tServicio);
11:    tolerancias_c = Obtener_Elementos(c, tolerancias);
12:    n_inicial = Calcular_Asignación_Inicial(estrategia, tArribos_c, tServicios_c, c.size( ));
13:    sim.Set_Parametro("Tasas_Arribo", tArribo_c);
14:    sim.Set_Parametro("Tasas_Servicio", tServicio_c);
15:    sim.Set_Parametro("Tolerancias", tolerancias_c);
16:    n+= Buscar_Asignación_Óptima(n_inicial, sim);    // metaheurística
17:    if c.size( ) = 1 then
18:        Comparar_Resultados_Erlang(sim);
19:    endif
20: end foreach
21: return n;
```

En las líneas 2, 3 y 4 se configuran los parámetros generales para la simulación, es decir los parámetros de escala de tiempo utilizado por las variables temporales del motor de simulación, de límite de tiempo de simulación y cantidad de repeticiones de cada experimento.

Cabe aclarar que la función Obtener_Elementos, utilizada en las líneas 9, 10 y 11, devuelve un vector que contiene los valores contenidos en el vector ingresado como segundo parámetro correspondiente a las actividades especificadas en el primer parámetro. Otra cuestión a tener en cuenta es que para el caso particular en que la unidad sólo contenga una actividad (líneas 17 a 19), los resultados de la simulación (probabilidad de encolar un trabajo, el tiempo promedio de espera en cola y el tiempo promedio de permanencia en el sistema), son comparados contra las fórmulas de Erlang y Little (ver fórmulas en 3.3 y el cálculo empírico en 3.4.3.4). Finalmente la función Calcular_Asignación_Inicial (línea 12) que, como explicita su nombre, realiza el cálculo de la asignación inicial de personal, depende de la estrategia especificada. A continuación damos el pseudocódigo de esta función:

Calcular_Asignación_Inicial

In $estrategia$: Estrategia utilizada en la asignación inicial (PROMEDIO, RO_PROMEDIO, PEOR_CASO, ADICIONAL)
In $tArribo$: Vector que contiene una serie de tasas de arribos
In $tServicio$: Vector que contiene una serie de tasas de servicios
In $nMuestra$: Contiene la cantidad de elementos en $tArribos$ y $tServicios$
Out n: Número inicial de personal

1: **docase** estrategia
2: **case** "PROMEDIO":
3: $\mathrm{n} = \mathrm{Techo}(\frac{\mathrm{Promedio(tServicio)}}{\mathrm{Promedio(tArribo)}} * \mathrm{nMuestra});$
4: **case** "RO_PROMEDIO":
5: $\mathrm{n} = \mathrm{Techo}(\mathrm{Promedio}\left(\frac{\mathrm{tServicio}}{\mathrm{tArribo}}\right) * \mathrm{nMuestra});$
4: **case** "PEOR_CASO":
5: $\mathrm{n} = \mathrm{Techo}(\frac{\mathrm{Mayor(tServicio)}}{\mathrm{Menor(tArribo)}} * \mathrm{nMuestra});$
6: **case** "ADICIONAL":
7: $\mathrm{n} = \mathrm{Techo}\left(\frac{\mathrm{Mayor(tServicio)}}{\mathrm{Menor(tArribo)}} * \mathrm{nMuestra}\right) * 2;$
8: **end_docase**
9: **return** n;

Donde las funciones Promedio, Mayor y Menor obtienen el valor promedio, mayor y menor respectivamente contenidos en un vector de reales y la función Techo retorna el entero más pequeño que no sea menor al valor ingresado como parámetro en la función.

4.2.4 Metaheurística

En la sección anterior dimos una idea general del funcionamiento del algoritmo de búsqueda de la solución, donde la receta utilizada para encontrar una solución factible consta de la aplicación de un método de búsqueda por bipartición, donde cada nueva región de exploración tiene la mitad de tamaño de la región anterior y la función de verificación determina la validez del resultado sugerido a partir del cual se determina un nuevo intervalo de búsqueda, en caso de existir uno.

A continuación damos el pseudocódigo que implementa esta función:

Buscar_Asignación_Óptima

In $n_{inicial}$: Determina la cantidad inicial de personal asignada a la unidad
In sim: Motor de simulación
Out $n_{óptimo}$: Número óptimo de personal
Var $cota_{superior}$: Contiene la última asignación de personal factible
Var $cota_{temporal}$: Contiene la asignación de personal del caso de prueba actual
Var $cota_{inferior}$: Contiene una asignación de personal inválida o no factible

1: $\mathrm{cota_{superior}} = \mathrm{cota_{temporal}} = \mathrm{n_{inicial}};$
2: $\mathrm{cota_{inferior}} = 0;$
3: **while** not fin **do**

```
4:      sim.Número_Servidores(cota_temporal);
5:      for i ← 1 to sim.Repeticiones()
6:              esfactible = Verificar_Asignación(sim);     // función de aptitud
7:              if not esfactible then
8:                      goto 11;
9:              endif
10:     endfor
11:     if esfactible then
12:             cota_superior = cota_temporal;  // Actualizo cota_superior con la nueva
13:                                             // asignación factible
14:             cota_temporal = Techo((cota_temporal + cota_inferior)/2);
15:             if cota_temporal = cota_inferior then
16:                     fin = true;
17:             endif
18:     else
19:             cota_inferior = cota_temporal;  // Actualizo cota_inferior con una nueva asignación no factible
20:             if cota_superior = cota_temporal then
21:                     cota_superior = cota_temporal * 2;      // Corrijo el intervalo de búsqueda
22:                     cota_temporal = cota_superior;
23:             else
24:                     cota_temporal = Techo((cota_temporal + cota_superior)/2);
25:                     if cota_temporal = cota_inferior then
26:                             fin = true;
27:                     endif
28:             endif
29:     endif
30: endwhile
31: n_óptimo = cota_superior;
32: return n_óptimo;
```

En la línea 4 se ingresa la cantidad de personal al simulador para cada iteración. De las líneas 5 a 10 se ejecutan las repeticiones independientes del escenario de simulación. En la línea 6 se invoca la función de aptitud, encargada de ejecutar la simulación y la comprobación de los resultados. En caso de que la solución no sea factible se finaliza la ejecución (líneas 7, 8 y 9) y se continúa con la búsqueda de nuevos casos. De las líneas 11 a 29 se encuentra la lógica que implementa el algoritmo de bipartición, que en los casos de éxito, disminuye la asignación del personal a la mitad (línea 14) y en el caso contrario los aumenta al doble (línea 21). Algo importante a tener en cuenta es que puede ocurrir que la asignación de recursos inicial no sea factible, lo que provocaría una corrección en la región de búsqueda. Dicho caso de corrección se encuentra contemplado en las líneas 20, 21 y 22.

4.2.5 Verificación de asignación

La función de verificación de asignación se encarga básicamente de responder si se cumple o no la restricción del modelo que establece que los tiempos de permanencia promedio en el sistema deben ser menores a los tiempos de tolerancia máxima por cada tipo de cliente, a partir de una asignación de personal dada. Para poder responder esta interrogante se elaboró un procedimiento que consta básicamente de dos pasos:

1. Simulación de una unidad de trabajo
2. Verificación de resultados

Para la implementación del primer paso se utilizó el motor de simulación OMNeT++ versión 4.2.2; el lector podrá encontrar más información sobre ésta herramienta en el Apéndice C. El paso de verificación de resultados, básicamente calcula el tiempo de servicio promedio, por cada tipo de cliente generado, valor que se compara con la tolerancia de cada uno de ellos. En caso de cumplirse la restricción antes mencionada, significa que se encontró una solución factible al problema, aunque esta no sea óptima (mínima).

4.2.5.1 Modelo de simulación

A partir de los conceptos básicos en los que se basa OMNeT++ (ver Apéndice C), estamos en condiciones de describir como implementamos el modelo que simula la realidad planteada en esta tesis.
Nuestro modelo solución está compuesto básicamente por los siguientes módulos:

- **Source:**
  - Es un módulo simple que contiene la lógica necesaria para generar muestras aleatorias e independientes con distribución exponencial. Éste módulo es el encargado de simular un tipo de cliente en el sistema. Para determinar el tipo de cliente, es posible especificar a partir de uno de sus parámetros el tiempo entre arribos para un tipo de cliente específico.
- **PassiveQueue:**
  - Es un módulo simple que implementa un administrador de la cola de clientes en el sistema. El nombre proviene del hecho que esta cola no realiza ninguna acción por iniciativa propia, depende enteramente de estímulos externos (la llegada de un cliente, o el pedido de un cliente desde un servidor). A partir de sus parámetros es posible especificar la capacidad máxima, la estrategia de encolamiento (pila o fifo) y la estrategia de distribución de trabajos entre los servidores disponibles (aleatoria, prioridad, roundRobin o demora mínima). Para nuestro caso la capacidad de la cola es infinita, la estrategia de encolamiento es fifo (First In First Out) y la distribución de trabajos es aleatoria.
- **Server:**
  - Es un módulo simple que contiene la lógica necesaria para simular un servidor capaz de atender cualquier tipo de cliente en el sistema. Su parámetro principal

permite especificar los tiempos de servicio de las actividades que es capaz de atender.

- **Sink:**
  - Es un módulo simple encargado de registrar datos estadísticos, los cuales son utilizados para los cálculos. Los dos datos más importantes que almacena son el tiempo total de permanencia en el sistema y el tiempo total en cola de cada trabajo, discriminado por tipo.
- **Simulador:**
  - Éste es el módulo principal y representa el modelo de simulación.
    Internamente está compuesto por los 4 modelos simples anteriores e interconecta los diferentes generadores de muestras (Sources) con la cola (PassiveQueue), la cola con los diferentes servidores (Servers) y los servidores con el recolector de resultados (Sink). Los dos parámetros principales que contiene permiten determinar la cantidad de generadores y servidores que se deben instanciar.

