Escuela Superior
de Ingeniería y Tecnología
Universidad de La Laguna

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Gestión dinámica de recursos usando lógica borrosa

Dynamic resource management using fuzzy logic

Adrián Prieto Curbelo

La Laguna, 3 de julio de 2019

Dña. **Vanesa Muñoz Cruz** con N.I.F. 78.698.687-R profesora Contratada Doctora, adscrita al Departamento de Ingeniería Informática y de Sistemas de la Universidad de La Laguna, como tutora

C E R T I F I C A

Que la presente memoria titulada:

“Gestión dinámica de recursos usando lógica borrosa”

ha sido realizada bajo su dirección por D. **Adrián Prieto Curbelo**,

con N.I.F. 54.052.860-P.

Y para que así conste, en cumplimiento de la legislación vigente y a los efectos oportunos firman la presente en La Laguna a 5 de julio de 2019

Agradecimientos

A lo largo de la elaboración de este trabajo, que ha tenido sus altibajos, he podido ir superando los obstáculos gracias al constante apoyo de muchas personas que me han hecho ver que con trabajo y dedicación se pueden conseguir los objetivos.

Quiero dar las gracias especialmente a mi tutora Vanesa Muñoz Cruz por su implicación en el presente Trabajo Fin de Grado, los conocimientos que me ha aportado y su constante apoyo.

También agradezco a mi familia que me ha demostrado que, aunque parezca que no se puede realizar todas las cosas que uno quiere, siempre existe alguna manera de conseguirlo, con uso de la constancia y de la motivación, y sobre todo, de no desfallecer en el intento.

Licencia

Resumen

La gestión de los hospitales no es un tema fácil, ya que existen numerosos procesos interdependientes y complejos. El enfoque que se hace en el paciente, que es atendido por diferentes profesionales y con posibles objetivos dispares, puede ocasionalmente convertirlo en un mero espectador de todo el proceso.

La evolución tecnológica, tanto en el diagnóstico como en el tratamiento médico, requiere una mayor especialización de los profesionales, además de contar con herramientas de ayuda para la toma de decisiones en varias áreas.

Dentro de dicha gestión y con el enfoque puesto en el desarrollo de herramientas automáticas para la ayuda en la toma de decisiones, nos centraremos en la gestión de las colas de pacientes organizadas por salas y realizaremos una aproximación a través de la teoría de las colas y la lógica difusa, a fin de desarrollar una herramienta de ayuda en la gestión de la reubicación de recursos entre distintas salas.

Este Trabajo Fin de Grado se centra en la implementación de un programa de ordenador que realice una simulación de colas de pacientes en un Hospital y la gestión de redistribución de recursos mediante métodos de toma de decisiones basados en lógica difusa.

La implementación se ha realizado utilizando lenguaje C++, librerías de entorno gráfico Qt y librerías de lógica difusa.

En el alcance del proyecto tenemos:

- *Desarrollar un entorno hospitalario simplificado consistente en 2 Salas, cada una con su cola de pacientes y un determinado número de doctores.*
- *Simular la evolución, en una jornada laboral, de la llegada de pacientes y su atendimiento.*
- *Tratar de definir parámetros de entrada y salida para que, utilizando lógica difusa, podamos decidir si reubicar los recursos para optimizar el atendimiento a los pacientes.*
- *Realizar una generalización a 2 conjuntos de recursos: Médicos y Enfermeras.*

Palabras Clave

Simulación, Hospital, Reubicación de recursos, Teoría de colas, Lógica difusa.

Abstract

Hospitals management are not an easy issue, with numerous interdependent and complex processes. The focus places on the patient, attended by different professionals with possible disparate objectives, may occasionally turn him into a mere spectator of the whole process.

The technological evolution both in medical diagnosis and treatment , requires a greater specialization of professionals, as well as having decision-making aid tools within several areas.

Inside that management and related with the development of automatic aid tools to aid in choice-making, we will focus on the management of patient queues organized by rooms and make an approximation through the theory of queues and fuzzy logic, in order to try to make a aid tool in the management of resource relocation between different rooms.

This End Degree Project focuses on the development of a computer program that simulates the administration of an Hospital patients queues and the relocation of resources using fuzzy logic in choice-making.

The implementation has been done using C ++ language, Qt graphical environment libraries and fuzzy logic libraries.

In the scope of the project we have:

- *Develop a simplified hospital environment consisting of 2 rooms, each with its patient queue and a certain number of doctors.*
- *Simulate the evolution, in a workday, of the arrival of patients and their attendance.*
- *Try to define input and output parameters so using diffuse logic we can decide if relocate resources in order to optimize patient care..*
- *Make a generalization to 2 sets of resources: Doctors and Nurses.*

Keywords

Simulation, Hospital, Resource reallocation, Queue theory, Fuzzy logic.

Índice general y contenido

Índice de figuras

1. Introducción

1.1. Contexto

Se desarrollará una aplicación para la gestión dinámica de recursos limitados y ya definidos utilizando lógica borrosa en un entorno de simulación, donde se determinará qué conjunto de ellos será el óptimo en un motor de colas determinado.

1.2. Objetivos

Para este proyecto, se ha dado forma a la idea tomando como ejemplo un hospital, que dispone tanto de distintos tipos de recursos (médicos, enfermeras…) como de diferentes colas de pacientes en las consultas (los pacientes que llegan a las distintas salas).

La principal objeto de la implementación es realizar una simulación que pueda visualizar, para las distintas salas, cuál sería la combinación óptima de recursos ya definidos previamente, con el objetivo de que las colas sean óptimas, durante un supuesto día de trabajo del hospital.

Una de las características que queremos alcanzar con este proyecto es que pueda ser escalable, es decir, que pueda ser implementado independientemente del número de tipos de recursos o salas creadas; y que su coste computacional sea lineal o menor.

Otra de las características es que pueda ser tratado como un programa de tiempo real, en el sentido de mostrar resultados con la mayor precisión a la vez que sea optimizado para asegurarse de que los resultados puedan ser visualizados en un tiempo determinado.

Los objetivos principales a alcanzar son:

- Desarrollar un entorno hospitalario simplificado consistente en 2 Salas, cada una con su cola de pacientes y un determinado número de doctores.
- Simular la evolución, en una jornada laboral, de la llegada de pacientes y su atendimiento.
- Tratar de definir parámetros de entrada y salida para que, utilizando lógica difusa, podamos decidir si reubicar los recursos para optimizar el atendimiento a los pacientes.
- Realizar una generalización a 2 conjuntos de recursos: Médicos y Enfermeras.

2. Metodologías utilizadas

Las principales metodologías que se han utilizado en el proyecto son la Teoría de Colas y la Lógica Borrosa. A continuación se comentará brevemente cada una de ellas.

2.1. Teoría de colas

La teoría de colas es el estudio matemático del comportamiento de líneas de espera, donde cada cola se puede considerar por tanto, una línea de espera para un determinado servicio. **[1,2]**

2.1.1. Clasificación de los sistemas de colas

El proceso básico de la mayor parte de las colas es el siguiente:

- Los clientes que requieren un servicio se generan a través del tiempo en una fase de entrada.
- Dichos clientes entran al sistema y se unen a una cola.
- En determinado momento se elige un miembro de la cola para proporcionarle el servicio, y el cliente sale del sistema de colas.

Un sistema de colas viene determinado por varias características:

1. **Dimensión de la cola.** Sería el tamaño máximo de clientes potenciales. Si la población es muy grande se supone infinito por simplicidad de cálculo.
2. **Modelo de llegada de clientes**. Describe el régimen de llegada de clientes a unirse al sistema. Existen dos clases básicas:
    a. Determinístico, en el cual clientes sucesivos llegan en un intervalo de tiempo fijo y conocido.
    b. Probabilístico, en el cual el tiempo entre llegadas sucesivas es incierto y variable. Destaca el modelo de llegadas exponenciales como una buena aproximación en muchos casos.
3. **Disciplina de la cola**. Sería el modo en que los clientes son seleccionados para ser atendidos. Las disciplinas más habituales son: FIFO (el primero que llega se atiende primero), LIFO (el último en llegar se atiende primero) y RSS ó SIRO (selección aleatoria).
4. **Modelo del servicio**. Describe la duración del servicio, esto es, el tiempo que un cliente es atendido. Análogamente al modelo de llegadas puede ser determinístico o probabilístico (caso destacado exponencial)..
5. **Número de servidores (dependientes).** Cuántos recursos (servidores o dependientes) iguales hay para atender determinada necesidad. Dichos recursos se ofrecen en paralelo pudiendo ser atendido un cliente por cualquiera de ellos.
6. **Número de estados del servicio.** Número de conjuntos de servidores que atienden a distintas necesidades.

2.1.2. Caso práctico: colas de pacientes en el hospital

En el ejemplo práctico que nos ocupa, colas de pacientes en un Hospital.

- Tenemos varias salas separadas físicamente. Esto nos da **colas de pacientes independientes.** En el estudio hemos considerado 2 salas.
- La **dimensión de las colas** es infinita.
- La **disciplina de las colas** es FIFO. El primero en llegar se atiende primero.
- El **método de llegadas** de clientes sería una exponencial, según una Poisson(λ), donde λ es el número medio de llegadas por unidad de tiempo. El λ lo vamos variando a lo largo del tiempo para reflejar que en las horas centrales de la jornada lleguen más pacientes.
- Modelo de servicio. Los tiempos de duración del servicio también se distribuirán exponencialmente una Poisson(μ) .
- Número de estados del servicio. Se ha realizado el estudio con 1 y 2 estados. Cada paciente puede esperar por doctores (caso 1) o doctores / enfermeras (caso 2).
- Número de servidores. Cada sala dispone de un cierto número de doctores / enfermeras. Los cuales en determinado momento se pueden reubicar a otra sala con una pérdida de tiempo por la reubicación (tiempo que tarda en estar disponible el recurso en la otra sala).

2.2. Lógica difusa

El concepto de conjunto borrosos fue introducido por primera vez por Lofti A. Zadeh en 1964, para representar y manipular datos que no eran precisos. Dicho concepto dio paso a la denominada Teoría de la Lógica Borrosa o Lógica Difusa, que se basa en el hecho de que, ciertas magnitudes pueden tomar valores que no se pueden clasificar en un conjunto determinado. El objetivo de la Lógica Difusa es por tanto, definir conjuntos que modelen las situaciones de imprecisión. **[3,4]**

A continuación se ilustrará con un ejemplo.

2.2.1. Conjuntos clásicos

Si tenemos un posible rango de valores, como por ejemplo temperaturas entre 0º y 50º y se definen los conjuntos de temperaturas FRÍO, TEMPLADO y CALIENTE, una manera de definición de conjuntos clásico podría ser:

- FRÍO si la temperatura está entre 0º y 20º
- TEMPLADO si la temperatura está entre 20º y 35º
- CALIENTE si la temperatura está entre 35º y 50º

