Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Optimización y simulación del no-wait Flow Shop Scheduling Problem

Optimization and simulation of the no-wait Flow Shop Scheduling Problem

Abel Delgado Falcón

La Laguna, 10 de septiembre de 2019

Doña **María Belén Melián Batista**, con N.I.F. 44.311.040-E Profesora Titular de Universidad adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutor y Don **Airam Expósito Márquez**, con N.I.F. 54.056.048-E personal investigador adscrito al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como cotutor

C E R T I F I C A (N)

Que la presente memoria titulada:

”*Optimización y simulación del no-wait Flow Shop Scheduling Problem*”

ha sido realizada bajo su dirección por D. **Abel Delgado Falcón**, con N.I.F. 79.060.797-E.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 10 de septiembre de 2019

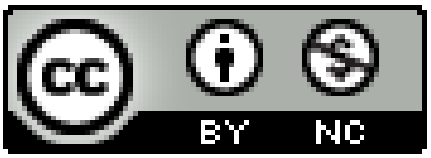
Agradecimientos

A mi familia y pareja por todo el apoyo recibido a lo largo de la carrera en los momentos más difíciles.

A mis tutores Mª Belén Melían Batista y Airam Expósito Márquez por toda la ayuda recibida a la hora de realizar el proyecto, en especial a Christopher Expósito Izquierdo por su paciencia conmigo y el acceso a su librería.

A mi tía allá donde estés.

Licencia



Resumen

Este trabajo tiene como finalidad la optimización y simulación de un problema de planificación de tareas conocido como 'flow shop scheduling', haciendo uso de la restricción 'no-wait', la cual impide que existan tiempos de espera entre la ejecución de una misma tarea en diferentes máquinas. Es un tipo especial de problema que surge a partir del 'job shop scheduling', añadiendo una restricción de orden para el modo en el que se realizan las tareas. El 'flow shop scheduling' se puede aplicar tanto al campo de la informática como al campo de la producción industrial u otros campos ajenos, como podría ser el paso de un paciente por una operación, donde tenemos tres tareas que se deben ejecutar en un orden estricto: el preoperatorio, la operación y el postoperatorio.

Para la fase de optimización, se han utilizado algoritmos heurísticos para intentar obtener una solución lo más aproximada posible a la esperada en el menor tiempo posible, ya que el problema es computacionalmente complejo y por ello se encuentra dentro del conjunto 'NP-Hard'.

De cara a la simulación, se ha implementado un pequeño programa basado en simulaciones por eventos discretos, que ha permitido conocer como se comporta la secuencia obtenida en la optimización bajo un cierto umbral de incertidumbre y con ello extraer datos para analizarlos.

Palabras clave: optimización, simulación, eventos discretos, búsqueda local, GRASP, VNS, 'flow shop', 'no-wait', 'Np-Hard'

Abstract

This work is aimed at optimizing and simulating a task planning problem known as 'flow shop scheduling', making use of the 'no-wait' restriction, which prevents waiting times between the execution of the same task on different machines. It's a special type of problem that comes from the 'job shop scheduling', adding an order restriction to the way in which tasks are performed. The 'flow shop scheduling' can be applied to the field of computer science and to the field of industrial production or other outside fields, such as the passage of a patient through an operation, where we have three tasks that must be executed in a strict order: preoperative, operation and postoperative.

For the optimization phase, heuristic algorithms have been used to try to obtain a solution as close as possible to the expected in the shortest possible time, since the problem is computationally complex and for that is inside the 'NP-Hard' set.

For the simulation, a small program based on simulations for discrete events has been implemented, which has allowed us to know how the sequence obtained in the optimization behaves under a certain threshold of uncertainty and with it extract data to analyze them.

Keywords: optimization, simulation, discrete events, local search, environments, GRASP, VNS, 'flow shop', 'no-wait', 'Np-Hard'

Índice general

Índice de Figuras

Índice de Tablas

Capítulo 1

Introducción

1.1. Contexto

La secuenciación, conocida como *scheduling*, se define como la asignación de recursos a tareas durante periodos de tiempo determinados, con objeto de optimizar una determinada medida de comportamiento [1]. En un sistema productivo *flow shop*[2] existe un conjunto de tareas que debe seguir un flujo unidireccional para ser debidamente procesado a través de diferentes fases o estaciones, donde cada una consta con una única máquina para procesar las diferentes operaciones que componen a la tarea.

Normalmente las máquinas adquiridas tienen características que difieren del resto, lo cual se traduce en diferentes tiempos de ejecución o diferente proceso de trabajos. Este tipo de sistemas es muy frecuente en la industria, donde para llevar a cabo el proceso de elaboración de un producto, se realizan un conjunto de operaciones unitarias necesarias para modificar las características de las materias primas. Sin embargo, esto no significa que el sistema no funcione en otros campos. Toda tarea que implique una planificación unidireccional puede ser modelada y tratada como un problema de secuenciación.

Actualmente las empresas tienen que ser muy competitivas para poder permanecer en el mercado y, por tal razón, estas deben ser flexibles al cambio ante cualquier imprevisto, aplicando herramientas con el propósito de volver sus procesos productivos mucho más eficiente. Esto conlleva a que las empresas necesiten invertir en investigación para encontrar la forma óptima de programar las tareas de tal modo que los tiempos de producción y/o fabricación se minimicen.

Para ofrecer una solución a este problema, lo ideal sería poder contar con una nueva metodología que permita optimizar el mínimo tiempo total

requerido para el procesamiento de todas las tareas, situación que afecta directamente al precio final del producto, ofreciendo ventajas competitivas en el mercado.

1.2. Objetivos

Este proyecto está orientado a resolver cualquier tipo de problema de secuenciación que cumpla las características de un *no-wait[3] flow shop*, mencionadas en el tercer capítulo de este documento.

El principal objetivo a cumplir es el desarrollo de un modelo de optimización que sea adaptable para cualquier instancia del problema recibida, sin importar número de tareas o máquinas. Se debe poder obtener una solución factible y que a su vez cumpla con los objetivos de un *flow shop*. Además, se desarrollará un modelo que permita simular el comportamiento en entornos con incertidumbre, haciendo uso de eventos discretos.

El modelo debe ser capaz de adaptarse a cambios, podríamos tener un secuencia de operaciones que deben pasar por tres fases: un preoperatorio, la operación y el postoperatorio; y necesitar adaptar la secuencia a cambios de última hora, como operaciones de emergencia [4].

Para realizar la fase de optimización, se analizarán y compararan los mejores algoritmos disponibles haciendo uso de heurísticas[5], las cuales van a permitir hallar soluciones en tiempo polinomial de la manera más eficiente posible.

Por último, al finalizar el trabajo, el objetivo de la fase de simulación es que se haya creado un modelo de simulación discreto[6] que permita realizar simulaciones con diferentes parámetros de entrada, obteniendo como salida datos que indiquen como se va a comportar una cierta permutación, obtenida en la fase de optimización, al llevarla a un entorno que contiene un umbral de incertidumbre.

1.3. Motivación

El estudio de algoritmos en la búsqueda de soluciones para entornos de flujo regular resulta de interés, dado que en la mayoría de las fábricas, los productos que se elaboran siguen un determinado orden de procesamiento y son introducidos en las máquinas de una manera ya predefinida. Por ello, tiene gran importancia la obtención de un buen algoritmo que sea capaz de encontrar un resultado óptimo o al menos, cercano a él.

Además, teniendo en cuenta la naturaleza de estas operaciones, tiene sentido aprovechar la optimización matemática y la simulación para mejorar los procesos. Así pues, se espera tener la capacidad de realizar ciertas recomendaciones a partir de los resultados que se observen en la simulación.

Actualmente es muy frecuente utilizar aplicaciones software en el campo de la optimización. Ciertos programas se ocupan de la totalidad de las tareas de optimización de bastantes empresas. Los problemas de programación de tareas aparecen con frecuencia en la vida real y se basan en la organización de una serie de trabajos que comparten un conjunto limitado de recursos cumpliendo diversas restricciones. El problema consiste en la asignación de dichos recursos a las tareas con el objetivo de optimizar un determinado criterio, normalmente relacionado con el tiempo o coste del proceso.

Estos problemas son considerados como muy complejos, por lo que su resolución mediante métodos exactos resulta muy costosa. Para ello se utilizan técnicas heurísticas que simplifican la búsqueda sin necesidad de analizar una a una todas las posibles soluciones.

1.4. Estructura de la memoria

Este documento esta compuesto por una estructura que hace referencia al orden de trabajo realizado para la elaboración del proyecto.

Este capítulo tratado conlleva una breve descripción general de la propuesta del TFG, donde se ha relatado un contexto general del problema y los objetivos que se persiguen alcanzar.

En el siguiente capítulo de la memoria, el número dos, se introducen conceptos básicos y el estado del arte de las herramientas, técnicas y tecnologías empleadas, separados por los dos temas principales tratados en el TFG (optimización y simulación).

El tercer capítulo describe el escenario general de un problema *flow shop* junto al modelo matemático, utilizado tanto en simulación como optimización.

El cuarto capítulo describe la propuesta de solución que se plantea con el binomio optimización-simulación para alcanzar los objetivos del proyecto, entrando más en detalle sobre cómo se programa el modelo.

El quinto capítulo describe la experimentación realizada con el problema construido. Esta se lleva a cabo para demostrar la utilidad para la que está planteada el modelo matemático, haciendo multitud de pruebas con diferentes entradas y parámetros.

Los capítulos seis y siete recogen las conclusiones obtenidas tras el desarrollo completo y las líneas futuras posibles de cara a mejoras del proyecto.

Por último, el octavo capítulo recoge el presupuesto de elaboración del proyecto.

Capítulo 2

Conceptos básicos: optimización y simulación

2.1. Optimización

Cuando hablamos de optimización de un problema, al final lo que queremos es elegir el mejor elemento de los disponibles en un conjunto[7]. Normalmente, la función objetivo $f(x)$ nos permite determinar cuál de los elementos disponibles es mejor en función a si queremos maximizar o minimizar. Generalmente, la optimización incluye el descubrimiento de los mejores valores de una función objetivo dado un dominio definido. Estos problemas son muy comunes para realizar el cálculo de la cantidad de cierto elemento que necesitas para acotar una superficie, ver el máximo o el mínimo de un área o un volumen, etc.

Los problemas de optimización normalmente vienen representados por la siguiente estructura:

$$f : A \to R$$

Donde si buscamos minimizar:

- $\forall x_0 \in A \to f(x_0) \geq f(x)$

O si buscamos maximizar:

- $\forall x_0 \in A \to f(x_0) \leq f(x)$

donde x_0 y x son elementos de A, un subconjunto del espacio *euclídeo* R^n

Muchos de los problemas de optimización pueden ser modelados mediante esta notación, dónde A es un subconjunto del espacio R con grado n y delimitado por un conjunto de restricciones, igualdades o desigualdades. Típicamente se denomina al dominio de A como espacio de búsqueda de elementos mientras que los elementos que se encuentran en su interior son candidatos de solución o soluciones factibles.

En el caso más sencillo, se tendrá un único objetivo (maximizar o minimizar) pero, en ciencias e ingeniería se dan, en bastantes ocasiones, problemas que requieren la optimización simultánea de más de un objetivo, son las llamadas funciones multiobjetivo.

2.1.1. Técnicas de optimización

Las técnicas de optimización[8] se han convertido en una poderosa herramienta para el diagnóstico y solución de múltiples problemas complejos presentes en las ciencias, que aportan elementos importantes en la toma de decisiones.

- **Algoritmos exactos:** son aquellos que terminan en un determinado momento, es decir, tienen un número finito de pasos. A destacar entre los métodos más usados encontramos el algoritmo *Símplex*, diseñado para resolver problemas de programación lineal sin restricciones en el número de variables. Las variantes del este fueron diseñados para la programación cuadrática u optimizaciones de redes y problemas combinatorios.

- **Métodos iterativos:** se utilizan para resolver los problemas que quedan fuera del alcance del algoritmo *Símplex* y resuelven normalmente problemas de programación no lineales. Dependiendo del tipo de evaluación, se pueden diferenciar entre *Hessianos, gradientes o valores de función*. En algunos casos, el coste computacional puede ser excesivamente alto.

- **Heurísticas:** se utilizan para proveer soluciones aproximadas a algunos problemas de optimización complejos cuyo cálculo es altamente costoso. Se puede basar en la experiencia propia del individuo y en la de otros para encontrar la solución más factible del problema. Hacen referencia a un conjunto de métodos o reglas que indican cuales son las acciones idóneas que pueden generar soluciones al problema. Es usado como un mecanismo que, a menudo, es muy efectivo para encontrar las soluciones, pero no se garantiza en todos los casos.

2.1.2. Espacio de búsqueda

En el campo de la optimización, se conoce como espacio de búsqueda al dominio de la función que está siendo optimizada. En el caso de algoritmos de búsqueda como los que se tratan en este trabajo, el espacio corresponde a todas las posibles soluciones factibles del problema, sin necesidad de obtener la mejor.

En la figura 2.1 tenemos una representación del espacio en dos dimensiones de una función cualquiera, donde los máximos y mínimos relativos son posibles buenas soluciones que cualquier algoritmo de búsqueda podría obtener, mientras que la solución ideal a la que querríamos llegar es el mínimo o máximo absoluto, dependiendo de si el problema busca minimizar o maximizar respectivamente.

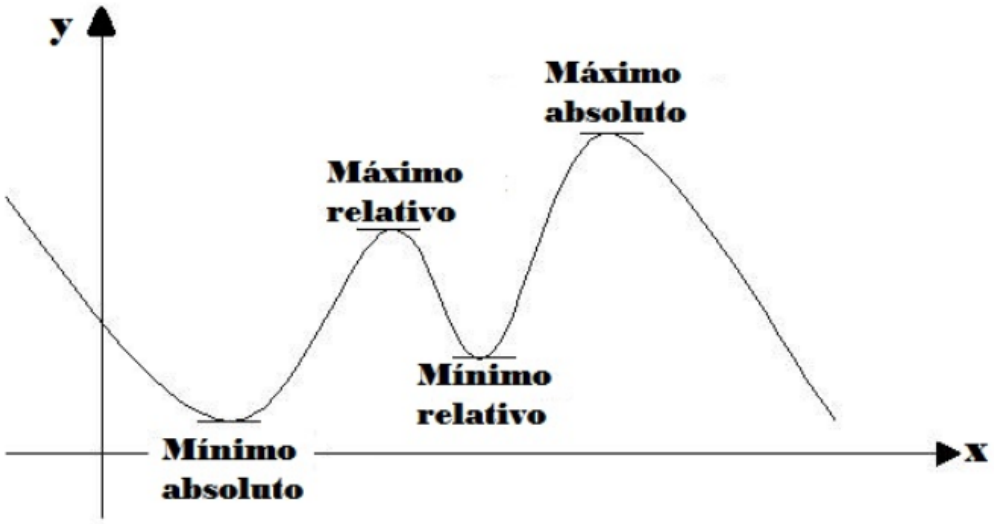


Figura 2.1: Espacio de búsqueda de un problema de optimización

2.1.3. Definición de las estructuras de entorno

Para poder encontrar soluciones en el espacio de búsqueda surgen las estructuras de entorno, las cuales asocian a cada solución del problema un conjunto de soluciones cercanas o vecinas según un criterio determinado. Generalmente, las estructuras de entorno se definen a través de movimientos que, aplicados sobre una solución, suministran otras.

2.1.3.1. Búsqueda local

En la búsqueda local[9] se parte de una solución inicial que generalmente es aleatoria y se hacen pequeños cambios a través de operadores que

modifican la solución hasta alcanzar un estado desde el cual no se puede alcanzar ningún estado mejor. Estos estados se conocen comúnmente como óptimo local y global (Ver Figura 2.2).

Normalmente se empieza con una solución inicial y el objetivo es ir realizando búsquedas en el vecindario, realizando movimientos cortos, con el objetivo de encontrar una solución mejor. Si se encuentra, se reemplaza la solución actual por la nueva y continua con el proceso, hasta que no se pueda mejorar la solución actual. Este proceso de búsqueda ayuda a encontrar óptimos locales de la solución, pero nunca garantiza que sea la mejor solución de las posibles (óptimo global).

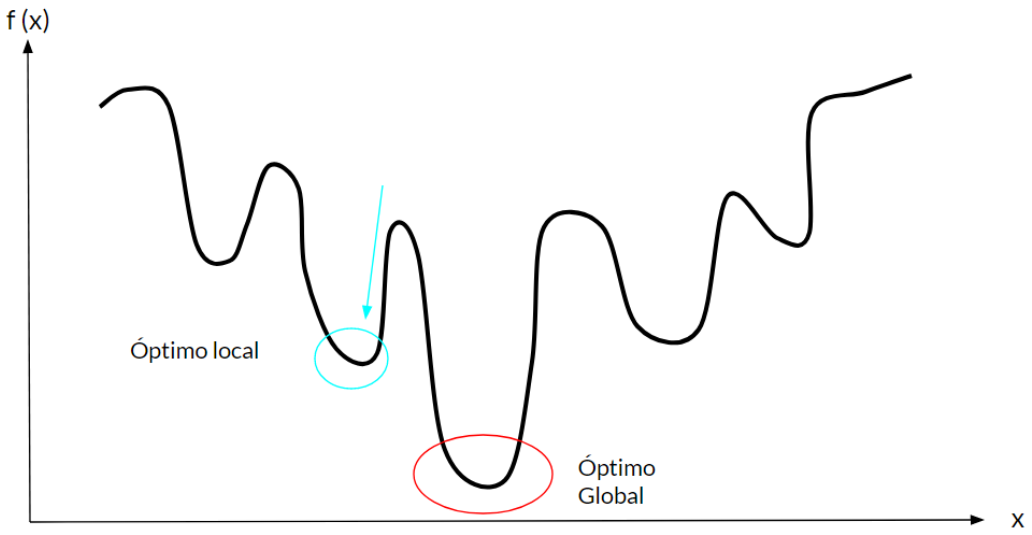


Figura 2.2: Óptimos de una búsqueda local

Dicha búsqueda puede utilizar diversos mecanismos de exploración, teniendo como lo más importantes a *Greedy* y *First better*(primero mejor).

- **Greedy**

  En una búsqueda *Greedy* se recorren todas los candidatos a los que se puede llegar a partir de una solución realizando un único movimiento (Ver Figura 2.3). Tras recorrerlos todos y evaluarlos, el mejor pasará a ser la nueva solución en base a la función objetivo.

