UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Evaluación de algoritmos para la simulacion estocástica de epidemias mediante sistemas P

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Gómez González, Sergio

Tutor/a: Sempere Luna, José María

Cotutor/a: Campos Frances, Marcelino

CURSO ACADÉMICO: 2022/2023

Índice

1. Introducción 3
2. Motivación 3
3. Objetivos 3
4. Estado del arte 3
5. Análisis del problema 4
6. Diseño de la solución 5
   6.1. Motor de simulación 5
   6.2. Algoritmos 7
      6.2.1. Algoritmo ARES 7
      6.2.2. Algoritmo de Gillespie 8
      6.2.3. Algoritmo K-Gillespie 8
      6.2.4. Algoritmo aleatorio 8
   6.3. Formato de escenarios de simulación 9
      6.3.1. Chaining 9
      6.3.2. Dos hábitats 9
      6.3.3. Nesting 10
      6.3.4. Anillo 11
7. Desarrollo de la solución 11
   7.1. Motor de simulación 12
      7.1.1. Librería *util* 12
      7.1.2. Archivo Multiset 13
      7.1.3. Librería *algorithm* 13
      7.1.4. Archivo simulator 15
   7.2. Escenarios de simulación 16
   7.3. Interfaz de usuario 17
8. Pruebas 17
   8.1. Prueba del motor de simulación 18
   8.2. Prueba de la implementación de los algoritmos 19
9. Experimentación 19
   9.1. Escenario *Chaining* 19
      9.1.1. ARES 20
      9.1.2. Aleatorio 20
      9.1.3. Gillespie maximal 20
      9.1.4. Gillespie minimal 21
   9.2. Escenario Dos Hábitats 22

9.2.1. ARES ... 22
9.2.2. Aleatorio ... 23
9.2.3. Gillespie maximal ... 23
9.2.4. Gillespie minimal ... 24
9.3. Escenario *Nesting* ... 25
9.3.1. Creación y disolución de membranas ... 25
9.3.2. Ciclo de objetos ... 27
9.4. Escenario Anillo ... 30
9.4.1. ARES ... 30
9.4.2. Aleatorio ... 31
9.4.3. Gillespie maximal ... 31
9.4.4. Gillespie minimal ... 32
10. Conclusiones ... 32
10.1. ARES ... 32
10.2. Gillespie maximal ... 33
10.3. Aleatorio ... 33
10.4. Gillespie minimal ... 33
11. Bibliografía ... 34

1. Introducción

La computación con membranas está demostrando ser de gran utilidad a la hora de simular la evolución de epidemias. En estos simuladores, basados en sistemas P, una de las tareas cruciales es elegir cuál de las reglas que gobiernan la evolución del simulador se va a aplicar. Para ello, se han creado diversos algoritmos como formas de solucionar este problema, comúnmente denominados algoritmos de elección de regla. En este trabajo se han probado algunos de estos algoritmos en diversos escenarios para observar cómo varía la evolución del simulador en función del algoritmo escogido.

Además, una vez escogida la regla, esta puede ejecutarse con máximo o mínimo paralelismo. Esto quiere decir que la regla se ejecuta exhaustivamente (tantas veces como sea posible) o una sola vez, respectivamente. En este trabajo también se abordan las diferencias entre estos dos tipos de ejecuciones.

En concreto, se han implementado y probado el algoritmo usado en ARES (Campos, y otros, 2015), un algoritmo aleatorio de creación propia, el algoritmo de Gillespie en su versión simple y el k-Gillespie, que es una generalización del anterior. Este último algoritmo es una aportación original de este trabajo.

2. Motivación

En la actualidad hay multitud de simuladores de sistemas P y cada cual usa el algoritmo de elección de regla que, en su momento, pareció más conveniente a la hora de diseñarlo. El objetivo de este estudio es establecer si estos simuladores serían equivalentes. Es decir, si se suministrase una misma configuración inicial a varios de estos simuladores ¿obtendríamos los mismos resultados? El objetivo de este estudio es responder a esta pregunta. En caso afirmativo los resultados obtenidos por cualquiera de los simuladores serían más sólidos. La simulación realizada por un simulador podría ser imitada por cualquier otro.

