Trabajo de Simulación en Grupo

Mikel García

Table of Contents

# 1 Formulación del problema

Como primera aproximación formulamos el problema como un problema de programación lineal. Las variables de decisión representan la cantidad de empleados contratados con tipo de jornada que comienzan su turno en la hora . Los índices y pueden tomar los valores y . Codificamos la jornada completa como , la reducida como y la parcial como . El índice toma valores naturales empezando en la hora a la que abre el supermercado por conveniencia, de esta manera se puede calcular la hora del fin de turno del empleado sumando al índice la duración del turno. El valor de depende de ya que, por ejemplo, un empleado con contrato a jornada completa tiene que trabajar 8 horas diarias, pero no puede terminar más tarde de las 21:00, hora de cierre del supermercado.

Se deduce la hora máxima a la que un empleado puede iniciar su turno a partir de esta regla.

* Jornada completa:
* Jornada reducida:
* Jornada parcial:

Por tanto, tenemos un total de 5 variables de tipo jornada completa, 7 de jornada reducida y 9 de jornada parcial. Es posible representar la solución como una lista de estas variables .

Respecto a la función objetivo, queremos minimizar el coste por contratación de los empleados. No se conoce el coste de la jornada completa, por lo que supondremos que tiene un valor . De esta manera podemos expresar el resto de los costes en función de esta cantidad. Por tanto, el coste de la jornada reducida es y el coste de la jornada parcial es . La función objetivo será la suma de estos costes

Por otro lado, el problema cuenta con una serie de criterios de calidad. Estos son:

1. La longitud media de la cola debe ser menor a dos clientes.
2. El 90% de los clientes esperan menos de 10 minutos.

No podemos traducir directamente estos criterios de calidad a restricciones del problema ya que no hay una expresión matemática evidente para relacionar las variables de decisión. Por el momento, para terminar de formular el problema vamos a introducir dos estadísticos , longitud media de la cola, y , cantidad de gente que espera menos de 10 minutos en la cola. Se puede entender estos estadísticos como funciones que dependen de .

Dada la naturaleza de las restricciones tendremos que simular las condiciones dadas por para tener una estimación de y .

# 2 Simulación

El objetivo es crear una simulación que contenga los elementos esenciales del sistema. Para entender su funcionamiento podemos ver el supermercado desde la perspectiva de un gerente. Sin pérdida de generalidad, podemos describir lo que pasa en un día.

Un cliente entra al supermercado dentro del horario de atención al cliente, pasa algún tiempo recorriendo los pasillos y cuando ha terminado de recoger los artículos que necesita se dirige a alguna de las cajas. En las cajas puede haber un empleado o no y los clientes solo pueden pagar por sus artículos en las cajas en las que hay un empleado. Los empleados pueden estar esperando para atender a un cliente o atendiendo a un cliente, en cuyo caso, se formará una única cola de clientes que esperan a que algún cajero vuelva a estar libre. La cola sigue una disciplina FIFO y el tiempo de espera del cliente depende del tiempo que pasan en la caja otros clientes y del número de empleados que haya en ese momento. La longitud de la cola, por otra parte, depende de la tasa de finalización de compra de los clientes y la tasa de servicio de los empleados. Ambas tasas pueden variar con el tiempo. Una vez el cliente realiza el pago en la caja abandona el supermercado.

## 2.1 Elementos del sistema

Sabiendo el funcionamiento del sistema podemos definir los elementos que participan en él. Los más evidentes son los clientes, las cajas y la cola.

Respecto a los clientes, conocemos la hora entrada al supermercado, la hora en la que finalizan su compra yendo a continuación a la cola o a una caja que esté libre y el tiempo que pasa en la caja o tiempo de pago.

Las cajas, por otra parte, pueden estar ocupadas con un empleado o no, es decir, pueden estar activas o cerradas. Y a su vez un empleado puede estar ocupado atendiendo un cliente o libre.

Por último, la cola es una lista que guarda los clientes en orden de incorporación.