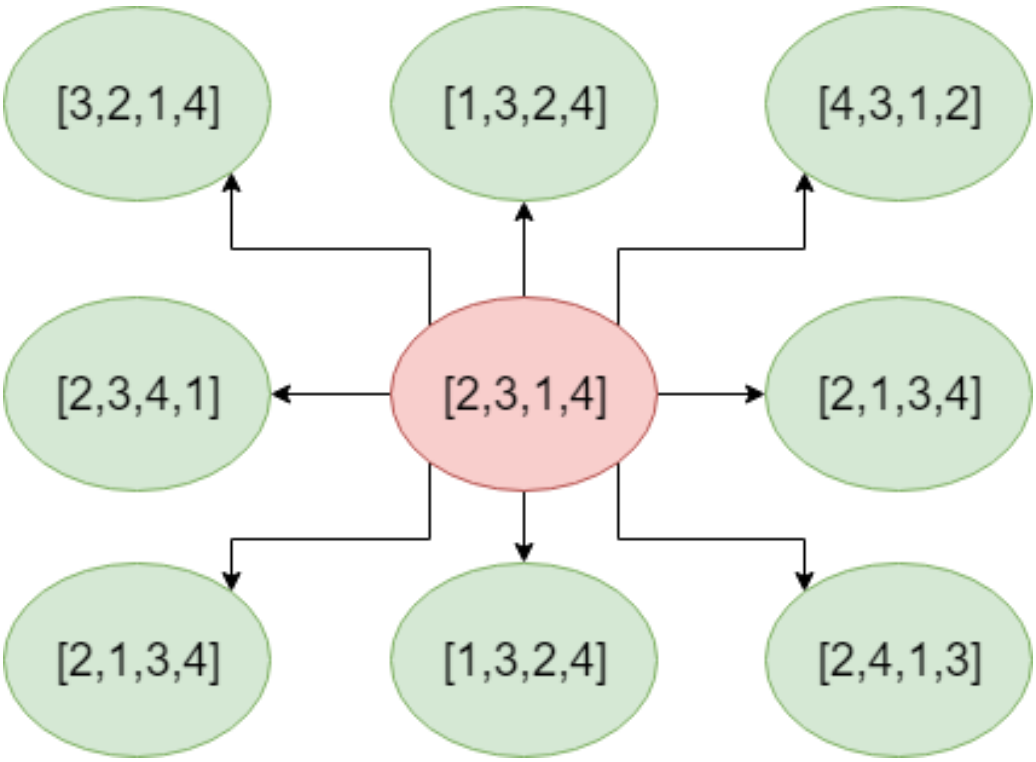


Figura 2.3: Ejemplo de búsqueda Greedy

El criterio de selección no es importante en este caso puesto que el algoritmo siempre recorrerá todas las posibilidades.

- **First better**

  En una búsqueda *primero mejor* se recorren todas los candidatos a los que se puede llegar a partir de una solución realizando un único movimiento tal y como se hace en *Greedy*, salvo que si en cualquier momento de la búsqueda se encuentra un candidato que es mejor que la solución actual, entonces se para y no se recorren el resto de candidatos, pasando a ser el encontrado la solución actual y ahorrando el coste de visitar a todos los vecinos posibles, siempre que se encuentre antes de visitarlos a todos.

  El criterio de selección es muy importante para este tipo de búsquedas, pues si se selecciona un buen candidato desde un principio al final se traduce en ahorro del tiempo de cómputo.

2.1.3.2. Movimientos

Cuando queremos visitar un vecino estamos realizando un movimiento dentro del entorno, el cual puede ser de tipo *Greedy* o *First better* como ya hemos visto. El tipo de movimiento escogido a la hora de mejorar la solución influye en mayor o menor medida dependiendo del problema tratado.

Para problemas de secuenciación de tareas, pueden considerarse los movimientos de reasignación, intercambio de dos o más tareas o inveersión de tareas.

- **Reasignación:** tomar una tarea que se encuentre asignada en una determinada posición y asignarla en otro diferente y que sea posible.
- **Intercambio:** escoger como mínimo dos tareas diferentes e intercambiar sus posiciones. Cuando el número de tareas escogidas es superior a dos, el método de intercambio puede variar, afectando a las soluciones que se pueden encontrar.
- **Inversión:** tomar un subconjunto de tareas e invertir las tareas que lo contienen sin alterar las posiciones del resto de las tareas, es decir, se intercambian entre ellas.

2.2. Simulación

Cuando hablamos de simular algo, en realidad lo que estamos haciendo es intentar imitar el funcionamiento de un sistema durante un intervalo de tiempo determinado, que parte de un estado inicial, dependiendo de los datos de entrada, y proporciona cierta información respecto al modelo o la estructura que se está simulando. De esta manera se pueden intentar prever errores antes de implementar el modelo en un entorno real.

Existen dos tipos de modelos orientados a las simulaciones:

- **Mental:** es el que utilizamos en nuestro día a día. Se utiliza antes de realizar una toma de decisión de cualquier tipo y que implica un resultado.
- **Computacional:** cuando el modelo ha sido especificado, se pasa a la implementación de un simulador. Para ello, se hace uso de paquetes de programas que contengan las rutinas necesarias para realizar el simulador.

2.2.1. Etapas para realizar la simulación

- **Definición del sistema:** se establece los elementos que se deben imitar, se especifican los objetivos del modelamiento y se conoce el objetivo de cada elemento.
- **Definición del modelo:** se define y construye el modelo con el cual se espera obtener unos resultados determinados. Es necesario formular bien todas las variables que componen el modelo y sus respectivas relaciones lógicas.
- **Definición de datos:** es muy importante definir con claridad y exactitud los datos que se van a utilizar en la simulación, siendo posible un mal resultado si los datos utilizados son incorrectos.
- **Traslación del modelo:** cuando se tiene un modelo correctamente definido, hay que decidir si se va a utilizar una aplicación diseñada para simular o utilizar un lenguaje de programación para codificar la propia simulación
- **Verificación:** hay que comprobar que el modelo cumple con todos los requisitos de diseño que se han ido realizando a lo largo del proyecto.

- **Validación:** se debe valorar las diferencias entre el funcionamiento del simulador y el sistema real que se está tratando de simular.
- **Experimentación:** consiste en diseñar una batería de pruebas que nos permita extraer unos resultados y a partir de estos realizar un análisis de los datos obtenidos.
- **Interpretación de resultados:** se deben analizar los datos obtenidos a través de la simulación y tomar las respectivas decisiones.
- **Documentación:** es necesaria una correcta documentación para ayudar a futuros modelos que puedan ir surgiendo a partir de simulaciones parecidas.

2.2.2. Ventajas y desventajas de la simulación

A pesar de que las herramientas de simulación proveen un potente entorno para realizar pruebas antes de llevarlo a un sistema real, tiene sus ventajas y desventajas dependiendo de la situación:

- **Ventajas**
  - Permite analizar y probar escenarios de un sistema dado.
  - Al ser resultados numéricos se los puede dar un grado de validación y/o precisión.
  - El estudio de un sistema en la simulación permite comprenderlo mejor y aportar datos que puedan perfeccionar el sistema más adelante.
  - Hay cambios en el sistema continuamente que podrían afectar al sistema real, pudiendo comprobarlos antes en la simulación.
- **Desventajas**
  - No se obtienen soluciones óptimas y por tanto la solución es inexacta debido a la incertidumbre.
  - Al simular haciendo uso de datos aleatorios no se puede medir el grado de precisión exactamente, ya que suele ser inexacto.
  - Desarrollar y validar un sistema es costoso a nivel de tiempo y de computación.
  - Todos los modelos desarrollados son, normalmente, únicos.

2.2.3. Simulación por eventos discretos

Una simulación basada en eventos discretos se utiliza en el campo de la informática para el modelado dinámico de sistemas. Está principalmente caracterizado por un control del tiempo que le permite avanzar en intervalos variables dependiendo del tipo de planificación de los eventos. Este tipo de sistemas tiene como requisito que las variables no cambien el comportamiento durante un intervalo simulado.

Para llevar la simulación a este proyecto, se ha escogido eventos discretos debido a la naturaleza del modelo del problema. Algunos de los conceptos principales que hay que tener en cuenta cuando se realiza este tipo de simulación son:

- **Tiempo de simulación:** el valor del tiempo que el simulador puede avanzar a una velocidad superior a la habitual de un reloj común, evolucionando así el estado de un sistema de forma acelerada.
    - **Simulación en tiempo acelerado:** el avance del tiempo de simulación es mayor de un segundo por cada segundo de tiempo real.
    - **Simulación en tiempo real:** el avance del tiempo de simulación exactamente de un segundo por cada segundo de tiempo real.
- **Entidad:** el sistema se modela sobre la base de entidades, formando un sistema compuesto. Pueden llegar a existir múltiples instancias de un tipo de entidad.
- **Evento:** suceso que hace cambiar las variables de estado del sistema. El tiempo de simulación permanece fijo siempre durante el procesamiento de un nuevo evento. Un evento puede pertenecer a una entidad o actor en el sistema.
- **Actividad:** secuencia de eventos pertenecientes a una entidad que cierran un ciclo funcional. A diferencia de un evento una actividad se desarrolla dentro de un intervalo de tiempo de simulación no puntual. Además, debe existir algún canal de comunicación entre los distintos procesos.

2.2.4. Estado del arte

Existen multitud de tipos de simulación actualmente (discreta, continua, combinación discreta-continua, determinística, dinámica, etc) y a su vez multitud de programas en el mercado para satisfacer todas las necesidades. Algunos de los más importantes en el mundo laboral y científico son:

- **Anylogic[10]:** es una herramienta que incluye todos los métodos de simulación más comunes. Incluye un lenguaje de modelado gráfico y también permite que los usuarios puedan ampliar los modelos de simulación utilizando Java. Los modelos se pueden basar en cualquiera de los principales paradigmas de simulación de modelado.
- **Simio[11]:** programa de simulación completamente orientado a objetos, siendo estos y sus procesos definidos en un entorno gráfico sin necesidad de programación. Permite introducir riesgos en la programación de una producción mediante su sistema.

- **Promodel[12]:** es una tecnología de simulación de eventos discretos que se utiliza para planificar, diseñar y mejorar sistemas de fabricación, logística y otros sistemas operativos nuevos o existentes. Permite representar con precisión los procesos del mundo real con el fin de llevar a cabo un análisis predictivo sobre los cambios potenciales

- **ServiceModel[13]:** Software que permite tanto simular como optimizar cualquier tipo de sistema de manufactura, ensamble, balanceo de líneas, logística y otras aplicaciones. Permite programación directa aunque no es prescindible. Contiene un módulo de optimización que evita el uso de prueba-error.

- **Powersim Studio[14]:** posee un lenguaje de modelado gráfico que da rapidez a la construcción de modelos y facilidad para la revisión de los mismos. Los ámbitos de aplicación de este software se hallan en el área empresarial. Son principalmente modelos financieros, gestión de clientes, análisis de producción, recursos humanos y desarrollo de nuevos productos. Carece de simulación en entornos 3D.

2.3. Optimización vs. simulación

La optimización y la simulación son disciplinas que sirven para ayudar a comprender, resolver y obtener soluciones de problemas matemáticos. Es por ello que ambos comparten ciertas similitudes:

- Sirven para resolver problemas independientemente de su tamaño de variables, es decir, no están limitados a problemas concretos.

- Ambos consiguen respuestas exactas o muy cercanas al óptimo global.

- Surgen a partir de modelos matemáticos.

- Parten de modelos reales.

- Aceptan cualquier tipo de planteamiento de problema, desde lineales hasta aleatorios o inexactos.

- Permiten el estudio de un escenario con restricciones.

Sin embargo, tal y como comparten características, son disciplinas que analizan el problema de maneras diferentes, y por tanto tienen sus diferencias.

- La simulación emula el comportamiento de un modelo real a través del tiempo repetidas veces y obtiene resultados para realizar medias métricas y extraer datos, mientras que la optimización ocurre en un momento estático y obtiene la mejor solución del modelo y, por tanto, realiza un análisis específico del problema.
- Dependiendo de la rigidez del modelo utilizado, durante una simulación el modelo puede ser contradictorio, mientras que en la optimización el modelo podría no obtener un resultado óptimo.
- De cara al análisis futuro, la optimización no permite obtener una mejor forma de resolver el modelo, ya que no se tiene en cuenta el tiempo, mientras que una simulación permite mejorar a largo plazo ineficiencias en el modelo.
- La simulación utiliza unas pautas para construirse a sí misma que describen el proceso, pero no hace que los resultados sean realmente óptimos, mientras que la optimización desde el principio tiene en cuenta todas las restricciones del modelo y su objetivo es encontrar el mejor óptimo.
- Una simulación que se ejecuta con incertidumbre permite que, a medida que se recogen datos, conocer como va a evolucionar con el tiempo, mientras que la optimización se aleja de su óptimo obteniendo resultados erróneos.

Capítulo 3

Problema

3.1. Descripción

El *flow shop* es un problema que surge a partir del *Job Shop Scheduling*. En este último, no existe un orden estricto de ejecución para todas las operaciones que deben realizarse en todos los trabajos, es decir, no tiene un flujo unidireccional como ocurre en el problema a tratar en este proyecto (ver figuras 3.1 y 3.2).

El problema base consta con las siguientes características proporcionadas por cualquier instancia:

- Entrada:
  - Un número n de tareas a realizar.
  - Un número m de máquinas por las que cada tarea realiza una operación.
  - Los tiempos de ejecución de cada operación o correspondiente a la tarea n en cada máquina m
- Salida:
  - Una permutación de las tareas n que cumpla todas las restricciones del problema y minimice el *makespan*

Cada tarea, se ejecutará tantas veces como máquinas tenga el problema, pudiendo ser el tiempo de ejecución diferente en cada máquina. A cada ejecución de una tarea en una determinada máquina se le conoce como operación.

La ejecución de una tarea debe realizarse en un orden estricto, el cual le diferencia del *job shop*. La primera operación se ejecuta en la primera

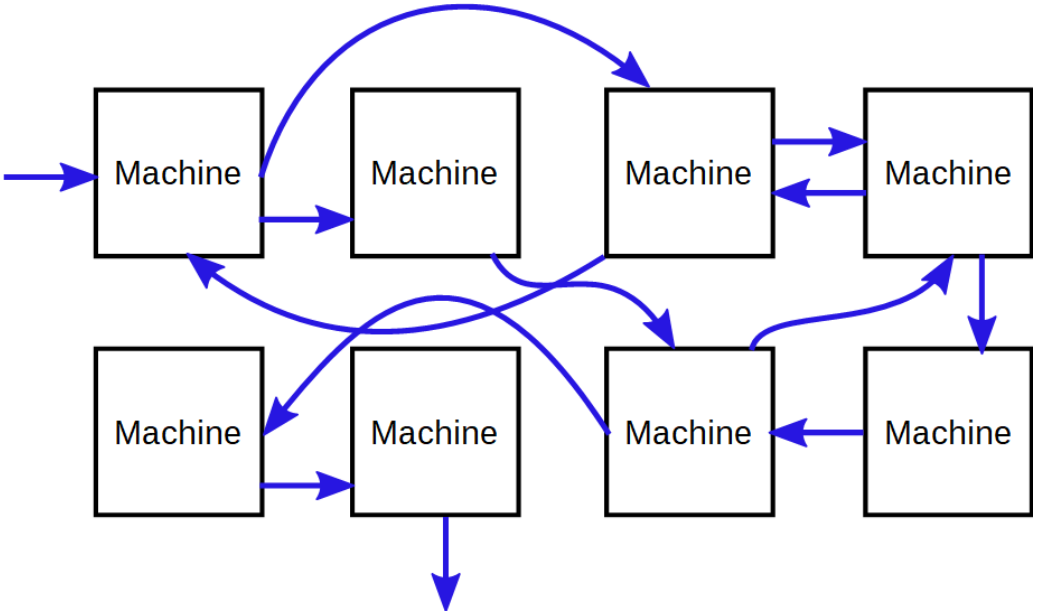


Figura 3.1: Flujo de un Job Shop

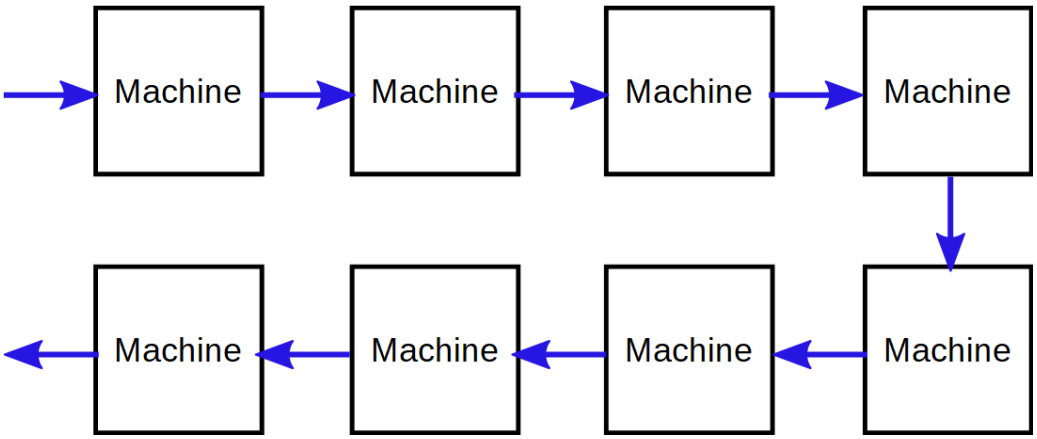


Figura 3.2: Flujo de un Flow Shop

máquina, luego la segunda operación en la segunda máquina, y así sucesivamente hasta completar la última operación. Sin embargo, las tareas pueden ejecutarse en cualquier orden, es decir, cualquier permutación es valida siempre que no exista alguna restricción que indique lo contrario. La definición del problema implica que el orden de trabajo es exactamente la misma para cada máquina, es decir, el flujo de ejecución de las operaciones de una tarea a través de las máquinas no puede variar. El principal problema a resolver es determinar la disposición óptima de la ejecución de tareas, de tal modo que cumpla la minimización del *makespan*.

Ninguna máquina va a poder ejecutar operaciones en paralelo, una vez se empiece una operación, se ejecutará el tiempo que tarde en esa máquina y no puede ser interrumpida.

Dependiendo de la variante del problema, pueden existir restricciones

de tiempo de inicio o final, podríamos tener tareas que deban acabar antes de cierto punto para darles prioridad o viceversa.

3.2. Objetivo

Principalmente se debe hallar la permutación óptima que garantice que cumple las restricciones de cualquier problema *flow shop*, teniendo en cuenta la restricción de *no-wait* entre operaciones de una misma tarea y garantizando que el *makespan* es el mínimo posible, siendo este el instante de tiempo en el que termina de ejecutarse la última operación de la última tarea.

La restricción no-wait ocurre cuando dos operaciones de una tarea deben ser procesadas sin ninguna interrupción entre ellas. Esta modificación es interesante, pues hay múltiples tareas en la vida cotidiana que se pueden beneficiar de una correcta optimización del modelo asociado, siempre que este cumpla las características del *flow shop*. El ejemplo más claro que podemos tomar es el paso de un paciente por una operación, donde tenemos tres tareas que se deben ejecutar en un orden estricto, como es el preoperatorio, la operación y el postoperatorio.