Para facilitar la comprensión de la estructura del modelo, a continuación presentamos una ilustración que representa los diferentes módulos y las interconexiones entre ellos:

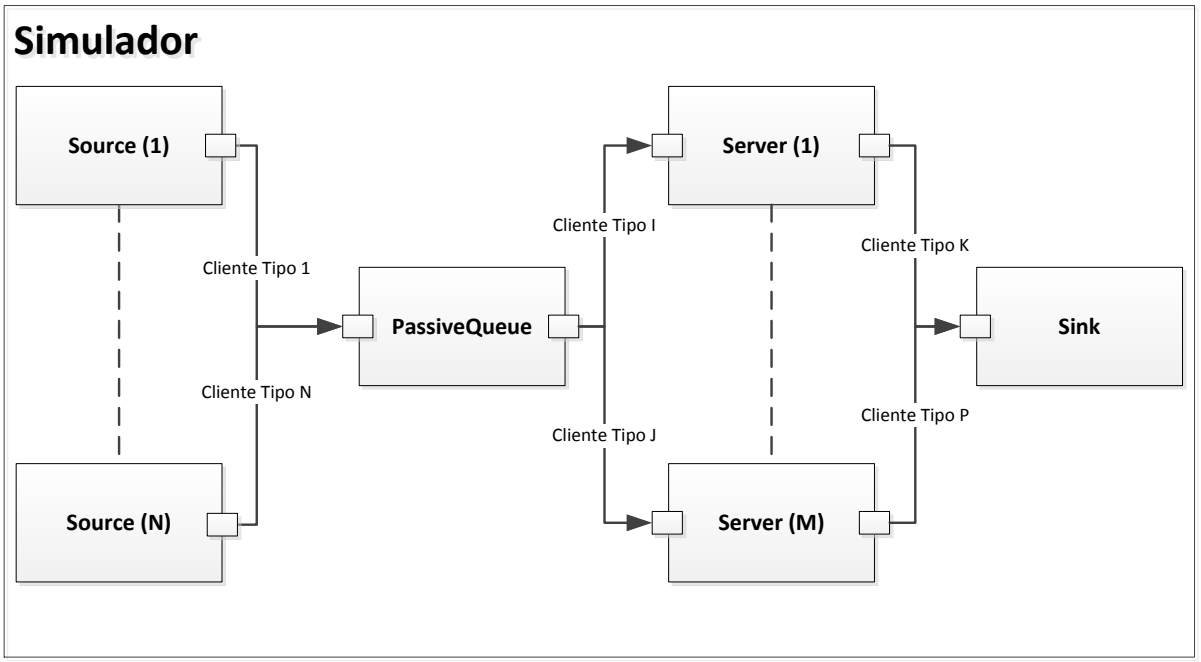

Figura 5 – Implementación del modelo de simulación

4.2.5.2 Implementación

Luego de introducir los conceptos más importantes que competen al framework de simulación, estamos en condiciones de mostrar el pseudocódigo de la función de verificación:

Verificar_Asignación

In sim: Motor de simulación
Out $esfactible$: Determina si la asignación cumple con la restricción 1.5

```
1: while not sim.Fin do        // sim.Fin = true si no quedan más eventos o se alcanzó el límite de tiempo
2:       módulo = sim.Siguiente_Módulo( );
3:       sim.Ejecutar_Evento(módulo);
4: endwhile
5: esfactible = true;
6: for i ← 1 to sim.Sources.size()   // Itero en la cantidad de generadores o actividades
7:       if sim.Sink.Obtener_Tiempo_Permanencia(i) >
8:          sim.Obtener_Parametro("Tolerancias")[i] then
9:                esfactible = false;
10:               goto 14;
11:      endif
12: endfor
13: return esfactible;
```

Las líneas 1 a 4 implementan el primer paso de verificación que consiste en la simulación de una unidad. Como se puede observar, a grandes rasgos respeta la estructura de funcionamiento de un simulador de eventos discretos, como se expone en la sección C.1 del Apéndice C.

Una vez finalizada la simulación, ya sea porque no hay más eventos para ejecutar o porque se alcanzó el límite de tiempo de ejecución (en nuestro caso sólo aplica el segundo criterio), se lleva a cabo la verificación de la restricción 1.3 del modelo matemático (líneas 5 a 11), donde para cada actividad atendida en la unidad simulada, se compara el tiempo de permanencia en el sistema calculado de cada tipo de cliente contra su tolerancia (líneas 7 y 8). Si el valor calculado es mayor, se finaliza el bloque iterativo de verificación debido a que el resultado ya no es satisfactorio (líneas 9 y 10).

En la línea 7, la función sim.Sink.Obtener_Tiempo_Permanencia(i) permite obtener los tiempos de permanencia registrados en el módulo de estadísticas Sink (ver sección anterior 4.2.5.1).

Un detalle que merece la pena recordar es, en la línea 15 del pseudocódigo de la sección 4.2.3 se configuró el parámetro "Tolerancias" en el simulador con el vector de tolerancias, de acuerdo a esto, en la línea 8 de éste pseudocódigo, se obtiene cada tolerancia utilizando el operador de indizado [] para buscar los elementos en dicho vector.

Capítulo 5 – Ejecución y análisis de pruebas

En esta sección mostraremos los resultados finales producto de la ejecución de la solución integral (coloreo y metaheurística) sobre un conjunto de instancias, que incluye la mayoría de las instancias del Grupo de Taller y algunas instancias de DIMACS. Luego de mostrar los resultados, haremos un análisis y daremos nuestras conclusiones sobre la información obtenida.

5.1 Infraestructura utilizada (hardware y software)

Para la ejecución de pruebas se utilizó un PC de ocho núcleos, con una frecuencia máxima de 3.6 GHz y arquitectura de 64 bits, con 8GB de memoria RAM, sistema operativo Windows 7 de 64 bits, OMNeT++ IDE como entorno de desarrollo y C++ como lenguaje de programación.

5.2 Ejecución de pruebas

Todas las pruebas fueron realizadas utilizando la estrategia de asignación de personal y generación de tiempo límite de simulación de acuerdo a lo analizado en Apéndice E.

Las pruebas se separaron básicamente en 2 grandes grupos. Para el primero, integrado por instancias de hasta 250 vértices (grafos pequeños), se ejecutaron una serie de 3 pruebas por cada una de ellas, salvo para la instancia de 250 debido al excesivo tiempo de ejecución requerido para resolverlo. Para el segundo, integrado por instancias de hasta 2000 vértices (grafos de gran tamaño) se ejecutó una sola prueba para cada tipo de algoritmo de coloreo disponible (LRLF y EXTRACOL_CSR_ATS, dando un total de 4 ejecuciones), porque se requiere mucho tiempo para la resolución de cada escenario de prueba. La principal idea detrás de la ejecución de instancias de gran tamaño, es poder demostrar la propiedad de agrupación de personal en las unidades expuesta en 3.2.4, donde se plantea básicamente la mejora en el cálculo del personal necesario a partir de una mejor distribución de actividades en unidades (menor cantidad de unidades). Como ya vimos en 2.7.5, EXTRACOL_CSR_ATS obtiene menor cantidad de colores para instancias de 1000 nodos en adelante que el resto de los algoritmos, lo que nos permitirá ver si se cumple o no la propiedad enunciada en el párrafo anterior.

A continuación presentamos los resultados obtenidos a partir de la ejecución de los escenarios de prueba:

- **Grupo 1:**

Nombre Instancia	Tipo Instancia	Algoritmo Coloreo	Nº Pruebas	Tiempo Virtual de Simulación	Tiempo Arribos Promedio
instancia_prueba_a25_u015_p050	GRUPO	LRLF	3	1058-h	0,0110208-h
instancia_prueba_a50_u015_p075	GRUPO	LRLF	3	1103-h	0,01148073-h
instancia_prueba_a75_u015_p025	GRUPO	LRLF	3	1018-h	0,01059919-h
instancia_prueba_a100_u015_p025	GRUPO	LRLF	3	988-h	0,01028826-h
instancia_prueba_a200_u015_p025	GRUPO	LRLF	3	970-h	0,01010337-h
instancia_prueba_a200_u015_p075	GRUPO	LRLF	3	1053-h	0,01096027-h
DSJC125.1	DIMACS	LRLF	3	33747-h	0,35152133-h
R250.5	DIMACS	LRLF	1	82508-h	0,85945333-h

Nombre Instancia	Mejor Coloreo	Asignaciones Personal	Tiempo Coloración (Segundos)	Tiempo Ejecución (Segundos)
instancia_prueba_a25_u015_p050	8	18	0,001	140,531
instancia_prueba_a50_u015_p075	7	29	0,001	469,183
instancia_prueba_a75_u015_p025	23	52, 53, 54	0,002	378,352
instancia_prueba_a100_u015_p025	27	71	0,004	307,096
instancia_prueba_a200_u015_p025	48	135	0,011	1203,61
instancia_prueba_a200_u015_p075	15	103	0,01	2540,57
DSJC125.1	49	329	0,002	1977,44
R250.5	7	482	0,017	94240,9

Tabla 11 – Grupo 1 – Pruebas finales

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1 y las columnas 2 a 6 muestran los diferentes parámetros que conforman cada escenario de prueba. La segunda tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran la menor cantidad de colores obtenidos, las diferentes asignaciones de personal para dicha cantidad de colores, el tiempo de ejecución del algoritmo de coloreo para la prueba que dio la mejor asignación de personal y finalmente el mejor tiempo de ejecución de la prueba en su totalidad para el caso en que se obtuvieron los mejores resultados (mejor coloración y mejor asignación de personal).