El supermercado también es un elemento del sistema, en concreto, puede verse como una sala de espera de capacidad infinita.

En consecuencia, tenemos las siguientes variables de estado:

* Número de clientes pendientes de entrar
* Número de clientes pendientes de terminar su compra
* longitud de la cola
* Estado de la caja, si hay un cajero en ella o no
* El estado del cajero, si está libre u ocupado
* El siguiente instante en el que el cajero está libre
* El número de cajeros del turno actual

Como se ha descrito antes, hay cierta interacción entre los elementos del sistema. Las variables de estado del sistema se actualizan en consecuencia de esta interacción.

* Al incorporarse a la cola, un cliente aumenta la longitud de esta.
* Si un cajero atiende a un cliente se actualiza su estado de libre a ocupado. Igualmente, si un cajero termina de atender a un cliente su estado se actualiza de ocupado a libre.
* Cuando un cajero atiende a un cliente se actualiza el momento en el que el cajero vuelve a estar libre sumando el tiempo de pago del cliente a la variable de estado.

Las demás variables de estado dependen del valor del reloj de simulación, por ejemplo, el estado de la caja se actualiza a las 9:00, 10:00, etc. Ocurre lo mismo para el número de cajeros del turno.

## 2.2 Calendario de eventos

El calendario de eventos es un elemento de la simulación que orquesta como esta ocurre. A través de los eventos presentes en él se actualizan las variables de estado. Para esta simulación tenemos que contar con un evento “Entrada de cliente”, otro evento “Fin de compra”, y tantos eventos “Cajero libre” como cajeros haya como máximo en el sistema. También necesitamos un evento “cambio de turno” y “fin de simulación”. Nótese que este último marca la finalización del día, y no la finalización de la ejecución de todos los días para los que estudiamos el sistema.

## 2.3 Variables acumuladoras

El principal interés que tiene simular el sistema es calcular los valores de las restricciones que dependen de y , que se calculan al final de la simulación. A lo largo de la simulación se recogen observaciones de unas variables que no participan en el sistema. Estas variables acumuladoras son las que después se utilizan para obtener el valor de los estadísticos.

Las variables acumuladoras serán , número de clientes en la cola en el instante y , el cliente i-ésimo ha esperado menos de 10 minutos en la cola. Si el cliente ha esperado más de 10 minutos , en caso contrario.

El cálculo del estadístico se hará a través de la media

Como puede valer o .

Por otro lado, para calcular la media se utilizará una integral que dividiremos entre el límite superior de la integral

Aunque debería de ser 21, sabemos que algunos clientes se incorporan a la cola a esa hora, por lo que, aunque la longitud de la cola solo disminuye desde ese momento, puede ser que siga habiendo clientes esperando a ser atendidos mucho después de esta hora.

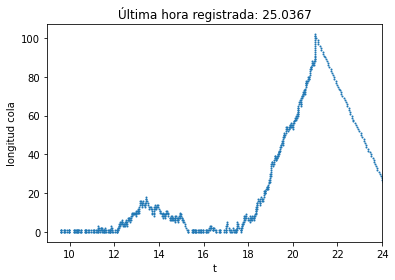


Figura 1: Longitud de la cola a lo largo de un día. Se observa que el servicio no termina hasta la madrugada del día siguiente

Si bien, considerar que el límite superior de la integral queda por encima de 21 no es suficiente, ya que se obtiene un valor de la media aceptable a costa de retrasarse en la hora de cierre del supermercado. Por este motivo definimos una función auxiliar

que utilizaremos para ponderar observaciones que queden por encima de la hora de cierre del supermercado.

De esta manera penalizamos que el servicio a los clientes finalice muy tarde obteniendo valores de longitud media más grandes si se supera la hora de cierre del supermercado.

# 3 Implementación

El sistema fue implementado en Python 3.10 desde cero, utilizando las librerías que ya vienen instaladas por defecto para implementar la mayor parte de este. Para añadir ciertas funcionalidades se utilizaron las librerías Pandas, para la lectura de los datos, Numpy, para operar arrays y matplotlib, para crear algunos gráficos. Se optó por el paradigma de la Programación Orientada a Objetos para implementar los elementos del sistema, reloj de simulación y calendario de eventos. Mientras que el paradigma de Programación Funcional se utilizó para implementar la simulación, la generación de soluciones y un método de búsqueda para hallar soluciones mejores.