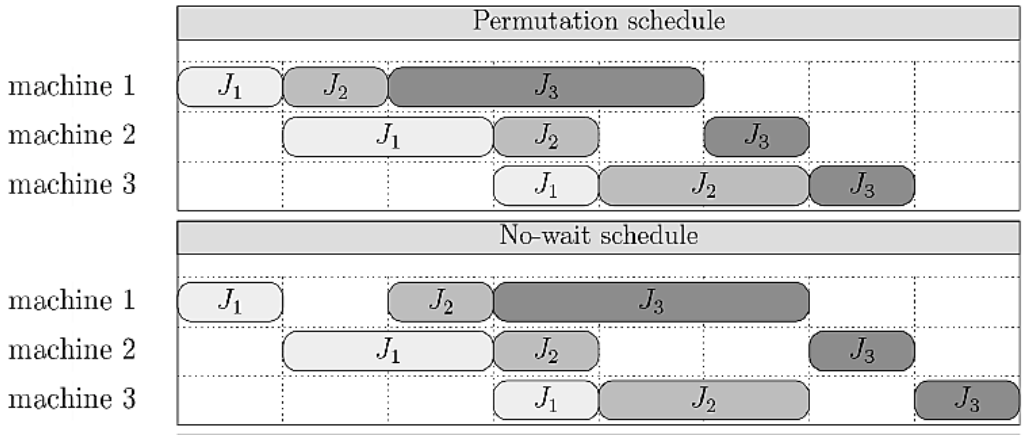


Figura 3.3: Comparación de un Flow Shop sin y con restricción no-wait

Tal y como se puede apreciar en la figura 3.3, una secuenciación de un problema *flow shop*, sin la restricción *no-wait*, viene determinada por una permutación de tareas que se irán ejecutando en su respectivo orden, de tal modo que no es necesario que tras la ejecución de una operación en su respectiva máquina, se ejecute la siguiente operación sin espera. Sin embargo, al llevar el problema a la restricción *no-wait*, nos encontramos que siempre que se empieza una tarea a ejecutar, deben ejecutarse todas las operaciones sin tiempo de espera entre ellas, siendo posible dejar

tiempo de espera entre tareas diferentes, pero nunca entre operaciones de una misma tarea. Para ver esto más claro, nos podemos fijar como varía el tiempo de inicio en la tarea *J2* de la figura 3.3.

La función objetivo del problema es minimizar el *makespan* final, sabiendo que es el tiempo total en procesar todos los trabajos. Para ello se utiliza la siguiente fórmula que permite realizarlo:

$$Min(C_{max}) = Min\left(\sum_{n=1}^{n-1}(t_{n,1} + D(n, n+1)) + \sum_{m=1}^{m} t_{n,m}\right)$$

Siendo $D(n, n+1) = D(p, q)$. Esta fórmula fue definida por *Reddi and Ramamoorthy* en 1972 [15].

$$D(p, q) = Max\left(\sum_{m=2}^{m} t_{p,m} - \sum_{m=1}^{m-1} t_{q,m}\ ,\ 0\right)$$

Donde:

- n: número de tareas a ejecutar.
- m: número de máquinas por las que debe pasar cada tarea.
- $t_{n,m}$: tiempo de ejecución de la tarea n en la máquina m.
- $D(p, q)$: el tiempo mínimo que debe existir entre el comienzo de la tarea p en la primera máquina después de haber sido ejecutado la tarea q.
- p: índice de la tarea siguiente a la ejecutada anteriormente, conocida como q.
- q: índice de una tarea cualquiera que ha sido ejecutada a través de todas las máquinas.

Por tanto, si representáramos como se está realizando el cálculo del *makespan*, podemos apreciarlo en la figura 3.4, donde las piezas a color rojo indican los tiempos que se están sumando en el primer cálculo más los tiempos de espera entre ellos para que se cumpla la restricción *no-wait* y finalmente el tiempo de ejecución de la última tarea en todas las máquinas. Con esto conseguimos realizar correctamente la definición de makespan, instante de tiempo en el que termina de ejecutarse la última operación de la última tarea.

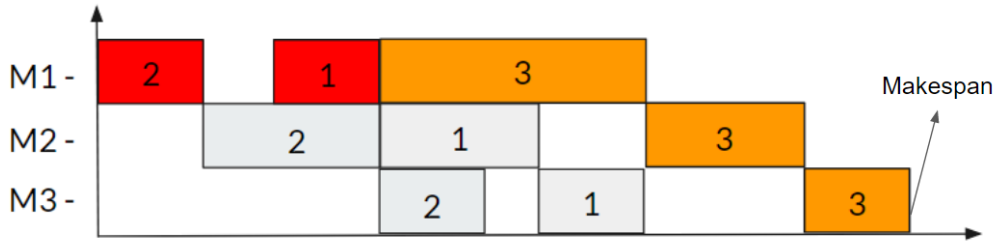


Figura 3.4: Makespan

Capítulo 4

Propuesta de solución

4.1. Lenguaje de programación

Para el desarrollo del TFG, se ha utilizado el lenguaje de programación Java[16], un lenguaje concurrente y orientado a objetos que fue diseñado para tener tan pocas dependencias de implementación como fuera posible.

La elección del lenguaje viene entre otras cosas debido a que la librería de optimización utilizada está escrita en Java y a los altos conocimientos adquiridos sobre el lenguaje.

4.2. Librería de optimización

Para desarrollar el problema se ha utilizado una librería de optimización que está siendo desarrollada por Christopher Expósito Izquierdo, Doctor en Informática y profesor contratado de la Universidad de La Laguna.

La librería ofrece una serie de herramientas con el objetivo de modelizar y resolver problemas de optimización. Proporciona la implementación de varias técnicas de optimización que se pueden utilizar para resolver una amplia gama de problemas. La librería está en desarrollo y actualmente el proyecto es privado.

Para este proyecto, se utilizó una combinación de la librería y código propio. Se utilizaron las clases que ayudan a definir el problema de optimización, la definición de una solución como una permutación de números enteros (que representan cada tarea) y un evaluador para el cálculo de la función objetivo, obteniendo así el *makespan* de una determinada permutación.

4.3. Optimización

4.3.1. Representación de la solución

La representación de una solución para el problema viene dada por un vector cuyos elementos representan cada tarea y el orden de ejecución de cada una, de izquierda a derecha. Tal y como se puede apreciar en la figura 4.1 para esa programación de tareas, la representación correspondiente es la indicada en la figura 4.2.

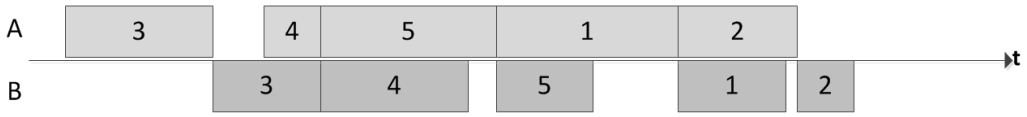


Figura 4.1: Programación de un flow shop no-wait

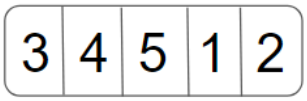


Figura 4.2: Representación de la solución de la figura 4.1

4.3.2. Movimientos implementados

- **Intercambio de dos elementos**

  En el *intercambio de dos elementos* se tienen dos pivotes los cuales se van a intercambiar de posición mientras que resto del vector permanece siempre igual, tal y como se puede apreciar en la figura 4.3. Hay que tener siempre en cuenta que los pivotes deben ser diferentes, evitando intercambiar el mismo elemento.

Figura 4.3: Movimiento de intercambio de dos elementos

- **Reinserción**

  En la reinserción se tienen también dos pivotes, el primero indica qué elemento va a ser reinsertado en el vector y el segundo indica la nueva posición del elemento a reinsertar. Como se puede ver en la figura 4.4 todos los elementos que se encuentran entre medias del elemento y la nueva posición son desplazados hacia el lado contrario al que se mueve el elemento. Hay que tener en cuenta, como en el intercambio de dos elementos, que no se debe escoger el mismo elemento para los dos pivotes y además es necesario evitar elementos que se encuentran juntos, pues estaríamos realizando un intercambio de dos elementos.

Figura 4.4: Movimiento de reinserción de un elemento

- **Inversión**

  En la inversión de elementos contamos con dos pivotes que nos indican en el vector que rango va a ser modificado. Para realizar el movimiento simplemente se altera inversamente el orden de los elementos que se encuentran dentro del rango especificado, tal y como se puede apreciar en la figura 4.5. Hay que tener en cuenta, como en los anteriores movimientos, que no se debe escoger el mismo elemento para los dos pivotes y además es necesario evitar elementos que se encuentran juntos, pues estaríamos realizando un intercambio de dos elementos.

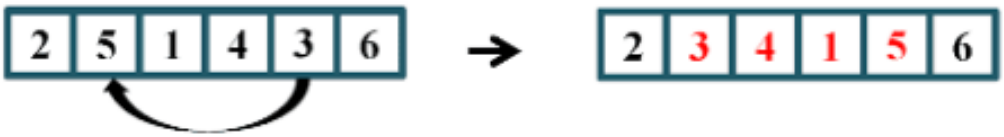


Figura 4.5: Movimiento de inversión de elementos

4.3.3. Algoritmos implementados

Para obtener un conjunto de soluciones que sea óptimo, cumpla con las restricciones del problema y sirva como entrada para la simulación que más adelante se realizará, se han implementado dos algoritmos diferentes: *GRASP* y *VNS*.

4.3.3.1. Greedy Randomized Adaptative Search Procedure (GRASP)

Este algoritmo dirige la mayor parte de su esfuerzo a construir soluciones de alta calidad que son posteriormente procesadas para obtener otras aún mejores. Normalmente este algoritmo no suele encontrar el óptimo global, siendo su salida la entrada de otros algoritmos heurísticos. El algoritmo *GRASP*[17] está compuesto por dos fases, una de construcción de una solución inicial y otra de procesamiento en la que se optimiza la solución generada en la fase anterior.

- **Fase de construcción:** se genera una lista restringida de candidatos (*RCL*) con un tamaño máximo y bajo un criterio, el cuál se determina según el problema. En esta lista irán los mejores candidatos a introducir en la construcción de la solución inicial. Se selecciona un candidato al azar y se introduce en la solución que se está construyendo. Se repite el proceso hasta que la solución inicial se considere completa y factible.
- **Fase de mejora:** la finalidad principal de esta fase es realizar una búsqueda local a partir de la solución generada para encontrar el óptimo local y obtener una buena solución. Se garantiza que a partir de esa solución no existe una mejor en ese entorno, lo cual permite a otros algoritmos usarla para diversificar la exploración de otras formas.
- **Actualización:** si la solución encontrada es mejor que la actual, se almacena. Dependiendo del criterio de parada, se devuelve la solución encontrada o se repite el proceso con el fin de encontrar otra posible solución mejor.

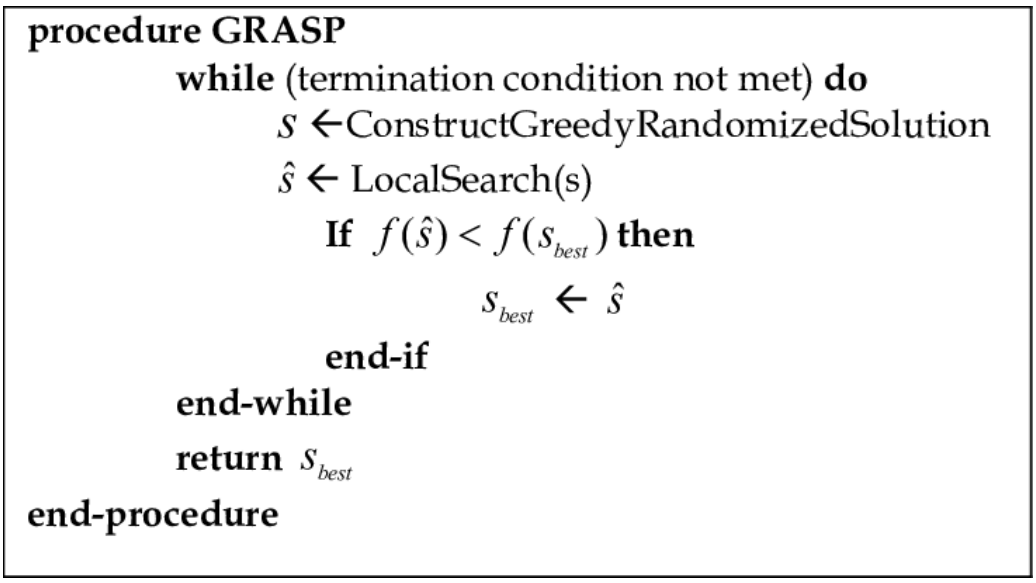


Figura 4.6: Pseudocódigo algoritmo GRASP

De cara a la futura experimentación, el algoritmo que se ha planteado consta con diferentes formas de ejecución, el cual brinda diferentes tipos de construcción para las soluciones y permite una exploración más amplia.

Los diferentes parámetros que el algoritmo acepta son:

- **Criterio de parada:** se le puede especificar al algoritmo dos tipos de parada: uno por número de iteraciones máximo, donde se fuerza al algoritmo a ejecutarse muchas veces con la intención de encontrar soluciones que raramente aparecen y pudieran ser mejores y otro por número de iteraciones sin mejora, donde se establece que si el algoritmo no mejora un número n de veces, entonces se para.

- **Tipo de generador de solución:** se ha utilizado un criterio de tiempo de ejecución, donde se calcula cuánto tiempo total va a estar una tarea realizando operaciones en las máquinas, y dependiendo de si se indica por mayor o menor tiempo, la construcción de la solución varia. Este criterio se aplica a la selección de candidatos para la *RCL*. La lista de candidatos del algoritmo *GRASP* tiene un tamaño que se puede modificar según convenga. Actualmente los tamaños utilizados para obtener resultados son tres y cinco.

- **Tipo de búsqueda local:** una vez tenemos la solución inicial construida, vamos a buscar un vecino mejor que nuestra solución actual. Para ello, se utiliza la búsqueda local con un movimiento que se define antes de comenzar. Todos los movimientos sirven, pero tras diversas ejecuciones el que mejor resultados provee es el *intercambio de dos elementos*. La búsqueda se puede realizar de dos formas: *Greedy*,

que recorrerá todos los vecinos y devolverá el mejor; o *first better*, que devuelve la primera solución encontrada que sea mejor que la actual.

4.3.4. Búsqueda por entornos variables (VNS)

En el algoritmo VNS[18] (*Variable Neighborhood Search*) se alternan una fase de construcción inicial, un proceso de sacudida sobre la solución conocido como *shaking* y una fase de mejora en la que se alternan diversos entornos para obtener una mejor exploración. Este proceso se repite hasta que se cumpla un cierto criterio de parada:

- **Número de iteraciones máximo:** se ejecutara el algoritmo un número máximo de iteraciones completas y luego se detiene, siempre quedándose con la mejor solución encontrada hasta el momento.
- **Número de iteraciones sin mejora:** se ejecutará el algoritmo hasta que no mejore la solución un número definido de veces, lo cual, a priori, nos permite asegurar que la solución es un buen óptimo local.

El algoritmo contiene una fase de construcción que se ha delegado al algoritmo GRASP, ya que provee una solución inicial robusta y que garantiza que es factible y además es un óptimo local.

Una vez hemos seleccionado nuestra solución inicial pasamos al proceso de diversificación, que en este caso es la sacudida (*shaking*). En este proceso se modifica la solución inicial aplicando algún movimiento de entorno. El número de veces que se repite la sacudida está determinado por el parámetro k, el cuál va incrementando hasta un máximo si no se encuentra una solución mejor o se resetea si se consigue obtener una solución mejor. El parámetro afecta al proceso de sacudida, indicando cuantas veces se va a realizar un movimiento sobre la solución inicial.

Una vez finalizado el proceso de sacudida, se comienza con el proceso de intensificación, el cual es conocido como *VND* (*Variable Neighbourhood Descent*). Este proceso realiza una búsqueda exhaustiva a través de diferentes entornos con sus respectivos movimientos permitiendo llegar a soluciones difíciles de obtener.

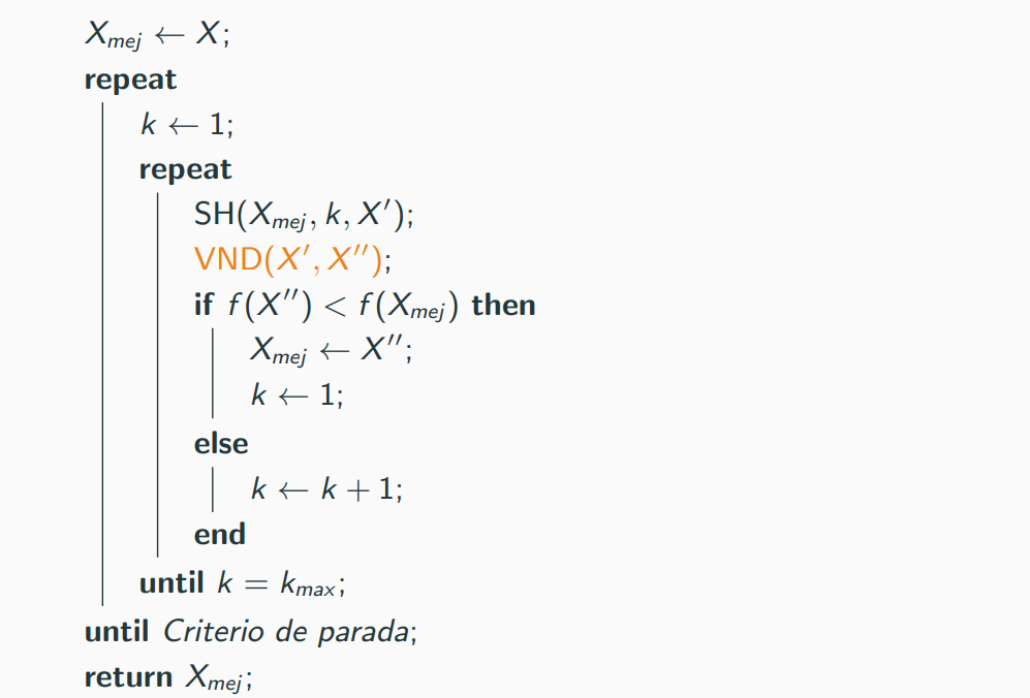


Figura 4.7: Pseudocódigo algoritmo VNS

4.3.4.1. VND

Es un tipo de búsqueda local que consiste en explorar el conjunto de soluciones aplicando diferentes movimientos de entorno repetidamente hasta que no se pueda mejorar la solución. De esta forma lo que se trata de encontrar es un óptimo local común para todas las estructuras de entorno definidas.

Partiendo de la primera estructura de entorno, si se logra mejorar la solución actual, se toma ésta como nueva solución y se vuelve a hacer la búsqueda local con el primer entorno, pero, si la solución no mejora, se cambia a la siguiente estructura de entorno y se intenta mejorar la solución. Si esta mejora, se repite el proceso desde la primera estructura de entorno, pero si no mejora, se pasa al siguiente entorno disponible, y así sucesivamente hasta que la solución no mejore con ninguno de los entornos.

4.3.5. Implementación

La fase de optimización fue diseñada para que se ejecuten todas las instancias disponibles y obtener los resultados con cada algoritmo implementado.

Durante una ejecución, se cargan los archivos individualmente y, para cada uno, se construye el problema acorde a los datos específicos de esa instancia y se le asocia un evaluador para poder comprobar cuan bueno es el *makespan*. Cuando el problema está construido con todos sus datos, se pasa por cada algoritmo que se haya definido, se resuelve y se guardan las soluciones obtenidas. El ciclo se repite mientras queden instancias sin resolver y algoritmos sin utilizar.

El diseño modular está pensado para poder añadir diferentes algoritmos heurísticos sin afectar al ciclo de vida de la fase de optimización.