3. Objetivos

En este trabajo estudiaremos, no sólo si confluyen en un resultado común, sino también cuántos pasos de ejecución necesitan y cuán costosos son dichos pasos.

Dividiremos este objetivo en varios más sencillos para resolver la incógnita. El primer subobjetivo, claro está, será la creación del motor de simulación basado en sistemas P. A este le sucederá la creación de varios escenarios de simulación, o configuraciones iniciales del simulador. Además, se deberá implementar cada uno de los algoritmos de elección de regla y adaptar el motor de simulación para integrarlos. Finalmente se probarán los diferentes algoritmos en todos los escenarios y de estas pruebas se extraerá la conclusión.

4. Estado del arte

Este trabajo se nutre del modelo de computación con membranas. Dicho modelo fue enunciado por primera vez en el artículo *Computing with membranes* (Paun, 1998). Este marco ha resultado de especial utilidad para el desarrollo de simuladores que emulan procesos químicos y biológicos. Como este trabajo se ha centrado en el área de la epidemiología, nos fijaremos más en dicho ámbito. En la Universidad Politécnica de Valencia se han publicado varios trabajos respecto de los sistemas P y la epidemiología. Dos de ellos han sobresalido de entre los demás: LOIMOS (Baquero, Campos, Llorens, & Sempere, 2021) y ARES (Campos, y otros, 2015). LOIMOS es un simulador epidemiológico desarrollado durante la epidemia COVID-

19, y en el cual trabajaron los tutores de este TFG, junto con otros compañeros. Por su parte, ARES es otro simulador especializado en el análisis de la resistencia a antibióticos. En él participaron también los tutores de este TFG trabajando con un grupo más amplio.

Estos son los proyectos que más han influenciado este trabajo por ser los más cercanos, no obstante, hay más ejemplos de aplicaciones que han seguido esta línea. Por citar alguno, podemos enunciar el trabajo *Simulating Multilevel Dynamics of Antimicrobial Resistance in a Membrane Computing Model* (Campos, y otros, 2019).

El modelo de computación con membranas se basa la definición de una estructura y un contenido, y unas reglas que rijan la evolución de dicha estructura y contenido. Así, se denomina estructura de un sistema P al anidamiento de membranas. Cada membrana puede tener un tipo diferente y puede haber varias membranas con el mismo tipo. De esta forma, cada membrana delimita una región del sistema. Estas regiones marcan el ámbito de aplicación de las reglas. Es decir, una regla puede aplicarse dentro de una membrana. Estas reglas gobiernan la evolución de esta estructura creando o disolviendo membranas, o incluso conservando las membranas, pero modificando el anidamiento de las mismas. Puede, por ejemplo, expulsarse una membrana de otra. Además, estas reglas pueden modificar el contenido de las membranas (en forma de objetos que simulan sustancias químicas) cambiando la naturaleza de ese contenido o cambiando su ubicación.

De esta forma, se simula el funcionamiento de una célula eucariota, en las que diferentes químicos se combinan en ciertos orgánulos para dar lugar a otros productos. Las reglas que modifican la estructura del sistema se inspiran en los procesos de reproducción celular.

5. Análisis del problema

Para obtener pruebas empíricas de la similitud de las evoluciones de un simulador que utilice diferentes algoritmos, deberemos crear dicho simulador de sistemas P. Existe una gran variedad de interpretaciones de los sistemas P enunciados por Paun. En este trabajo hemos utilizado los descritos por tuplas del tipo (V, H, μ, Ω, R), donde:

- V es el alfabeto de objetos. Los objetos en los sistemas P emulan sustancias químicas que interactúan entre sí dentro de una célula.
- H es el alfabeto de tipos de membrana. Las membranas se corresponden con los elementos del mismo nombre en la biología celular. Contienen, en principio, a los objetos, aunque estos últimos pueden atravesarlas en ocasiones.
- μ denota la estructura inicial del sistema P. Este puede ser descrito en forma de cadena de texto, así, las membranas se representan de la forma "$[]_{Tipo}$", donde Tipo ha de estar contenido en H. Los objetos, por su parte, serán representados como letras. Así, los elementos situados entre la apertura y cierre de un corchete están ubicados en una membrana. Esto es pura notación y puede utilizarse cualquier otra, pero en este trabajo usaremos esta cuando se expongan ejemplos y aclaraciones.
- $\Omega = \bigcup \omega_i$ es el conjunto de objetos iniciales contenidos en cada región. Así, cada ω_i denotaría el conjunto de objetos contenidos en la región *i*.
- R es el conjunto de reglas que rigen la evolución del sistema P. Están sujetas a unos tipos expuestos por Gh. Paun en la definición de los sistemas y estudios posteriores. Estos tipos son los siguientes:
    - Evolución: es la sustitución de un conjunto de objetos por otro conjunto de objetos, de la forma $x, y \in V^*: x \to y$.

- Comunicación hacia dentro: un conjunto de objetos o membranas es absorbido por otra membrana, de la forma $x, y \in V^*, M \in H: x[\quad]_M \rightarrow [y]_M$ o, en el segundo caso, $M, N \in H: [\quad]_N[\quad]_M \rightarrow [[\quad]_N]_M$.
- Comunicación hacia fuera: un objeto o membrana es expulsado por otra membrana, de la forma $x \in V^*, M \in H: [x]_M \rightarrow y[\quad]_M$, o, en el segundo caso, $M, N \in H: [[\quad]_N]_M \rightarrow [\quad]_M[\quad]_N$. Si el elemento expulsado traspasa la membrana más externa, desaparece.
- Disolución de membrana: se elimina la membrana en cuestión y su contenido se libera a la membrana que la contenía, en caso de existir.

Será necesario, pues, que el motor de simulación por pasos implementado pueda manejar estos conceptos para poder obtener unos resultados válidos. También se deberán crear casos concretos de sistemas P sobre los que realizar las pruebas. Será necesario, asimismo, fijar un patrón para proporcionar los sistemas P al motor de simulación.

Por último, se habrá de implementar una serie de algoritmos de elección de regla y adaptar el simulador para su correspondiente uso.

6. Diseño de la solución

En el presente apartado se presentará el diseño del simulador de sistemas P que se propone crear. Se especificará el diseño del motor de simulación, los algoritmos de elección de regla y el formato de entrada del simulador.

6.1. Motor de simulación

Para implementar el motor de simulación se seguirá el siguiente diagrama UML basado en la arquitectura del simulador ARES (Campos, y otros, 2015):