## 3.1 representación de la solución

Dada la relación entre el optimizador y el simulador una solución debe estar en un formato con el que ambos puedan trabajar. Cada uno utiliza aspectos distintos de la solución, por ejemplo, el optimizador necesita conocer el valor de las variables de decisión para poder asignar un valor de la función objetivo en función de estas, mientras que el simulador solo necesita conocer la cantidad de empleados que hay en un turno sin importar su tipo de jornada u hora de inicio.

Por este motivo se tomará la representación descrita en la formulación del problema como representación base y se crea una representación alternativa de la solución que se utilizará como input para el simulador. Dicha representación se calculará primero, haciendo sumas acumuladas de un turno a otro para los tres tipos de jornadas, dicho de otro modo, si un empleado empieza su turno a las 9 terminará su jornada a las 13:00, 15:00 o 17:00 en función de su tipo de jornada. Por lo tanto, debe contar en todos los turnos entre su hora de inicio y su hora de finalización. En segundo lugar, haremos la suma por turnos de empleados de los tres tipos de jornadas. De esta manera tenemos una representación por turnos de la solución que indicará el número de empleados disponibles para atender clientes por hora

## 3.2 Clases

Aunque todos los elementos del sistema deben de formar parte de la simulación no todos son implementados como clase. El criterio para implementar un elemento como clase ha sido su complejidad. Si la interacción con un elemento resulta muy aparatosa como para lidiar con ella en la propia simulación el código para dicha interacción encapsula dentro de un método de la clase. En otro caso se trabaja con el elemento del sistema como una tupla o lista.

### 3.2.1 Cola

La clase Cola cuenta con una lista en la que se pueden agregar o quitar clientes, representados por una tupla que contiene la hora de incorporación a la cola y su tiempo de pago. Además, cuenta con dos variables acumuladoras; TiemposEspera que registra el tiempo que pasa entre que un cliente se pone a la cola y que es atendido, y longitudCola que registra la longitud de la cola cada vez que se actualice, así como el instante en el que se produzca este cambio.

Dados los estadísticos que queremos estudiar, en lugar de definir las variables acumuladoras con independencia de las clases se han definido dentro de la Cola, cada vez que esta se inicializa las variables acumuladoras se inicializan vacías.

Consta de un método aumentarCola que añade una tupla a la cola que contiene la información del cliente junto con el tiempo actual. De igual manera actualiza el acumulador longitudCola.

Por otro lado, tenemos el método actualizar. Su función es comprobar si el cliente ha esperado más de 10 minutos, guardando el valor 0 si es así en TiemposEspera y guardando 1 en otro caso, y añadir la longitud de la cola en el nuevo instante.

La clase cuenta con otros métodos para eliminar elementos de la lista devolviendo dicho elemento, mostrar la cola por pantalla u obtener su longitud. los dos últimos se implementaron para dar información de la simulación al usuario en caso de que la necesite.

### 3.2.2 Cajero

Esta clase se ha creado con el propósito de simular el servicio que presta un empleado en el supermercado. Tiene varios métodos que permiten gestionar el estado y su interacción con la cola de clientes. En esta clase los métodos de representación de Python han sido sobrescritos para proporcionar una salida que pueda entender el usuario.

El constructor de la clase inicializa las variables ocupado, si está atendiendo a un cliente, y activo, si hay un empleado o no de servicio, con el valor False. La clase también contiene un atributo cliente, que tomará como valor la tupla que representa al cliente cuando el empleado esté atendiendo un cliente y un atributo cuandoLibre que indica el próximo momento en el que el cajero podrá atender a un cliente si este se mantiene activo. Para este último atributo tomamos un valor arbitrariamente grande y conocido mientras el cajero no esté atendiendo a un cliente.