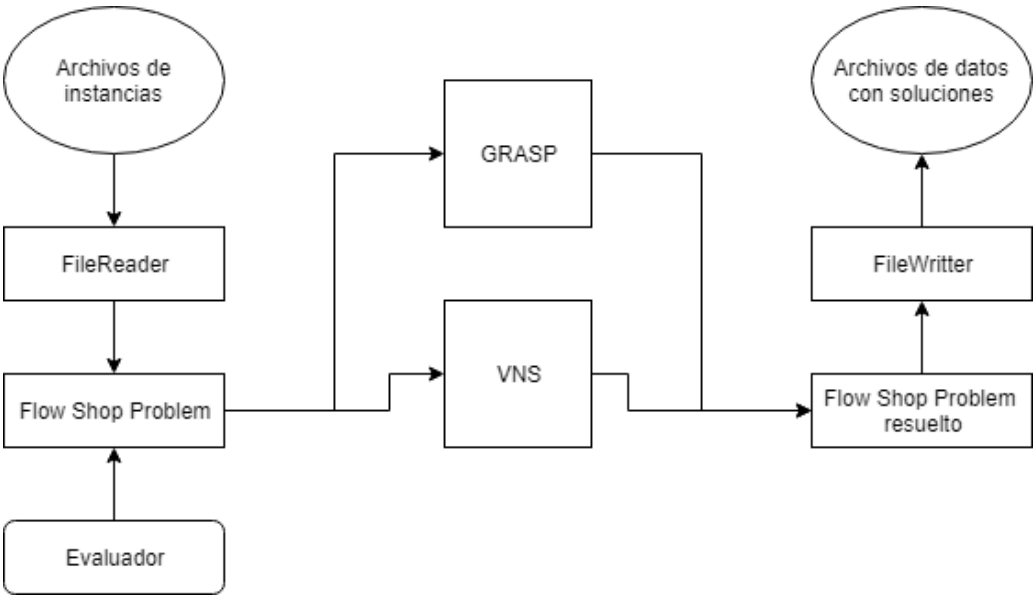


Figura 4.8: Fase de optimización

Ambos algoritmos(*GRASP y VNS*) están diseñados para ser utilizados como cajas negras que se pueden configurar en base a los parámetros que aceptan. Un esquema que los representa viene en la figura 4.9, donde dados unos parámetros, el algoritmo recibe una instancia del problema que contiene los datos necesarios (número de tareas, número de máquinas, tiempos de ejecución, etc) y devuelve el problema resuelto con el mejor *makespan* encontrado. Las diferentes combinaciones de parámetros se pueden fijar o variar para realizar diferentes pruebas que permitan investigar que combinaciones son mejores para el *no wait flow shop*.

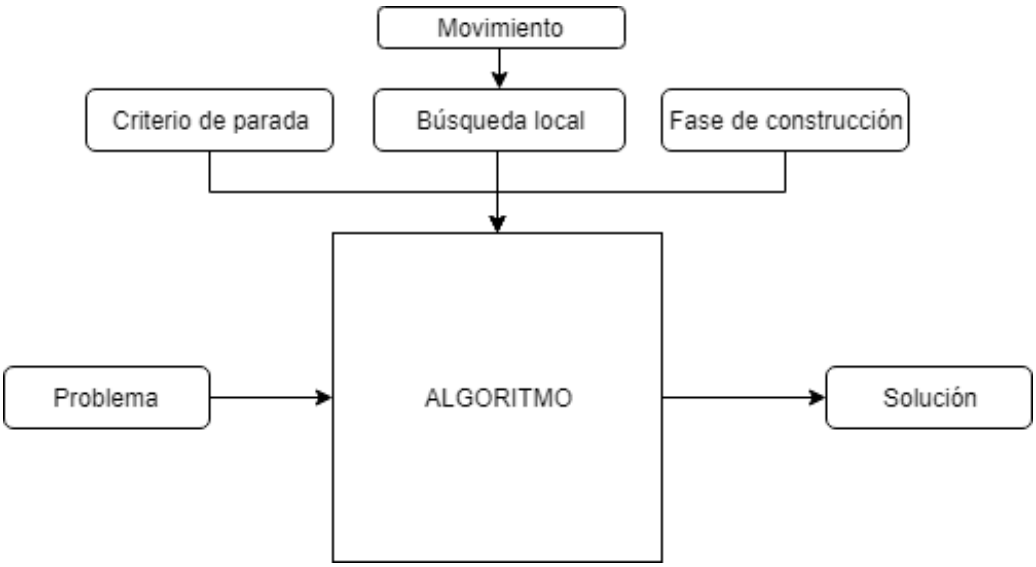


Figura 4.9: Esquema de cajas negras para los algoritmos

La fase de construcción GRASP que se utiliza en los dos algoritmos, construye una solución inicial que es factible para el problema. Esto se realiza con los datos que se obtienen de la instancia cargada y los parámetros que se fijan en el momento de inicio de cada algoritmo.

Se tiene una 'piscina' con los posibles candidatos y se van extrayendo en base al criterio de selección, que se han definido por mayor o menor tiempo de ejecución de la tarea en todas las máquinas, y se completa la RCL. Cuando está llena, se extrae un elemento aleatoriamente y se introduce en la solución inicial que se está construyendo de izquierda a derecha y se repite el proceso hasta que no queden tareas sin asignar.

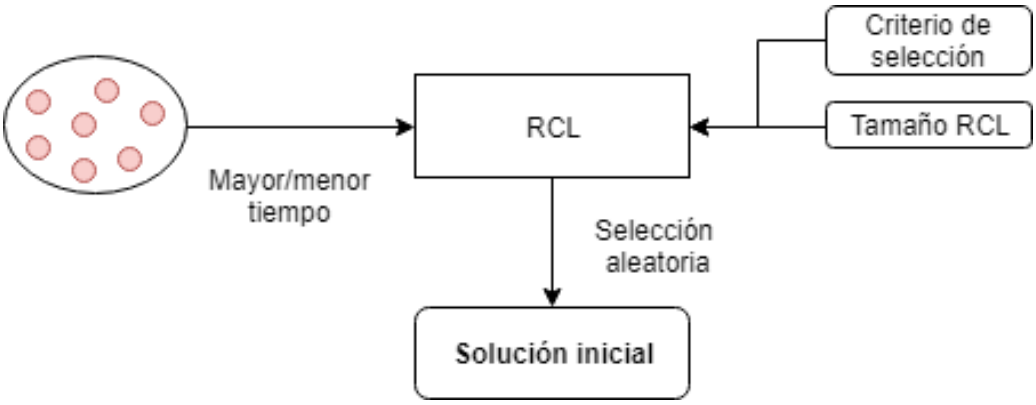


Figura 4.10: Fase de construcción GRASP

Por último, la búsqueda local genera vecinos a partir de la solución actual. Dependiendo de si es Greedy o primero mejor, parará en un instante u otro, pero el esquema de generación es siempre el mismo. El tipo de movimiento que se aplique, afectará a que vecino se va a visitar en el siguiente ciclo de la búsqueda del entorno.

4.4. Simulación

4.4.1. Línea temporal

Cuando simulamos en realidad estamos introduciendo eventos en un cierto instante de tiempo. Esto se puede representar como una línea temporal, que inicia en el instante cero y acaba cuando la última operación de la última tarea termina, es decir, el *makespan* viene a ser el final de la línea temporal. A nivel interno del programa, esto se interpreta como una cola de sucesos, lo cual permite emular el comportamiento de la línea temporal.

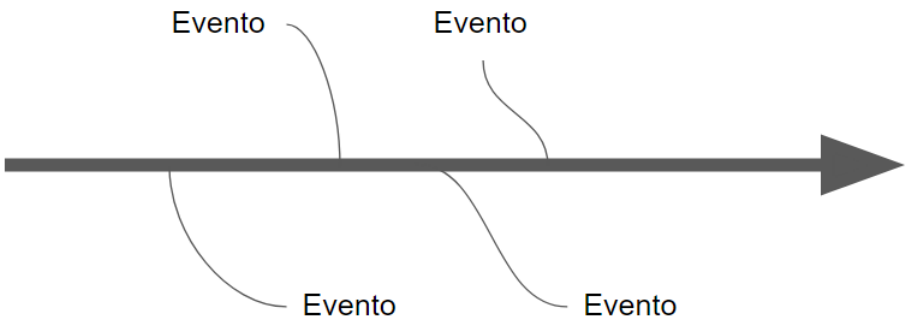


Figura 4.11: Línea de tiempo con eventos

4.4.2. Evento

Un evento dentro de la simulación de un *flow shop* viene a ser una entidad que ocurre en cierto instante de tiempo y que contiene información para el simulador. Cuando un evento ocurre en el simulador, efectúa un cambio de estado alterando la línea temporal y por ende el proceso de simulación. La información que contiene es el instante de tiempo en que ocurre ese evento.

Se distinguen dos tipos de eventos:

- **Comienzo de una operación:** se produce cuando una operación inicia su ejecución en alguna máquina.
- **Fin de una operación:** se produce cuando una operación finaliza de ejecutarse en la misma máquina que su evento de comienzo.

Por tanto, cada operación tendrá dos eventos asociados, conocidos como eventos exógenos.

4.4.3. Umbral de incertidumbre

Tal y como se ha mencionado anteriormente, un evento consta de información acerca del tiempo en que ocurre el evento en el simulador, pero para que una operación tenga dos eventos (inicio y final), se debe calcular un tiempo de ejecución que contenga un factor de incertidumbre. Este calculo es necesario para poder ejecutar la simulación un número de veces definido y comprobar como evoluciona el sistema con tiempos diferentes en cada operación.

Para obtener el tiempo que tarda en realizarse una operación, se toma como una variable aleatoria, donde el tiempo final será:

$$T_f = t_e \ \pm \ v$$

donde:

- T_f: tiempo final de ejecución en la simulación.
- t_e: tiempo de ejecución obtenido de la instancia del problema. Se mantiene fijo en la optimización y es lo que se espera que tarde una operación en ejecutarse normalmente.
- v: valor obtenido aleatoriamente de una campana de Gauss como en la figura 4.12 y que depende del factor de incertidumbre que se permita, indicado en porcentaje. El valor final puede ser negativo o positivo.

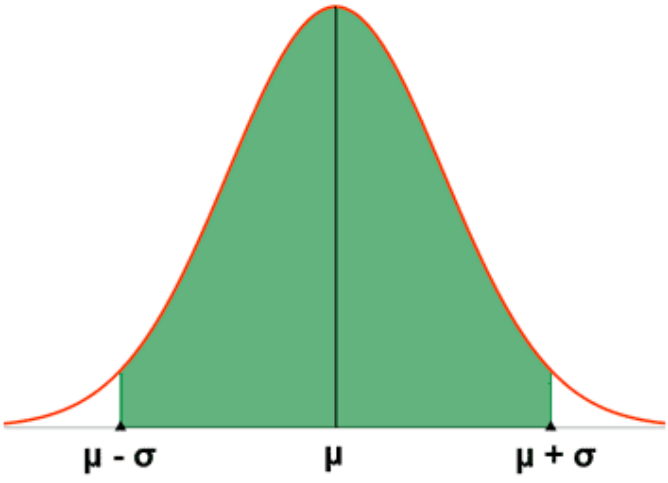


Figura 4.12: Campana de Gauss

4.4.4. Planificación de eventos

Una vez definidos los elementos básicos de una simulación, hay que tener en cuenta que la planificación de eventos es crucial para poder llevar a cabo una correcta simulación y varía dependiendo del problema de uso. Para el tratado a lo largo de este documento en particular, hay que tener en cuenta que podemos necesitar dejar un tiempo de espera entre la ejecución de una tarea y la siguiente.

Debido a la naturaleza del problema, al realizar una simulación donde existe un umbral de incertidumbre, no se puede saber, a priori, el tiempo exacto de espera entre una tarea y la siguiente. Es por ello que la planificación de eventos se realiza tarea a tarea, siendo necesario planificar todas las operaciones de una tarea en sus correspondientes máquinas antes de calcular el tiempo de espera entre tareas. Una vez conocido ese tiempo, entonces se puede proceder a repetir de nuevo la planificación de la siguiente tarea hasta que no quede ninguna por simular.

4.4.5. Implementación

La fase de simulación fue diseñada para modificar la salida de la fase de optimización y comprobar como se comporta el sistema. La simulación recibe como entrada el valor del *makespan* obtenido para poder realizar comparaciones, la secuenciación de tareas asociada al valor y un porcentaje de incertidumbre a establecer, que se indica como parámetro.

La idea principal es introducir una instancia en la optimización, calcular la mejor solución de todos los algoritmos y, por último, pasar por una fase de simulación con eventos discretos que permita extraer conclusiones.

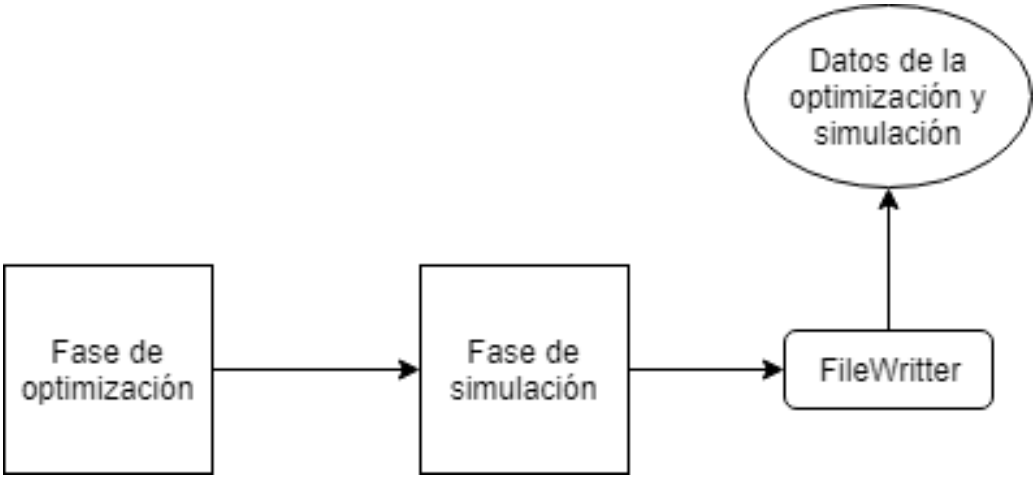


Figura 4.13: Fase de optimización

El proceso de simulación sigue un flujo continuo cuando se ejecuta y que se puede convertir a una máquina de estados que explica su funcionamiento interno.

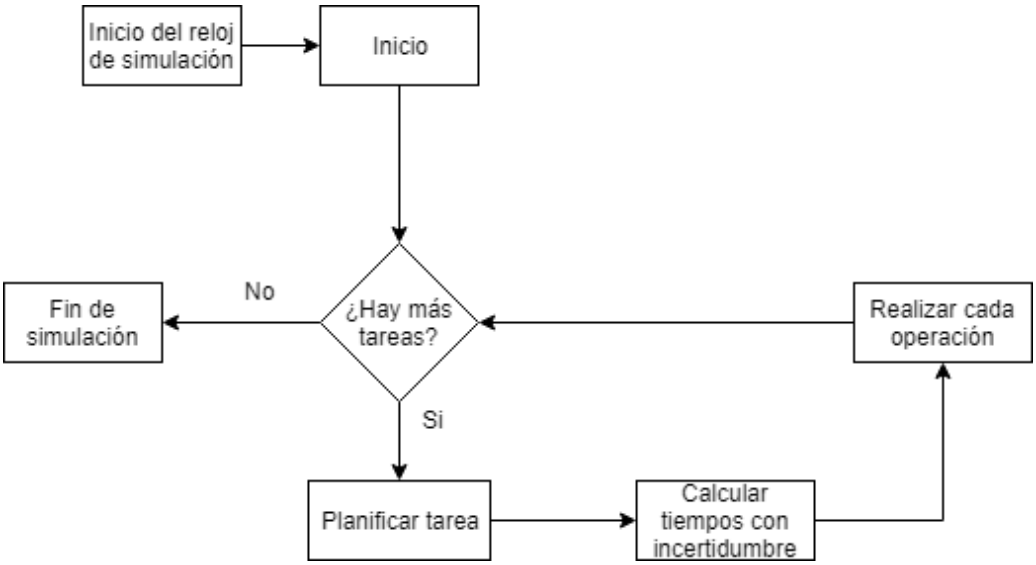


Figura 4.14: Proceso de simulación

Cada tarea se planificará en todas sus máquinas antes de pasar a planificar la siguiente tarea de la secuencia. Durante la planifcación de una operación, primero se precalculan los tiempos con incertidumbre, escogiendo un valor aleatorio en base al porcentaje introducido por parámetro, y luego se crean los eventos de inicio y fin por cada operación de la tarea. Una vez creados, los eventos se introducen en su lugar correspondiente y se repite hasta planificarlos todas las tareas del problema.

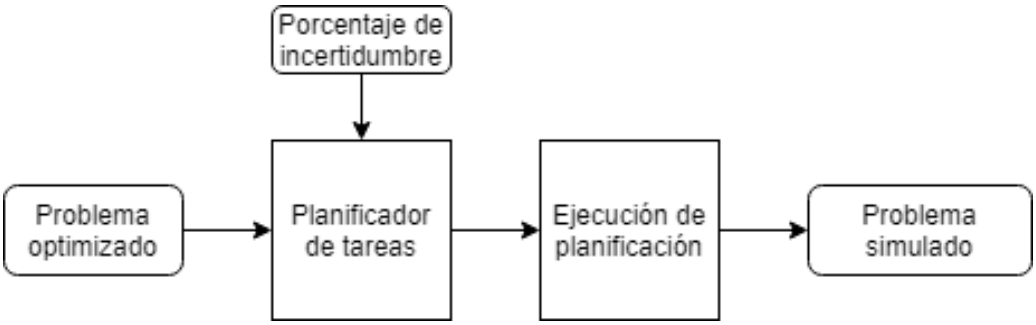


Figura 4.15: Implementación del proceso de simulación

Finalmente, cuando todos los eventos han sido planificados, se realiza una ejecución para comprobar como varía el tiempo final del *makespan* con respecto al obtenido en la optimización.

Capítulo 5

Experimentación

5.1. Especificaciones de la máquina

La máquina en la que se realizaron toda la experimentación tiene las siguientes características a tener en cuenta:

- CPU: Intel Core i7 4770K 3.5GHz Turbo Boost 3.9GHz, 4 núcleos
- 16GB de RAM
- Sistema Operativo Windows 10 Pro

5.2. Instancias utilizadas

Las instancias utilizadas para este problema en particular han sido creadas haciendo uso de un pequeño programa que genera datos aleatorios. El programa genera tiempos de ejecución a partir de un número de máquinas y número de tareas a realizar introducidos por parámetro. La salida que se obtiene es un fichero como el que se puede ver en la figura 5.1

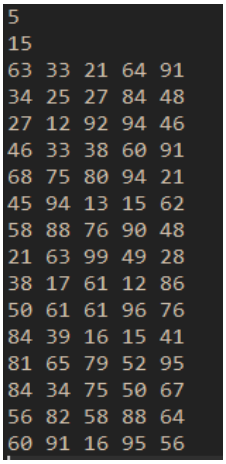


Figura 5.1: Instancia del problema

La primera fila hace referencia al número de máquinas, la siguiente al número tareas y, por último, el resto de filas son ordenadamente el tiempo de ejecución de cada operación, escrito en forma de matriz, donde las máquinas son columnas y las tareas filas.