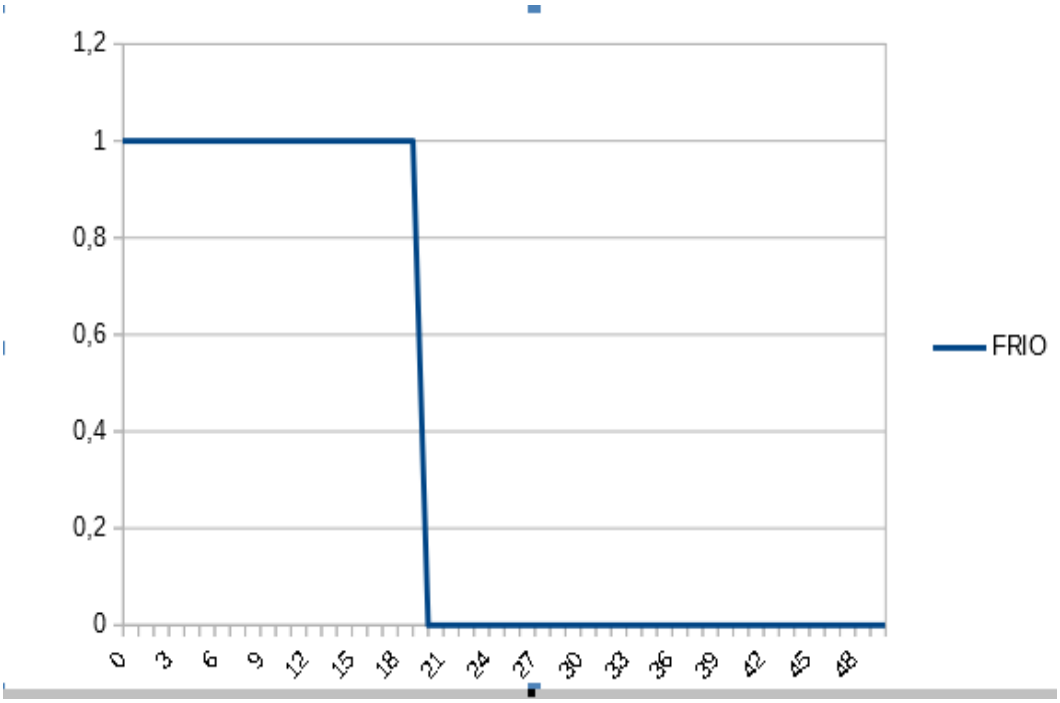


Figura 1: Conjunto FRIO

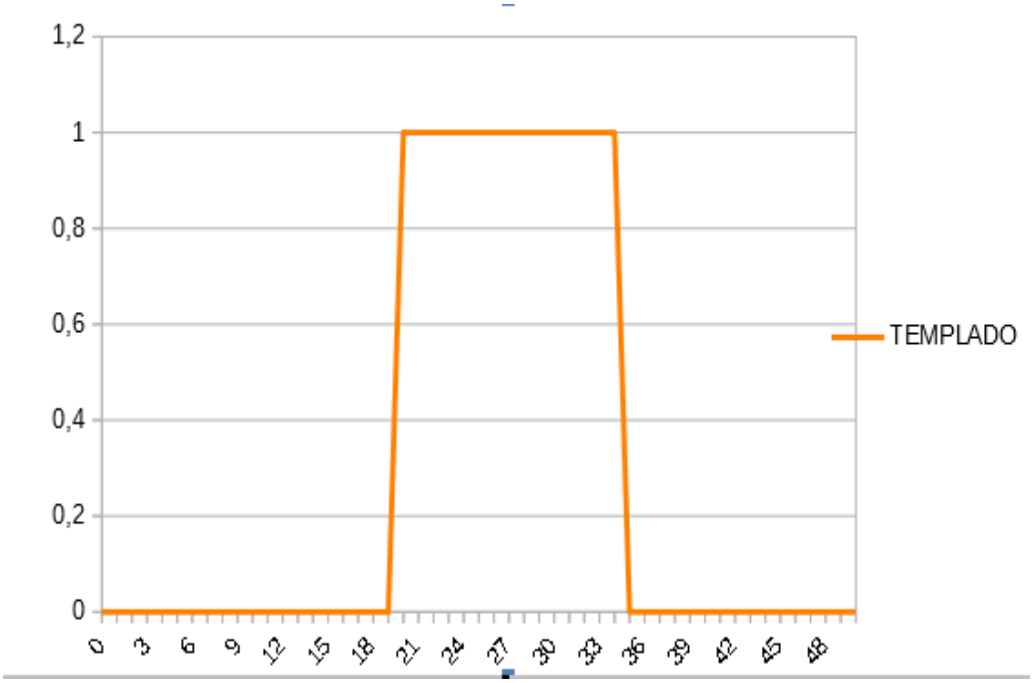


Figura 2: Conjunto TEMPLADO

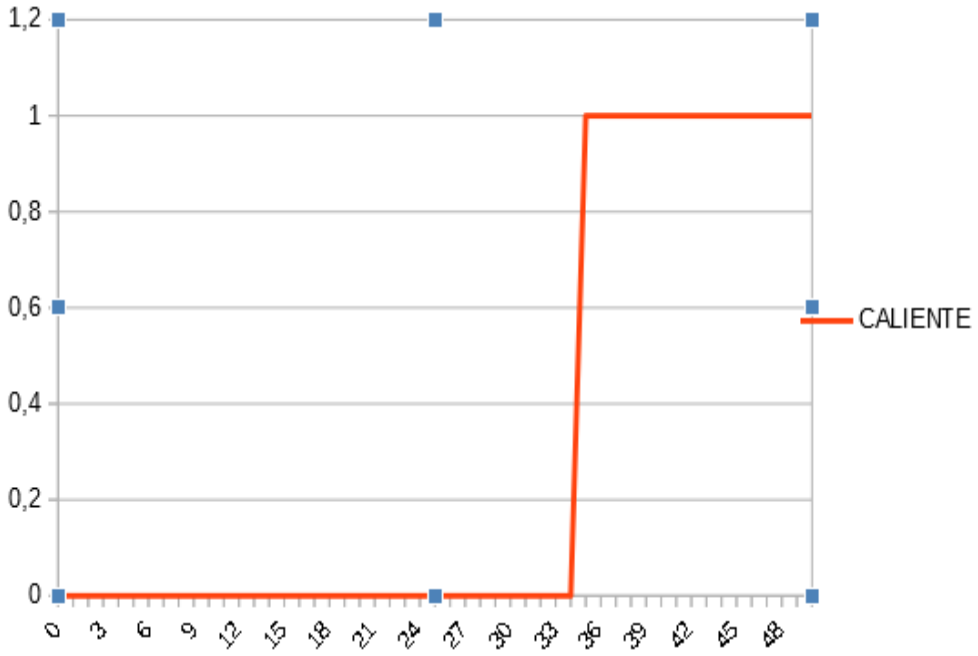


Figura 3: Conjunto CALIENTE

```
FRIO { t | t <= 20 }
TEMPLADO { t | (t >20) AND (t <=35) }
CALIENTE { t | t >35 }
```

teniendo entonces una distribución de valores en conjuntos clásicos. Un cierto valor t se sabría si está o no en un determinado conjunto. Si suponemos que 0 es falso y 1 es cierto, el preguntar si un valor está en un conjunto solo daría valores discretos 0 o 1.

2.2.2. Conjuntos difusos

Los conjuntos difusos vienen determinados por una función característica. Estas funciones sitúan en el eje X los posibles valores de la variable; y en el eje Y su grado de pertenencia a dicho conjunto que varía de cero (FALSO) a uno (CIERTO). Algunas de las funciones más utilizadas podrían ser las funciones trapezoidales y las funciones gaussianas. Nótese que un conjunto clásico es un caso especial de conjunto difuso cuya función característica es una función escalonada con valores 0 y 1.

Por ejemplo, sobre un rango de temperaturas podríamos usar funciones gaussianas para definir los siguientes conjuntos difusos de de temperatura:

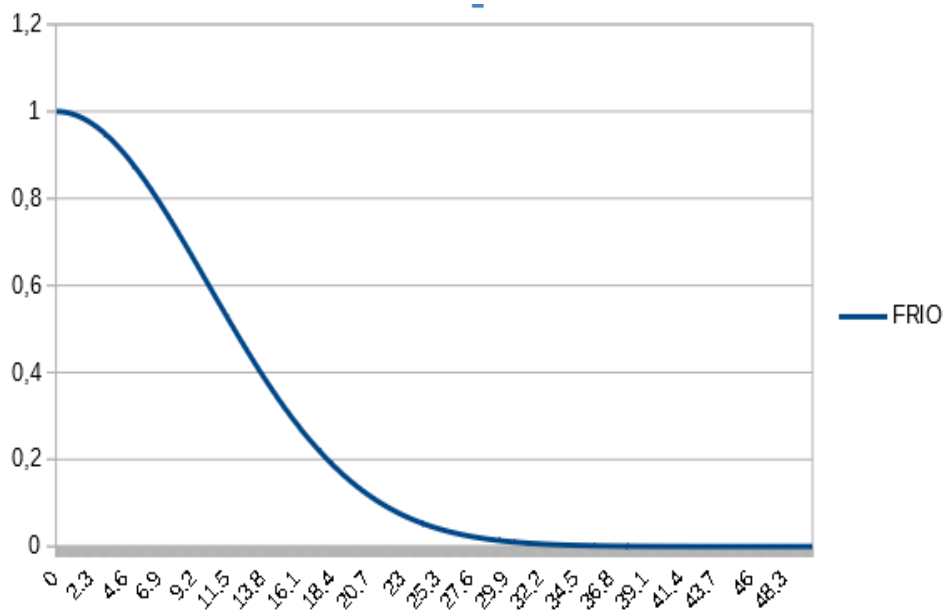


Figura 4: Conjunto difuso FRIO

De la gráfica podríamos decir que para t=9º nos da aprox 0,8 que quiere decir que hace bastante frío. Sin embargo t=35º nos da aprox 0, que quiere decir que no hace frío en absoluto.