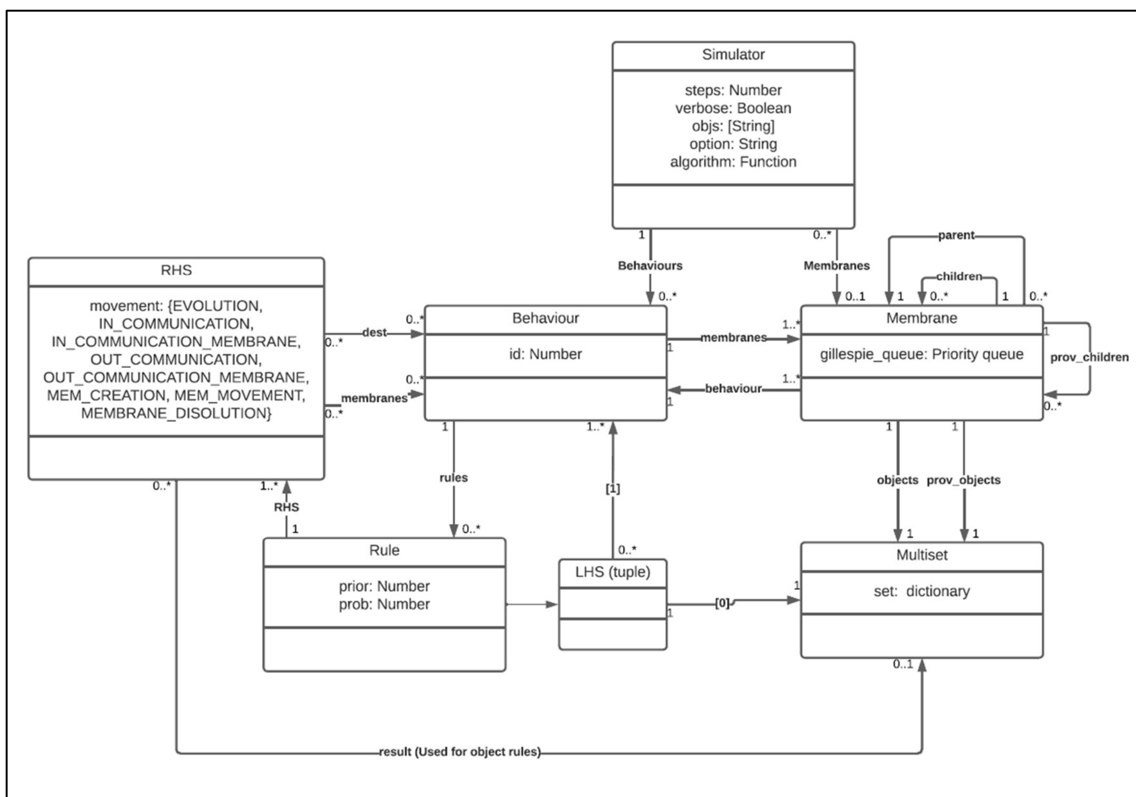


Figura 1: diagrama UML del motor de simulación

De esta forma, la clase *Simulator* sería el motor de simulación en sí mismo, y el resto de las clases compondrían el resto de los conceptos.

Empezando por *Simulator*, podemos ver que en todo momento tenemos almacenado un atributo *option* que contendrá el nombre del algoritmo a ejecutar. Este algoritmo se implementará en una función a parte cuya referencia se guardará en *algorithm*. Contiene también un diccionario que contiene los tipos de membrana y una referencia a la membrana entorno. También almacenará el alfabeto de objetos y los pasos que ha ejecutado. Finalmente, el motor incluye una función *verbose* para proporcionar información más allá de la ejecución.

Multiset es una clase auxiliar que facilita la gestión de multiconjuntos. Incluirá operaciones como la inclusión de un nuevo elemento, su eliminación, el conteo de elementos de un tipo y operaciones relacionadas con la lógica de conjuntos.

Respecto de la clase *Membrane*, representa una membrana de un sistema P. Tiene enlaces tanto a la membrana padre como a las hijas y una lista de hijas provisionales. Esta última se usa para almacenar las hijas que la membrana tendrá en el siguiente paso de ejecución. Las referencias a los objetos y objetos provisionales tienen una función análoga. Contiene, asimismo, una cola de prioridad para utilizar el algoritmo de Gillespie. Su uso se explicará junto con el algoritmo. Por último, se guarda una referencia al tipo de membrana.

La clase *Behaviour* es una forma de optimizar espacio, dado que en muchas membranas actuarán las mismas reglas, tendrán el mismo comportamiento. Así se guardarán las membranas en las que actúa este comportamiento, las reglas que lo definen y su identificador.

Una regla (*Rule*) tiene, en primera instancia, una parte izquierda (requerimientos para aplicarse) y una parte derecha (resultado). Así la parte izquierda contendrá los objetos y tipos de membrana necesarios para activar la regla. La parte derecha se compone de varios tipos de regla (codificados como *RHS*), que identifican los diferentes tipos de producciones de una regla. Sin embargo, esto habrá que explicitarlo en la definición del sistema P. Además, se incluye una prioridad y una propensión o probabilidad de aplicación.