Para realizar diferentes pruebas, se construyeron varios archivos con distinto número de máquinas y tareas. Las nomenclatura utilizada para los nombres de las instancias es la siguiente: $data\_n$, con $n = 1, 2, .., 15$.

- Las instancias que van del 1 al 5 contienen 3 máquinas y 25 tareas.
- Las instancias que van del 6 al 10 contienen 5 máquinas y 15 tareas.
- Las instancias que van del 11 al 15 contienen 10 máquinas y 10 tareas.

Sin embargo, para extraer los resultados en cada una de ellas, se han escogido 2 instancias de cada grupo, y la idea es extraer que parámetros y algoritmo de los implementados son mejores para la optimización. Por otra parte, de cara a la simulación, se usarán los mejores resultados obtenidos para cada instancia con el mejor algoritmo detectado en la fase de optimización y sus respectivos mejores parámetros.

5.3. Resultados de la optimización

5.3.1. Comparativa de algoritmos

Dado el diseño de los algoritmos, recordamos que tanto el algoritmo *GRASP* como el *VNS*, aceptan las mismas combinaciones de parámetros, sin embargo, el algoritmo *VNS* no utiliza la *RCL*, simplemente recibe el parámetro porque delega su fase de construcción al *GRASP*. Las diferentes combinaciones son:

- Tamaño de lista restringida de candidatos: { 3, 5 }
- Criterio de parada: {Máximo de iteraciones(2500), número de iteraciones sin mejora(20)}
- Movimiento de búsqueda local: { Reinserción, inversión, mover dos elementos }
- Búsqueda local: { Greedy, primero mejor }

Por lo tanto, si mezclamos los parámetros existen 24 posibilidades. Para obtener datos fiables, cada instancia que estamos utilizando se ha ejecutado 30 veces con cada configuración de parámetros y se hallará la media de cada resultado para poder comparar las diferentes combinaciones y decidir cuál es mejor.

Vamos a comenzar por comprobar que tipo de movimiento funciona mejor a la hora de optimizar en ambos algoritmos y sin tener en cuenta los parámetros de cada uno de ellos. Para ello, se realizará la media de todos los *makespan* que utilizaron cada movimiento de los existentes y tiempo medio de ejecución, mostrando los datos en la gráfica de la figura 5.2.

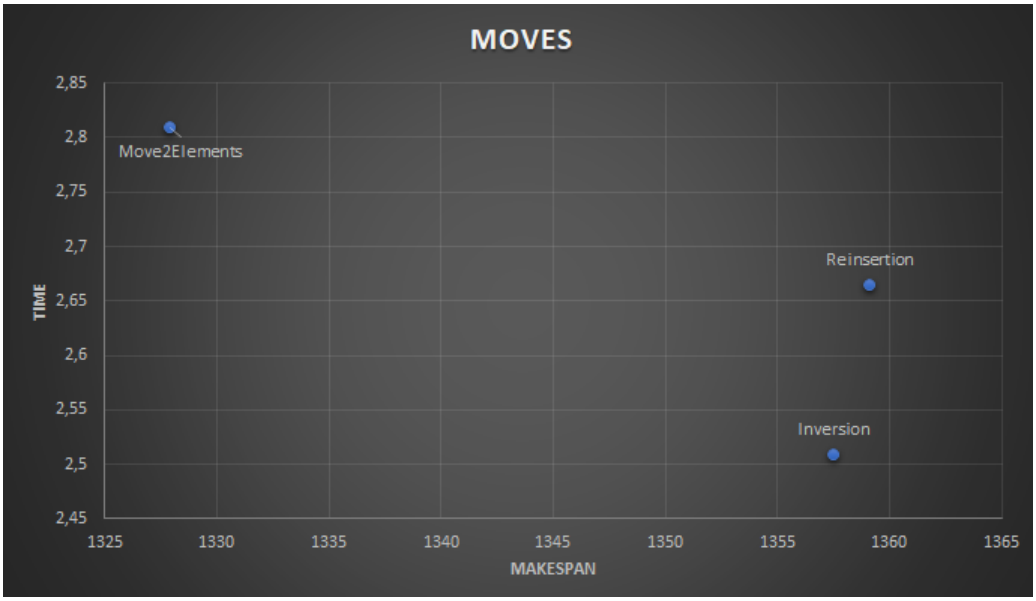


Figura 5.2: Diferencia de los tres tipos de movimiento

Como se puede observar, los datos nos permiten apreciar una diferencia en los resultados, sobre todo de cara al *makespan*, que es el objetivo principal a minimizar de este problema. Mientras que los movimientos *inversión* y *reinserción* son capaces de obtener un resultado medio que oscila en el intervalo [1355-1360], el movimiento *mover dos elementos* es capaz de conseguir una media considerablemente inferior al resto, que a pesar de hacerlo en algunas décimas de tiempo más, no es una diferencia tan significativa como la de obtener un *makespan* menor.

Por tanto, a raíz de estos datos, se decidió utilizar *mover dos elementos* como principal movimiento de los espacios de búsqueda del algoritmo *GRASP* y, por ello, a partir de este punto los datos analizados contendrán únicamente búsquedas locales con dicho movimiento, pues el algoritmo *VNS* va a usar todos los movimientos siempre en su búsqueda local tal y como se comentó previamente.

Ahora que conocemos que movimiento nos garantiza un mejor resultado, vamos a intentar reducir el tiempo de cálculo medio analizando los parámetros de los algoritmos. Comenzaremos por el algoritmo *GRASP*, donde si modificamos el criterio de parada obtenemos la media de resultados que podemos observar en el siguiente gráfico:

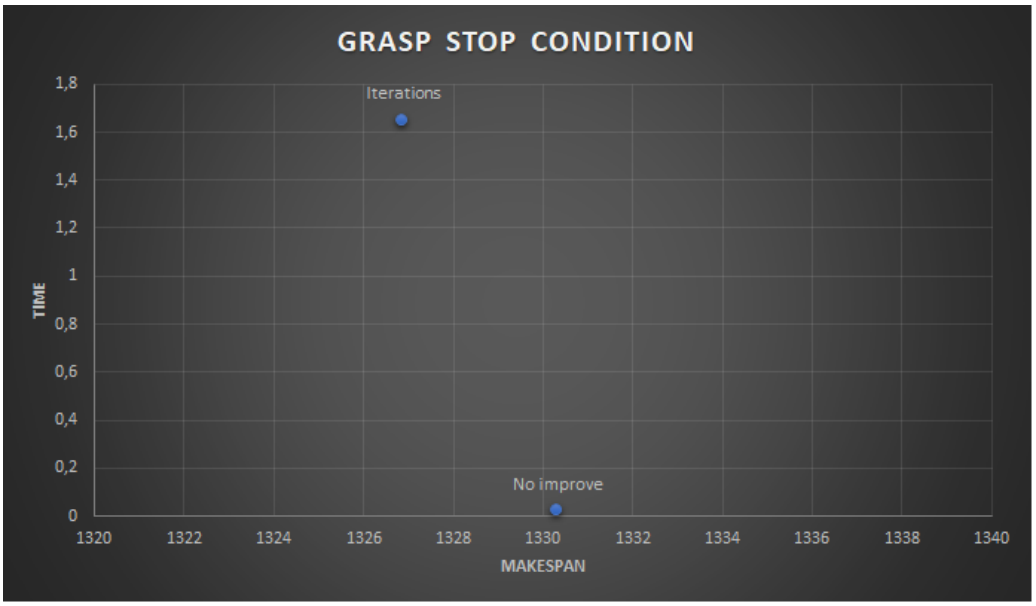


Figura 5.3: Diferencia de las condiciones de parada en el algoritmo *GRASP*

Se puede observar que ambos criterios consiguen un *makespan* medio muy aproximado, por lo cual apenas existe una diferencia, sin embargo, si que se puede apreciar una diferencia significativa en el tiempo de cálculo medio. Si recordamos, el criterio de *iteraciones* fuerza al algoritmo a ejecutarse 2500 veces, mientras que el de *no mejora*, parará si no consigue mejorar durante 20 veces seguidas. Esto lo podemos traducir a que el algoritmo que utiliza el movimiento seleccionado como mejor anteriormente, es capaz de lograr una muy buena aproximación del mejor *makespan* sin necesidad de utilizar tantas iteraciones, convirtiéndose directamente en menos tiempo de cómputo.

Continuando con el algoritmo *VNS*, vamos a extraer la misma información para ver si realmente es decisivo el tipo de criterio de parada en algoritmos diferentes. Para ello, tenemos la gráfica de la imagen 5.4 que refleja la media de datos para su respectiva ejecución.

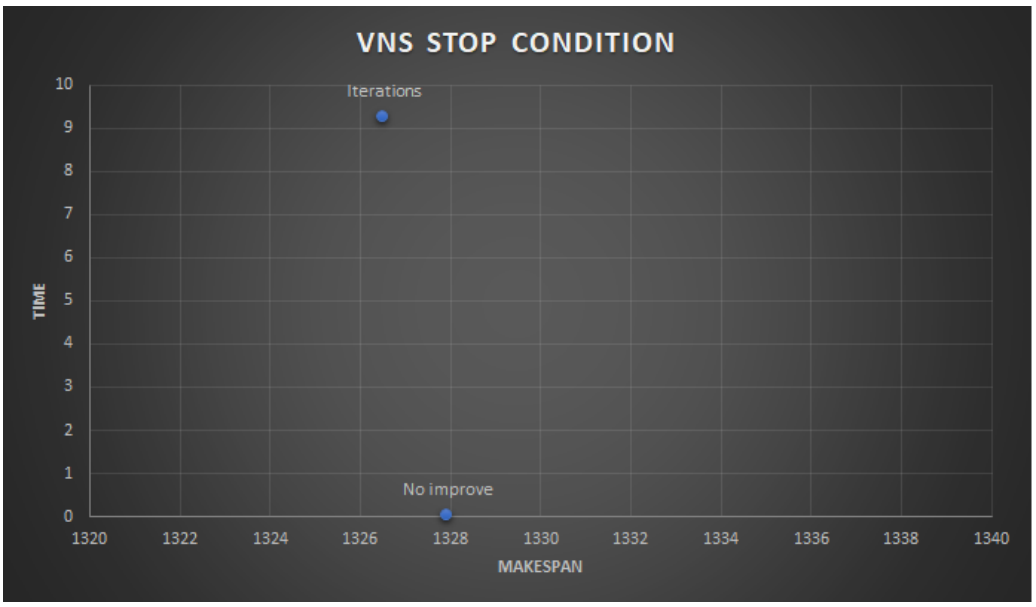


Figura 5.4: Diferencia de las condiciones de parada en el algoritmo *VNS*

Se puede observar que en este algoritmo, la diferencia de tiempo es mucho más clara, ya que, si recordamos, su búsqueda local hace uso de todos los movimientos intensificando la búsqueda mucho más, y por ello, el tiempo aumenta casi exponencialmente con cada iteración más necesaria.

Por tanto, ambos algoritmos funcionan mejor en tiempo de cómputo y minimizando el *makespan* haciendo uso del criterio de *no mejora*.

Para intentar demostrar que hasta ahora las decisiones tomadas eran correctas, se intentó realizar una nueva ejecución para comprobar cuanto tiempo aproximado era necesario para llevar a cabo de nuevo los cálculos y sus respectivos resultados. Anteriormente, para conseguir los mismos resultados era necesario ejecutar el programa durante más de 10 horas, mientras que en este punto, el cálculo necesario apenas llega a los 10 minutos consiguiendo soluciones igual de buenas. En definitiva, analizar los resultados ayuda a encontrar la mejor forma de optimizar ambos algoritmos.

A pesar de que ambos algoritmos van por buen camino, vamos a intentar comprobar si el resto de parámetros influyen directamente en el cálculo y tiempo de ejecución. A continuación, vamos a estudiar como afecta el modo en que se explora el entorno de vecinos. La gráfica de la figura 5.5 permite observar los datos analizados.

En el gráfico se han incluido tanto la media de ambos algoritmos en búsqueda *greedy* y *primero mejor*, como la media de ambos algoritmos por separado. Tal y como se puede apreciar, los datos apenas cambian

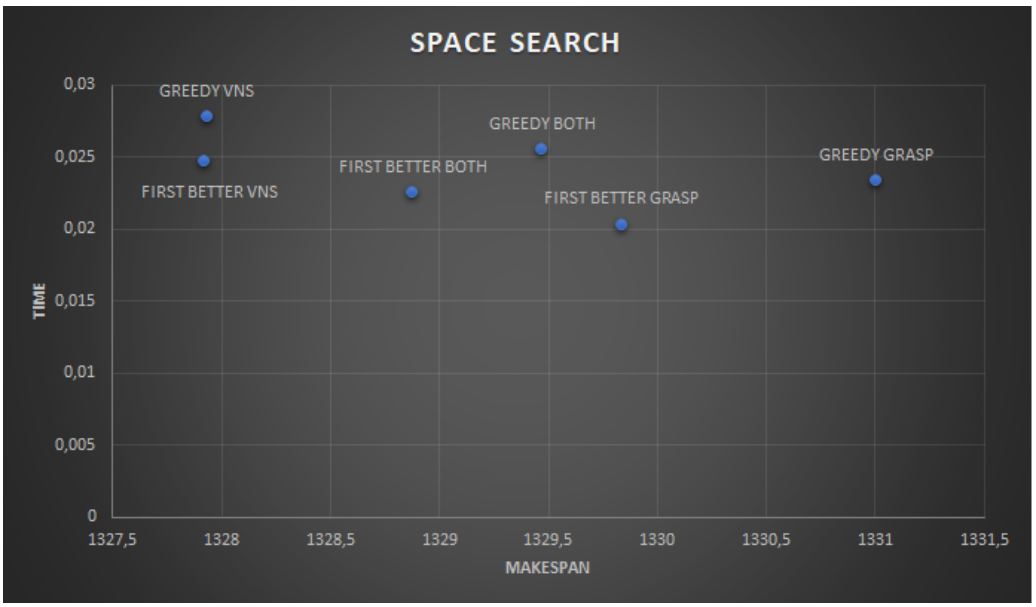


Figura 5.5: Diferencia del espacio de búsqueda en ambos algoritmos

al seleccionar uno u otro tipo de criterio de búsqueda en el espacio, por ello, elegir uno u otro apenas afecta para estas instancias utilizadas. Por tanto, vamos a seleccionar el criterio de *primero mejor*, ya que computacionalmente utiliza menos recursos al explorar menos vecinos del espacio, obteniendo mejores soluciones con un tiempo apenas notable, pero menor que el *greedy*.

Por último, vamos a analizar los parámetros que afectan a la fase de construcción del algoritmo *GRASP*, el cual se utiliza también por el *VNS*, quien delega su construcción al propio algoritmo como se mencionó anteriormente.

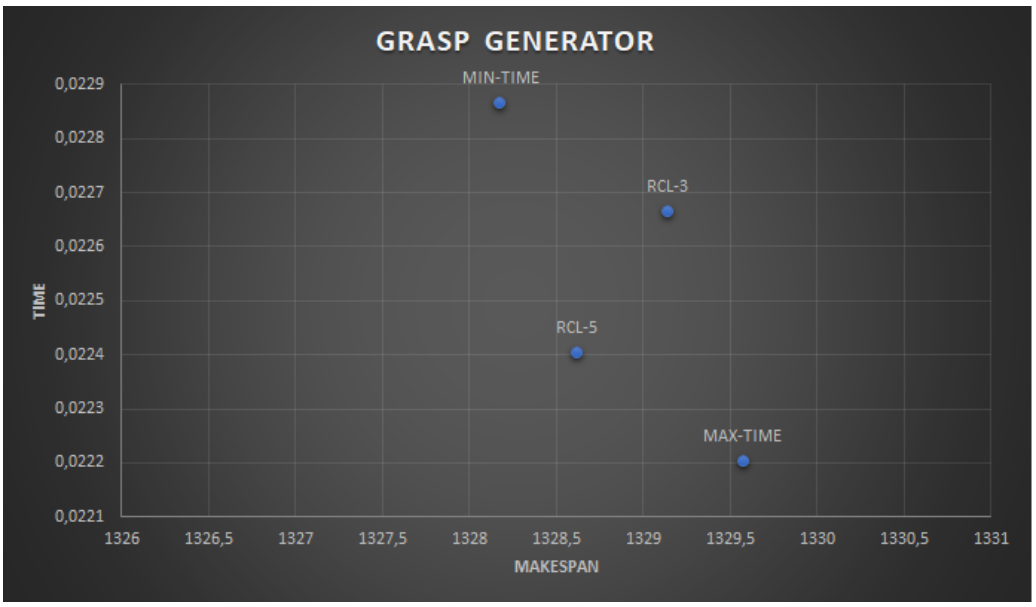


Figura 5.6: Tipo de generación en la fase de construcción *GRASP*

Nuevamente podemos apreciar que apenas existen mejoras entre elegir generación por *RCL* de tamaño 3 o 5 y selección de candidato por *mayor* o *menor* tiempo. No obstante, se decidió escoger una RCL con tamaño 3, ya que computacionalmente es mas costoso tratar con listas de tamaño 5. También se ha escogido el criterio de máximo tiempo, ya que, en este tipo de problema que estamos tratando, se ha observado que los algoritmos tienden a secuenciar las tareas que más tiempo conllevarán en las primeras posiciones.

Para terminar, vamos a hacer una recapitulación donde las mejores combinaciones para ambos algoritmos son:

Algoritmo	GRASP	VNS
Criterio de parada	Nº veces sin mejora	Nº veces sin mejora
Tamaño RCL	3	-
Selección candidato	Mayor tiempo	-
Búsqueda local	Primero mejor	Primero mejor
Movimiento	Mover 2 elementos	Todos

Cuadro 5.1: Resumen de mejores parámetros para los algoritmos

Por último, tenemos una comparación de ambos algoritmos al resolver las instancias 100 veces con los mejores parámetros de la tabla 5.7.

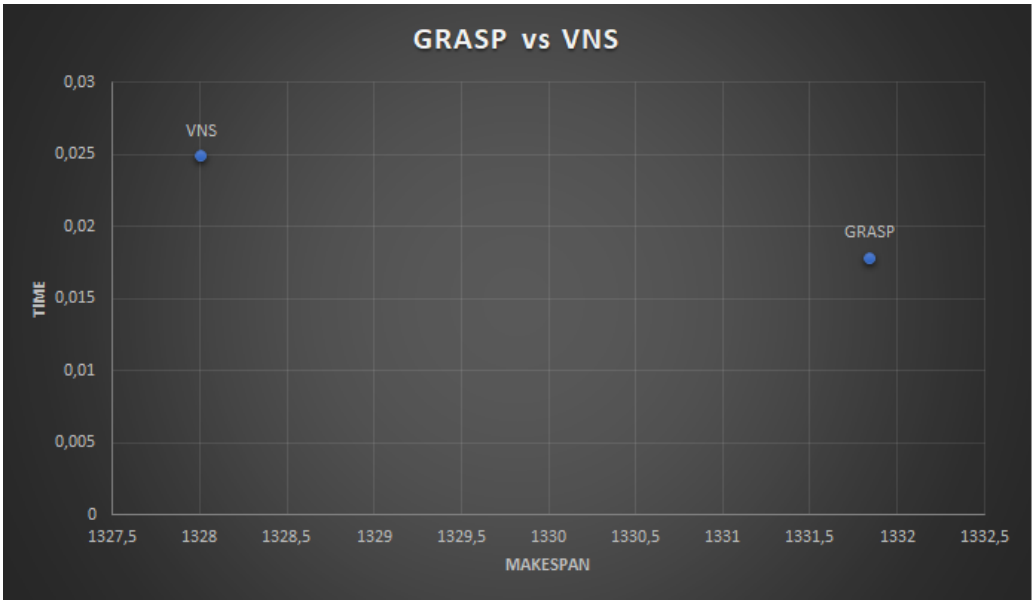


Figura 5.7: Comparación de ambos algoritmos

Podemos apreciar, que el algoritmo que mejor optimiza es el *VNS*, obteniendo mejores resultados para el *makespan* en un tiempo superior de 0.05 milésimas de segundos.