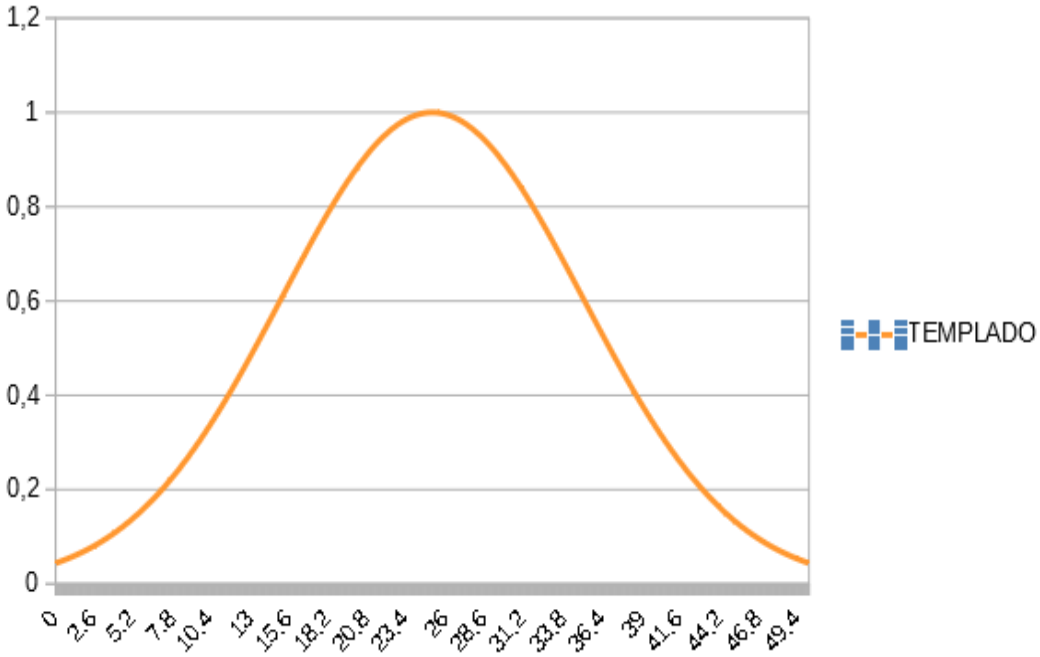


Figura 5: Conjunto difuso TEMPLADO

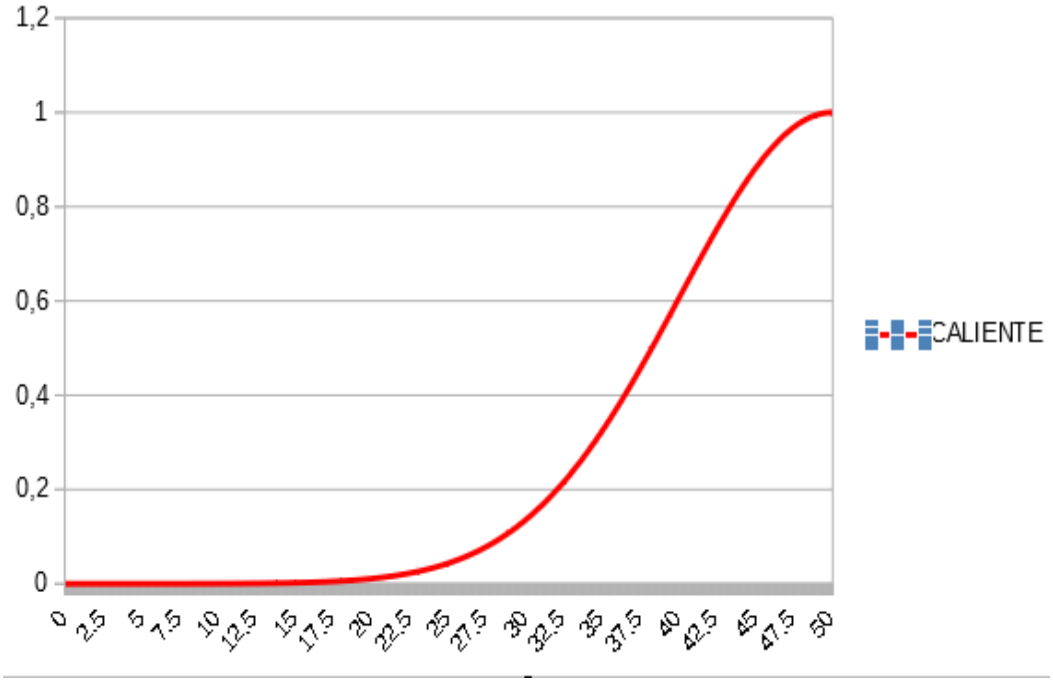


Figura 6: Conjunto difuso CALIENTE

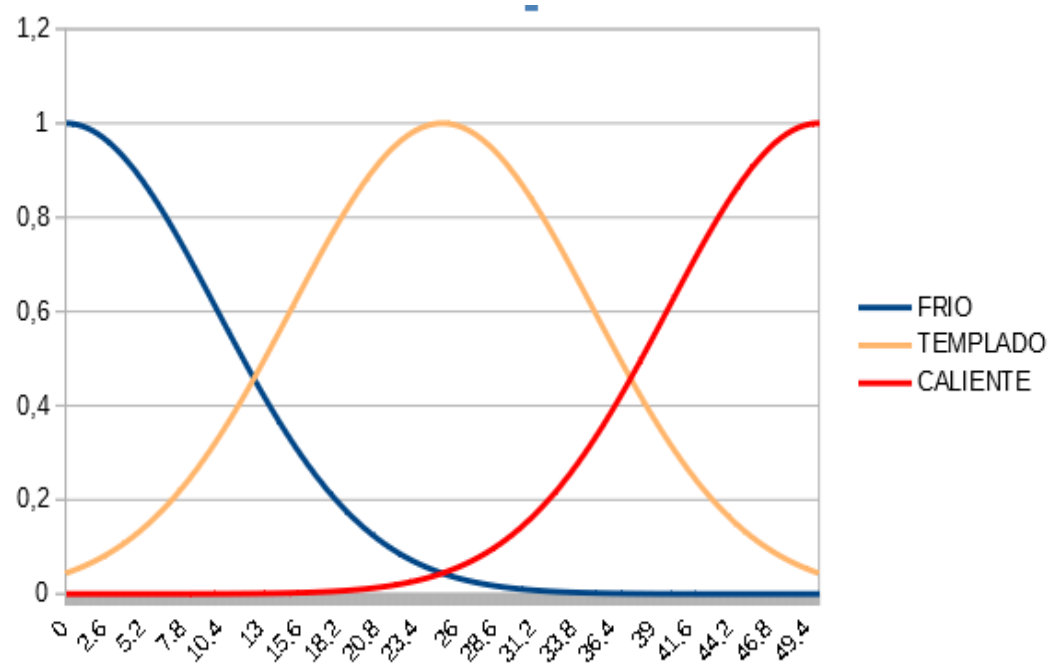


Figura 7: Conjuntos difusos FRIO, TEMPLADO y CALIENTE

Otro ejemplo podría ser definir sobre un rango de coste eléctrico unos conjuntos difusos usando funciones trapezoidales.

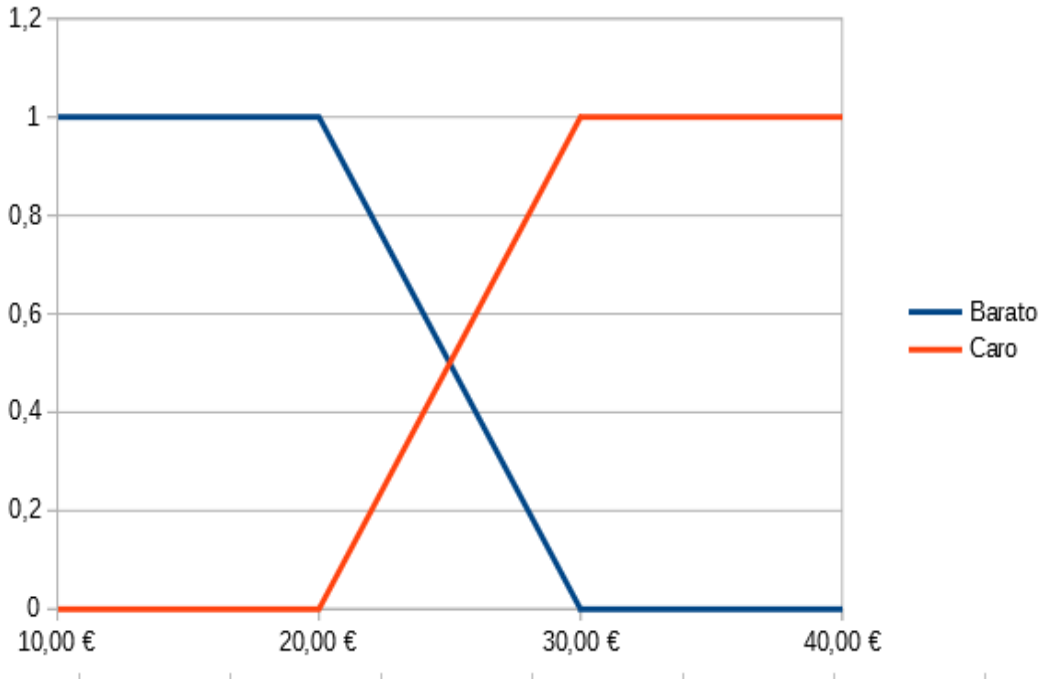


Figura 8: Conjuntos difusos BARATO y CARO con funcion característica trapezoidal

De la gráfica podríamos decir que para c=25€ nos da 0,5 que es y no es caro, y es y no es barato, esto es, dicho vulgarmente, no es caro ni barato sino todo lo contrario.

2.2.3. Sistema basado en lógica difusa

En un sistema basado en lógica difusa podemos distinguir los siguientes apartados:

- **Entradas**: se toman variables de entrada. Sobre dichas variables se definen conjuntos difusos mediante sus funciones características.
- **Salidas**: Sobre las variables de salida se definen unos conjuntos difusos
- **Mecanismo de Inferencia**: Tendremos una serie de reglas difusas que relacionan los conjuntos difusos de entrada con los conjuntos difusos de salida. Las reglas tienen la forma **Si u es A [AND|OR …] entonces v es B**. Donde, u es variable entrada, A conjunto entrada, v variable salida, B conjunto salida.
- **Difusor**: A cada variable se le asigna un grado de pertenencia a cada uno de los conjuntos difusos considerados.
- **Desdifusor:** Partiendo del conjunto difuso obtenido, se usan métodos de desborrosificación (centroide, bisector, etc.), para obtener el valor de las variables de salida.

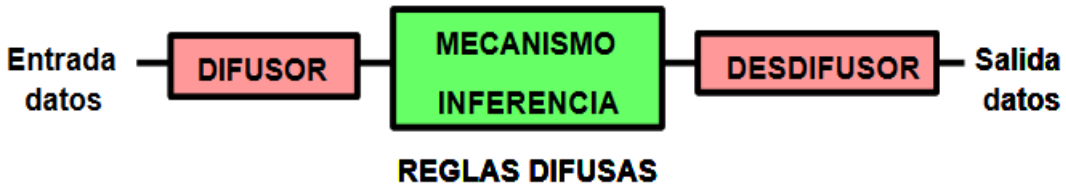


Figura 9: Toma de decisiones con lógica difusa

2.2.4. Toma de decisiones usando lógica difusa

En el proceso de toma de decisiones clásico, se definirían una serie de conjuntos clásicos sobre las entradas y una serie de reglas completamente definidas, de manera que para todos los casos posibles se obtenga si ejecutamos o no las acciones correspondientes.

Análogamente, en el proceso de toma de decisiones mediante lógica difusa tenemos, para una determinada acción.

- Entradas: se toman variables de entrada y sobre ellas se definen conjuntos difusos.
- Salida: Sobre el rango [0,1] se definen unos conjuntos difusos, con las acciones a tomar.
- Tendremos una serie de reglas difusas, que relacionan los conjuntos de entrada con las acciones de salida.

El funcionamiento del proceso es:

- Dado unos valores determinados de entrada se obtiene una salida y para cada acción que se quiera realizar se obtendría un valor entre 0 y 1, que podría definirse como la probabilidad de ejecutar dicha acción.
- Luego se interpretarán dichos valores de salida para decidir si se ejecuta o no la acción. Por ejemplo, una posible interpretación sería considerar que si el valor de salida para una determinada acción es mayor que 0,5 se ejecute la acción y no se ejecute en caso contrario.

2.2.5. Ejemplo de toma de decisiones usando lógica difusa

Podríamos considerar el siguiente ejemplo:

- Entradas:
  - variable temperatura con conjuntos [**FRÍA, TEMPLADA, CALIENTE**].
  - variable **Coste Eléctrico** con conjuntos difusos [**BARATO, CARO]**.
- Salida:
  - Sobre el rango [0,1] los conjuntos difusos **[ENCENDER CALEFACCIÓN, ENCENDER AIRE ACONDICIONADO]**.

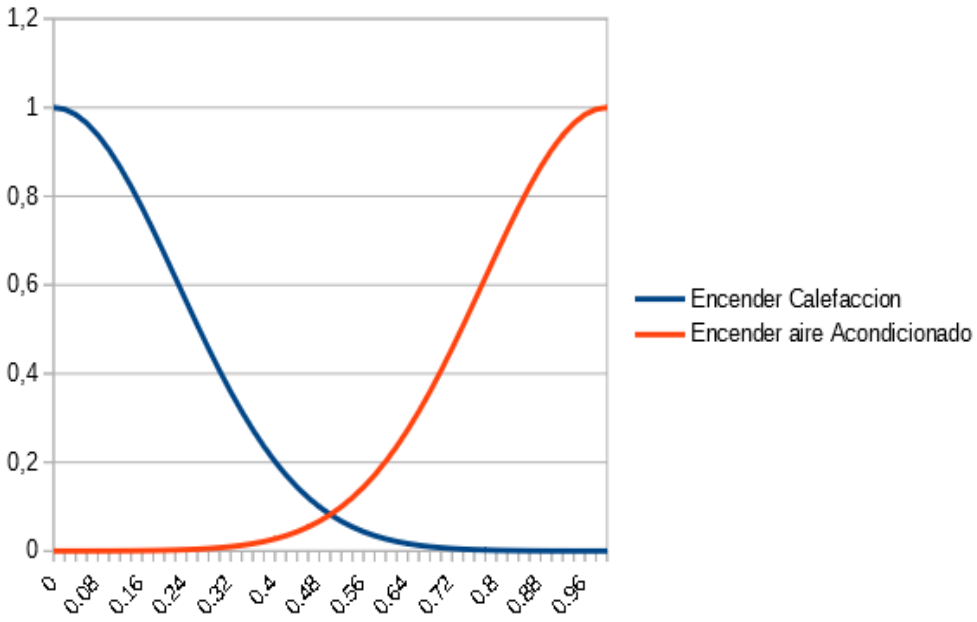


Figura 10: Conjuntos difusos ENCENDER CALEFACCION y ENCENDER AIRE ACONDICIONADO

- Reglas difusas:
  - SI temperatura es **FRÍA** Y Coste Eléctrico es **BARATO** entonces **ENCENDER_CALEFACCION**;
  - Si temperatura es **CALIENTE** entonces **ENCENDER AIRE ACONDICIONADO**;

El funcionamiento del proceso es, que dado unos valores determinados de entrada t,c de temperatura y coste eléctrico, se obtiene un valor de salida f(t,c) entre 0 y 1. y para ese valor f(t,c) se pueden obtener las funciones **ENCENDER_CALEFACCION(f(t,c)) y ENCENDER_AIRE_ACONDICIONADO(f(t,c)**, que podrían leerse como la probabilidad de ejecutar dicha acción.

- Podríamos determinar que encendemos la calefacción / aire acondicionado cuando el valor sea mayor que 0,6 y apagamos la calefacción / aire acondicionado, si el valor es menor que 0,4.

3. Planificación y cronograma

Se muestran a continuación el plan de trabajo y el cronograma estimativo que se definieron para la realización del trabajo.

3.1. Plan de trabajo

1. Buscar o implementar una librería de lógica difusa para el lenguaje de programación elegido.
2. Crear valores iniciales para la simulación.
3. Realizar una simulación simple donde se aplique la lógica difusa.
4. Diseñar gráficas necesarias para visualizar los datos posibles.
5. Posibilidad de implementar tomando como prioridad la escalabilidad; aumentando tanto el número de colas y entornos, como de tipos de variables.
6. Posibilidad de implementar tomando como prioridad en sistema de tiempo real como asignación de prioridades, optimización de operaciones, temporización.

3.2. Cronograma

Se ha empleado un total de 12 semanas. Para ello, se han planificado todas las actividades anteriores de esta manera

1. Actividad 1 → 1 semana.
2. Actividad 2 → 2 semanas.
3. Actividad 3 → 3 semanas.
4. Actividad 4 → 2 semanas.
5. Actividad 5 → 2 semanas.
6. Actividad 6 → 2 semanas.

4. Herramientas utilizadas

Se ha implementado la mayor parte en C++, en concreto, utilizando el framework de Qt (https://doc.qt.io/), debido al gran repertorio de librerías que proporcionan acerca de gráficos y manejo de ventanas.

Para la lógica difusa, se utiliza una **librería externa y de open source** llamada FuzzyLite (https://fuzzylite.com/), que tiene soporte tanto para C++ para Java.

Esta librería también consta de una interfaz propia en Qt (llamada QtFuzzyLite), pero existen varios motivos para crear un nuevo programa de simulador en vez de utilizar QtFuzzyLite:

- Aunque la librería está considerada como open source, la interfaz y programa no lo es (de hecho tiene una licencia para usuarios finales).
- Los objetivos a alcanzar son tan específicos para este programa que no es viable utilizarlo.