Lo primero que salta a la vista es que, para diferentes coloraciones, aunque la cantidad de colores sea la misma, se puede ver que podemos llegar a obtener diferente cantidad de personal (celdas amarillas), lo que determina que la asignación es fuertemente dependiente de la combinación de actividades que se agrupan bajo una misma unidad. Además, pudimos

observar que el tiempo de ejecución tiende a aumentar a medida que se incrementa la densidad de aristas en el grafo (celdas verdes). Como la coloración se realiza sobre el grafo complemento, las instancias con mayor densidad contienen pocas aristas en su versión complementada, lo que tiende a disminuir la cantidad de vecinos que tienen los vértices, por lo tanto se tiende a generar coloraciones de pocos colores, donde cada color contiene un gran número de vértices. Como se puede observar en la tabla, la diferencia en el tiempo total de ejecución no la hace el algoritmo de coloración sino la ejecución de la metaheurística. Debido a que los escenarios de prueba fueron elaborados para ejecutar aproximadamente la misma cantidad total de eventos, supusimos que el incremento en el tiempo de ejecución estaba condicionado por las estructuras de datos que utiliza el motor de simulación, hecho que queda demostrado por el excesivo tiempo de ejecución de la prueba sobre la instancia R250.5 (celdas violetas), donde se puede observar que para dicha instancia se generan solamente 7 colores, 4 de ellos con 41, 43, 48 y 49 vértices (dato extraído del detalle de la prueba). Lo anterior muestra que éste framework de simulación tiene problemas para escalar a medida que se aumenta la cantidad de generadores de mensajes en el modelo (en la versión secuencial, sin incluir ningún mecanismo de computación paralela), o sea de unidades con mayor cantidad de actividades.

- **Grupo 2:**

Nombre Instancia	Tipo Instancia	Algoritmo Coloreo	Nº Pruebas	Tiempo Virtual de Simulación	Tiempo Arribos Promedio
DSJC1000.1	DIMACS	LRLF	1	24302-h	0,25313983-h
DSJC1000.1	DIMACS	EXTRACOL	1	24302-h	0,25313983-h
C2000.5	DIMACS	LRLF	1	59897-h	0,62392233-h
C2000.5	DIMACS	EXTRACOL	1	59897-h	0,62392233-h

Nombre Instancia	Mejor Coloreo	Asignaciones Personal	Tiempo Coloración (Segundos)	Tiempo Ejecución (Segundos)
DSJC1000.1	279	2456	0,131	15011
DSJC1000.1	261	2358	105,572	17830,8
C2000.5	195	5615	1,603	70465,3
C2000.5	187	5421	436,25	76834,2

Tabla 12 – Grupo 2 – Pruebas finales

La primera tabla muestra el nombre de las instancias utilizadas en la columna 1 y las columnas 2 a 6 muestran los diferentes parámetros que conforman cada escenario de prueba. La segunda tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 5 muestran la menor cantidad de colores obtenidos, las diferentes asignaciones de personal para dicha cantidad de colores, el tiempo de ejecución del algoritmo de coloreo

para la prueba que dio la mejor asignación de personal y finalmente el mejor tiempo de ejecución de la prueba en su totalidad para el caso en que se obtuvieron los mejores resultados (mejor coloración y mejor asignación de personal). Como ya mencionamos, el objetivo de estas pruebas es simplemente corroborar la propiedad enunciada en 3.2.4. Queda explícito en los resultados que para los casos que se utilizó EXTRACOL, se generaron coloraciones más pequeñas y el cálculo de la cantidad óptima de recursos también fue menor (celdas amarillas), si bien no podemos asegurar que esto se de en todos los casos ya que no elaboramos un conjunto más amplio de pruebas. Al generar coloraciones más pequeñas, se tienden a generar más conjuntos independientes con mayor cantidad de vértices lo que provoca, al igual que en el caso anterior, que el motor de simulación ejecute durante más tiempo (celdas verdes). Esto queda explícito en la siguiente tabla donde se extrajeron los resultados de la coloración para las 4 pruebas:

Cantidad de Vértices	#C. Indep. - EXTRACOL - DSJC1000.1	# C. Indep. - LRLF - DSJC1000.1	# C. Indep. - EXTRACOL - 2000.5	# C. Indep. - EXTRACOL - 1000.1
14	0	0	18	0
13	0	0	41	6
12	0	0	43	18
11	0	0	20	63
10	0	0	17	69
9	0	0	13	24
8	0	0	10	10
7	0	0	4	3
6	0	0	6	0
5	80	9	5	1
4	106	147	1	0
3	41	121	4	0
2	19	2	2	0
1	15	0	3	1

Tabla 13 – Grupo 2 – Detalles de coloración

Ambas ejecuciones de EXTRACOL generan mayor cantidad de conjuntos independientes de mayor tamaño, acumulando 824 vértices en el caso EXTRACOL sobre DSCJ1000.1 contra 633 vértices para LRLF sobre la misma instancia, solamente tomando en cuenta los conjuntos de 4 y 5 vértices y 1301 vértices en el caso EXTRACOL sobre C2000.5 contra 294 vértices para LRLF sobre C2000.5 solamente tomando en cuenta los conjuntos de 12, 13 y 14 vértices (celdas amarillas), o sea, EXTRACOL acumula en los primeros conjuntos más del 50% de la cantidad de vértices (o actividades) del grafo.

5.2.1 Resultados grupo módulo de taller

A raíz de los resultados obtenidos por el grupo de taller, mostramos a continuación una tabla comparativa que muestra las mejores asignaciones generadas por la solución del grupo (ver [43]) contra las mejores asignaciones obtenidas por nuestra solución (datos extraídos de la sección anterior). A continuación se muestran los resultados comparativos:

Nombre Instancia	k - LRLF	Mejor Asig.	Tiempo Ejecución (seg.)	Algoritmo Grupo Taller	k - Grupo Taller	Mejor Asig.	Tiempo Ejecución (seg.)
instancia_prueba_a25_u015_p050	8	18	140,531	Estable - 1	9	20	142,374
instancia_prueba_a50_u015_p075	7	29	469,183	Estable - 3	8	29	273,687
instancia_prueba_a75_u015_p025	23	52	378,352	Goloso - 6	26	67	727,046
instancia_prueba_a100_u015_p025	27	71	307,096	Goloso - 6	31	96	1379,096
instancia_prueba_a200_u015_p025	48	135	1203,61	Goloso - 6	56	199	4773,404
instancia_prueba_a200_u015_p075	15	103	2540,57	Estable - 3	18	177	2362,039

Tabla 14 – Resultados finales – Grupo de Taller

Esta tabla muestra el nombre de las instancias utilizadas en la columna 1; las columnas 2 a 4 muestran la cantidad de colores óptima a partir de la aplicación del algoritmo LRLF en cada instancia, la mejor asignación de personal obtenida para dichas coloraciones y el tiempo de ejecución total de procesamiento de cada escenario; las columnas 5 a 8 muestran el algoritmo del grupo de taller que mejor asignación de personal obtuvo, la cantidad de colores obtenidas por el algoritmo de la columna anterior, la mejor asignación asociada a dicha coloración y finalmente el tiempo de ejecución total de su solución.

En esta oportunidad volvimos a obtener mejores resultados que los generados por el grupo de taller, tanto desde el punto de vista de la coloración, como de la asignación de personal, salvo para la instancia **instancia_prueba_a50_u015_p075**, donde a pesar de generar una coloración más pequeña (7 colores contra 8), ambas soluciones encontraron la misma cantidad de personal (celdas amarillas). Salvo por la instancia donde se produce el empate, las demás soluciones cumplen la propiedad de asignar menor cantidad de personal cuando se tienen coloraciones de menor tamaño, o dicho de otra manera cuando la coloración produce menor cantidad de unidades (celdas verdes).

Con respecto al tiempo de ejecución, como nuestra solución depende de la cantidad de eventos que se generan en el simulador (a mayor cantidad, se producen mejores estimaciones, ver calibración simulador en Apéndice E) y de la estrategia de asignación inicial de personal, es difícil comparar desde el punto de vista temporal cuál es superior. De igual manera podemos ver que, los tiempos de ejecución obtenidos por nuestra solución para las instancias

instancia_prueba_a50_u015_p075 (469,183 segundos contra 273,687 segundos) e **instancia_prueba_a200_u015_p075** (2540,57 segundos contra 2362,039 segundos) fueron superiores a los del grupo de taller (celdas naranja) e inferiores para el resto de las instancias (celdas violetas).

5.3 Trabajo a futuro

En esta sección enumeramos algunas mejoras que pueden servir de inspiración en futuras investigaciones sobre este tema, y que por cuestiones de alcance no fueron realizadas en esta tesis.

Como vimos en 1.4.2 el funcionamiento de las unidades puede ser considerado de forma independiente (ver restricción 1.5). Debido a esta característica vemos que es posible aplicar técnicas de programación paralela para simular el funcionamiento de las distintas unidades de forma concurrente. De esta forma puede incluirse un análisis de medidas de desempeño tales como la medición del speedup, de la paralelicibilidad y de la eficiencia computacional [64]. Por su parte, OMNeT++ permite la ejecución paralela de modelos sin tener que hacer modificaciones de código, en particular es capaz de incorporar cualquier biblioteca estándar para programación paralela bajo el paradigma de comunicación de procesos mediante pasaje de mensajes (MPI), aunque el autor recomienda el uso de DeinoMPI.

Otra línea de trabajo a considerar en el futuro, podría ser la incorporación de la capacidad de especificar parámetros de calidad de servicio para la asignación de personal. Con esto nos referimos por ejemplo a configurar el porcentaje aceptable de abandonos de clientes, indicar el largo promedio de cola aceptable, entre otros. Esto permitiría realizar un análisis de impacto de como las distintas configuraciones condicionan la asignación de recursos.