5.4. Resultados de la simulación

Ahora que hemos obtenido qué parámetros optimizan correctamente las instancias utilizadas, vamos a extraer de la primera instancia el mejor valor de *makespan* y sus respectiva secuenciación de tareas. La idea es simular un número definido de veces la secuenciación con un valor de incertidumbre y ver como se comporta a lo largo del tiempo.

Para llevar a cabo la simulación vamos a definir tres diferentes tipos de incertidumbres: el primero va a ser de 10 %, el segundo un 50 % y el último de un 75 %.

Cada simulación será ejecutada un total de 500 veces y se realizará una media del *makespan final* obtenido de cada simulación con su respectivo valor de incertidumbre. Los datos estarán agrupados y se podrá observar cuanto varía el *makespan* obtenido en la fase de optimización y el obtenido en la simulación, permitiendo conocer como evoluciona un sistema *flow shop* en un entorno real donde no existe certeza de cara al tiempo de ejecución.

Los datos obtenidos de la fase de optimización para cada instancia son los que siguen en la tabla siguiente:

Instancia	**Makespan**	**Secuenciación**
data_1	1585.0	[19, 6, 2, 23, 18, 7, 8, 1, 24, 25, 14, 15, 11, 16, 21, 5, 17, 10, 12, 13, 22, 4, 20, 3, 9]
data_14	1176.0	[10, 2, 8, 5, 4, 3, 7, 6, 1, 9,]

Cuadro 5.2: Mejores soluciones de dos instancias diferentes

Tras realizar la simulación, para el fichero de data_1 se obtienen lo siguiente al realizar una descripción estadística:

- **Incertidumbre al 10 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1585,0
Mínimo	1578,866
Máximo	1591,515
1º Cuartil	1583,832
Mediana	1585,511
3º Cuartil	1586,997
Media	1585,488
Varianza	5,266
Desviación típica	2,295

Cuadro 5.3: Estadísticas de simulación con 10 % de incertidumbre.

La media con una incertidumbre del 10 % se asemeja al *makespan* que se obtuvo de la fase de optimización, garantizando que el sistema se comporta como se espera cuando hay bastante certeza de como se van a distribuir los tiempos. Esto se puede ver reflejado mayormente en la varianza de los datos, la cual tiene un valor muy pequeño.

- **Incertidumbre al 50 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1585,0
Mínimo	1559,287
Máximo	1632,507
1º Cuartil	1584,639
Mediana	1592,988
3º Cuartil	1601,254
Media	1592,764
Varianza	133,306
Desviación típica	11,546

Cuadro 5.4: Estadísticas de simulación con 50 % de incertidumbre.

Para el 50 %, ya se puede empezar a notar como los datos se desvían del resultado obtenido en la optimización. La media del *makespan* se sitúa bastantes décimas por encima indicando que podría tardar más cuando se desconoce un la mitad de los tiempos que se van a emplear en cada tarea. La varianza entre las muestras aumenta notoriamente indicando la existencia de una desviación en los datos mucho mayor.

- **Incertidumbre al 75 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1585,0
Mínimo	1559,869
Máximo	1648,181
1º Cuartil	1588,808
Mediana	1599,399
3º Cuartil	1611,160
Media	1599,975
Varianza	247,151
Desviación típica	15,721

Cuadro 5.5: Estadísticas de simulación con 75 % de incertidumbre.

Por último, añadir un gran valor de incertidumbre como es el 75 % provoca que los datos obtenidos se disparen y se alejen bastante del resultado original. La varianza nos indica que los datos de los que se disponen varían demasiado entre ellos y, por tanto, no se sabe con certeza un tiempo exacto para el *makespan final*. La simulación provoca resultados muy desparejos, dado que la media se encuentra muy elevada por encima del valor inicial.

Para el fichero de data_14 se obtienen datos que se muestran en los siguientes puntos. Cabe destacar, que la información que se extrae es la misma, a medida que aumenta la incertidumbre, crece la varianza y la desviación de las muestras.

- **Incertidumbre al 10 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1176,0
Mínimo	1172,866
Máximo	1185,514
1º Cuartil	1177,825
Mediana	1179,510
3º Cuartil	1181,003
Media	1179,488
Varianza	5,255
Desviación típica	2,195

Cuadro 5.6: Estadísticas de simulación con 10 % de incertidumbre.

- **Incertidumbre al 50 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1176,0
Mínimo	1153,287
Máximo	1226,507
1º Cuartil	1178,624
Mediana	1186,987
3º Cuartil	1195,273
Media	1186,763
Varianza	133,0306
Desviación típica	11,446

Cuadro 5.7: Estadísticas de simulación con 50 % de incertidumbre.

- **Incertidumbre al 75 %:**

Estadístico	Datos
No. de observaciones	500
Makespan	1176,0
Mínimo	1144,868
Máximo	1243,180
1º Cuartil	1183,775
Mediana	1194,398
3º Cuartil	1206,232
Media	1194,975
Varianza	248,656
Desviación típica	15,705

Cuadro 5.8: Estadísticas de simulación con 75 % de incertidumbre.

Finalmente, la simulación siempre determina que cuanta más incertidumbre existe, más tiempo va a requerir ejecutar una secuenciación, nunca determina que podría ocurrir de media un *makespan* menor al obtenido por la optimización, asemejando parámetros como la varianza y al desviación típica en simulaciones de diferentes instancias para los mismos tipos de incertidumbre.

Capítulo 6

Conclusiones y líneas futuras

6.1. Conclusiones

En la sección 1.2 se estableció como objetivo desarrollar un proyecto capaz de rendir bien bajo cualquier instancia del problema *flow shop*. Este objetivo se ha cumplido adecuadamente, siendo necesario desarrollar un software capaz de ejecutar una instancia grande en un periodo de tiempo bastante corto y obteniendo resultados muy favorables.

Se determinó que el algoritmo que mejor rinde cumpliendo los objetivos es el *VNS* haciendo uso de los siguientes parámetros:

- **Criterio de parada:** número de veces sin mejora.
- **Tipo de búsqueda local:** primero mejor.
- **Fase de construcción GRASP**
  - **Tamaño de la RCL:** 3.
  - **Criterio de selección de candidato:** mayor tiempo de ejecuciónc.

Los resultados computacionales obtenidos en la fase de experimentación, demuestran que el algoritmo *VNS* es ligeramente mejor que el *GRASP*, resultando los valores del *makespan* más bajos. Sin embargo, los tiempo de ejecución son mayores por algunas décimas, siendo no suficientemente elevados como para tener importancia en la selección del algoritmo.

También, se ha conseguido juntar ambas fases del proyecto en una. El programa puede encargarse de realizar una optimización y utilizar los datos obtenidos como entrada para la fase de simulación, la cual se basa en eventos con incertidumbre.

Se debe tener en cuenta que las capacidades del modelo de simulación están limitadas durante el proyecto debido a que se decidió utilizar un software propio y creado desde cero, en lugar de utilizar otros programas.

6.2. Líneas futuras

Un proyecto dedicado a la optimización y simulación de un problema determinado como es este tiene infinitas posibilidades de futuro, sin embargo, las más interesantes a conseguir tienen que ver con la fase de optimización.

Los algoritmos genéticos[19] son los más comunes en el campo de la optimización y se ha demostrado en otros artículos que utilizarlos permite obtener buenas soluciones en tiempos polinomiales y, por tanto, sería muy interesante su uso en este problema. No sólo hay que quedarse con el algoritmo genético, hay muchos más que se pueden implementar y analizar en busca de una mejor optimización.

Jugar con los parámetros de los diferentes algoritmos permitió obtener diferentes datos cada cual más interesante, sin embargo existen otros tipos de movimiento que se pueden implementar y diferentes criterios de parada o selección que no se han tratado en este proyecto.

La librería que se utilizó provee un sistema para generar soluciones vecinas haciendo uso del esquema de la figura 6.1, pero por falta de tiempo no fue posible incluirlo y se realizó uno propio. La principal razón por la que no se pudo utilizar, es la necesidad de implementar una función que determine cuanto varía el *makespan* al realizar un movimiento sobre la permutación sin necesidad de realizar el cálculo desde el principio.

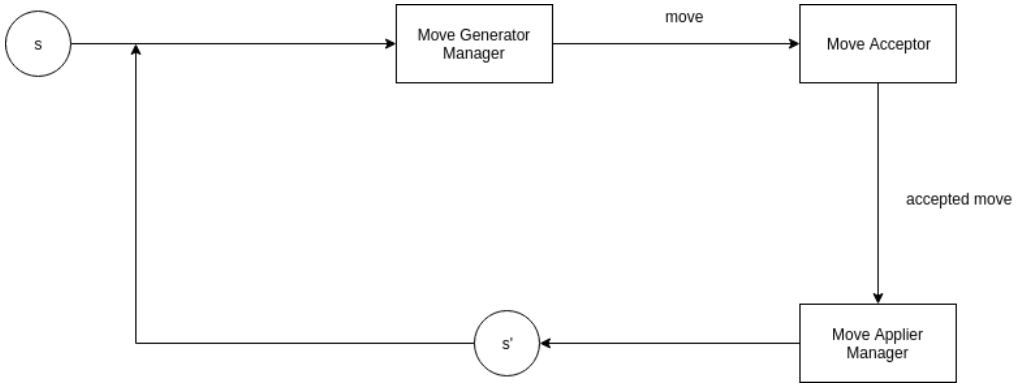


Figura 6.1: Esquema de generación de vecinos a partir de una solución

Hasta ahora se ha tratado con una instancia en un fichero de texto y de este se extraen diferentes datos, pero se podría proveer al software con

una potente interfaz que permita interactuar con la fase de optimización y sobre todo con la simulación.

Capítulo 7

Conclusions and future lines

7.1. Conclusions

In section 1.2 the main objective was to develop a project which performs correctly in any instance of the *flow shop* problem. This objective has been fulfilled correctly, being necessary to develop a software capable of running a large instance in a fairly short period of time and obtaining very favourable results.

It was determined that the algorithm which performs the best, fulfilling the objectives, is the *VNS* making use of the following parameters:

- **Stop criterion:** number of times without improvement.
- **Local search type:** first better.
- **GRASP construction phase**
  - **RCL Size:** 3.
  - **Candidate selection criterion:** maximum execution time.

The computational results obtained in the experimentation phase shows that the *VNS* algorithm is slightly better than the *GRASP*, resulting in the lowest *makespan* values possibles. However, the execution times are longer by a few tenths, being not high enough to have importance in the selection of the algorithm.

Also, it has been possible to combine both phases of the project into one. The program can do an optimization and use the data obtained as input for the simulation phase, which is based on events with uncertainty.

It should be noted that the capabilities of the simulation model are limited during the project because was decided to use an own software and created from scratch, instead of using other programs.

7.2. Future lines

A project dedicated to the optimization and simulation of a given problem, such as this, has infinite possibilities for the future, however, the most interesting to achieve have to do with the optimization phase.

Genetic algorithms[19] are the most common in the field of optimization and it has been demonstrated in other articles that using them allows to obtain good solutions in polynomial times and, therefore, it would be very interesting to use them in this problem. Not only it's necessary to keep the genetic algorithm, there are many more that can be implemented and analyzed looking for a better optimization.

To play with the parameters of the different algorithms allowed to obtain different data each one more interesting, however there are other types of movement that can be implemented and different criteria of stop or selection that have not been treated in this project.

The library that was used provides a system to generate neighboring solutions using the scheme in figure 7.1, but due to lack of time it was not possible to include it and i made one myself. The main reason why we can't apply, it's the need to implement a function that determines how much the makespan varies when making a movement on the permutation without the need to perform the calculation from the beginning.

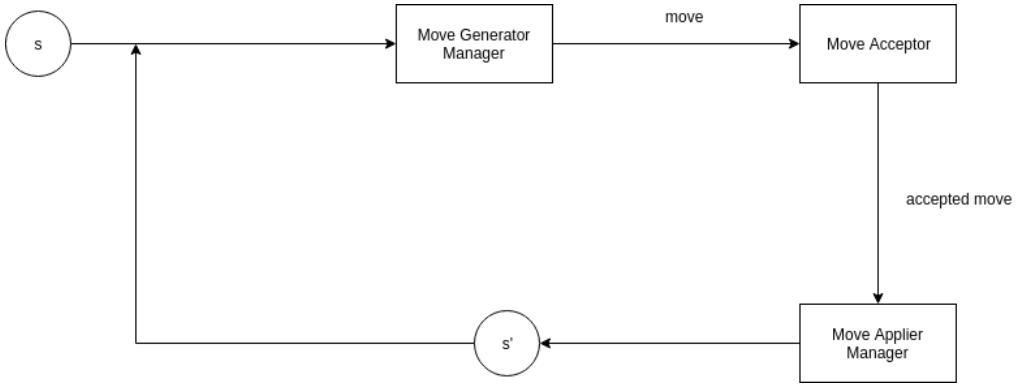


Figura 7.1: Scheme of generating neighbors from a solution

Up to now, we have dealt with an instance in a text file and from this we extract different data, but we could provide the software with a powerful interface that allows to interact with the optimization phase and above all with the simulation.

Capítulo 8

Presupuesto

8.1. Presupuesto de trabajo

El presupuesto estimado necesario del proyecto se basa en las horas de trabajo dedicadas. Por otra parte, no es necesario la adquisición de ningún tipo de hardware o software de pago, pero en la tabla se añadirá el coste de contratación de un servidor privado que contiene una alta capacidad de cómputo para instancias mucho más pesadas.

Tipo	**Coste**
Servidor	119 €/mes
Trabajo dedicado	4800 €
	5157 €

Cuadro 8.1: Resumen de prespuesto

Apéndice A

Generador de instancias

A.1. Algoritmo de generación

```
public class Generator {

        /** The Constant MIN_RANGE. */
        private static final Integer MIN_RANGE = 10;

        /** The Constant MAX_RANGE. */
        private static final Integer MAX_RANGE = 100;

        /** The Constant ENDLINE. */
        private static final String ENDLINE = "\n";

        /**
         * Generate all the data file.
         *
         * @param fileName the file name
         * @param machines the machines
         * @param tasks the tasks
         * @throws Exception the exception
         */
        public static void generateData(File fileName, Integer machines, Integer tasks)
            throws Exception {
                if (fileName.exists()) {

                        FileWriter writer = new FileWriter(fileName);

                        writer.write(machines + ENDLINE);
                        writer.write(tasks + ENDLINE);

                        Integer[][] randomData = randomData(machines, tasks);

                        for (int i = 0; i < tasks; i++) {
                                for (int j = 0; j < machines; j++) {
                                        writer.write(randomData[i][j] + " ");
                                }
                                writer.write(ENDLINE);
                        }

                        writer.close();
                } else {
```

```
                throw new IllegalArgumentException("The file does not exist.");
            }
        }

        /**
         * Random data generator.
         *
         * @param machines the machines
         * @param tasks the tasks
         * @return the integer[][]
         */
        public static Integer[][] randomData(Integer machines, Integer tasks) {
            Integer[][] data = new Integer[tasks][machines];

            for (int i = 0; i < tasks; i++) {
                for (int j = 0; j < machines; j++) {
                    data[i][j] =
                        ThreadLocalRandom.current().nextInt(MIN_RANGE,
                        MAX_RANGE + 1);
                }
            }

            return data;
        }

        /**
         * The main method.
         *
         * @param args the arguments
         */
        public static void main(String[] args) {
            try {
                if (args.length != 3)
                    throw new Exception(
                                    "This program expect 3 arguments: file
                                       to export, number of machines and
                                       number of tasks.");

                generateData(new File(args[0]), Integer.parseInt(args[1]),
                    Integer.parseInt(args[2]));

            } catch (Exception error) {
                System.err.println(error.getMessage());
                error.printStackTrace();
            }
        }

}
```

Apéndice B

Optimización

B.1. Evaluador

```
public class FlowShopMakespanEvaluator extends
    EvaluatorSingleObjectiveFunction<PermutationSolution<FlowShopProblem>> {

        @Override
        public void evaluate(PermutationSolution<FlowShopProblem> solution) {
                super.setObjectiveFunctionValue(makespan(
                solution.getOptimizationProblem().getMachines(),
                Arrays.stream(solution.getRepresentation()).boxed().toArray(Integer[]::new),
                solution.getOptimizationProblem().getTimes()));
        }

        @Override
        public void fillSolution(PermutationSolution<FlowShopProblem> solution) {}

        /**
         * Determine the minimum delay time between the completion of job Jp
         * and the initiation of Jq using the Reddi and Ramamoorthy formula
         *
         * @param p the previous task p
         * @param q the actual task q
         * @param machines the number of machines
         * @param times the times of execution
         * @return the delay time between the completion of job Jp
         * and the initiation of Jq
         */
        public static int delayTime(Integer p, Integer q, Integer machines, Integer[][]
            times) {
        int delay = 0;

        for (int i = 1; i < machines; i++)
            delay += times[p - 1][i];

        for (int i = 0; i < machines - 1; i++)
            delay -= times[q - 1][i];

        return (delay <= 0) ? 0 : delay;
    }
```

```
    /**
     * Calculate the makespan of a given schedule
     *
     * @param machine the machines
     * @param tasks the schedule
     * @param times the times
     * @return the makespan
     */
    public static double makespan(Integer machines, Integer[] tasks, Integer[][] times) {
        double makespan = 0.0;

        // For every task, get the time of execution
        for (int i = 0; i < tasks.length; i++) {
          makespan += times[tasks[i] - 1][0]; // Add the time of execution to the
              makespan

          // If the task is not the first, then we calculate the delay time between
          // the previous task and the current, because the waiting time also counts
          // for the makespan
          if (i != 0)
            makespan += delayTime(tasks[i-1], tasks[i], machines, times);

          // If the current task is the last, then we take all the
          // execution times in every machine
          if (i == (tasks.length -1))
            for (int j = 1; j < machines; j++)
                makespan += times[tasks[i] - 1][j];
        }

        return makespan;
    }
}
```