En la clase hay definidos dos métodos “getter” estaActivo y estaOcupado que devuelven el valor de las variables activo y ocupado respectivamente.

El método ocupar recibe una instancia de Cola y de Reloj. Marca al cajero como ocupado y toma el primer cliente en la cola actualizando la variable cuandoLibre con el tiempo en que el cajero estará libre después de atender al cliente. También llama al método actualizar de la cola añadiendo una observación más a las dos variables acumuladoras. Por último, devuelve el cliente que se va a atender. liberar, por otra parte, cumple un papel antagónico al de ocupar, cambia ocupado a False y devuelve el cliente que acaba de ser atendido.

Los métodos getCuandoLibre devuelve el tiempo en que el cajero estará libre. Mientras que el método setCuandoLibre establece el tiempo en que estará libre.

Los métodos setCliente y getCliente establecen y obtienen, respectivamente, el cliente actual atendido por el cajero.

Por último, contamos con un método cerrarCaja que marca el cajero como inactivo.

### 3.2.3 Cajas

La implementación de esta clase no es estrictamente necesaria, si bien, proporciona una interfaz más cómoda para trabajar con un conjunto de instancias de Cajero, instanciar el número de cajeros que requiere la simulación, iterar sobre ellos y efectuar los cambios de turno cuando sean necesarios. Como esta clase está diseñada para crear instancias de Cajero también necesitamos sobrescribir los métodos de representación de la clase para que así la representación ya implementada de los cajeros sea legible. Igualmente, se debe hacen una implementación de los métodos \_\_iter\_\_ y \_\_next\_\_ para poder iterar sobre la colección.

La clase cuenta con dos métodos, el primero, cambiarCajeros, se utilizará únicamente para los cambios de turno y recibirá una lista con los cambios definidos al principio de la simulación. Mientras que el segundo, algunaOcupada, itera sobre todos los cajeros para comprobar si alguno está atendiendo a un cliente. Que este método devuelva False será la condición de parada de la simulación.

### 3.2.4 Reloj

Aunque el paso del tiempo en la simulación podría registrarse con una variable numérica se ha optado por implementar una clase sencilla para seguir el avance de la simulación. Esencialmente tiene dos métodos; getTiempo para obtener el instante actual de la simulación (que ya hemos visto que se necesita en métodos de otras clases), y actualizar.

El interés que tiene implementar esta clase es que, en el método actualizar, se comprueba el valor actual del tiempo con el siguiente momento al que se quiere avanzar. Esto no evita que los eventos ocurran de manera desordenada, pero en caso de que se quiera avanzar a un tiempo menor que el actual se lanzará una excepción parando como consecuencia la simulación. En fases tempranas del desarrollo era especialmente útil para evitar el desperdicio de tiempo y esfuerzo computacional dentro de una simulación incompleta o defectuosa.

Al igual que otras clases, cuenta con métodos de representación propios.

### 3.2.5 Calendario

El calendario es la clase más compleja de la implementación. Se dará una descripción breve de sus métodos y constructor, sin por ello pasar por alto los detalles más importantes.

* \_\_init\_\_: es el constructor de la clase. Recibe varios parámetros, como una lista de entradas, una lista de fin de compras, un número máximo de cajeros, el número de cajeros por turno y la diferencia entre turnos. En este método, se inicializan los atributos de la clase.
* siguienteEvento: devuelve el siguiente evento que ocurre en el calendario, este será aquel con el tiempo más pequeño.
* actualizar: Al final de la iteración se actualiza del calendario aquellos eventos que necesiten un nuevo tiempo.
* actualizarCajeroLibre: actualiza el valor del próximo evento “CajeroLibre”. La razón por la que la actualización de la ocupación del cajero tiene un método propio es que al iterar en la simulación siempre se comprueba si hay un cajero libre que pueda atender a un cliente. En este caso se actualiza el evento dando el tiempo de finalización de servicio. Por otro lado, si el cajero no está ocupado el evento recibirá un valor arbitrariamente grande.
* cambiarTurno: se produce siempre que el reloj llegue a en punto, recibe una lista con el número de cajas que deben ser abiertas o cerradas, pero no indica qué cajas tiene que cambiar su estado. Para abrir cajas se escogerán aquellas que estén cerradas. Por otro lado, se priorizará el cierre de las cajas que terminen antes de atender clientes.
* finalizarDia: Se da un valor arbitrario y grande a los eventos que se sabe que no van a volver a ocurrir en el día simulado. Puede ocurrir que se llegue a las 21:00 y siga habiendo clientes en la cola. Con este método se evita que se lance la misma instancia del mismo evento incurriendo en un bucle infinito.
* finalizar: verifica si todos los eventos en el calendario han terminado.