Las actividades que consideramos en esta tesis tienen la característica de ser atómicas, es decir que son indivisibles y constan de un sólo paso. Otra alternativa sería poder incorporar además actividades que consten de varios pasos (procesos compuestos por múltiples actividades atómicas). Con esto se buscaría obtener soluciones que se acerquen más a la realidad que se plantea en las empresas. También como forma de hacer más real la solución se podrían considerar las capacidades distintas del personal para el cumplimiento de las diferentes tareas (existencia de tasas de servicio diferenciales para una misma actividad en distintos servidores).

La solución que implementamos considera que una asignación es factible si es exitosa para todas las repeticiones del experimento. Una opción más flexible podría consistir en que una solución sea considerada factible si la mayoría de las repeticiones son exitosas. Sería de esperar que con esta estrategia se puedan obtener mayor cantidad de soluciones factibles.

También consideramos que son útiles de implementar en soluciones futuras las modificaciones planteadas en el trabajo del Grupo de Taller (ver sección 6.3 en [43]).

Finalmente el objetivo más ambicioso sería la aplicación de la herramienta de optimización en empresas reales de forma de poder estimar el costo mínimo requerido para el correcto funcionamiento de la organización, evitar tiempos muertos y superposición de actividades, reconocer unidades sub y sobre dimensionadas y asegurar niveles aceptables de satisfacción para los clientes a partir de la restricción de los tiempos de espera.

Capítulo 6 – Conclusiones

En esta tesis estudiamos un problema correspondiente al área de gestión de la fuerza laboral que consiste en el dimensionamiento de empresas de alto porte, a partir de la asignación eficiente de actividades compatibles en unidades organizacionales y la asignación óptima de personal a cada una de ellas.

Para resolver este problema fue necesario dividirlo en dos fases. La primera se corresponde con la asignación de actividades compatibles en unidades, problema combinatorio identificado con la coloración de grafos, donde las actividades son representadas por los vértices, y las relaciones de compatibilidad por sus aristas. La segunda fase consiste en la asignación óptima de personal a las unidades obtenidas en la fase anterior, problema de naturaleza estocástica donde aplicamos aspectos de teoría de colas y métodos de Montecarlo.

En la fase de coloración realizamos un relevamiento de algunos de los algoritmos de mayor desempeño presentes en la literatura para resolver instancias de diversos tamaños. Del análisis de resultados para estos algoritmos podemos concluir que para encontrar coloraciones óptimas con grafos de menos de 1000 vértices el algoritmo que mejor se desempeño fue el LRLF, y para instancias mayores o iguales a 1000 vértices la mejor opción en cuanto a la calidad de la solución, sin considerar el tiempo de ejecución fue la variante CSR_ATS de EXTRACOL. Adicionalmente nuestros resultados fueron comparados con los mejores casos obtenidos por el grupo de taller, para dos de sus instancias (instancia_prueba_a200_u015_p025 e instancia_prueba_a200_u015_p075). Para ambos casos el LRLF generó mejores coloraciones y tiempos de ejecución que los algoritmos del grupo de taller.

Para la segunda fase se construyó una metaheurística que utiliza el resultado de la coloración obtenida a partir de alguno de los algoritmos seleccionados e implementa una búsqueda por bipartición para resolver el problema de asignación óptima de personal. Este algoritmo utiliza una función de aptitud que evalúa la factibilidad de la asignación actual a partir de la simulación de cada unidad y el chequeo de las restricciones de tiempo propuestas en el modelo matemático. Para verificar el correcto funcionamiento del simulador utilizamos como caso de estudio la simulación de una unidad con una sola actividad. Los resultados obtenidos fueron comparados con las fórmulas de Little (largo promedio de cola y tiempo promedio de permanencia en el sistema) y Erlang C (probabilidad de encolar un cliente), obteniendo un error relativo promedio menor al 1% para el caso en que se generaron 96000 eventos. Esto fue posible debido a las características del modelo que considera que los clientes demandan servicios siguiendo un proceso de Poisson, y el personal atiende las solicitudes de los clientes con tasa exponencial.

En las pruebas finales se consideraron instancias de 25, 50, 75, 100, 200, 250, 1000 y 2000 vértices. Estas instancias se clasificaron en dos grupos. En el primero (instancias hasta

250 vértices) se pudo observar que para diferentes coloraciones, a pesar de que la cantidad de colores era el mismo, se obtuvo diferente cantidad de personal (un resultado un tanto intuitivo) lo cual demuestra que la asignación depende de la combinación de actividades agrupadas bajo la misma unidad. Además se pudo observar que el tiempo de ejecución aumenta con el incremento del tamaño de las unidades, lo que nos llevó a pensar en una pérdida de performance del motor de simulación, hecho que confirmamos con la instancia R250.5. En el segundo grupo (instancias con 1000 vértices o más) las pruebas se enfocaron en demostrar empíricamente la propiedad enunciada en 3.2.4. Los resultados obtenidos de las pruebas fueron contundentes y permitieron comprobar que a medida que se encontraban coloraciones con menor cantidad de colores, también se minimizaba la cantidad de personal asignado por unidad. Este resultado también fue evidenciado en el análisis comparativo que hicimos contra los resultados del grupo de taller, ya que en todos los casos, con excepción de uno (caso de empate en la asignación para la instancia **instancia_prueba_a50_u015_p075**, con 29 personas en ambos resultados), obtuvimos mejores coloraciones y por ende menores asignaciones de personal.

Apéndice A – Colas en un banco

Por lo general a las personas no les gusta hacer cola en los bancos y en consecuencia con el fin de mejorar el nivel de servicio, las instituciones bancarias se concentran en averiguar cuestiones como:

- El número promedio de personas que esperan en el banco (es decir, la longitud de la cola).
- Cuanto tiempo los cajeros están inactivos (en relación al tiempo de trabajo).

Supongamos una situación hipotética en la que un cierto banco está dispuesto a emplear hasta 5 cajeros y como mínimo a 1.

Supuestos del modelo:

- La distribución del tiempo que les toma a los cajeros llevar a cabo una tarea es exponencial, con una media de 2 minutos y una desviación estándar de 5/4 de minutos.
- Eventualmente no existe un límite para el largo de la cola, ya que el banco tiene una superficie muy grande.
- Los clientes arriban con una distribución de Poisson, con una media de 25 clientes por hora.
- El servicio se brinda utilizando como disciplina de cola FIFO.
- Esto es una cola $M/M/x$ donde $1 \leq x \leq 5$.

Consideramos la siguiente notación:

- x: Número de cajeros (servidores)
- L_q: Largo promedio de la cola
- W: Porcentaje de tiempo inactivo de los cajeros (en espera)
- λ: Tasa de arribos promedio (25 clientes por hora)
- μ: 1/ (Tiempo promedio de atención de clientes) = $\frac{1}{2}\ cliente/min\ * 60\ min/hora$ = 30 clientes por hora
- ρ_s: Cantidad promedio de trabajo para cada servidor, por hora

Utilizamos los valores anteriores para calcular L_q y W.

$$\rho = \frac{\lambda}{\mu} = \frac{25}{30} = \frac{5}{6}$$

$$\rho_s = \frac{\lambda}{x\mu} = \frac{\rho}{x} = \frac{5}{6x}$$

p_0, L_q, W y el tiempo de inactividad se calculan usando las siguientes formulas:

$$p_0 = \left(\sum_{i=0}^{x-1} \frac{\rho^i}{i!} + \frac{\rho^x}{x!\,(1-\rho_s)} \right)^{-1}$$

$$L_q = \frac{\rho^x \rho_s p_0}{x!\,(1-\rho_s)^2} = \frac{5^{x+1} p_0}{6^{x+1} x!\, x \left(1 - \frac{5}{6x}\right)^2}$$

$$W = \frac{1}{\mu} + \frac{L_q}{\lambda} = \frac{1}{30} + \frac{L_q}{25}$$

$$Tiempo\ Inactivo = 1 - \rho_s = 1 - \frac{5}{6x}$$

Los resultados de las ecuaciones variando desde 1 a 5 cajeros son los siguientes:

$Cajeros$	ρ_s	$\frac{\rho^i}{i!}$	p_0	L_q	W	$Inactivo$
1	0,833	0,833	0,167	4,167	0,2	0,167
2	0,417	0,347	0,412	0,175	0,04	0,583
3	0,278	0,096	0,432	0,022	0,034	0,722
4	0,208	0,02	0,434	0,003	0,033	0,792
5	0,167	0,003	0,435	0,000	0,033	0,833

Tabla 15 – Colas en un banco – resultados de ecuaciones

Observando la tabla se puede concluir que 1 o 2 cajeros deberían ser contratados, ya que esto mantiene una cola corta, pero sin que los cajeros estén ociosos durante mucho tiempo [51].

Apéndice B – Problema de Buffon

Se tienen sobre un plano una serie de paralelas, siendo 2d la distancia entre cada dos de ellas. Se lanza al azar sobre el plano una aguja de longitud $2l < 2d$ y se trata de hallar la probabilidad de que la aguja corte a alguna paralela [65].

Sea x la distancia del centro de la aguja a la paralela más próxima, y ω el ángulo que forma con ella (ver Figura 6). Suponemos que x tiene una distribución uniforme en el intervalo $(0, d)$, es decir, que el centro de la aguja se toma al azar en un punto del segmento perpendicular a las paralelas. Suponemos también que ω es independiente de x y que tiene también una distribución uniforme en el intervalo $(0,\ \pi)$.