B.2. Problema de optimización

```
public class FlowShopProblem extends OptimizationProblem{
        /** The number of machines. */
        private Integer machines;

        /** The number of tasks. */
        private Integer tasks;

        /** The times of each task in each machine. */
        private Integer[][] times;

        /**
         * Instantiates a new flow shop problem.
         */
        public FlowShopProblem() {
                super();
                setName("Flow shop problem");
                setDescription("TODO");
                setEvaluator(null);
                times = new Integer[0][0];
        }
```

```java
/**
 * Instantiates a new flow shop problem.
 *
 * @param machines the machines
 * @param tasks the tasks
 * @param times the times
 */
public FlowShopProblem(Integer machines, Integer tasks, Integer[][] times) {
        super();
        this.machines = machines;
        this.tasks = tasks;
        this.times = times;
}

/**
 * Add a value in the time matrix.
 *
 * @param machine the machine
 * @param task the task
 * @param value the value
 */
public void addTime(Integer machine, Integer task, Integer value) {
        times[machine][task] = value;
}


/**
 * Resizes the times matrix.
 */
public void resize() {
        times = new Integer[getTasks()][getMachines()];
}

/* (non-Javadoc)
 * @see com.kaizten.opt.problem.OptimizationProblem#toString()
 */
@Override
public String toString() {
        StringBuilder builder = new StringBuilder();
builder.append("Name=");
if (hasName())
    builder.append(getName());

builder.append("\n");
builder.append("Description=");
if (hasDescription())
    builder.append(getDescription());

builder.append("\n");
builder.append(getEvaluator());

builder.append("\nMachines=" + getMachines());
builder.append("\ntasks=" + getTasks());
builder.append("\nTimes:\n");
for (int i = 0; i < getTimes().length; i++) {
        for (int j = 0; j < getTimes()[i].length; j++) {
```

```
                        builder.append(times[i][j] + " ");
                }
                builder.append("\n");
        }
                return builder.toString();
        }

        /** Getters and setters. **/

        public Integer getMachines() {
                return machines;
        }

        public void setMachines(Integer machines) {
                this.machines = machines;
        }

        public Integer getTasks() {
                return tasks;
        }

        public void setTasks(Integer tasks) {
                this.tasks = tasks;
        }

        public Integer[][] getTimes() {
                return times;
        }

        public void setTimes(Integer[][] times) {
                this.times = times;
        }
}
```

B.3. GRASP

```
public class FlowShopCustomGRASP {

        /** The optimization problem. */
        private FlowShopProblem optimizationProblem;

        /** The best solution. */
        private PermutationSolution<FlowShopProblem> bestSolution;

        /** The initial solution. */
        private PermutationSolution<FlowShopProblem> initialSolution;

        /** The local search. */
        private LocalSearch localSearch;

        /** The iterations. */
        private Integer iterations;

        /**
         * Instantiates a new grasp algorithm to solve a given problem.
```

```
 *
 * @param optimizationProblem the optimization problem
 */
public FlowShopCustomGRASP(FlowShopProblem optimizationProblem) {
        super();
        setOptimizationProblem(optimizationProblem);
        setLocalSearch(new LocalSearch());
        graspConfig();
}

/**
 * Grasp initial configuration.
 */
private void graspConfig() {
        setInitialSolution(null);
        setBestSolution(null);
        setIterations(0);
}

/**
 * Runs the grasp algorithm.
 *
 * @param criterion the stop criterion
 * @param localSearchType the local search type
 * @param generator the generator for the CL
 * @throws Exception the exception
 */
public void run(StopCriterion criterion, LocalSearchType localSearchType,
    CandidateListGenerator generator)
                throws Exception {

        // If the stop criterion is number of times without improvement
        if (criterion == StopCriterion.NUM_IMPRO) {
                Integer noImprove = 0;
                graspConfig();
                while (noImprove < Constants.NO_IMPROVED) {
                        iterations++;
                        initialSolution =
                            GraspInitialSolution.runGenerator(optimizationProblem,
                            generator);
                        localSearch.setInitialSolution(initialSolution);
                        initialSolution =
                            localSearch.findOptimum(localSearchType);
                        if (bestSolution != null
                            || bestSolution.getObjectiveFunctionValue(0)
                            > initialSolution.getObjectiveFunctionValue(0)) {
                                // Clone the initial solution as best solution
                                bestSolution =
                                    (PermutationSolution<FlowShopProblem>)
                                                initialSolution.clone();
                                noImprove = 0; // Reset no improve counter
                        } else
                                noImprove++;
                }

                // If the stop criterion is a maximum number of iterations
        } else if (criterion == StopCriterion.ITERATIONS) {
```

```
                graspConfig();
                Integer iteration = 0;
                while (iteration < Constants.MAX_ITERATIONS) {
                        iterations++;
                        initialSolution =
                            GraspInitialSolution.runGenerator(optimizationProblem,
                            generator);
                        localSearch.setInitialSolution(initialSolution);
                        initialSolution =
                            localSearch.findOptimum(localSearchType);
                        iteration++;
                        if (bestSolution != null
                            || bestSolution.getObjectiveFunctionValue(0)
                                > initialSolution.getObjectiveFunctionValue(0)) {
                            // Clone the initial solution as best solution
                            bestSolution =
                                (PermutationSolution<FlowShopProblem>)
                                initialSolution.clone();
                        }
                }
        } else
                throw new IllegalArgumentException("Invalid stop criterion for
                    grasp.");
}

/** Getters and Setters **/

public FlowShopProblem getOptimizationProblem() {
        return optimizationProblem;
}

public void setOptimizationProblem(FlowShopProblem optimizationProblem) {
        this.optimizationProblem = optimizationProblem;
}

public PermutationSolution<FlowShopProblem> getBestSolution() {
        return bestSolution;
}

public void setBestSolution(PermutationSolution<FlowShopProblem> bestSolution) {
        this.bestSolution = bestSolution;
}

public PermutationSolution<FlowShopProblem> getInitialSolution() {
        return initialSolution;
}

public void setInitialSolution(PermutationSolution<FlowShopProblem>
    initialSolution) {
        this.initialSolution = initialSolution;
}

public LocalSearch getLocalSearch() {
        return localSearch;
}

public void setLocalSearch(LocalSearch local) {
```

```
                this.localSearch = local;
        }

        public Integer getIterations() {
                return iterations;
        }

        public void setIterations(Integer iterations) {
                this.iterations = iterations;
        }
}
```

B.4. Fase de construcción GRASP

```
public class GraspInitialSolution {

        /**
         * Run the initial solution generator.
         *
         * @param problem the flow shop scheduling problem
         * @param generator the generator type
         * @return the initial solution built
         */
        public static PermutationSolution<FlowShopProblem> runGenerator(FlowShopProblem
            problem,
                        CandidateListGenerator generator) {
                PermutationSolution<FlowShopProblem> solution = new
                    PermutationSolution<FlowShopProblem>(problem,
                                problem.getTasks());

                Random random = new Random();

                // Call the LC generator
                List<Integer> candidateList = generateCandidateList(problem.getTasks());

                // Call the order by criterion list
                if (generator == CandidateListGenerator.MIN_TIME) {
                        candidateList = orderList(candidateList, problem, generator);
                } else if (generator == CandidateListGenerator.MAX_TIME) {
                        candidateList = orderList(candidateList, problem, generator);
                }

                // Builds the RCL and selects random elements from it until the CL and
                    RCL are empty and the solution is built
                List<Integer> restrictedCandidateList = new ArrayList<Integer>();
                Integer count = 0;
                while (!candidateList.isEmpty() || !restrictedCandidateList.isEmpty()) {
                        while (restrictedCandidateList.size() < Constants.RCL_SIZE &&
                            !candidateList.isEmpty()) {
                                restrictedCandidateList.add(candidateList.get(0));
                                candidateList.remove(0);
                        }
                        int element = random.nextInt(restrictedCandidateList.size());
                        solution.setValue(count, restrictedCandidateList.get(element));
                        restrictedCandidateList.remove(element);
```

```
                count++;
        }
        return solution;
}

/**
 * Generates the candidates list.
 *
 * @param number the number of candidates
 * @return the list
 */
private static List<Integer> generateCandidateList(Integer number) {
        List<Integer> list = new ArrayList<Integer>(number);
        for (int i = 0; i < number; i++) {
                list.add(i + 1);
        }
        return list;
}

/**
 * Order the candidate list.
 *
 * @param candidateList the candidate list
 * @param problem the flow shop scheduling problem
 * @param generator the generator type
 * @return the candidate list
 */
private static List<Integer> orderList(List<Integer> candidateList,
    FlowShopProblem problem,
                CandidateListGenerator generator) {
        Map<Integer, Integer> helper = new HashMap<Integer, Integer>();

        // Builds a map with his task id and the sum of time on all machines
        for (Integer i : candidateList) {
                Integer sum = 0;
                Integer times[][] = problem.getTimes();
                for (int j = 0; j < times[i - 1].length; j++) {
                        sum += times[i - 1][j];
                        helper.put(i, sum);
                }
        }
        candidateList.clear();

        // Finds the maximum or minimum entry on the map, arranges it in its
            corresponding
        // position and removes it from the map.
        for (int i = 0; i < problem.getTasks(); i++) {
                Map.Entry<Integer, Integer> tempEntry = null;
                for (Map.Entry<Integer, Integer> entry : helper.entrySet()) {
                        if (generator == CandidateListGenerator.MAX_TIME) {
                                if (tempEntry == null || entry.getValue() >
                                    tempEntry.getValue()) {
                                        tempEntry = entry;
                                }
                        } else if (generator == CandidateListGenerator.MIN_TIME)
                            {
```

```
                                        if (tempEntry == null || entry.getValue() <
                                            tempEntry.getValue()) {
                                                tempEntry = entry;
                                        }
                                }
                        }
                        candidateList.add(tempEntry.getKey());
                        helper.remove(tempEntry.getKey());
                }
                return candidateList;
        }
}
```

B.5. VNS

```
public class VNS {
        /** The optimization problem. */
        private FlowShopProblem optimizationProblem;

        /** The best solution. */
        private PermutationSolution<FlowShopProblem> bestSolution;

        /** The initial solution. */
        private PermutationSolution<FlowShopProblem> initialSolution;

        /** The local search. */
        private LocalSearch localSearch;

        /** The iterations. */
        private Integer iterations;

        /** The random generator. */
        private Random random;

        /** The moves for the local search. */
        @SuppressWarnings("rawtypes")
        private Move[] moves;

        /**
         * Instantiates a new VNS algorithm to solve a given problem.
         *
         * @param optimizationProblem the optimization problem
         */
        public VNS(FlowShopProblem optimizationProblem) {
                super();
                setOptimizationProblem(optimizationProblem);
                setLocalSearch(new LocalSearch());
                vnsConfig();
                random = new Random();
                moves = new Move[] {new Move2Elements(), new ReInsertion(), new
                    Inversion() };
        }

        /**
         * VNS initial configuration.
```

```
 */
private void vnsConfig() {
        setInitialSolution(null);
        setBestSolution(null);
        setIterations(0);
}

/**
 * Run the algorithm VNS.
 *
 * @param criterion the stop criterion
 * @param localSearchType the local search type
 * @param generator the generator for the RCL
 * @throws Exception the exception
 */
public void run(StopCriterion criterion, LocalSearchType localSearchType,
    CandidateListGenerator generator) throws Exception {
        vnsConfig();
        // The initial solution is built using GraspInitiaSolution generator
        initialSolution = GraspInitialSolution.runGenerator(optimizationProblem,
            generator);
        initialSolution.evaluate();

        bestSolution = initialSolution.clone();

        if (criterion == StopCriterion.ITERATIONS) {
                while (iterations < Constants.MAX_ITERATIONS) {
                        Integer k = 1;
                        while (k <= Constants.K_MAX) {
                                shakeSolution(k);
                                VND(localSearchType);
                                if (bestSolution.getObjectiveFunctionValue(0) >
                                    initialSolution.getObjectiveFunctionValue(0))
                                    {
                                        bestSolution = initialSolution.clone();
                                        k = 1;
                                } else
                                        k++;
                        }
                        iterations++;
                }
        } else if (criterion == StopCriterion.NUM_IMPRO) {
                Integer noImprove = 0;
                while (noImprove < Constants.NO_IMPROVED) {
                        Integer k = 1;
                        while (k <= Constants.K_MAX) {
                                shakeSolution(k);
                                VND(localSearchType);
                                if (bestSolution.getObjectiveFunctionValue(0) >
                                    initialSolution.getObjectiveFunctionValue(0))
                                    {
                                        bestSolution = initialSolution.clone();
                                        k = 1;
                                        noImprove = 0;
                                } else {
                                        k++;
                                        noImprove++;
```

```
                                }
                        }
                        iterations++;
                }
        }
}

/**
 * Run the vnd algorithm
 * @throws Exception the exception
 */
private void VND(LocalSearchType localSearchType) throws Exception {
        Boolean exit = false;
        PermutationSolution<FlowShopProblem> tempSolution =
            initialSolution.clone();

        // Solution to explore
        localSearch.setInitialSolution(tempSolution);
        localSearch.setBestSolution(null); // Prevent other solutions saved

        do {
                // Move 2 elements
                localSearch.setMove((Move2Elements) moves[0]); // Set the move
                    for the local search
                // Solution to explore
                localSearch.setInitialSolution(tempSolution);
                localSearch.setBestSolution(null); // Prevent other solutions
                    saved
                tempSolution = localSearch.findOptimum(localSearchType);
                tempSolution.evaluate();
                if (tempSolution.getObjectiveFunctionValue(0) <
                    initialSolution.getObjectiveFunctionValue(0)) {
                        initialSolution  = tempSolution.clone();
                        // Jump to next iteration if one better was found, so it
                            starts again with the initial environment
                        continue;
                }

                // Reinsertion
                localSearch.setMove((ReInsertion) moves[1]); // Set the move for
                    the local search
                // Solution to explore
                localSearch.setInitialSolution(tempSolution);
                localSearch.setBestSolution(null); // Prevent other solutions
                    saved
                tempSolution = localSearch.findOptimum(localSearchType);
                tempSolution.evaluate();

                if (tempSolution.getObjectiveFunctionValue(0) <
                    initialSolution.getObjectiveFunctionValue(0)) {
                        initialSolution  = tempSolution.clone();
                        // Jump to next iteration if one better was found, so it
                            starts again with the initial environment
                        continue;
                }

                // Inversion
```

```
                localSearch.setMove((Inversion) moves[2]); // Set the move for
                    the local search
                // Solution to explore
                localSearch.setInitialSolution(tempSolution);
                localSearch.setBestSolution(null); // Prevent other solutions
                    saved
                tempSolution = localSearch.findOptimum(localSearchType);
                tempSolution.evaluate();

                if (tempSolution.getObjectiveFunctionValue(0) <
                    initialSolution.getObjectiveFunctionValue(0)) {
                        initialSolution  = tempSolution.clone();
                        // Jump to next iteration if one better was found, so it
                            starts again with the initial environment
                        continue;
                }

                exit = true;
        } while (!exit);
}


/**
 * Shake the initial solution k times, selecting a random element and a random
     index where
 * the element will be reinserted.
 *
 * @param k number of times a shake will be executed
 */
private void shakeSolution(Integer k) {
        ReInsertion reinsertion = new ReInsertion(); // The reinsert move

        while (k > 0) {
                int element =
                    random.nextInt(initialSolution.getRepresentation().length);
                int insertIndex =
                    random.nextInt(initialSolution.getRepresentation().length);

                while (element == insertIndex) {
                        insertIndex = random.nextInt(
                        initialSolution.getRepresentation().length);
                }

                int[] newRepresentation = reinsertion.reInsertion(
                                            initialSolution.getRepresentation(),
                                                element, insertIndex);

                // Update the solution with the new representation after
                    reinsert an element
                for (int i = 0; i < initialSolution.getRepresentation().length;
                    i++)
                        initialSolution.setValue(i, newRepresentation[i]);

                k--;
        }
        initialSolution.evaluate();
}
```

```
        /** Getters and Setters **/

        public FlowShopProblem getOptimizationProblem() {
                return optimizationProblem;
        }

        public void setOptimizationProblem(FlowShopProblem optimizationProblem) {
                this.optimizationProblem = optimizationProblem;
        }

        public PermutationSolution<FlowShopProblem> getBestSolution() {
                return bestSolution;
        }

        public void setBestSolution(PermutationSolution<FlowShopProblem> bestSolution) {
                this.bestSolution = bestSolution;
        }

        public PermutationSolution<FlowShopProblem> getInitialSolution() {
                return initialSolution;
        }

        public void setInitialSolution(PermutationSolution<FlowShopProblem>
            initialSolution) {
                this.initialSolution = initialSolution;
        }

        public LocalSearch getLocalSearch() {
                return localSearch;
        }

        public void setLocalSearch(LocalSearch localSearch) {
                this.localSearch = localSearch;
        }

        public Integer getIterations() {
                return iterations;
        }

        public void setIterations(Integer iterations) {
                this.iterations = iterations;
        }
}
```

B.6. Búsqueda local

```
public class LocalSearch {

        /** The initial solution. */
        private PermutationSolution<FlowShopProblem> initialSolution;

        /** The best solution. */
        private PermutationSolution<FlowShopProblem> bestSolution;

        /** The move. */
        private Move<PermutationSolution<FlowShopProblem>> move;

        /**
         * Instantiates a new local search.
         *
         * @param initialSolution the initial solution
         */
        public LocalSearch() {
                setInitialSolution(null);
                setBestSolution(null);
                setMove(null);
        }

        /**
         * Find local optimum with the chosen move.
         *
         * @param localSearchType the local search type
         * @return the permutation solution
         * @throws Exception the exception
         */
        public PermutationSolution<FlowShopProblem> findOptimum(LocalSearchType
            localSearchType) throws Exception {
                if (move == null)
                        throw new Exception("You are not setting a move.");

                if (initialSolution == null)
                        throw new Exception("You are not setting an initial solution.");

                bestSolution = initialSolution.clone();
                bestSolution.evaluate();
                Boolean isLocalOptimum = true;

                // Search the local optimum of the initial solution.
                // If the move.nextSolution returns a better solution, it will be a
                    local optimum
                do {
                        if (localSearchType == LocalSearchType.GREEDY)
                                initialSolution =
                                    move.nextSolutionGreedy(initialSolution);
                        else if (localSearchType == LocalSearchType.FIRST_BETTER)
                                initialSolution =
                                    move.nextSolutionFirstBetter(initialSolution);

                        initialSolution.evaluate(); // Calculate the makespan of this
                            solution
```

```
                        // Check if the new makespan is better than the actual solution.
                        if (bestSolution.getObjectiveFunctionValue(0) >
                            initialSolution.getObjectiveFunctionValue(0)) {
                                bestSolution = initialSolution.clone();
                                isLocalOptimum = false;
                        } else
                                isLocalOptimum = true;

                } while (!isLocalOptimum);

                return bestSolution;
        }

        /** Getters and Setters **/

        public void setMove(Move<PermutationSolution<FlowShopProblem>> move) {
                this.move = move;
        }

        public Move<PermutationSolution<FlowShopProblem>> getMove() {
                return this.move;
        }

        public PermutationSolution<FlowShopProblem> getInitialSolution() {
                return initialSolution;
        }

        public void setInitialSolution(PermutationSolution<FlowShopProblem>
            initialSolution) {
                this.initialSolution = initialSolution;
        }

        public PermutationSolution<FlowShopProblem> getBestSolution() {
                return bestSolution;
        }

        public void setBestSolution(PermutationSolution<FlowShopProblem> best) {
                this.bestSolution = best;
        }
}
```