5. Modelado básico

5.1. Gestión de colas de pacientes de un Hospital

Como se ha comentado en capítulos anteriores, el objetivo de este trabajo, es la implementación de un **software de simulación** de las colas de pacientes en un Hospital utilizando **la teoría de colas,** y la reubicación de recursos utilizando la toma de decisiones mediante **lógica difusa**.

La redistribución de recursos se puede considerar muy importante para la optimización, y se puede aplicar en modelos de **la teoría de colas**. Para este trabajo lo hemos enfocado de esta manera:

- Un conjunto de recursos finito
- Varias colas donde los recursos realizan un trabajo.

Para adaptarlo a una situación real, se va a realizar el modelado y simulación de un **Centro de Salud** u Hospital, donde:

- Las **colas** serán representadas por varias salas o consultas que disponga el hospital. En ellas, pacientes irán entrando, esperando a ser atendidos.
- Los **recursos** que tenemos que asignar están representados por el personal del hospital, que atenderán a los pacientes conforme van llegando.

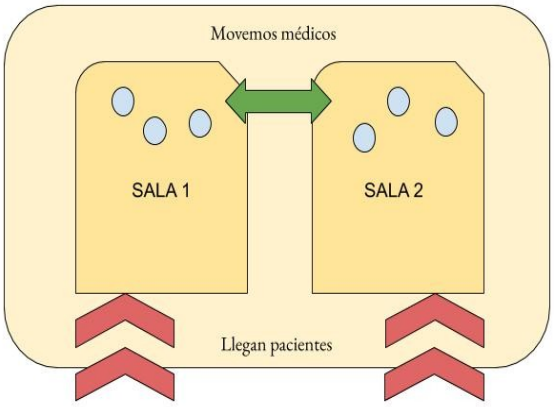
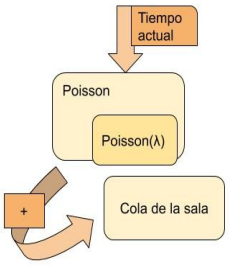


Figura 11: Esquema general

Aclarado esto, con la simulación se pretende emular el comportamiento tanto de las colas como de los recursos a lo largo de una jornada laboral, con el objetivo también de proporcionar una solución óptima a la simulación para que:

- Se aprovechen en todo lo posible los recursos disponibles.
- Las colas de pacientes no se saturen, o al menos, que haya una proporción ideal de recursos con el tamaño de la cola.

5.2. Esquema general



Para realizar la lógica de la simulación, nos hemos basado en el siguiente algoritmo:

- Los pacientes van llegando en un intervalo de tiempo a cada una de las salas.
- Se calculan los parámetros de las salas que sean importantes para la lógica difusa.
- Se realiza la operación de inferencia (**desdifusor** o **defuzzificación**). Esto genera un único valor determinista del estado de la sala por cada una de ellas.
- Se utiliza un algoritmo de selección, con un doble objetivo, **escoger las salas** que necesitan distribuir los recursos y determinar **si es óptimo** realizar este cambio.
- Si las condiciones lo permiten, reubicamos los recursos de la sala.

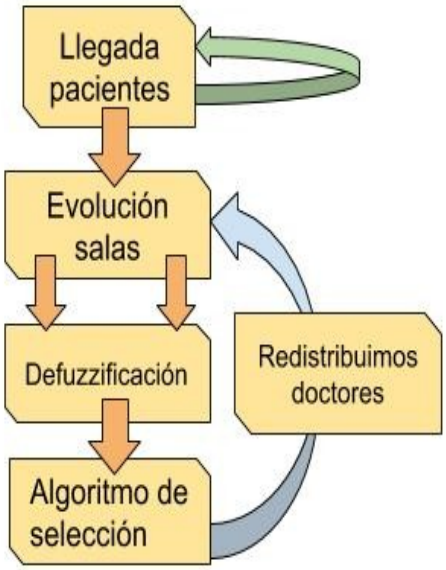


Figura 12: Algoritmo general

Todos los procesos anteriormente mencionados serán realizados en un bucle, de manera que la propia simulación tenga una **retroalimentación** a partir del estado anterior.

5.3. Modelo de llegada de pacientes.

Aunque al principio podemos suponer por simplificación que la llegada de pacientes es constante, esto no sería eficaz para nuestro programa, e incluso nos perjudicaría a la hora de distribuir los recursos. Una distribución lineal o uniforme tampoco se puede aplicar en esta simulación por los mismos motivos, ya que todas las salas tendrían un comportamiento similar.

En la teoría de colas, se recomienda realizar una **distribución estadística**, como la distribución de Poisson, que expresa, a partir de una frecuencia de ocurrencia media, la probabilidad de que ocurra un determinado número de eventos durante cierto período de tiempo, es decir, probabilidad de ocurrencia de sucesos.

Esta distribución asegura:

- Que el comportamiento de la función, comparando a distribuciones polinómicas, no sea constante, dando una sensación de no determinismo y aleatoriedad.
- Que aunque asignemos pacientes cada intervalo de tiempo X, podemos asignarlos

también en otro intervalo más pequeño X' divisor de X, porque se cumple la propiedad :

$$Y = \sum_{i=1}^{N} Poisson(\lambda_i) \sim Poisson(\sum_{i=1}^{N} \lambda_i)$$

La gráfica sería de esta manera:

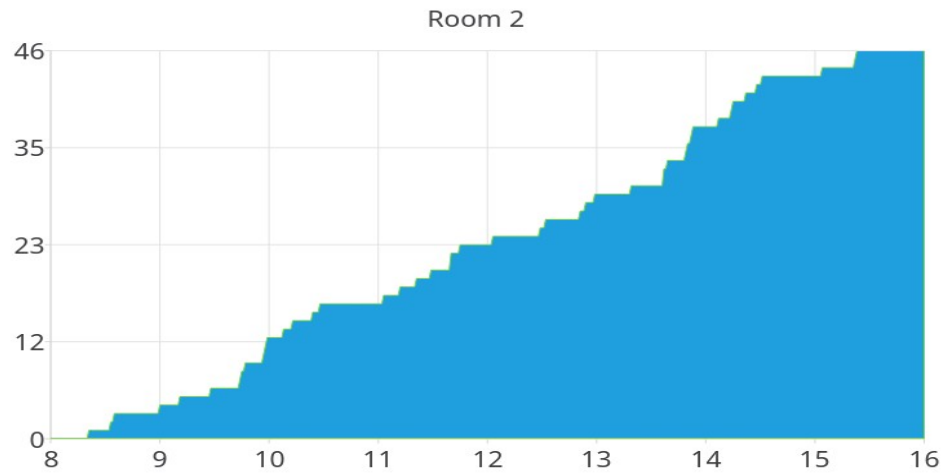


Figura 13: Llegada de pacientes con una distribución de Poisson

Pero como vemos, la gráfica es casi **lineal** con respecto al tiempo**,** dando un toque de artificialidad con una frecuencia de llegadas de pacientes constante a lo largo de la jornada. Una gráfica lineal en una situación real implica que los pacientes tengan poca libertad de aleatoriedad, de forma de que pueden **predecir y controlar fácilmente**. Queremos que la simulación se prepare también a situaciones inesperadas.

- Un ejemplo de esto es que no existen los **picos o máximos**. En una situación real, existen las horas puntas o simplemente un accidente a gran escala.

Para romper esta linealidad, se añade otra **Distribución de Poisson** para dar más **aleatoriedad**. Este proceso consistirá en realizar una distribución de poisson a través del tiempo, y utilizar este valor en otra distribución de poisson para obtener el resultado final.

El resultado lo podemos ver en la siguiente gráfica. Esta representa los pacientes que van entrando, **acumulados** de una sala.

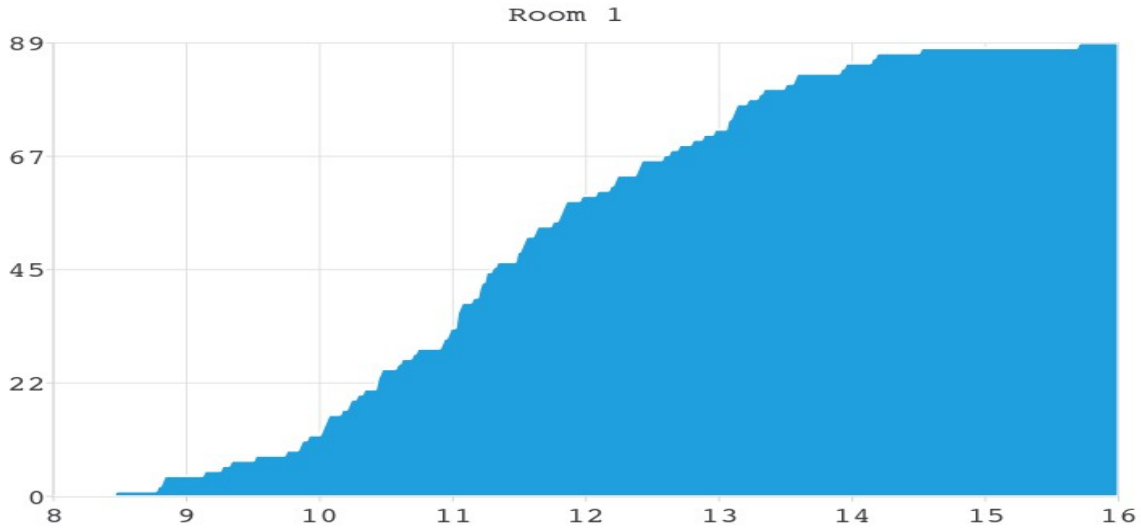


Figura 14: LLegada de pacientes con distribucion de Poisson modulada en el tiempo

Como vemos, no tiene una regresión lineal, a diferencia de la gráfica anterior, sino que posee el comportamiento de una distribución normal: aumenta frecuencia de llegadas hacia la mitad de la jornada, haciéndose menor a primera y última hora.

6. Implementación lógica difusa

6.1. Uso de la lógica difusa en el programa.

Previamente se ha explicado cómo realizar la inferencia de forma teórica. En este capítulo se mostrará la aplicación en el proyecto.

El programa lanza la evaluación de reubicación de recursos cada cierto tiempo. Este período de tiempo es configurable, pero :

- Es necesario que haya variaciones significativas en la sala.
- En esta simulación, el cálculo de si han llegado nuevos pacientes, si entran en consulta, si salen de consulta, si se van a casa o pasan a otra cola, se efectúa cada minuto, por lo tanto el tiempo de evaluación de reasignación de recursos ha de ser mayor a 1 minuto.
- El tiempo en que un recurso pasa de una sala a otra se ha establecido en 15 min (una vez que acaben con el atendimiento del paciente si lo hubiese), y no parece conveniente volver a evaluar si se cambian recursos cuando estén moviéndose recursos, se ha optado por tener a las salas estables antes de volver a evaluar.
- Por simplicidad se ha configurado un tiempo para evaluar, esto es, el proceso de decisión de reubicación basado en lógica difusa, en 1 hora, pudiéndose modificar este parámetro dentro del programa.
- Si se decide bajar el tiempo de evaluación por debajo del tiempo de reubicación de los recursos, habría que contemplar que se estén moviendo recursos en ese lapso de tiempo.

De cualquier manera, el cálculo de las entradas es instantáneo, no se tiene en cuenta lo ocurrido desde la anterior evaluación.

En cada sala, se definen dos entradas, **Tolerancia de la cola (TC)** y **Tiempo de espera (TE)**, y una salida, **Necesidad de reubicación de recursos (NR)**, que dan como resultado un número significante del estado de la sala.

- **TC**: Para cada sala y grupo de recursos, se calcula la *tolerancia de la cola* como el número total de pacientes que están esperando en todas las colas de esa sala en el instante **t**.
- **TE**: Para cada sala y grupo de recursos, se calcula el *tiempo de espera* como el máximo de los tiempos que llevan esperando en cola los pacientes en el instante **t**. Como hemos implementado colas **FIFO** (First in First Out), éste se corresponde al primer paciente de cada cola.
- **NR**: Para cada sala y grupo de recursos, la *necesidad de reubicación* sería cuan perentorio sería que hubiese más personal para ese grupo de recursos para atender a los pacientes.

Como sabemos, para cada uno de los parámetros tenemos funciones características de los diferentes conjuntos difusos considerados, mostrándose una gráfica de representación de dichas funciones, que tienen un rango de valores comprendido entre [0,1].

Esto nos surge un problema más: en nuestro programa sólo almacenamos datos de origen entero, con límite [0,∞]. Por ello se han normalizado los valores, aplicándose una transformación T(p):

$$T(p)=1-e^{-p/\theta}$$

donde:

- p, θ son números mayores que 0.
- T(p) es un valor en el rango [0,1). si p varía en [0,∞)
- Si θes muy pequeña, un número pequeño como p=5 puede hacer que T=1 fácilmente. Si por el contrario, tiene un valor grande, es muy difícil acercarse a T=1. Por este motivo, y sabiendo que $T(\theta)=0.632212$**,** elegimos θ dependiendo de los **valores medios de p.**

A continuación se muestra un ejemplo de los conjuntos borrosos y las reglas utilizadas en el sistema.