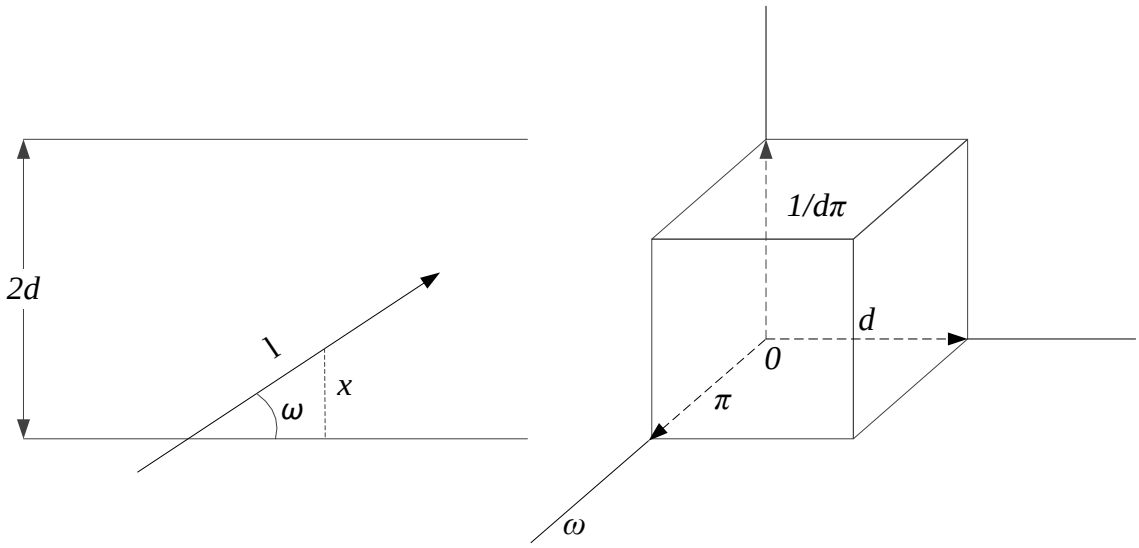

Figura 6 – Representación del problema de Buffon

Todas las posibles posiciones de la aguja están determinadas por los valores x y ω de los puntos del rectángulo de lados $(d,\ \pi)$. La función de densidad conjunta será:

$$\frac{I}{d\pi} \quad \text{para} \begin{cases} 0 \leq x \leq d \\ 0 \leq \omega \leq \pi \end{cases}$$

La aguja corta a alguna paralela cuando se cumple la condición:

$$x \leq l\ sen\ \omega$$

Luego la probabilidad se encontrará calculando la siguiente integral doble:

$$p = \iint_{x \leq l\ sen\ \omega} \frac{I}{d\pi} dx d\omega = \frac{I}{d\pi} \int_0^{\pi} d\omega \int_0^{l\ sen\ \omega} dx =$$

$$= \frac{I}{d\pi} \int_0^{\pi} d\omega [x]_0^{l\ sen\ \omega} = \frac{I}{d\pi} \int_0^{\pi} l\ sen\ \omega d\omega = \frac{I}{d\pi} [-\cos\omega]_0^{\pi} =$$

$$= \frac{I}{d\pi} [-cos\pi + \cos 0] = \frac{2l}{d\pi}$$

De la fórmula anterior, se deduce:

$$\pi = \frac{2l}{dp}$$

Siendo l y d constantes, y si de alguna forma calculáramos p, tendríamos determinado π.

Si lanzáramos la aguja un gran número de veces N y contáramos el número n de ellas en que la aguja corta a alguna paralela, la frecuencia relativa $\frac{n}{N}$ es una estimación de p, es decir:

$$\pi \sim \frac{2l}{d} : \frac{n}{N}$$

Si hacemos que la distancia entre las paralelas fuese el doble de la longitud de la aguja, o sea

$$2d = 4l, \qquad \frac{2l}{d} = I$$

Con lo cual

$$\pi \sim I : \frac{n}{N} = \frac{N}{n}$$

Es decir, dividiendo el número de lanzamientos por el número de veces que la aguja corta a alguna paralela, obtendríamos un valor aproximado del número π, con lo cual un problema determinístico se ha resuelto de forma experimental.

Por lo tanto, un problema analítico, la determinación del número π, que la geometría clásica resuelve por el método de los perímetros, se reduce a determinar una probabilidad, y después estimar esta por una frecuencia relativa.

Apéndice C – Motor de simulación

C.1. **Introducción**

OMNeT++ es un framework de simulación de eventos discretos modular y orientado a objetos, escrito en C++, que provee la infraestructura y herramientas necesarias para construir simulaciones sobre modelos (ver [66]). Algunas aplicaciones en las que puede utilizarse este framework son:

- Modelado de protocolos
- Modelado de redes de colas
- Modelado de diferentes redes de comunicaciones
- Validación de arquitecturas de hardware

En general permite modelar y simular cualquier sistema donde se pueda aplicar una aproximación de eventos discretos.

Un simulador de eventos discretos, es un sistema donde las transiciones de estados (eventos) ocurren en instancias de tiempo concretas, y el tiempo de ocurrencia de los eventos es instantáneo (cero), o despreciable (ver capítulo 4 del manual de OMNeT++ en [66]). El tiempo transcurrido dentro del simulador al ejecutar un modelo, se denomina tiempo de simulación o virtual ya que no tiene ninguna relación con el tiempo real o de CPU. Cada evento generado es almacenado en una estructura, comúnmente llamada FES (Future Event Set o Conjunto de Eventos Futuros).

En general este tipo de simuladores (como OMNeT++) funciona siguiendo el siguiente esquema expresado en forma de pseudocódigo:

Simulador_Eventos_Discretos

```
inicialización... ...      // Incluye la construcción del modelo y la inserción inicial de eventos en el FES
while not FES.Vacío y not simulador.Fin do
        evento = FES.Obtener_Primer_Evento();
        tiempo = evento.Tiempo_Ocurrencia();
        procesar_evento(evento);      // Puede incluir la inserción de nuevos eventos en el FES
                                      // o la eliminación de eventos existentes.
endwhile
finalizar simulación … …  // Registro de resultados, datos estadísticos, limpieza de estructuras, etc.
```

C.2. **Conceptos básicos**

El objeto más importante dentro de OMNeT++ son los **modelos** (o redes), objetos que utiliza el motor de simulación para llevar a cabo una simulación. Todos los modelos son

especificados utilizando un leguaje propietario llamado NED (Network Description o Descripción de Red).

Un modelo está compuesto por varias piezas constructivas más pequeñas llamadas **módulos** los cuáles interactúan unos con otros mediante el envío de mensajes (o eventos). Existen básicamente dos tipos de módulos, los **simples** y los **compuestos**. Un módulo compuesto puede contener múltiples módulos de ambos tipos, los cuales a su vez pueden estar interconectados a través de **compuertas** entre sí y/o con el módulo contenedor y se comunican mediante el envío de mensajes. No existe un límite en lo referente a niveles de composición.

El modelo, en definitiva es un módulo compuesto y ocupa la raíz en la jerarquía de composiciones.

Como ya se mencionó, las compuertas son las interfaces de comunicación de los módulos y pueden ser sólo de entrada, sólo de salida y bidireccionales. Una compuerta de entrada y una de salida pueden ser conectadas a través de una **conexión**. Las conexiones solamente pueden ser creadas dentro de un módulo compuesto.

Debido a la estructura jerárquica de los módulos, los mensajes viajan a través de una cadena de conexiones, partiendo y llegando siempre de un módulo simple.

Los módulos simples, son los módulos más bajos en la jerarquía ya que no pueden contener otros módulos. Sin embargo, a diferencia de los módulos compuestos, contienen el comportamiento que especifica cómo debe funcionar el modelo (contiene la implementación de los algoritmos en lenguaje C++).

Ambos tipos de módulos pueden contener **parámetros** de tipo string, numérico o booleano. Cómo los parámetros son representados como objetos, además de almacenar constantes, tiene la capacidad de actuar como fuente de números aleatorios con cierta distribución de probabilidad, como por ejemplo: uniforme, exponencial, etc.

A continuación presentamos una ilustración de los conceptos definidos previamente:

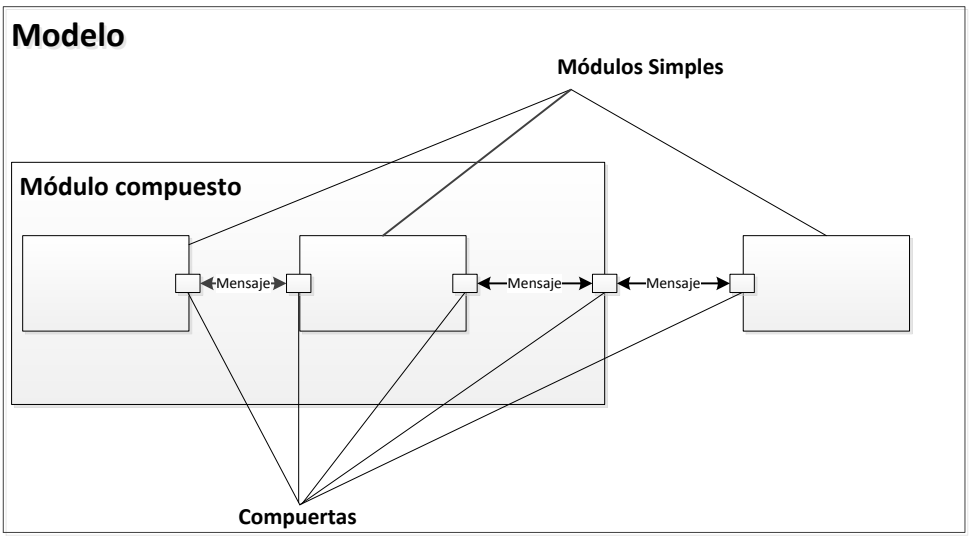

Figura 7 – Componentes de un modelo OMNeT++

Apéndice D – Archivos de parametrización

Los archivos con extensión .sim contienen los parámetros de configuración que permiten especificar el comportamiento del modelo implementado para ser ejecutado por el framework de simulación. Cada archivo de parametrización, representa un escenario diferente de prueba y debe contener al menos 10 líneas de configuración que son obligatorias. Todas las líneas del archivo deben respetar el siguiente formato:

clave = valor [//Comentario]

Dónde:

- **clave** es el nombre interno del parámetro utilizado dentro del simulador.