B.7. Movimientos

```
public abstract class Move<S extends PermutationSolution<FlowShopProblem>> {

        /**
         * Next better solution, if exists, using greedy exploration.
         *
         * @param solution the initial solution
         * @return the solution if was found
         */
        public abstract S nextSolutionGreedy(S solution);

        /**
         * Next better solution, if exists, using first better exploration
```

```
         *
         * @param solution the initial solution
         * @return the solution if was found
         */
        public abstract S nextSolutionFirstBetter(S solution);
}

public class Inversion extends Move<PermutationSolution<FlowShopProblem>> {

        @Override
        public PermutationSolution<FlowShopProblem>
            nextSolutionGreedy(PermutationSolution<FlowShopProblem> solution) {
                PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
                bestSolution.evaluate();

                int first = 0;
                int second = 1;

                // It will exit after having gone through all the solutions,
                // no matter if it was found or not
                while (first != solution.getRepresentation().length - 1) {
                        int[] newRepresentation =
                            inversion(solution.getRepresentation(), first, second);

                        // Update the solution with the new representation after
                            reinsert an element
                        for (int i = 0; i < solution.getRepresentation().length; i++)
                                solution.setValue(i, newRepresentation[i]);

                        solution.evaluate(); // Evaluate the solution

                        // After evaluate the move, check if it's better
                        if (solution.getObjectiveFunctionValue(0) <
                            bestSolution.getObjectiveFunctionValue(0))
                                bestSolution = solution.clone();

                        if (second == solution.getRepresentation().length - 1) {
                                first++;
                                second = first + 1;
                        } else
                                second++;
                }
                return bestSolution;
        }

        @Override
        public PermutationSolution<FlowShopProblem>
            nextSolutionFirstBetter(PermutationSolution<FlowShopProblem> solution) {
                PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
                bestSolution.evaluate();

                int first = 0;
                int second = 1;

                Boolean found = false; // A better solution found

                // It will exit if found or if finished without found
```

```
        while (first != solution.getRepresentation().length - 1 && !found) {

                int[] newRepresentation =
                    inversion(solution.getRepresentation(), first, second);

                // Update the solution with the new representation after
                    reinsert an element
                for (int i = 0; i < solution.getRepresentation().length; i++)
                        solution.setValue(i, newRepresentation[i]);

                solution.evaluate(); // Evaluate the solution

                if (solution.getObjectiveFunctionValue(0) <
                    bestSolution.getObjectiveFunctionValue(0)) {
                        bestSolution = solution.clone();
                        found = true;
                }

                if (!found) {
                        if (second == solution.getRepresentation().length - 1) {
                                first++;
                                second = first + 1;
                        } else
                                second++;
                }
        }
        return bestSolution;
}

/**
 * It inverts all the elements between two given index
 *
 * @param array the array to invert
 * @param first first index
 * @param second second index
 * @return array with elements inverted
 */
private int[] inversion(int[] array, int first, int second) {
        while (first < second) {
                array = swap(array, first, second);
                first++;
                second--;
        }
        return array;
}

/**
 * Swap two elements in an array
 *
 * @param array the array
 * @param first first element index
 * @param second second element index
 * @return array with elements swapped
 */
private int[] swap(int[] array, int first, int second) {
        int aux = array[first];
        array[first] = array[second];
```

```
                array[second] = aux;
                return array;
        }

        @Override
        public String toString() {
                return "Inversion";
        }
}

public class Move2Elements extends Move<PermutationSolution<FlowShopProblem>> {

        @Override
        public PermutationSolution<FlowShopProblem>
            nextSolutionGreedy(PermutationSolution<FlowShopProblem> solution) {
                PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
                bestSolution.evaluate();

                int firstElement = 0;
                int secondElement = firstElement + 1;

                // It will exit after having gone through all the solutions, no matter
                    if it was found or not
                while (firstElement != solution.getRepresentation().length - 1) {

                        solution.swap(firstElement, secondElement);
                        solution.evaluate();

                        if (solution.getObjectiveFunctionValue(0) <
                            bestSolution.getObjectiveFunctionValue(0))
                                bestSolution = solution.clone();

                        if (secondElement == solution.getRepresentation().length - 1) {
                                solution.swap(firstElement, secondElement);
                                firstElement++;
                                secondElement = firstElement + 1;
                        } else {
                                solution.swap(firstElement, secondElement);
                                secondElement++;
                        }
                }
                return bestSolution;
        }

        @Override
        public PermutationSolution<FlowShopProblem>
            nextSolutionFirstBetter(PermutationSolution<FlowShopProblem> solution) {
                PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
                bestSolution.evaluate();

                int firstElement = 0;
                int secondElement = firstElement + 1;

                Boolean found = false; // A better solution found

                // It will exit if found or if finished without found
```

```
                while (firstElement != solution.getRepresentation().length - 1 &&
                    !found) {

                        solution.swap(firstElement, secondElement);
                        solution.evaluate();

                        if (solution.getObjectiveFunctionValue(0) <
                            bestSolution.getObjectiveFunctionValue(0)) {
                                bestSolution = solution.clone();
                                found = true;
                        }

                        if (!found) {
                                if (secondElement == solution.getRepresentation().length
                                    - 1) {
                                        solution.swap(firstElement, secondElement);
                                        firstElement++;
                                        secondElement = firstElement + 1;
                                } else {
                                        solution.swap(firstElement, secondElement);
                                        secondElement++;
                                }
                        }
                }
                return bestSolution;
        }

        @Override
        public String toString() {
                return "Move2Elements";
        }
}

public class ReInsertion extends Move<PermutationSolution<FlowShopProblem>> {

        @Override
        public PermutationSolution<FlowShopProblem>
            nextSolutionGreedy(PermutationSolution<FlowShopProblem> solution) {
                PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
                bestSolution.evaluate();

                int element = 0;
                int insertIndex = 1;

                // It will exit after having gone through all the solutions, no matter
                    if it was
                // found or not
                while (element != solution.getRepresentation().length) {
                        // Avoid reinsert in the same position
                        if (element != insertIndex) {
                                int[] newRepresentation =
                                    reInsertion(solution.getRepresentation(), element,
                                    insertIndex);

                                // Update the solution with the new representation after
                                    reinsert an element
```

```
                        for (int i = 0; i < solution.getRepresentation().length;
                            i++)
                                solution.setValue(i, newRepresentation[i]);

                        solution.evaluate(); // Evaluate the solution

                        // After evaluate the move, check if it's better
                        if (solution.getObjectiveFunctionValue(0) <
                            bestSolution.getObjectiveFunctionValue(0))
                                bestSolution = solution.clone();

                        if (insertIndex == solution.getRepresentation().length -
                            1) {
                                element++;
                                insertIndex = 0;
                        } else
                                insertIndex++;

                } // If the element and the index are the same, check the index
                    is not at the end
                else if (insertIndex == solution.getRepresentation().length - 1)
                        element++;
                else
                        insertIndex++;
        }
        return bestSolution;
}

@Override
public PermutationSolution<FlowShopProblem>
    nextSolutionFirstBetter(PermutationSolution<FlowShopProblem> solution) {
        PermutationSolution<FlowShopProblem> bestSolution = solution.clone();
        bestSolution.evaluate();

        int element = 0;
        int insertIndex = 1;

        Boolean found = false; // A better solution found

        // It will exit if found or if finished without found
        while (element != solution.getRepresentation().length - 1 && !found) {

                // Avoid reinsert in the same position
                if (element != insertIndex) {
                        int[] newRepresentation =
                            reInsertion(solution.getRepresentation(), element,
                            insertIndex);

                        // Update the solution with the new representation after
                            reinsert an element
                        for (int i = 0; i < solution.getRepresentation().length;
                            i++)
                                solution.setValue(i, newRepresentation[i]);

                        solution.evaluate(); // Evaluate the solution
```

```
                        if (solution.getObjectiveFunctionValue(0) <
                            bestSolution.getObjectiveFunctionValue(0)) {
                                bestSolution = solution.clone();
                                found = true;
                        }

                        if (!found) {
                                if (insertIndex ==
                                    solution.getRepresentation().length - 1) {
                                        element++;
                                        insertIndex = 0;
                                } else
                                        insertIndex++;
                        }
                } // If the element and the index are the same, check the index
                    is not at the end
                else if (insertIndex == solution.getRepresentation().length - 1)
                        element++;
                else
                        insertIndex++;
        }
        return bestSolution;
}

/**
 * Reinsert an element in the array
 *
 * @param array the array with the elements
 * @param element the element to move
 * @param index the new position of the element
 * @return an array with the element reinserted
 */
public int[] reInsertion(int[] array, int element, int index) {

        int tempElement = array[element]; // Save the element

        return addPos(removeElement(array, element), index, tempElement);
}

/**
 * Using an index, it removes the element located in that index
 *
 * @param array the array with elements
 * @param index the position of the element
 * @return an array without the element
 */
private static int[] removeElement(int[] array, int index) {

        // Create another array of size one less
        int[] result = new int[array.length - 1];

        // Copy the elements except the index
        // from original array to the other array
        for (int i = 0, k = 0; i < array.length; i++) {

                // if the index is the removal element index
                if (i == index) {
```

```
                        continue;
                }

                // if the index is not the removal element index
                result[k++] = array[i];
            }
            return result;
        }

        /**
         * Adds to the array an element in the specified index
         *
         * @param array the array where the element will be inserted
         * @param index the index where it will be inserted
         * @param element the element to be inserted
         * @return an array with the element inserted and the others shifted
         */
        private static int[] addPos(int[] array, int index, int element) {
            int[] result = new int[array.length + 1];

            // Keep all the elements before the index
            for (int i = 0; i < index; i++)
                result[i] = array[i];

            result[index] = element; // Set the element at index

            // Keep all the elements after the index
            for (int i = index + 1; i < array.length + 1; i++)
                result[i] = array[i - 1];

            return result;
        }

        @Override
        public String toString() {
            return "Reinsertion";
        }
}
```

Apéndice C

Simulación

C.1. Evento

```
public class Event implements Comparable<Event> {

        private String id; // The id for know what event is it
        private double time; // The moment when the event happens

        /**
         * Instantiates a new event.
         *
         * @param time the time
         * @param id the id
         */
        public Event(double time, String id) {
        this.time = time;
        this.id = id;
    }

        /**
         * Sets the parameters.
         *
         * @param time the time
         * @param id the id
         */
        public void setParameters(double time, String id) {
        this.time = time;
        this.id = id;
    }

    /**
     * Gets the id.
     *
     * @return the id
     */
    public String getId() {
        return id;
    }

    /**
     * Gets the time.
     *
```

```
     * @return the time
     */
    public double getTime() {
        return time;
    }

    /* (non-Javadoc)
     * @see java.lang.Comparable#compareTo(java.lang.Object)
     */
    @Override
    public int compareTo(Event event) {
        if (this.getTime() > event.getTime())
            return 1;
        else if (this.getTime() < event.getTime())
            return -1;
        else
            return 0;
    }
}
```

C.2. Simulación de eventos discretos

```
public class Simulation {

        /** Variables that holds information about the optimized problem */

        /** The optimization problem. */
        private FlowShopProblem optimizationProblem;

        /** The schedule. */
        private int[] schedule;

        /** Class variables **/

        /** The time line for the events. */
        private PriorityQueue<Event> timeLine;

        /** The total makespan. */
        private double makespan;

        /** The random variable. */
        private Random random;

        /** The uncertainity. */
        private double uncertainty;


        /**
         * Instantiates a new simulation.
         */
        public Simulation() {
                this.optimizationProblem = null;
                this.schedule = null;
                this.makespan = 0;
                this.timeLine = new PriorityQueue<Event>();
```

```
        this.random = new Random();
        this.uncertainty = 1.0;
}

/**
 * Runs the simulation.
 * @throws Exception
 */
public void run() throws Exception {
        makespan = 0;

        if (optimizationProblem == null || schedule == null)
                throw new Exception("You must set an optimization problem and a
                    schedule");

        // Make sure the time line is clear and the makespan is 0
        timeLine.clear();
        makespan = 0.0;

        // Precalculate the times of execution with uncertainty
        double[][] simulationTimes = buildSimulationTimes(uncertainty);

        // Some helpers for hold time
        double helper = 0.0;
        double helper2 = 0.0;
        for (int i = 0; i < schedule.length; i++) { // For each task
                if (i == 0) { // If the task is the first
                        for (int j = 0; j <
                            getOptimizationProblem().getMachines(); j++) { //
                            For each machine
                                // Create the start event of the task in the
                                    machine j, add the time to the helper
                                // The event needs the exact moment where it
                                    happens
                                Event inicio = new Event(helper, ("Ini_" + (i +
                                    1) + "_" + (j + 1)));
                                // Parse is used for prevent more than four
                                    decimal numbers
                                helper = Double.parseDouble(String.format
                                        (Locale.ROOT, "%.4f",
                                        (helper + simulationTimes[schedule[i] -
                                            1][j])));
                                Event fin = new Event(helper, ("Fin_" + (i + 1)
                                    + "_" + (j + 1)));

                                // In the helper2 we hold the finish time of the
                                    first task.
                                // This is for calculate the delay between tasks
                                    after it finishes without lost the
                                    information
                                if (j == 0)
                                        helper2 = helper;

                                // Add the events to the time line
                                timeLine.add(inicio);
                                timeLine.add(fin);
                        }
```

```
                    } else {
                            for (int j = 0; j <
                                getOptimizationProblem().getMachines(); j++) {
                                    if (j == 0)
                                            helper =
                                                Double.parseDouble(String.format(
                                                    Locale.ROOT, "%.4f", (helper2 +
                                                        delayTime(schedule[i - 1],
                                                        schedule[i],
                                                            getOptimizationProblem().
                                                            getMachines(),
                                                                simulationTimes))));

                                    Event inicio = new Event(helper, ("Ini_" + (i +
                                        1) + "_" + (j + 1)));
                                    helper = Double.parseDouble(
                                                String.format(Locale.ROOT, "%.4f",
                                                (helper +
                                                    simulationTimes[schedule[i] -
                                                    1][j])));
                                    Event fin = new Event(helper, ("Fin_" + (i + 1)
                                        + "_" + (j + 1)));
                                    if (j == 0)
                                            helper2 = helper;

                                    timeLine.add(inicio);
                                    timeLine.add(fin);
                            }
                    }
            }
            simulateEvents();
    }

    private void simulateEvents() {
    while (!timeLine.isEmpty()) {
        Event event = timeLine.poll();
        makespan = event.getTime();
    }
}

    /**
     * Builds the simulation times using uncertainty.
     *
     * @param uncertainty the uncertainty
     * @return the double[][] matrix with the new times
     */
    private double[][] buildSimulationTimes(double uncertainty) {
            double[][] simulationTimes = new
                double[schedule.length][getOptimizationProblem().getMachines()];
            for (int i = 0; i < getOptimizationProblem().getTimes().length; i++) {
                    for (int j = 0; j <
                        getOptimizationProblem().getTimes()[i].length; j++) {
                            simulationTimes[i][j] = generateRandomTime(
                                                    getOptimizationProblem()
                                                    .getTimes()[i][j], uncertainty);
                    }
            }
```

```
            return simulationTimes;
    }

    /**
     * Generate the execution time for a task adding an uncertainty.
     *
     * @param value the value
     * @param uncertainty the uncertainty
     * @return the value generated
     */
    private double generateRandomTime(double value, double uncertainty) {
            double discount = (uncertainty * value) / 10.0;
            double max = value + discount;
            double min = value - discount;
            return Double.parseDouble(String.format(Locale.ROOT, "%.4f", min + (max
                - min) * random.nextDouble()));
    }

    /**
     * Determine the minimum delay time between the completion of job Jp
     * and the initiation of Jq using the Reddi and Ramamoorthy formula
     *
     * @param p the previous task p
     * @param q the actual task q
     * @param machines the number of machines
     * @param times the times of execution
     * @return the delay time between the completion of job Jp
     * and the initiation of Jq
     */
    private double delayTime(int p, int q, int machines, double[][] times) {
    double delay = 0;

    for (int i = 1; i < machines; i++)
        delay += times[p - 1][i];

    for (int i = 0; i < machines - 1; i++)
        delay -= times[q - 1][i];

    return (delay <= 0.0) ? 0.0 :  Double.parseDouble(String.format(Locale.ROOT,
        "%.4f", delay));
}

    /** Getters and Setters **/

    public FlowShopProblem getOptimizationProblem() {
            return optimizationProblem;
    }

    public void setOptimizationProblem(FlowShopProblem optimizationProblem) {
            this.optimizationProblem = optimizationProblem;
    }

    public int[] getSchedule() {
            return schedule;
    }
```

```
	public void setSchedule(int[] schedule) {
		this.schedule = schedule;
	}

	public PriorityQueue<Event> getTimeLine() {
		return timeLine;
	}

	public void setTimeLine(PriorityQueue<Event> timeLine) {
		this.timeLine = timeLine;
	}

	public double getMakespan() {
		return makespan;
	}

	public void setMakespan(double makespan) {
		this.makespan = makespan;
	}

	public Random getRandom() {
		return random;
	}

	public void setRandom(Random random) {
		this.random = random;
	}

	public double getUncertainty() {
		return uncertainty;
	}

	public void setUncertainty(double uncertainity) {
		this.uncertainty = uncertainity;
	}
}
```

Bibliografía

[1] Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Pinedo, M. L. (2012), 2012.

[2] Flow shop scheduling. https://en.wikipedia.org/wiki/Flow_shop_scheduling.

[3] Hamilton Emmons and George Vairaktarakis. *The No-Wait Flow Shop*. Springer, Boston, MA, 2012.

[4] Xianxia Zhang Bing Wang, Xingbao Han. Predictive-reactive scheduling for single surgical suite subject to random emergency surgery. 2015.

[5] El-Ghazali Talbi. *Metaheuristics: From Design to Implementation*. Wiley, 2009.

[6] George S. Fishman. *Discrete-Event Simulation: Modeling, Programming, and Analysis*. Springer, 2001.

[7] Melvyn Jeter. *Mathematical Programming: An Introduction to Optimization*. Marcel Dekker, Inc. (1683), 1986.

[8] Chander Mohan and Kusum Deep. *Optimization Techniques*. New Age Science, 2009.

[9] Jan K. Lenstra Emile L. Aarts and Emile Aarts. *Local Search in Combinatorial Optimization*. Princeton University Press, 1997.

[10] Anylogic. https://www.anylogic.com.

[11] Simio. https://www.simio.com/index.php.

[12] Promodel. http://promodel.com.mx/promodel/.

[13] Service model. http://cort.as/-Oynq.

[14] Powersim studio. https://www.powersim.com/.

[15] Ramamoorthy Reddi, S.S. On the flow shop sequencing problem with no-wait in process. 1972.

[16] Java home page. `https://www.java.com/es/`.

[17] Mauricio G.C. Resende and Celso C. Ribeiro. *Optimization by GRASP: Greedy Randomized Adaptive Search Procedures*. Springer, 2016.

[18] Angelo SifalerasSaid SalhiJack Brimberg. *Variable Neighborhood Search*. Springer, 2018.

[19] IMRAN ALI CHAUDHRY and ABDUL MUNEM KHAN. Minimizing makespan for a no-wait flowshop using genetic algorithm. 2012.