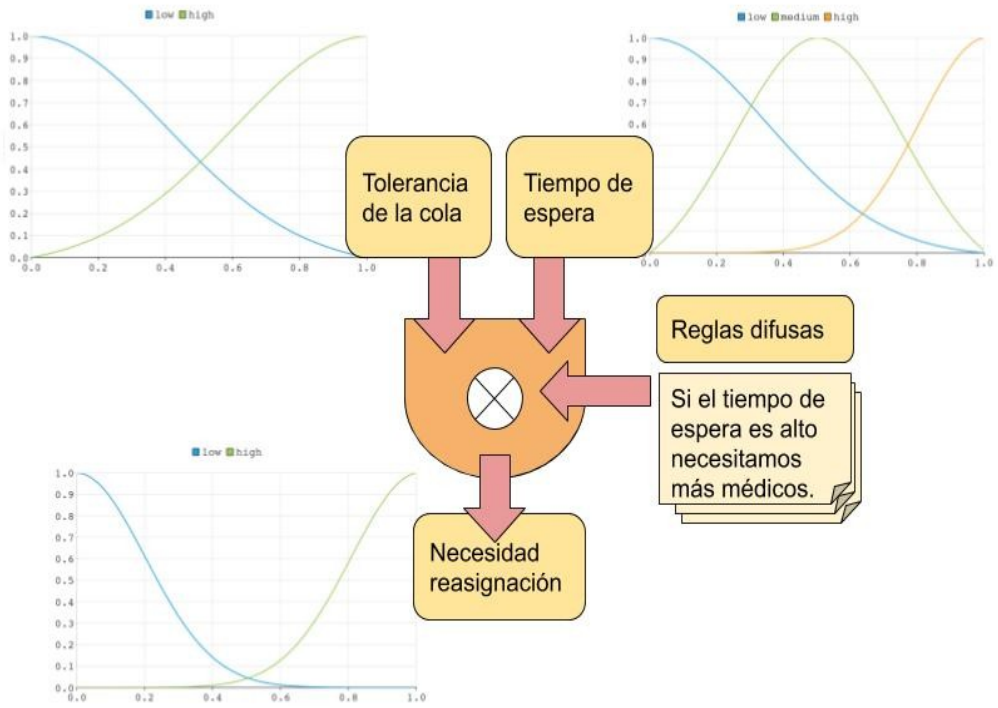


Figura 15: Lógica difusa: Entradas, Salidas y Motor de Inferencia.

Y también añadimos una serie de reglas, propias del sistema:

Valores de Entradas		Salidas
Tolerancia de la cola	Tiempo máximo de espera	Necesidad de distribución
Alto		Alto
Bajo	Medio	Bajo
Bajo	Bajo	Bajo

Vemos que nuestra serie de reglas es inexacta (no existe una entrada para Bajo - Alto), pero el **sistema puede aproximar y estimar** el valor de salida en caso de las variables de entrada tomen dichos valores.

Las gráficas utilizadas son las siguientes:

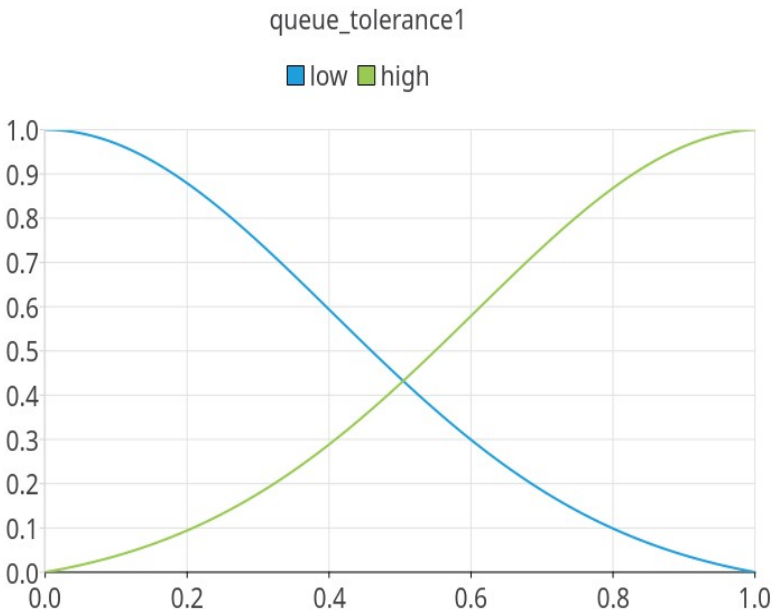


Figura 16: Tolerancia de la cola : BAJA y ALTA

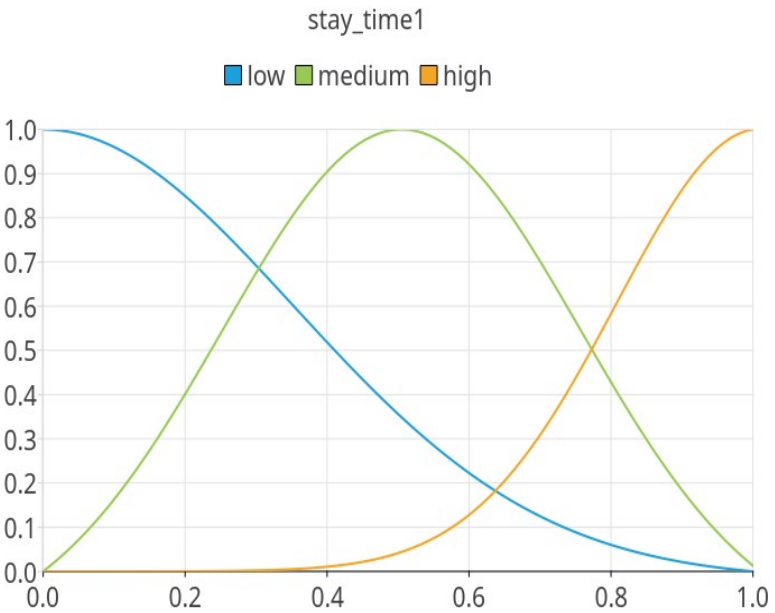


Figura 17: Tiempo de espera: BAJA, MEDIA y ALTA

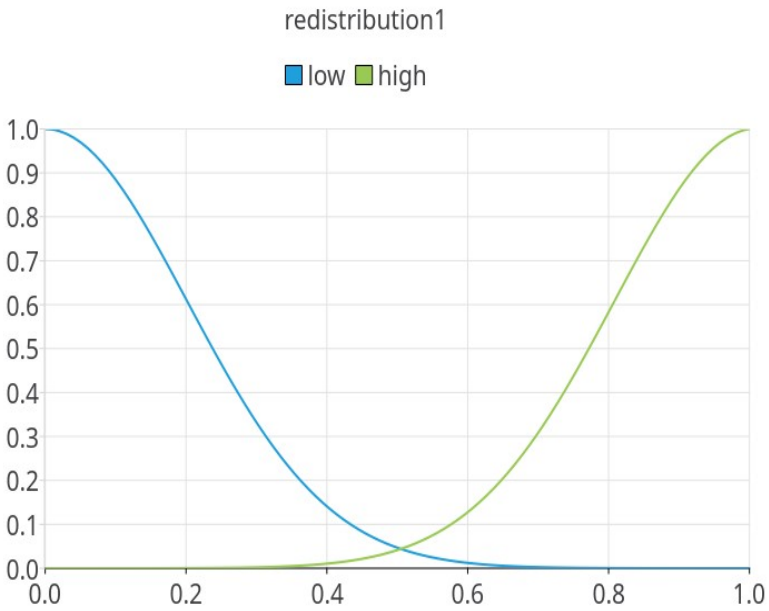


Figura 18: Necesidad de reubicación: BAJA y ALTA

Cabe destacar que tanto **el número de reglas** utilizadas como de valores que describamos en cada gráfica **influye en la exactitud** del sistema. Cuantos más conjuntos borrosos y más reglas se definan, más situaciones se estarán calificando y por lo tanto más exacto será el sistema.

6.2. Algoritmo de selección

En el anterior paso, obtenemos en cada sala los parámetros de entrada TC y TE, para conseguir un valor NR.

Tras recibirlo, necesitamos realizar una toma de decisiones donde los pasos son los siguientes:

- Elegimos las salas candidatas para cambiar. Para ello, se ha elegido la sala con la NR más alta y la NR más baja.
- Decidimos si es viable transferir recursos entre ambas salas. Para ello, se han implementado dos límites: uno inferior para el RR bajo y uno superior para el RR alto.
- Después de cumplir los requisitos anteriores, t**ransmitiremos como máximo un recurso en cada intervalo de tiempo** definido en el sistema. Los motivos son los siguientes:
    - Tener una **retroalimentación** asegurando que siempre se

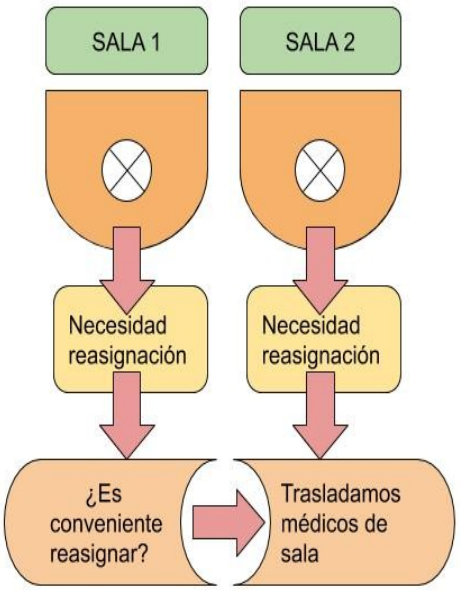


Figura 19: Algoritmo de Selección: Reubicación de recursos

cumplen los requisitos anteriores.

- Se ha planteado también el hecho de que existe un **margen de tiempo** donde el recurso, tras ser reubicado, **no se pueda utilizar** (esto simula el hecho real de que un médico, por ejemplo, necesite un tiempo para moverse físicamente a la otra sala).

- Teniendo un **intervalo de tiempo mayor al tiempo de reubicación** del recurso, evita **posibles livelocks** de los recursos. Esto puede suceder en caso de que, si un recurso que está siendo reubicado acaba de llegar a una sala, **no sea reubicado de nuevo** a otra sala, y así indefinidamente, llegando incluso a inhabilitar el recurso.

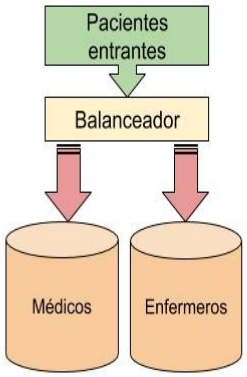
7. Implementaciones adicionales

En este capítulo se mostrarán las implementaciones adicionales realizadas en el sistema, en base a los objetivos definidos previamente.

7.1 Interfaz gráfica

Se optó por un diseño claro y sencillo: un menú con botones que mostrarán diversas ventanas cuando los pulsemos:

- Se muestra para cada una de las salas, botones que llevan a información detallada de la sala, como la gráfica individual o los recursos utilizados. En este último se permiten modificar los parámetros desde la propia ventana.



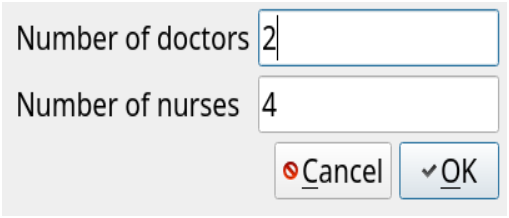
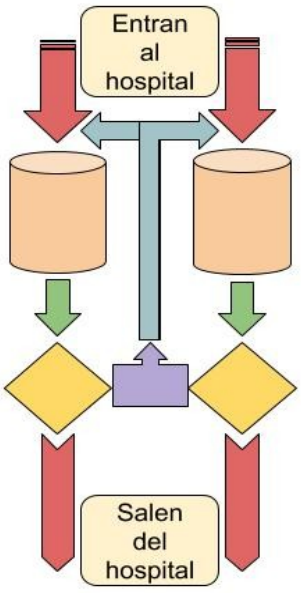


Figura 20: Cantidad de Recursos



- En la parte inferior, se crean botones para información y acciones propias de la simulación en general. Ejemplos serían realizar la simulación y ver datos relevantes de todo el hospital.