- **valor** es el dato que se quiere asociar a la clave.
- **//Comentario** es un metadato opcional y permite describir en lenguaje natural cualquier tipo de comentario. Éste metadato no es tenido en cuenta por la aplicación, es utilizado simplemente con fines informativos.

Como ya mencionamos, la estructura mínima del archivo debe contener 10 líneas, sin importar el orden, las cuales pasamos a detallar en el cuadro siguiente:

Clave	Descripción
simtime-scale	Escala de tiempo que utilizan las variables temporales. Valores posibles: [-18, -1].
sim-time-limit	Límite de tiempo de simulación. El formato es N-U, donde N es un natural y U es la unidad temporal (s - segundos, min = minutos, h = horas)
repeat	Repeticiones de cada experimento
Simulador.queue.sendingAlgorithm	Estrategia de envío de mensajes a los servidores. Valores posibles: priority, random, roundRobin, minDelay.
totalSources	Cantidad total de actividades
Simulador.source[*].interMeanArrivalTimes	Tiempos promedios de arribos
Simulador.server[*].serviceMeanRateTimes	Tiempos promedios de servicios
Simulador.client[*].tolerancesMeanTimes	Tolerancias de los clientes
Simulador.outputResidenceTimeUnit	Unidad de tiempo en la que se desea visualizar el tiempo promedio de espera en cola y el tiempo promedio de permanencia en el sistema. Valores posibles: s, min, h.
Simulador.outputVirtualTimeUnit	Unidad de tiempo en la que se desea visualizar el tiempo virtual de simulación. Valores posibles: s, min, h.

Tabla 16 – Estructura de los archivos de parametrización

Apéndice E – Calibración Simulador

Previo a la ejecución de los experimentos finales, realizamos una serie de pruebas para poder definir la cantidad de muestras (valor que se traduce en el tiempo de simulación límite) necesarias a generar, que nos permitan estimar correctamente la cantidad de recursos necesarios en cada unidad.

El escenario macro de prueba consta de una sola actividad, donde básicamente se variaron dos parámetros, el tiempo límite de simulación y el tiempo de arribos. El objetivo de variar el tiempo límite de simulación es la de generar mayor cantidad de muestras, y la intención de variar el tiempo de arribo fue la de modelar básicamente 2 tipos de escenarios, uno donde éste tiempo es mayor al de servicio y otro donde ocurre lo contrario. A continuación, mostramos en formato tabular los diferentes escenarios elaborados para calibrar la herramienta:

Límite tiempo	Nº escenario	Tiempo Arribo (segundos)	Tiempo Servicio (segundos)	Tolerancia (segundos)	Cantidad aproximada de eventos
1 día (8 horas)	1	12	20	25	2400
	2	36	20	25	800
1 semana (40 horas)	3	12	20	25	12000
	4	36	20	25	4000
1 mes (160 horas)	5	12	20	25	48000
	6	36	20	25	16000
6 meses (960 horas)	7	12	20	25	288000
	8	36	20	25	96000

Tabla 17 – Escenarios de calibración simulador

Para cada uno de los escenarios anteriores, se ejecutaron una serie de 3 pruebas, de los cuales se obtuvieron los siguientes resultados:

Nº escenario	GAP - Erlang C Promedio	% Error Erlang C Promedio	GAP – Tiempo Espera Promedio	% Error Tiempo Espera Promedio	GAP – Tiempo Permanencia Promedio	% Error Tiempo Permanencia Promedio	Tiempo Ejecución Promedio (segundos)
1	0,018618667	0,062111912	0,34506933	7,67434187	0,389981185	1,591993676	0,169333333
2	0,014830333	0,122795106	0,28747767	17,19116448	1,175538305	5,424165945	0,045
3	0,011787667	0,039323681	0,29428549	6,544909256	0,31599349	1,289958741	0,792666667
4	0,006209333	0,051413257	0,13722137	8,205838008	0,12567976	0,579911239	0,178333333
5	0,004469	0,014908594	0,16403125	3,648054908	0,192695844	0,786629141	3,147
6	0,002000333	0,016562753	0,07661198	4,581396622	0,246370541	1,13680234	0,668666667
7	0,001665	**0,005554444**	0,08212836	**1,826534805**	0,084097038	0,34330362	18,75933333
8	**0,001074**	0,008892716	**0,0656264**	3,924458531	**0,071394654**	**0,329429036**	3,959666667

Tabla 18 – Resultados preliminares para calibración simulador

La tabla anterior muestra, de izquierda a derecha el número de escenario de prueba, el error absoluto promedio para el cálculo de la probabilidad de encolar un cliente (Erlang C), el porcentaje de error (o error relativo) promedio para Erlang C, el error absoluto promedio para el cálculo del tiempo de espera en cola (Little), el porcentaje de error del mismo dato, el error absoluto promedio en el cálculo del tiempo de permanencia en el sistema (Little), y el porcentaje de error promedio para el mismo dato.

Un hecho que era de esperar es el incremento en el tiempo de ejecución a partir del aumento en el tiempo de simulación y el decremento del tiempo entre arribos, esto se traduce directamente en un aumento en la cantidad de eventos (o llegadas de clientes) generados en la simulación.

De la tabla anterior, se puede observar que para el escenario 8, en general, se obtienen los resultados promedio con menor error absoluto y relativo (números en rojo). Aunque éste escenario presenta un resultado subóptimo en el cálculo del error relativo promedio de Erlang C (0,0089%) y el mejor tercer valor en el error relativo promedio del tiempo de espera en cola (3,92), donde en ambos casos los mejores resultados se obtuvieron en el escenario de ejecución 7 (0,00565% y 1,82% respectivamente), el costo de ejecución en tiempo (3,96 segundos) es 4,74 inferior al del escenario 7 (18,76 segundos). De éste análisis consideramos aceptable los valores obtenidos por el escenario número 8, de lo cual, a priori podemos decir que se debe cumplir que el tiempo de arribo debe ser por lo menos 96000 veces menor al tiempo de simulación para generar como mínimo la cantidad de eventos considerada en el escenario 8. El resultado anterior, no determina cómo se debe aplicar la relación de tiempos para unidades con múltiples actividades. Cómo el tiempo límite de simulación es un parámetro global dentro de la simulación y aplica a todas las actividades por igual dentro de una misma unidad, para asegurar la creación de la cantidad de eventos establecida (96000) para cada una de ellas, el tiempo de simulación debe calcularse a partir de la siguiente relación:

$$tiempo_{límite} = Máximo(\{tiempo_{arribo} / \ tiempo_{arribo} \in Unidad_{simulada}\}) * 96000$$

De esta forma, aseguramos una cota inferior de 96000 eventos para todas las actividades contenidas en la unidad. Sin embargo, para no quedarnos solamente con éste resultado inicial, decidimos generar otra serie de pruebas para evaluar el tiempo real de ejecución y la calidad de los resultados a partir de la variación de la estrategia de asignación inicial y del cálculo del tiempo límite de simulación, siempre tomando en cuenta la misma coloración para cada instancia. Con esta nueva serie de pruebas se busca ajustar un poco mejor los parámetros de calibración. Los resultados obtenidos fueron:

Instancia	Estrategia Asignación	Eventos	Cantidad Óptima de Personal	Tiempo
instancia_prueba_a25_u015_p050	ADICIONAL	COTA MÍNIMA	18	955,647
	PROMEDIO	COTA MÍNIMA	18	432,956
	PEOR_CASO	PROMEDIO	18	234,705

	PROMEDIO	PROMEDIO	18	**147,531**
	RO_PROMEDIO	PROMEDIO	18	**141,811**
instancia_prueba_a50_u015_p075	ADICIONAL	COTA MÍNIMA	30	4221
	PEOR_CASO	PROMEDIO	30	887,038
	PROMEDIO	PROMEDIO	30	**499,235**
	RO_PROMEDIO	PROMEDIO	30	**500,838**

Tabla 19 – Ajustes de generación de eventos simulador

En la tabla anterior, la columna estrategia de asignación específica el método de cálculo inicial de personal a asignar en cada unidad, tal como se describió en el pseudocódigo **Calcular_Asignación_Inicial** en 4.2.3. La columna eventos determina el método de cálculo para el tiempo límite de simulación, donde el valor **"COTA MÍNIMA"** asegura un piso de 96000 eventos para cada actividad, mientras que **"PROMEDIO"** busca asegurar un promedio de ejecución de 96000 eventos por actividad ($\frac{total_{eventos}}{total_{actividades}} = 96000$).

Con respecto a los resultados, se puede observar que no hay diferencia en la cantidad de personal calculado, sin embargo vemos un decremento interesante en el tiempo total de ejecución de la prueba cuando se considera tanto el cálculo promedio de asignación inicial de personal (PROMEDIO y RO_PROMEDIO) como la cantidad promedio de eventos a ejecutar (PROMEDIO).

Para la primer instancia (instancia_prueba_a25_u015_p050) se ejecutó un caso más de prueba (estrategia = PROMEDIO, eventos = PROMEDIO) para visualizar la evolución completa del tiempo de ejecución.

De los resultados se desprende que los casos que mejor performance muestran son aquellos que consideran promedios para ambas dimensiones (estrategia y eventos), sin mostrar diferencias significativas de tiempos de ejecución entre las estrategias PROMEDIO y RO_PROMEDIO.