## 3.3 Simulacion

Para poder abordar el problema, se ha descompuesto el bucle de simulación en dos partes. Por un lado, se ha programado la función probarSolucion que recibe como parámetros el dataframe con los datos de los clientes y una posible solución del problema en su representación alternativa. Por otro lado, contamos con el probador de soluciones que es la simulación, implementada con el mismo nombre. Como este es un caso de simulación con punto de finalización, se consideran los días como simulaciones distintas.

Recordemos que los datos están guardados en un dataframe que contiene una columna que marca el día al que pertenecen los datos. Podemos hacer consultas al dataframe de manera que nos devuelva todas las observaciones pertenecientes a un día concreto. Este subconjunto de observaciones del dataframe es el que le pasaremos al simulador en forma de listas, cada columna será una lista. Además de las listas EntradaCliente, FinCompra y TiempoPago se debe calcular la lista de cambios de turno, que contiene la diferencia de caja activas de un turno a otro. Dados estos elementos lo único que falta es instanciar las clases descritas en el anterior apartado e inicializarlas en función de los valores de la representación alternativa de la solución. Con esto podemos lanzar la simulación.

La simulación devuelve el objeto Cola, que contiene las variables acumuladoras. A través de dos funciones, calcularMedia y calcularPercentil se resumen los datos de las variables acumuladoras, calculando una integral en el primer caso y sumando el número de unos y ceros en el otro. En ambos casos se devuelve una lista con los valores antes mencionados y el número de instancias que se han utilizado para calcularlos . La idea es guardar las distintas listas para poder evaluar la solución de manera global, haciendo la suma de las filas y dividiendo la primera posición ente la segunda o evaluar la solución día por día, dividiendo directamente las primeras posiciones entre las segundas. La evaluación global se utilizará para calcular la factibilidad de la solución, así como su fitness. Mientas que la segunda se utilizará para ver la evolución de los estadísticos a lo largo de los días.

Como se ha mencionado antes, la simulación reproduce los eventos que ocurren durante un día. Esencialmente, se trata de un bucle while que iterará mientras haya personas pendientes pagar, ya sea porque están en la cola o porque estén siendo atendidas en ese momento y por tanto los cajeros aún están ocupados. Se necesitan estas dos condiciones porque, por un lado, puede ser que se haya eliminado el último elemento de tiemposPago y que los cajeros sigan atendiendo clientes. Si solo se utilizara la primera condición la simulación terminaría antes de tiempo. Por otro lado, si solo nos fijásemos en si los cajeros están libres o no en el momento en el que estuvieran todos libres terminaría la simulación sin importar si hay clientes pendientes de entrar en el supermercado, terminar su compra o en la cola.

Al principio del bucle se obtiene el siguiente evento del calendario y se avanza el reloj al instante marcado por el evento. A continuación, hay cuatro condicionales para los eventos “entrada”, “finCompra”, “cajeroLibre” y “cambioTurno”. Cada uno supone una rutina alternativa en la ejecución del programa:

* “entrada”: elimina el primer elemento de la lista EntradaCliente.
* “finCompra”: elimina el primer elemento de la lista finCompra y se obtiene su tiempo de pago asociado eliminándolo también de TiempoPago. Añadimos este tiempo a la cola junto con la hora actual del reloj a fin de calcular más tarde el tiempo de espera.
* “cajeroLibre”: Se busca qué cajero ha terminado de servir a un cliente dentro del conjunto de cajas. Se libera el cajero y por último se actualiza el evento “cajeroLibre”.
* “cambioTurno”: Obtenemos la diferencia de cajeros entre los dos turnos para la nueva hora. Si hay cambio se cierran o abren las cajas que sean necesarias, en otro caso no se hace nada.