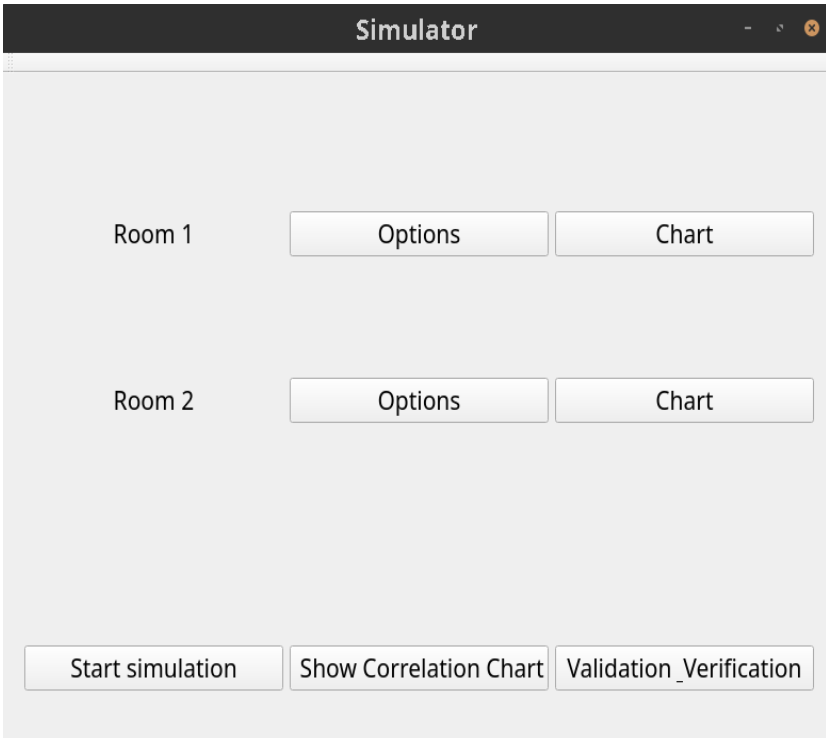


Figura 21: Menú General

7.2 Varios recursos

Esta nueva funcionalidad que se quiere implementar conlleva un cambio importante del diseño del código.

- La estructura general sigue siendo la misma: un simulador dispone de distintas salas. Esto supone que parte del código siga intacto.

- La mayor parte de los cambios residen en la **estructura de la sala**: también existe una jerarquía dentro de la misma, donde se guardan los **recursos como una estructura** más.

Estos recursos actúan de **forma independiente** y tienen su propio comportamiento, incluyendo su propia cola de pacientes.

Para simplificar cuáles de estos pacientes de los que hemos establecido anteriormente van a cada cola, se utiliza un **balanceador**, asegurándose que un porcentaje va destinado a los médicos y el resto a los enfermeros.

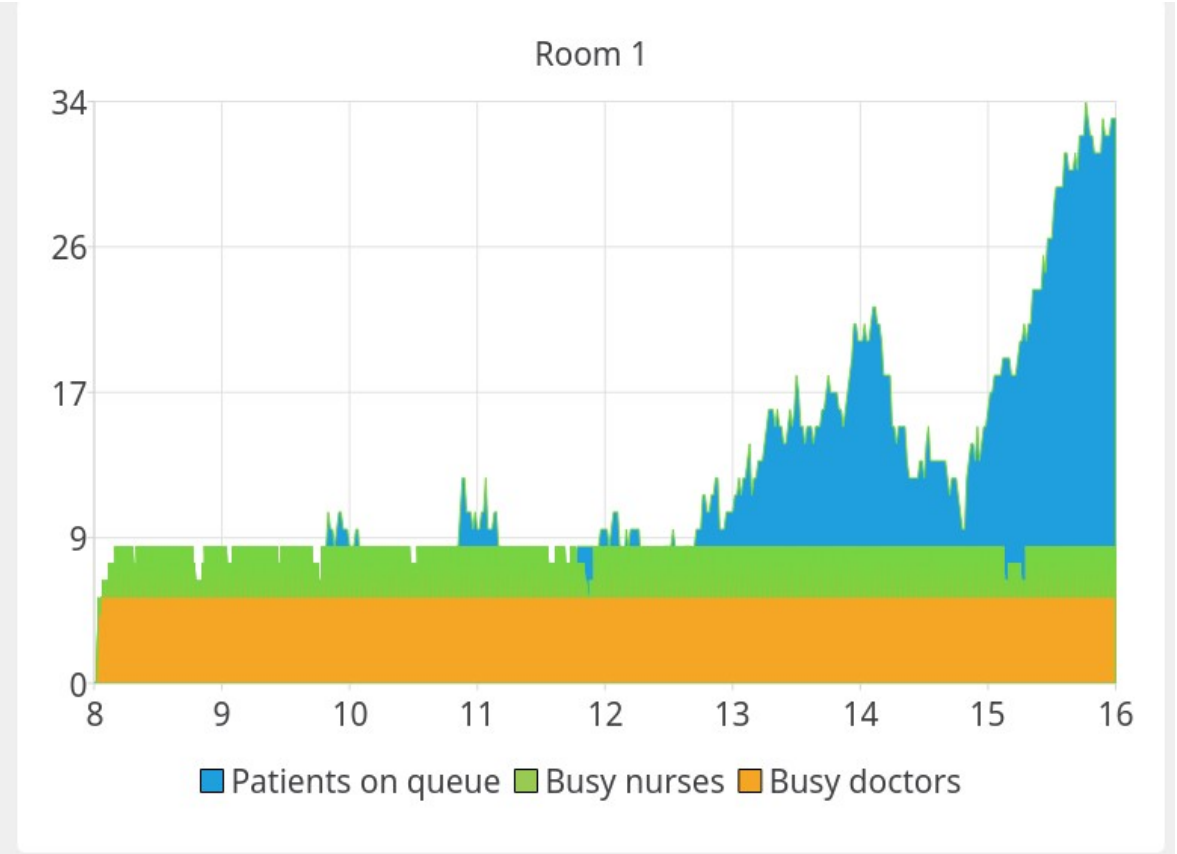
Sin embargo, esto conlleva que los recursos no tengan ningún tipo de dependencia entre ellos (en el caso real no siempre es así). Para generar alguna dependencia, lo haremos en las colas de pacientes.

Después de que cualquier recurso acabe de tratar al paciente, éste pasará por un mecanismo, que elegirá si el paciente **será enviado a casa,** o será enviado a la **cola de otro tipo de recurso** (desde un enfermero a un médico, o desde un médico a un

enfermero). Esto en un caso real se puede interpretar como una secuencia de tareas que envuelven distintos tipos de recursos (por ejemplo, un análisis de sangre realizado por un enfermero para ser atendido después por el médico).

Esta modificación se realizará en el bucle principal, con el objetivo de no alterar el propio bucle que se había diseñado anteriormente.

En cuanto a la representación gráfica, para cada uno de los dos tipos de recursos (doctores y enfermeras), se mostrarán gráficamente los recursos ocupados, además del total de pacientes en espera de la sala.



Los parámetros globales que afectan al sistema, como los límites y las reglas, no se han cambiado. Solamente se ha modificado el coste en tiempo de operación de cada recurso y su número.

8. Verificación y validación

8.1. Objetivos

Se ha añadido al sistema la funcionalidad de **realizar grandes cantidades** de simulaciones, con el fin de realizar una **aproximación estadística** del conjunto y, en conclusión, evaluar optimizaciones y mejoras en el sistema.

Para ello, necesitamos unos parámetros que sean relevantes a partir de la cola. A partir de la teoría de colas, utilizaremos estos:

- **Tiempo de espera**: concretamente el tiempo que un paciente ha estado esperando en cola para ser atendido.
- **Tiempo medio de espera:** La media del Tiempo de espera para todos los pacientes atendidos en todas las salas.
- **Tiempo del servicio**: este parámetro se define como el tiempo desde que el paciente llegue al hospital hasta que se marche después de haber sido completamente atendido.
- **Tiempo medio del servicio**: La media del Tiempo de servicio para todos los pacientes atendidos en todas las salas.
- **Pacientes no atendidos**: el número de pacientes que, después del fin de jornada, aún no han sido atendidos. En el caso de utilizar varios recursos, **se suman** todos los pacientes que estén esperando al fin de jornada.

  Dado estos criterios, consideramos que un **sistema es mejor o más eficiente** que otro si todos estos parámetros son menores.

8.2. Pruebas y tests obtenidos.

Para este paso, se definirán los parámetros que necesitemos evaluar de la teoría de colas para determinar que una simulación es eficiente. Para ello, utilizamos un caso simple:

- Utilizamos 2 salas, donde:
  - Haya 5 médicos y 3 enfermeros en una; y 2 médicos y 4 enfermeros en otro.
  - En ambas salas, existe la mitad de probabilidad de los pacientes de ser atendido inicialmente tanto por un médico como por un enfermero.
  - Después de ser atendido por un médico, hay un 90% de probabilidades de marcharse y 10% de ser atendido después por un enfermero.
  - Después de ser atendido por un enfermero, hay un 40% de probabilidades de marcharse y 60% de ser atendido después por un médico.

- Para la lógica difusa se utilizarán unos conjuntos y reglas similares a las expuestas anteriormente en el apartado de implementación de lógica difusa..

  Después, comparamos tanto utilizando la lógica difusa (intercambiando recursos) como si no lo tuviera (un sistema fijo, sin intercambio de recursos).

Sistema fijo:

Waiting time average (in minutes)	21.1608
Total spent time average (in minutes)	69.04
Average of unattended patients	55.84
Std desv of waiting time	11.1621
Std desv of total time	93.3051
Std desv of unattended patients	112.203

Figura 22: Sistema sin reubicación de recursos

Utilizando lógica difusa:

Waiting time average (in minutes)	20.9003
Total spent time average (in minutes)	65.8
Average of unattended patients	50.76
Std desv of waiting time	9.55403
Std desv of total time	100.916
Std desv of unattended patients	135.367

Figura 23: Sistema con reubicación de recursos mediante lógica difusa

Como se puede ver en la comparativa, aún con unas reglas sencillas y unas gráficas simples, se nota una pequeña mejora tanto en el tiempo de servicio (un 7% menos), como del tiempo de espera (un 1%). Esta mejora es más notable en los pacientes que no han sido atendidos (un 12%)

El hecho de que la desviación estándar sea mayor en todos los atributos utilizando lógica difusa (aumentando en 16%, 8% y 20% respectivamente) significa que la **variación** de estos valores es más grande, resumiendo en que es más inestable predecir los resultados (es más probable tanto de ser menor como de ser mayor que la media).

9. Conclusiones y líneas futuras

9.1. Conclusiones

Como conclusiones podríamos destacar la viabilidad de la lógica difusa en la toma de decisiones, en este caso dentro de la problemática de movimiento de recursos entre salas para optimizar el funcionamiento de colas de pacientes.

9.2. Líneas futuras

Para que el esquema esbozado en este proyecto tenga posibilidades de implantarse, habría que poder contemplar la realización de las siguientes tareas:

- Ampliar el programa actual de simulación flexibilizando los parámetros y contemplando más variedad de casos reales.
- Realizar la conexión con sistemas de recogida de datos y toma de decisiones.
- Estudio de requisitos para la implantación real del sistema, incluyendo costes, riesgos, tiempos, formación, mantenimientos, actualizaciones, securización, acceso a datos de carácter personal, ...

9.2.1. Ampliación de la Simulación.

Ampliar las funcionalidades del programa y flexibilizar el programa para que pueda adaptarse a una o varias casuísticas específicas. Habría que ampliar el funcionamiento de las simulaciones en los siguientes aspectos:

- Poder definir un número determinado de salas.
- Poder definir un número de determinado de recursos.
- Poder establecer el diagrama de flujo de los pacientes a consumir los diferentes recursos.
- Obtener parámetros de funcionamiento de las colas reales para poder ajustar la simulación que se asemeje lo más posible al caso real.
- Poder modificar las reglas de lógica difusa para que se adapte a un aprovechamiento óptimo de los recursos, en la mayoría de los casos.
- Poder describir la posibilidad o no de movimiento de recursos humanos entre salas, dependiendo de las plazas y las capacidades o habilidades de los recursos: por ejemplo determinar si una persona (recurso) está capacitada para desempeñar determinado puesto y es posible un movimiento de puesto. Esto puede requerir la conexión con el sistema de gestión de recursos humanos de la empresa.
- Horarios, dedicación, posibles horarios y/o jornadas de descanso, de los recursos humanos implicados en esta simulación.

9.2.2. Ampliación de conectividad con el entorno.

Ampliación de las funcionalidades para hacerlo operativo en un entorno real. El programa esbozado tiene como finalidad la integración dentro de un entorno en funcionamiento y que requerirá la implementación de:

- Conexión con sistemas dispensadores de tickets de colas gestionados centralizadamente.
- Conexión con el sistema de recursos humanos de la empresa para determinar el personal que puestos de trabajo ocupa, ubicación y posible movilidad a otros puestos de similar categoría, horarios, descanso del personal, .. etc
- Conexión con el sistema de entrada y salida de pacientes de atendimiento por parte de un recurso.
- Conexión con sistema de notificaciones de movimientos de recursos.