De todo lo anterior, concluimos que para la ejecución de pruebas se adoptará la estrategia PROMEDIO de asignación inicial de personal y para el cálculo del tiempo de simulación límite se utilizará el cálculo PROMEDIO de eventos a generar.

Apéndice F – Generador de archivos de parametrización

Para construir los diferentes archivos de parametrización utilizados por el framework de simulación para generar los casos de prueba, implementamos en C++ un generador de archivos por fuera de la solución de la metaheurística. Este generador toma de la entrada los diferentes valores que se deben especificar para la construcción de los archivos sim. El generador recibe como parámetros el nombre del archivo de parametrización con extensión y opcionalmente puede recibir el parámetro "promedio" que discutiremos más adelante en esta sección.

Los valores a ingresar al generador por la entrada estándar son los siguientes:

- El valor de escala correspondiente a la clave **simtime-scale** de la Tabla 16. Este valor es un entero negativo entre -18 y -1. El generador solicita este valor de la siguiente manera:
    - Ingrese la escala de tiempo (entero negativo entre -18 y -1):
- La unidad de tiempo para el límite de simulación correspondiente a la clave **sim-time-limit.** El valor para sim-time-limit se calcula dentro del generador, luego veremos en que consiste este cálculo. Los posibles valores a ingresar son "h", "m" o "s" que se traducen a horas, minutos o segundos. El generador solicita este valor de la siguiente manera:
    - Ingrese unidad de tiempo para el límite de simulación (h, m, s):
- La cantidad de repeticiones que se desean realizar del experimento. Este valor se corresponde al de la clave **repeat.** Este valor es un entero mayor a 0. El generador solicita este valor de la siguiente manera:
    - Ingrese las repeticiones para el experimento:
- La cantidad de actividades que se quieren simular. Este valor se corresponde al de la clave **totalSources.** Este es un entero mayor a 0. El generador solicita este valor de la siguiente manera:
    - Ingrese la cantidad de actividades:
- La unidad de tiempo en la que se quieren expresar los tiempos entre arribos, los tiempos entre servicios y las tolerancias de los clientes. Los correspondientes tiempos son calculados dentro del generador, más adelante veremos cómo. La unidad elegida se aplica a todos los tiempos anteriores. Los posibles valores a ingresar son "h", "m" o "s" que se traducen a horas, minutos o segundos. En el archivo de parametrización generado esta unidad es la unidad de los tiempos que aparecen para las claves **Simulador.source[*].interMeanArrivalTimes**, **Simulador.server[*].serviceMeanRateTimes** y

Simulador.client[*].tolerancesMeanTimes. El generador solicita el valor para la unidad de la siguiente manera:

- Ingrese unidad de tiempo para los tiempos entre arribos, los tiempos entre servicios y la tolerancia del cliente (h, m, s):

- La unidad de tiempo para visualizar el tiempo promedio de espera en cola y el tiempo promedio de permanencia en el sistema. La unidad ingresada es la correspondiente al valor para la clave **Simulador.outputResidenceTimeUnit.** Los posibles valores a ingresar son "h", "m" o "s" que se traducen a horas, minutos o segundos. El generador solicita el valor para esta unidad de la siguiente manera:
  - Ingrese unidad de tiempo para el tiempo promedio de espera en cola y el tiempo promedio de permanencia en el sistema (h, m, s):

- La unidad de tiempo para visualizar el tiempo virtual de simulación. La unidad ingresada es la correspondiente al valor para la clave **Simulador.outputVirtualTimeUnit.** Los posibles valores a ingresar son "h", "m" o "s" que se traducen a horas, minutos o segundos. El generador solicita el valor para esta unidad de la siguiente manera:
  - Ingrese unidad de tiempo para el tiempo virtual de simulación (h, m, s):

F.1. **Cálculos que se realizan en el generador**

Una vez que se ingresan por la línea estándar los valores detallados anteriormente el generador construye los archivos de parametrización y calcula los tiempos promedios entre arribos, los tiempos promedios entre servicio, las tolerancias de los clientes y el tiempo total de simulación.

Los tiempos promedios entre arribos se calculan mediante la generación de un número aleatorio entre los valores de las variables **inicio** y **fin** más un valor aleatorio entre 0 y 1. Las variables **inicio** y **fin** toman los siguientes valores:

$$inicio = 1;$$

$$fin = 2 + (rand(\,)\%\,99);$$

Es decir que para la variable **fin** se generan valores entre 2 y 100. Por lo que los valores de los tiempos promedios entre arribos se calculan así:

$$t_{arribo[i]} = inicio + rand(\,)\%fin + srand48(\,), \qquad con\ 0 \leq i \leq \#actividades$$

Los tiempos promedios entre servicios se calculan de la misma manera, es decir que se generan valores aleatorios entre **inicio** y **fin** más un valor aleatorio entre 0 y 1:

$$t_{servicio[i]} = inicio + rand(\,)\%fin + srand48(\,), \qquad con\ 0 \leq i \leq \#actividades$$

Los tiempos de tolerancia de los clientes deben ser mayores a los tiempos promedios entre servicios y para asegurarnos de ello, los valores para los tiempos de tolerancia los generamos como un número aleatorio entre el tiempo de servicio más un 15 % de holgura y **fin,** más un valor aleatorio entre 0 y 1:

$$tolerancia[i] = t_{servicio[i]} * 1{,}15 + rand(\,)\%fin + srand48(\,),$$

$$con\ 0 \leq i \leq \#actividades$$

Para calcular el tiempo de simulación total existen dos alternativas:

- Si se ingresó el parámetro opcional **promedio** como parámetro del generador esto indica que se desea realizar un cálculo promedio del tiempo de simulación que consiste en la suma de todos los tiempos de arribo multiplicado por 96000 y luego dividido entre la cantidad de actividades. En este caso estamos considerando el tiempo necesario para simular 96000 eventos por cada actividad y luego lo promediamos. El cálculo es el siguiente:

$$tiempo_{temp} += t_{arribo[i]} + 96000$$

$$tiempo = Techo\big((tiempo_{temp}/actividades) * \ Obtener\_Factor\_Conversion\big)$$

- Si se no se ingresó el parámetro **promedio** significa que se quiere realizar el cálculo del tiempo de simulación para el peor caso. En este caso lo que se hace para obtener el tiempo de simulación es multiplicar el valor correspondiente al tiempo de arribo más lento (valor de la variable **fin**) por 96000. El cálculo es el siguiente:

$$tiempo = Techo(fin * 96000 * Obtener\_Factor\_Conversion)$$

Finalmente la función **Obtener_Factor_Conversion** permite convertir las unidades de los tiempos promedios de arribos a las unidades ingresadas para el tiempo de simulación es decir, si los tiempos promedios de arribos estaban expresados en minutos y el tiempo de simulación quiere expresarse en horas, es necesario convertir la variable "tiempo" de minutos a horas.

Bibliografía

[1] M. Butler, «Business Execution and Workforce Management,» 2011.

[2] «The Human Capital Management Applications Report, 2005-2010,» AMR Research.

[3] «Contact Centre Workforce Management Market Report,» DMG Consulting, 2009.

[4] M. T. Procesos de Poisson, «Introduccion a la Investigacion de Operaciones - Facultad de Ingeniería,» 2012. [En línea]. Available: http://www.fing.edu.uy/inco/cursos/io/. [Último acceso: Agosto 2012].

[5] L. Euler, «Solutio problematis ad geometriam situs pertinentis,» *Academiae scientiarum Petropolitanae,* vol. 8, p. 128–140, 1741.

[6] J. J. Sylvester, «Chemistry and algebra,» *Nature,* vol. 17, p. 284–284, 1878.

[7] H. Heesch, «Untersuchungen zum Vierfarbenproblem,» Bibliographisches Institut, Mannheim, 1969.

[8] N. Robertson, D. Sanders, P. Seymourc y R. Thomas, «The Four-Colour Theorem,» *Journal of Combinatorial Theory, Series B,* vol. 70, nº 1, pp. 2-44, 1997.

[9] D. de Werra, «An introduction to timetabling,» *European Journal of Operations Research,* vol. 19, pp. 151-162, 1985.

[10] D. Rafaeli, D. Mahalel y J. Prashker, «Heuristic Approach to Task Scheduling: 'Weight' and 'Improve' Algorithms,» *International Journal of Production Economics,* vol. 29, nº 2, pp. 153-156, 1993.

[11] T. A. Feo y M. Resende, «Flight Scheduling and Maintenance Base Planning,» *Management Science,* vol. 35, nº 12, pp. 1415-1432, 1980.

[12] S. Firouzian y M. N. Jouy Bari , «Coloring fuzzy graphs and traffic light problem,» *The Journal of Mathematics and Computer Science,* vol. 2, p. 431–435, 2011.

[13] E. Beltrami y L. Bodin, «Networks and Vehicle Routing for Municipal Waste Collection,» *Networks,* vol. 4, p. 65–94, 1973.

[14] M. Gamache, A. Hertz y J. O. Ouellet, «A graph coloring model for a feasibility problem in monthly crew scheduling with preferential bidding,» *Computers and Operations Research,* vol. 38, nº 8, p. 2384–2395, 2007.

[15] N. Zufferey, P. Amstutz y P. Giaccari, «Graph colouring approaches for a satellite range scheduling problem,» *Journal of Scheduling,* vol. 11, nº 4, p. 263–277, 2008.

[16] G. J. Chaitin, M. A. Auslander, A. K. Chandra, J. Cocke, M. E. Hopkins y P. W. Markstein, «Register allocation via coloring,» *Computer Languages,* vol. 6, nº 1, p. 1981.