Después de gestionar los eventos iteramos sobre los cajeros. Para cada cajero se comprueba si está ocupado, si está activo y si hay una cola de clientes. Si el cajero no está ocupado, está activo y hay cola se ocupa el cajero y se actualiza el valor del evento “cajeroLibre”.

Tras suficientes iteraciones la lista TiemposPago quedará vacía, eso implica que todos los clientes han sido atendidos o bien, están haciendo cola. Como no podemos asegurar si aún hay cola no se parará la simulación y los cajeros seguirán atendiendo clientes hasta que la cola quede vacía. Cuando se halla atendido al último cliente todos los cajeros estarán libres y por tanto se podrá finalizar la simulación devolviendo las variables acumuladoras.

## 3.4 Algortimo de Busqueda

Gracias al simulador podemos evaluar las soluciones del problema obteniendo unos resultados razonables. Si bien, no sabemos las características que tiene una buena solución. Como primera aproximación las soluciones que se generan son aleatorias para adquirir cierto conocimiento del desempeño de estas en el problema. A continuación, se utilizará una técnica de búsqueda inteligente para encontrar mejores soluciones.

La técnica escogida es Path Relinking. Como se ha generado cierta cantidad de soluciones contamos con un conjunto de soluciones candidatas que podemos utilizar para este algoritmo. Path Relinking consiste en, a partir de dos soluciones, construir un camino de soluciones intermedias que conecte las soluciones escogidas. Las soluciones intermedias son evaluadas para comprobar si dentro de ese camino existe una solución mejor a que las utilizadas para generar ese camino. La hipótesis subyacente es que las soluciones de buena calidad están cerca unas de otras.

Se valoraron distintas variantes de esta técnica según la función utilizada para escoger la mejor solución del camino y la forma de escoger las soluciones que forman los extremos del camino.

Se optó por definir una función fitness para evaluar las soluciones en base al valor de la función objetivo y la factibilidad de las restricciones. Esta función permite comparar unas soluciones con otras de forma relativa. La función fitness también será utilizada para elegir las soluciones extremas del camino.

### 3.4.1 Función fitness

Una solución del problema es evaluada en base a tres criterios: un valor relativo basado en el valor de la función objetivo, el grado de factibilidad de la primera restricción y el grado de factibilidad de la segunda.

Respecto al primero de los criterios, no conocemos el rango de valores en el que la función objetivo está definida y además necesitamos que los tres criterios estén en una escala parecida para evitar que uno domine sobre los otros. Por esta razón el valor de la función objetivo se escala por el método de mínimo-máximo

De esta manera pasamos del rango de valores en el que se encuentran los valores de las soluciones candidatas al intervalo , haciendo que soluciones con un valor de bajo tengan valores más cercanos a uno, mientras que a más alto sea el valor más cercano será a cero.

Por otra parte, se define los criterios de factibilidad como restricciones blandas. Nuestro objetivo es que si la restricción se cumple con el valor límite de la restricción se le asigne el valor . Si la restricción se cumple con un valor por debajo del valor limite le asignaremos un valor cercano a uno, en caso contrario se le asignará un valor cercano a cero. Para modelar este comportamiento se ha utilizado la función sigmoide

que en lugar de recibir como input x recibirá un input que adapte el punto de inflexión, donde la sigmoide vale , al valor límite de la restricción

Esta función se utiliza para calcular la factibilidad de ambas restricciones. En el siguiente grafico podemos ver su efecto sobre una de las restricciones

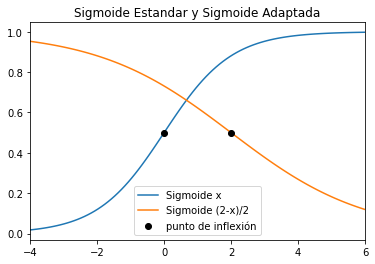


Figura 2: Sigmoide sobre la restricción de la longitud de la cola.