9.2.3. Implantación.

El objetivo fundamental de este programa es ser implantado en un entorno real y dar solución a la toma de decisiones de movimiento de recursos entre salas para optimizar las colas de espera de pacientes. En las fases de implantación podríamos señalar:

9.2.3.1. Análisis de costes y riesgos.

- Determinar el alcance del proyecto, con un análisis de costes y riesgos.
- Ver si hay decisión y oportunidad de abordarlo.

9.2.3.2. Implantación de un PILOTO en modo TEST.

- Determinar el alcance inicial del piloto.
- Realizar el despliegue de la solución.
- Impartir formación entre el personal implicado en este piloto.
- Poner en funcionamiento con datos reales, pero usar las decisiones obtenidas por el programa para evaluar la bondad de la solución y comparar con el funcionamiento sin tomar dichas decisiones.
- Evaluar si las decisiones tomadas por el programa son provechosas, son factibles de realizar, redundan en un mejor funcionamiento en la gestión de los recursos y el personal implicado ve como una mejora la posible implantación..

9.2.3.3. Implantación del PILOTO.

- Poner en funcionamiento el piloto con datos reales, haciendo caso de las

decisiones obtenidas por el programa.

- Evaluar si las decisiones tomadas por el programa han sido provechosas, han sido factibles de realizar, han redundan en un mejor funcionamiento en la gestión de los recursos y el personal implicado ha visto como una mejora la posible implantación.

9.2.3.4. Implantación progresiva.

- Suponiendo el éxito del piloto, hacer extensible la solución al resto del entorno de una manera progresiva y con el menor perjuicio para usuarios y personal.

10. Summary and Conclusions

The implementation has been done using C++ language, along with Qt graphical environment libraries and fuzzy logic libraries.

We have developed a minimal hospital environment consisting of 2 rooms, each with 2 set of resources: Doctors and Nurses, which each of them have their own patient queue.

We have simulated the evolution, in a workday, of the arrival of patients based on a Poisson distribution.

When a resource becomes free, based on a FIFO queue, this resource will attend the next patient.

Every certain period of time, we evaluate the queues' state and try to make a decision based on fuzzy logic.

To evaluate the queues state we have used 2 parameters:

- *"**Queue Tolerance**" which means the numbers of patients waiting on this queue*
- *"**Waiting TIme**" which stores the maximum waiting time in the queue.*

*With the aid of fuzzy logic we obtain a number "**Resource Reallocation Need**" in order to determine if we command to certain resources to move.*

Finally, we have tested on a bunch of simulations, with and without resources relocations, to compare both, taking into consideration more parameters:

- *"Waiting Average Time", which shows the average of the time spent by patients in queue.*
- *"Total Spent Average Time" which values the average of the time spent by patients from its arrival to the departure completely from hospital.*
- *Unattended patients, which garners patients who are still waiting until the end of the workday schedule and they haven't been attended.*

The results confirm that, even with a simple fuzzy system, we have better results when applying resource allocation using fuzzy logic that when we do not relocate resources at all, confirming that fuzzy logic it is a good approach to optimize and minimize the hospital patients queues, even with different types of resources .

11. Presupuesto

Tipo	Objeto	Tiempo	Coste
Hardware:	Ordenador para ejecutar la aplicación desarrollada.		1000€
Software:	Sistema Operativo: Linux		0€
Software:	Herramientas: C++, Qt (Open Version), , defuzzy library		0€
Desarrollo:	Coste de programación del programa de simulación de colas de pacientes en varias salas de un hospital, contemplando la toma de decisiones en mediante lógica difusa de movimiento de recursos entre salas para mejorar el tiempo de espera.	250 horas	3000 €
Total			**4000 €**

Anexos y Bibliografía.

Bibliografía

1. García Sabater, José Pedro. Aplicando Teoría de Colas en Dirección de Operaciones. UPV. 2015.

2. Donald Gross. Queueing Theory. 2008.

3. Zadeh, L. A. Fuzzy Logic. 1988

4. Zadeh, L.A. Fuzzy Sets. Information and Control, 8, 338-353. 1965

Apéndice. Algoritmos utilizados

loop.cpp

```
/*****************************************************************************
*
* Fichero loop.cpp
*
*****************************************************************************
*
* AUTORES : Adrian Prieto Curbelo
*
*
* FECHA:  01/04/2019
*
*
* DESCRIPCION: Bucle general del simulador de colas de pacientes en varias salas
*              de un Hospital, Con posibilidad de mover recursos entre Salas
*              aplicando toma de decisiones basadas en lógica difusa.
*
*
*****************************************************************************/

#include <simulator.h>

QVector<double> Simulator::debug_loop()
{
   const int number_of_resources = rooms_[0].number_of_resources();
   const int debug_time = total_time_;

   QVector<QVector<double>> rr;
   /* first resize by resources, then by number of rooms */
   rr.resize(number_of_resources);
   for (int i=0; i<rr.size(); i++) rr[i].resize(rooms_.size());

   QVector<int> change ;
   change.fill(-1, number_of_resources);
   QVector<int> min_it, max_it;
   min_it.resize(number_of_resources); max_it.resize(number_of_resources);

   int min_pos;

   //tools to capture room's general values
   unsigned long total_patients = 0;
   unsigned long unattended_patients = 0;
   QVector<unsigned long> record_time;
   record_time.resize(NUM_PARAMETERS);



   //initial log
   for (int j=0; j<rooms_.size(); j++)
   {  rooms_[j].clean_room();
      rooms_[j].log_one_cycle();
   }


   for (int i=0; i< debug_time; i++)
   {
```

```
      for (int j=0; j<rooms_.size(); j++)
        {


          total_patients += rooms_[j].add_patients_to_queue(i, debug_time);

          //quitar un tiempo de operación
          for (int k=0; k<rooms_[j].number_of_resources(); k++)
             {rooms_[j].get_resource(k)->remove_one_operation();
             }
        }



      if (i % ASK_TIME == 0)
      {
        // para cada recurso queremos conseguir los valores mínimos y
         // máximos como su índice

         QVector<double> max_; max_.fill(-1, number_of_resources);
         QVector<double> min_; min_.fill(2, number_of_resources);

         for (int j= 0; j< rooms_.size(); j++)
         {
            for(int k=0; k<rooms_[j].number_of_resources(); k++)
             {
                 // los obtenemos con la operación de lógica difusa
                 rr[k][j ] =  defuzzy(k, j);
                 if (rr[k][j ] < min_[k])
                 {
                    min_it[k] = j;
                    min_[k] = rr[k][j];
                 }
                 if (rr[k][j] > max_[k])
                 {
                    max_it[k] = j;
                    max_[k] = rr[k][j];
                 }
             }
          }

          // comprobamos si algunos de los recursos se puede mover
           // ( si hay recursos libres y se cimplen los límites establecidos)


           for(int k=0; k<number_of_resources; k++)
              if ((max_[k] > LIM_MAX) && (min_[k] < LIM_MIN))
              {
                 min_pos = rooms_[min_it[k]].get_resource(k)->ok_to_remove();
                 if (min_pos >= 0)
                 { change[k] = DESP_TIME;
                    rooms_[min_it[k]].get_resource(k)->remove_one_resource(min_pos);
                    std::cout << "At " << 8+ (i/60)  <<  ":" << i%60 << " a resource of type "<<
k+1 << " has left from Room" << min_it[k] << " to go to Room" << max_it[k] << std::endl;
                  }
              }

      }


       // si este recurso ya llegó a su destino añadir un recurso
      for (int k=0; k<change.size(); k++)
       { change[k] --;
         if (change[k] == 0)
            {
```

```
                rooms_[max_it[k]].get_resource(k)->add_one_resource();
                //std::cout << "At " << 8+ (i/60)  <<  ":" << i%60 << " a doctor has arrived from
Room" << min_it << " to Room" << max_it << std::endl;
             }
       }


      for (int j= 0; j< rooms_.size(); j++)
      {


         rooms_[j].assign_patients_to_idle_resources(record_time);
         rooms_[j].log_one_cycle();


      }
   }

   //calculated unattended patients at the end of the schedule
   for (int i=0; i<rooms_.size(); i++)
      for (int j=0; j<rooms_[i].number_of_resources(); j++)
   {
      unattended_patients +=( rooms_[i].get_resource(j)->queue_size() );
   }

   QVector <double> simulator_time;
   simulator_time.resize(NUM_PARAMETERS);
   simulator_time[WAITING_TIME] = (1.0 * record_time[WAITING_TIME]) / ( total_patients -
unattended_patients);
   simulator_time[TOTAL_TIME] = record_time[TOTAL_TIME] / ( total_patients -
unattended_patients);
   simulator_time[UNATTENDED_PATIENTS] = unattended_patients;

   std:: cout << "Waiting Time: " << simulator_time[WAITING_TIME] << std::endl;
   std:: cout << "Total Time: " << simulator_time[TOTAL_TIME] << std::endl;
   std:: cout << "Unattended Patients: " << simulator_time[UNATTENDED_PATIENTS];

   return simulator_time;
}

void Simulator::verification(double& mean_waiting_time, double& mean_total_time, double&
mean_unattended_patients, double& desv_waiting_time,  double& desv_total_time, double&
desv_unattended_patients)
{
   const int NUM_SIMULATIONS = 100;
   QVector<QVector<double> > records_;
   records_.resize(NUM_SIMULATIONS);
   for (int i=0; i<records_.size(); i++) records_[i].resize(NUM_PARAMETERS);

   double sum_waiting_time = 0;
   double sum_total_time = 0;
   double sum_unattended_patients = 0;

   for (int i=0; i<NUM_SIMULATIONS; i++)
     {
       records_[i] = debug_loop();
       sum_waiting_time += records_[i][WAITING_TIME];
       sum_total_time += records_[i][TOTAL_TIME];
       sum_unattended_patients += records_[i][UNATTENDED_PATIENTS];
     }

   mean_waiting_time = sum_waiting_time / NUM_SIMULATIONS;
   mean_total_time = sum_total_time / NUM_SIMULATIONS;
   mean_unattended_patients = sum_unattended_patients / NUM_SIMULATIONS;
```

```
   for (int j=0; j<NUM_SIMULATIONS; j++)
   {
     desv_waiting_time += pow((records_[j] [WAITING_TIME] - mean_waiting_time),2);
     desv_total_time += pow((records_[j][TOTAL_TIME] - mean_total_time), 2) ;
     desv_unattended_patients += pow((records_[j][UNATTENDED_PATIENTS] -
mean_unattended_patients), 2) ;
   }

   desv_waiting_time = sqrt (desv_waiting_time);
   desv_total_time = sqrt(desv_total_time);
   desv_unattended_patients = sqrt (desv_unattended_patients);

}
```

room.cpp

```
/*****************************************************************************
*
* Fichero room.cpp
*
*****************************************************************************
*
* AUTORES : Adrian Prieto Curbelo
*
*
* FECHA:  01/04/2019
*
*
* DESCRIPCION: Funciones necesarias para el manejo de salas.
*
*
*****************************************************************************/

#include "room.h"

int Resource::get_random_poisson(double median)
{
   static std::random_device rd;
   std::poisson_distribution<int> dist(median);
   return dist(rd);
}

void Room::log_one_cycle()
{

  Log_room_entry single_entry;

   single_entry.number_of_resources.resize(number_of_types);
   single_entry.queue.resize(number_of_types);
   single_entry.queue_tolerance.resize(number_of_types);
   single_entry.stay_time.resize(number_of_types);

 for (int i=0; i<number_of_types; i++)
 {   for(int j=0; j<resources[i].resource_size(); j++)
         if (resources[i].get_resource(j) != 0) single_entry.number_of_resources[i]++;

     single_entry.queue[i] = resources[i].queue_size();
     single_entry.queue_tolerance[i] = resources[i].get_tolerance_queue();
     single_entry.stay_time[i] = resources[i].get_stay_time();
 }

  log.push_back(single_entry);
```

```
}

int Room::add_patients_to_queue(int time_, int total_time_)
{
      const double poisson_median = resources[0].get_normal_dist_value(double (1.0 * time_ /
total_time_));
      const int more_patients = resources[0].get_random_poisson(double (1.0 * poisson_median /
1.9));

      //const int more_patients = resources[0].get_random_poisson(0.1);

      for (int i=0; i<more_patients; i++)
      { const double random_number = ((rand() % 100) * 1.0) / 100;
        for(int j=initial_prob.size()-1; j>=0; j--)
        { if(initial_prob[j] <= random_number)
          { resources[j].add_one_patient();
            break;
           }
        }
      }

      return more_patients;
}

void Room::assign_patients_to_idle_resources(QVector<unsigned long>& record)
{

   for (int i=0; i<number_of_types; i++)
     {   for(int j=0; j<resources[i].resource_size(); j++)
         { if (resources[i].get_resource(j) == 0)
           {  //move previous patient
              if (resources[i].get_count(j) != 0)
              {
                 const double random_number = ((rand() % 100) * 1.0) / 100;
                 for(int k=resource_prob[i].size()-1; k>=0; k--)
                  { if(resource_prob[i][k] <= random_number)
                     { resources[k].add_one_patient(); break;
                     }
                  }
              }
              resources[i].add_count(j);

              //when a patient is attended
              if(! resources[i].no_queue())
              {
                 // change 0 (idle) to time_operation (busy) to doctor
                 int operation_time = resources[i].get_operation_time();

                 resources[i].set_resource(j,operation_time);

                 record[WAITING_TIME] += operation_time;
                 record[TOTAL_TIME] += (operation_time + resources[i].get_first_patient());

                 // remove patient from queue
                 resources[i].substract_one_patient();

              }
           }
         }
     }
}
```