[17] A. Koster, «Frequency Assignment- Models and Algorithms, PhD thesis,» Universiteit Maastricht, Minderbroedersberg 4 6211 LK Maastricht, The Netherlands, 1999.

[18] M. R. Garey, D. S. Johnson y H. C. So, «An application of graph coloring to printed circuit testing,» *IEEE Transactions on Circuits and Systems,* vol. 23, pp. 591-599, 1976.

[19] P. Burzyn, F. Bonomo y G. Durán, «Computational complexity of edge modification problems in different classes of graphs,» *Electronic Notes in Discrete Mathematics,* vol. 18, pp. 41-46, 2004.

[20] E. Schaeffer, «Complejidad computacional de problemas y el análisis y diseño de algoritmos,» Universidad Autónoma de Nuevo León, San Nicolás de los Garza, México, 2011.

[21] M. R. Garey y D. S. Johnson, Computers and intractability: A guide to the theory of NP-completeness, San Francisco: W.H. Freeman and Company, 1979.

[22] S. Cook, «The complexity of theorem-proving procedures,» de *Proc. 3rd Ann. ACM Symposium on Theory of Computing Machinery*, New York, 1971, pp. 151-158.

[23] R. M. Karp, «Reducibility Among Combinatorial Problems,» de *Complexity of Computer Computations*, New York, R. Miller and J. Thatcher (Eds.), Plenum Press, 1972, pp. 85-103.

[24] M. R. Garey, D. S. Jhonson y L. Stockmeyer, «Some simplified NP-complete problems,» de *Proceedings of the Sixth Annual ACM Symposium on Theory of Computing*, New York, ACM, 1972, pp. 47-63.

[25] D. Brélaz, «New Methods to Color the Vertices of a Graph,» *Communications of the ACM*, vol. 22, nº 4, p. 251–256, 1979.

[26] D. W. Matula, G. Marble y J. D. Isaacson, «Graph Coloring Algorithms,» de *Graph Theory and Computing*, New York, Edited by R. C. Read, Academic Press, 1972, p. 109–122.

[27] S. H. Smith y T. A. Feo, «A GRASP for Coloring Sparse Graphs,» de *Working Paper*, Austin, Texas, Department of Mechanical Engineering, University of Texas, 1990.

[28] E. Malaguti y P. Toth, «A survey on vertex coloring problems,» Dipartimiento di Elettronica e Sistemistica University of Bologna Viale Risorgimento, Bologna, Italy, 2009.

[29] F. T. Leighton, «A graph coloring algorithm for large scheduling problems,» *Journal of Research of the National Bureau of Standards*, vol. 84, nº 6, p. 489–503, 1979.

[30] A. Hertz y D. de Werra, «Using Tabu Search Techniques for Graph Coloring,» *Computing*, vol. 39, p. 345–351, 1987.

[31] P. Galinier y J. K. Hao, «Hybrid evolutionary algorithms for graph coloring,» *Journal of Combinatorial Optimization*, vol. 3, nº 4, p. 379–397, 1999.

[32] C. A. Morgenstern, «Distributed coloration neighborhood search. In Johnson, D.S., Trick, M.A. (eds) Cliques,Coloring, and SatisfiabilitySatisfiability: 2nd DIMACS Implementation Challenge, 1993,» de *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, Providence, RI, American Mathematical Society, 1996, p. 335–358.

[33] N. Funabiki y T. Higashino, «A minimal-state processing search algorithm for graph coloring problems,» *IEICE Transactions Fundamentals*, Vols. %1 de %2E83-A, nº 7, p. 1420–1430, 2000.

[34] P. Hansen y N. Mladenovic, «Variable neighborhood search: Principles and applications,» *European Journal of Operatrational Research*, vol. 130, pp. 449-467, 2001.

[35] C. Avanthay, A. Hertz y N. Zufferey, «A variable neighborhood search for graph coloring,» *European Journal of Operational Research*, 2003.

[36] Q. Wu y J.-K. Hao, «Coloring large graphs based on independent set extraction,» LERIA, Université d'Angers, 2 Boulevard Lavoisier, 49045 Angers Cedex 01, France, 2011.

[37] Z. Lü y J. K. Hao, «A memetic algorithm for graph coloring,» *European Journal of Operational Research*, vol. 200, nº 1, pp. 235-244, 2010.

[38] A. Mehrotra y M. A. Trick, «A column generation approach for graph coloring,» *INFORMS Journal on Computing*, vol. 8, pp. 344-354, 1996.

[39] M. Chiarandini, G. Galbiati y S. Gualandi, «Efficiency issues in the RLF heuristic for graph coloring,» de *MIC 2011: The IX Metaheuristics International Conference*, Udine, Italy, 2011.

[40] M. Chams, A. Hertz y D. de Werra, «Some experiments with simulated annealing for coloring graphs,» *European Journal of Operational Research*, vol. 32, pp. 260-266, 1987.

[41] C. Felurent y J. A. Ferland, «Genetic and hybrid algorithms for graph coloring,» *Annals of Operations Research*, vol. 63, pp. 437-461, 1996.

[42] S. Balaji, V. Swaminathan y K. Kannan, «A Simple Algorithm to Optimize Maximum Independent Set,» *AMO – Advanced Modeling and Optimization*, vol. 12, nº 1, 2010.

[43] A. Descoins, F. Mangino, I. Badiola y M. D'Ambrosio, «Algoritmos de Estructuración de Empresas basados en Coloración de Grafos,» Facultad de Ingeniería - Universidad de la República, Montevideo, Uruguay, 2011.

[44] «Graph Coloring Instances,» [En línea]. Available: http://mat.gsia.cmu.edu/COLOR/instances.html.

[45] D. Johnson, Aragon, McGeoch y Schevon, «Optimization by Simulated Annelaing: An

Experimental Evaluation; Part II, Graph Coloring and Number Partitioning,» *Operations Research,* vol. 31, pp. 378-406, 1991.

[46] J. Culberson, «Culberson's Graph Coloring Research,» [En línea]. Available: http://webdocs.cs.ualberta.ca/~joe/Research/color.html.

[47] J. Black, C. Martel y H. Qi, «Graph and hasing algorithms for modern architectures : Design and performance,» *Proceedings of 2nd International Workshop on Algorithm Engineering, WAE 98',* pp. 37-48, 1998.

[48] G. Leon, «Universidad autónoma de Tamaulipas - Simulación de Sistemas,» [En línea]. Available: http://uat.gustavoleon.com.mx/SSU3%20-%20%20Teoria%20de%20Colas.pdf. [Último acceso: Agosto 2012].

[49] M. Moliterno, «Teoría de Líneas de Espera,» [En línea]. Available: http://www.geocities.ws/mdmoli/archivos/ioii4/unidad4.html.

[50] E. López Rubio, «Lenguajes y Ciencias de la Computación - Universidad de Málaga,» [En línea]. Available: http://www.lcc.uma.es/~ezeqlr/ios/Tema5.pdf. [Último acceso: Agosto 2012].

[51] «Blog de Andrew Ferrier - Applications Of Queueing Theory,» [En línea]. Available: http://www.andrewferrier.com/oldpages/queueing_theory/Henry/index.html. [Último acceso: Agosto 2012].

[52] V. B. Iversen, Teletraffic Engineering Handbook - Technical University of Denmark, 2001.

[53] J. Sethuraman y M. S. Squillante, Optimal Stochastic Scheduling in Multiclass Parallel Queues.

[54] O. Jouini, Z. Aksin y I. Dallery, «Queueing Models for Full-Flexible Multi-class Call Centers with Real Time Anticipated Delays,» *International Journal of Production Economics.*

[55] O. Jouini, A. Pot, G. Koole y Y. Dallery, «Real-Time Scheduling Policies for Multiclass Call Centers with Impatient Customers,» 2009.

[56] S. Ding, «Multi-class Fork-Join queues & The stochastic knapsack problem,» 2011.

[57] A. Sleptchenko, «Multi-class, multi-server queues with non-preemptive priorities».

[58] A. Sleptchenko, M. van der Heijden y A. van Harten, «Effects of finite repair capacity in multi-echelon, multi-indenture service part supply systems,» Faculty of Technology and management, University of Twente, 2000.

[59] A. van Harten y A. Sleptchenko, «On multi-class multi-server queueing and spare parts management,» 2000.

[60] Facultad de Ingeniería, «Material del curso de Métodos de Monte Carlo,» 2012.

[61] «Monte Carlo Simulation - Tossing a Coin,» [En línea]. Available: http://staff.argyll.epsb.ca/jreed/math7/strand4/4203.htm. [Último acceso: Enero 2013].

[62] Rubino, Gerardo; Tuffin, Bruno;, Rare Event Simulation using Monte Carlo Methods, INRIA, Rennes, France: John Wiley & Sons, Ltd., 2009.

[63] T. H. Cormen, C. E. Leiserson y R. L. Rivest, Introduction to Algorithms, MIT Press, 2000.

[64] S. Nesmachnow y S. Iturriago, «Facultad de Ingenieria- Curso de Computacion de alta performance,» 2012. [En línea]. Available: http://www.fing.edu.uy/inco/cursos/hpc/material/clases/Tema4-2012.pdf.

[65] V. Jimenez Diez de Artazcoz, «El método de Montecarlo y sus aplicaciones,» [En línea]. Available: http://es.scribd.com/doc/40917536/Metodo-de-Monte-Carlo-y-Sus-Aplicaciones.

[66] A. Varga, Manual de usuaruo OMNeT++, Budapest, Hungría: OpenSim Ltd., 2011.

[67] G. Koole, «Performance analysis and optimization in customer contact centers,» Vrije Universiteit, Department of Mathematics, Amsterdam, Holanda.

Tabla de Figuras

Índice de tablas