La imagen de los tres criterios usados se encuentra comprendida en el intervalo . Para obtener la función fitness haremos la media de los tres

# 4 Resultados

La simulación implementada se ejecutó un total de 30 veces para crear el conjunto de soluciones candidatas. Mientras que el algoritmo de Path Relinking se aplicó un total de 20 veces. Se asumió un máximo de 5 empleados por número y un coste base . Al utilizar Path Relinking con la función fitness esta parte del programa se convirtió en la más costosa por tener que simular cada una de las soluciones intermedias que se forman parte del camino.

Tanto para las soluciones halladas simulando como las halladas por Path Relinking el valor del percentil siempre es igual a uno, mientras que los valores del coste y la longitud media de la cola varía según la configuración de la solución. Respecto a la longitud de la cola a lo largo de los días, para las distintas soluciones obtenidas se observa que todas están comprendidas en el intervalo . En líneas generales las soluciones parecen adaptarse a la intensidad de la tasa de los días, por ejemplo, todas las soluciones muestran un valor bajo el día 12. Aunque también se ve como algunas soluciones tienen valores medios de cola más bajos que otras.

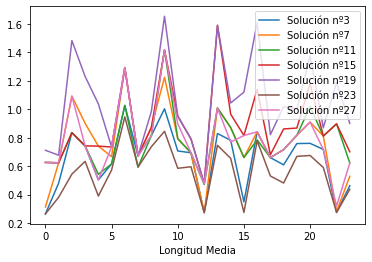


Figura 3: Valor medio de la cola según el día de la semana para distintas soluciones. Nótese la solución 23 tiene valores más bajos, mientras que la solución 19 tiene los valores más altos

Analizando la salida del simulador podemos ver que la solución 23 es más costosa pero tiene menos tiempo de espera, mientras que la solución 19 es menos costosa pero tiene más tiempo de espera. Ambas son factibles, pero la primera tiene un sobre coste en personal con contrato a jornada completa y jornada reducida, mientras que la segunda minimiza el número de empleados con contratos más caros.

| nº | Jornada Completa | Jornada Reducida | Jornada Parcial | Coste | Cola Media | Percentil |
| --- | --- | --- | --- | --- | --- | --- |
| 3 | 14 | 19 | 19 | 4060.00 | 0.65 | 1.00 |
| 7 | 7 | 14 | 23 | 3200.00 | 0.76 | 1.00 |
| 11 | 9 | 23 | 21 | 4000.00 | 0.78 | 1.00 |
| 15 | 8 | 12 | 27 | 3380.00 | 0.89 | 1.00 |
| 19 | 5 | 8 | 22 | 2460.00 | 1.05 | 1.00 |
| 23 | 14 | 14 | 19 | 3660.00 | 0.56 | 1.00 |
| 27 | 8 | 8 | 23 | 2820.00 | 0.78 | 1.00 |

*Tabla 1: Número de empleados por jornada, valor del objetivo y valores de las restricciones*

A pesar de este coste computacional los resultados que proporciona Path Relinking son mejores que los obtenidos solo simulando. Tanto el valor de la función objetivo como la longitud de la cola suelen tener valores más bajos en las soluciones obtenidas por Path Relinking, mientras los clientes son atendidos en menos de 10 minutos en ambos conjuntos de soluciones.

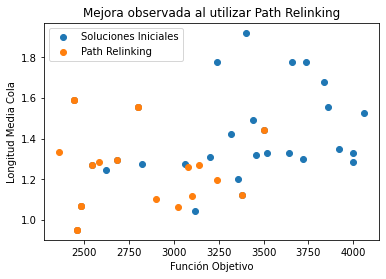


Figura 4: Valor de la función objetivo contra la longitud media de la cola para las soluciones obtenidas.