room.h

```
/*****************************************************************************
*
* Fichero room.h
*
/*****************************************************************************
*
* AUTORES : Adrian Prieto Curbelo
*
*
* FECHA:  01/04/2019
*
*
* DESCRIPCION: Estructuras de datos para manejar las salas
*
*
*****************************************************************************/

#ifndef ROOM_H
#define ROOM_H

#include <structs_constants.h>

class Resource
{
   int patient_tolerance = 8;
   int time_tolerance = 60;
   int desp_time;
   int ask_time;
   double median = 0.4;
   double desv = 0.2;

   int initial_resources;

   QString name;

   int operation_cost = 10;
   QVector<int> patient_queue;
   QVector<int> resource_status;
   QVector<int> resource_count;

public:
   /* gets */
   inline int get_resource(const int pos) {return resource_status[pos]; }
   inline int get_operation_time() {return get_random_poisson(operation_cost);}
   inline int get_count(int pos){return resource_count[pos];}
   inline int queue_size() {return patient_queue.size();}
   inline int resource_size() {return resource_status.size();}
   inline bool no_queue() {return patient_queue.isEmpty();}
   inline QString name_(){return name;}
   inline int get_first_patient() {return patient_queue[0];}
   inline int get_initial_resources() {return initial_resources;}


   /* sets */
   inline void add_one_patient() {patient_queue.push_back(0);}
   inline void substract_one_patient(){patient_queue.pop_front();}
   inline void add_count(int pos){resource_count[pos]++;}
   inline void set_resource(const int pos, const int value) {resource_status[pos] = value;}
   inline void set_name(const QString name_){name = name_;}
   inline void add_one_resource()
      {resource_status.push_back(0);
       resource_count.push_back(0);
      }
```

```
    inline void remove_one_resource (const int element)
       {resource_status.erase(resource_status.begin() + element);
        resource_count.erase(resource_count.begin() + element);
       }

    /* stadistical values */
    inline double get_normal_dist_value(double point) {return exp(-(point-median)*(point-
median) / (2*desv*desv));}
    inline double get_tolerance_queue () {return 1-(exp(-((1.0 *
queue_size())/patient_tolerance)));}
    inline double get_tolerance_queue (const int integer) {return 1-(exp(-((1.0 *
integer)/patient_tolerance)));}
    inline double get_stay_time() {return patient_queue.empty() ? 0 : 1-(exp(-((1.0
*patient_queue[0])/time_tolerance))); }
    int get_random_poisson(double);

    /* cycle operations */
     void remove_one_operation()
    {    for (int i=0; i<patient_queue.size(); i++)  patient_queue[i]++;
         for (int i=0; i<resource_status.size(); i++) if (resource_status[i] > 0) resource_status[i]--;
    }

    void one_cycle(int, int);
    void prompt(int);

    int ok_to_remove()
    {
       for (int i=0; i<resource_status.size(); i++)
          if (resource_status[i]== 0) return i;
       return -1;
    }

    inline void set_med(const double med_) {median = med_;}
    inline void set_desv(const double desv_) {desv = desv_;}
    inline void resource_resize(const int size, const int value = 0)
    {
      initial_resources = size;
      resource_status.fill(value, size);
      resource_count.fill(value, size);
    }

    void clear_patient_queue () {patient_queue.clear();}


};

class Room
{
       const int number_of_types;

       QVector<Resource> resources;

       QVector<double> initial_prob;
       /* In  resource[i]
        * probability a patient in queue will be attended by a resource of type i
        */
       QVector<QVector<double>> resource_prob;
        /* In resource[i][j]
        * probabilty that a patient will go to resource of type j after being attended by
resource type i.
        * resource[i][i] is the prob the same patient can go home.
        * Check that sums of resource[i][X] equals 1;
        */
       QVector<Log_room_entry> log;
```

```
   public:
      /* constructor methods */
      Room(): number_of_types (2)  {set_number_of_types();}
      inline void set_initial_number_resources(const int type, const int a)
{resources[type].resource_resize(a, 0);}
      void set_number_of_types()
      {
         resources.resize(number_of_types);
         initial_prob.resize(number_of_types);
         resource_prob.resize(number_of_types);

         for(int i=0; i<number_of_types; i++)
         {  initial_prob[i] = (1.0 * i) / number_of_types;
            resource_prob[i].resize(number_of_types);
         }

         resources[0].set_name("doctor");
         resources[1].set_name("nurse");
      }

      /* gets & sets */
      inline int get_log_size () { return log.size(); }
      inline int get_log_resource_status(const int resource, const int page) { return
log[page].number_of_resources[resource]; }
      inline int get_log_queue( const int resource, const int page) { return
log[page].queue[resource]; }
      inline int get_last_queue(const int resource){return log.last().queue[resource];}
      inline double get_last_stay_time(const int resource){return log.last().stay_time[resource];}
      inline double get_last_queue_tolerance(const int resource){return
log.last().queue_tolerance[resource];}
      inline Resource* get_resource(const int pos) { return &(resources[pos]); }
      inline int number_of_resources() {return number_of_types;}
      inline void set_resource_prob(int first, int second, double value) {resource_prob[first]
[second] = value;}
      inline void set_med_desv(const int resource, const double med, const double desv)
{resources[resource].set_med(med); resources[resource].set_desv(desv);}

      /* used for loop */
      void log_one_cycle();
      int add_patients_to_queue(int, int);
      void assign_patients_to_idle_resources(QVector<unsigned long>&);
      void clean_room()
      {
         for(int i=0; i<number_of_types; i++)
         {
            resources[i].resource_resize(resources[i].get_initial_resources());
            resources[i].clear_patient_queue();
         }
         log.clear();
      }
};

#endif // ROOM_H
```

simulator.cpp

```
/***************************************************************************
*
* Fichero simulator.cpp
*
***************************************************************************
```

```
*
* AUTORES : Adrian Prieto Curbelo
*
*
* FECHA:  01/04/2019
*
*
* DESCRIPCION: Simulador
*
*
*********************************************************************************/
#include "simulator.h"
#include "ui_simulator.h"

Simulator::Simulator(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::Simulator)
{
    ui->setupUi(this);

    ui->notification->setAlignment(Qt::AlignCenter);

    get_variables_from_file();

    //show current fuzzy rules
    /*open_window_with_io_graph(input_[0].id, input_[0].term);
    open_window_with_io_graph(input_[2].id, input_[2].term);
    open_window_with_io_graph(output_[0].id, output_[0].term);
    */

    start = false;
    build_room();
    createMenu();
}

Simulator::~Simulator() { delete ui; }

double Simulator::get_tolerance_queue(const int resource, const int pos, const double
time_interval)
{
    const double c = rooms_[pos].get_resource(resource)->get_normal_dist_value(time_interval);
    const int d = rooms_[pos].get_resource(resource)->get_random_poisson(c);
    return rooms_[pos].get_resource(resource)->get_tolerance_queue(d);
}

void Simulator::build_room()
{
    const int number_of_rooms = 2;

    rooms_.resize(number_of_rooms);

    rooms_[0].set_initial_number_resources(0, 5);
    rooms_[0].set_initial_number_resources(1, 3);
    rooms_[1].set_initial_number_resources(0, 2);
    rooms_[1].set_initial_number_resources(1, 4);

    rooms_[0].set_med_desv(0, 0.7, 0.6);

    rooms_[0].set_resource_prob(DOCTOR, DOCTOR, 0.9);
    rooms_[0].set_resource_prob(DOCTOR, NURSE, 1);
    rooms_[0].set_resource_prob(NURSE, NURSE, 0.4);
    rooms_[0].set_resource_prob(NURSE, DOCTOR, 1);

    rooms_[1].set_resource_prob(DOCTOR, DOCTOR, 0.9);
    rooms_[1].set_resource_prob(DOCTOR, NURSE, 1);
```

```
    rooms_[1].set_resource_prob(NURSE, NURSE, 0.4);
    rooms_[1].set_resource_prob(NURSE, DOCTOR, 1);

    //rooms_.last().rechange_paramethers(0.5, 0.1, 5);
    //rooms_[0].set_initial_number_doctors(INITIAL_NUMBER_DOCTORS_0);
    //rooms_[1].set_initial_number_doctors(INITIAL_NUMBER_DOCTORS_1);
}

double Simulator::defuzzy(const int resource, const int room_number)
{
    //do it for each resource
    const double stay_time = rooms_[room_number].get_last_stay_time(resource);
    const double queue_tolerance = rooms_[room_number].get_last_queue_tolerance(resource);

    engine = new Engine;

    InputVariable* in_queue_tolerance = input_[room_number].variable;
    InputVariable* in_stay_time = input_[room_number + rooms_.size()].variable;

    OutputVariable* out_ = output_[room_number].variable;

    engine->addInputVariable(in_stay_time);
    engine->addInputVariable(in_queue_tolerance);
    engine->addOutputVariable(out_);

    //add rules
    RuleBlock* rules = new RuleBlock;
    rules->setEnabled(true);
    rules->setConjunction(new AlgebraicProduct);
    rules->setDisjunction(new AlgebraicSum);
    rules->setImplication(new AlgebraicProduct);
    rules->setActivation(new General);
    rules->addRule(Rule::parse("if queue_tolerance" + std::to_string(room_number+1) + " is high
then redistribution" + std::to_string(room_number+1) + " is high" ,engine));
    rules->addRule(Rule::parse("if queue_tolerance" + std::to_string(room_number+1) + " is low
and stay_time" +  std::to_string(room_number+1) + " is medium then redistribution" +
std::to_string(room_number+1) + " is low" ,engine));
    rules->addRule(Rule::parse("if queue_tolerance" + std::to_string(room_number+1) + " is low
and stay_time" +  std::to_string(room_number+1) + " is low then redistribution" +
std::to_string(room_number+1) + " is low" ,engine));
    engine->addRuleBlock(rules);


    in_stay_time->setValue(stay_time);
    in_queue_tolerance->setValue(queue_tolerance);

    engine->process();
    return out_->getValue();
}
```

simulator.h

```
/**************************************************************************************
*
* Fichero simulator.h
/**************************************************************************************
*
* AUTORES : Adrian Prieto Curbelo
*
*
* FECHA:  01/04/2019
```

```
*
*
* DESCRIPCION: Estructuras de datos para la simulacion
*
*********************************************************************************/

#ifndef SIMULATOR_H
#define SIMULATOR_H

#include "room.h"

namespace Ui {
class Simulator;
}

using namespace fl;

class Simulator : public QMainWindow
{
    Q_OBJECT

public:
    explicit Simulator(QWidget *parent = nullptr);
    ~Simulator();
    void set_initial_state();
    void create_engine_from_file();
    void set_initial_values();

    void main_loop();
    void build_graph(bool, int);

    /* auxiliar functions */
    std::string search_file();
    void timeout();
    void read_default_file();

    void get_variables_from_file();
    void build_all_graphs();
    void build_side_layout();

    QVector<double> debug_loop();
    void verification(double&, double&, double&, double&, double&, double&);

    void clean_simulator();
    void build_room();
    void simple_algorithm_one_cycle();
    double defuzzy(const int resource, const int number_room);


    /* rooms' functions */
    double get_tolerance_queue(const int resource, const  int room, const double time_interval);
    double get_stay_time(const int resource, const int pos ){ return
rooms_[pos].get_resource(resource)->get_stay_time();}

    /* gui functions */
    void createMenu();
    void build_single_patient_chart(const int room);
    void open_window_single_patient_graph(const int room);
    void open_window_correlation();
    void build_room_correlation_chart();
    void build_io_graph(const QString&, const QVector<term_graph>&);
    void open_window_with_io_graph(const QString&, const QVector<term_graph>&);
    void open_window_change_parameters(const int room_number);
```

```
private:
    Ui::Simulator *ui;

    fl::Engine* engine;
    QVector<Fuzzy_Input> input_;
    QVector<Fuzzy_Output> output_;

    QChart* chart;

    const QString current_file = "fll/tfg_simple.fll";

    QButtonGroup* group = new QButtonGroup(this);
    QChartView* chartView;

    int time_ = 0;
    const int total_time_ = 8*60;

    QVector<Room> rooms_;

    QVBoxLayout* main_layout;
    QVBoxLayout* rooms_gui_list;

    bool start;
};

#endif // SIMULATOR_H
```