Finalmente, *RHS* representa el tipo de regla, codificado en el atributo *movement*. El resto de las referencias tendrán una connotación distinta en función de cuál sea el tipo de regla:

- EVOLUTION: el resultado de la derivación se guardará en *result*. Las otras referencias quedan vacías.
- IN_COMMUNICATION: la membrana que absorberá los objetos tendrá el tipo apuntado por *dest*. Los objetos absorbidos se guardarán en *result*.
- IN_COMMUNICATION_MEMBRANE: la membrana que absorberá los objetos tendrá el tipo apuntado por *dest*. Las membranas absorbidas serán de los tipos de la lista *membranes*.
- OUT_COMMUNICATION: los objetos expulsados se guardan en *result*. El resto de las referencias no se usan.
- OUT_COMMUNICATION_MEMBRANE: las membranas expulsadas tienen el tipo apuntado por la lista *membranes*.
- MEM_CREATION: la membrana creada tendrá el tipo apuntado por *dest* y contendrá los objetos de *result*.
- MEMBRANE_DISOLUTION: el tipo de la membrana a disolver será el apuntado por *dest*. El contenido de dicha membrana será liberado en la membrana madre.

- MEM_MOVEMENT: las membranas del tipo indicado por *membranas* serán movidas a la membrana del tipo *dest*. La membrana "origen" será inferida usando la información de la parte izquierda de la regla. Es una suerte de atajo para mover una membrana del interior de otra al interior de una tercera. Tanto la membrana origen como la destino deben ser hijas de la misma membrana.

En cada paso de ejecución del motor de simulación se analizarán las reglas que pueden ejecutarse en cada membrana del simulador. Tras dicho análisis se ejecutarán las reglas decretadas por el algoritmo de elección de regla aplicado.

6.2. Algoritmos

El motor de simulación expuesto necesita una forma de elegir las reglas que se van a ejecutar un número determinado de veces. Para seleccionarlas, se implementarán los algoritmos como funciones de una librería. A continuación, se detallan los algoritmos implementados.

6.2.1. Algoritmo ARES

Es un algoritmo que aplica las reglas de forma exhaustiva. Es decir, aplicará todas las reglas tantas veces como sea posible, respetando los conflictos de competencia entre reglas y las prioridades. Es el algoritmo utilizado en el simulador ARES (Campos, y otros, 2015).

Utiliza como entrada el contenido de la membrana y las reglas que pueden actuar en ella, junto con sus prioridades y propensiones. Tras algunas iteraciones devolverá el número de veces que será aplicada cada regla.

Consta de siete pasos:

1. Crear los bloques de competencia: se agruparán todas las reglas en este tipo de bloques. Una regla pertenecerá a un bloque si alguno de los elementos de su parte izquierda coincide con alguno de la parte izquierda de otra regla del bloque y tienen la misma prioridad. Es decir, si la regla en cuestión compite con alguna otra regla del bloque. No es necesario que haya competencia entre todas las reglas del bloque; basta con la competencia entre pares.
2. Si el bloque solo contiene una regla, se devolverá el número máximo de veces que pueda ser aplicada en la membrana y se continúa con el paso siete. En caso contrario no se hace nada y se continúa con el siguiente paso.
3. Se calcula el número máximo de veces que podría ser ejecutada cada regla del bloque si no hubiese competencia. Llamaremos a estos valores vector de ejecución.
4. Se calcula el número de objetos de cada tipo que se consumirían si todas las reglas se ejecutasen las veces indicadas por el vector de ejecución. Nos referiremos a estos valores como vector de consumo.
5. Se divide el número de objetos de cada tipo que hay en la membrana entre la respectiva cantidad del vector de consumo. Si el valor mínimo obtenido en este paso es mayor o igual que 1, ejecutamos el paso siete. Si dicho valor es menor, ejecutaremos el paso seis.
6. Se multiplica el valor obtenido en el paso 5 por los respectivos valores del vector de ejecución. Continuaremos ejecutando el paso 4.
7. Se multiplica el vector de ejecución por la propensión de cada regla. Después se redondeará el vector obtenido y se devolverá dicho vector como el número de veces que ha de aplicarse cada regla del bloque. Si quedan más bloques por analizar, se ejecutará el paso dos con el siguiente.

Huelga decir que un mismo objeto o membrana no puede ser utilizado por varias reglas en un mismo paso de ejecución.

6.2.2. Algoritmo de Gillespie

Es un algoritmo que, en su versión más básica, devuelve una regla de un conjunto respetando las prioridades y propensiones. Utiliza como entrada el contenido de una membrana y las reglas que actúan en ella, junto con sus prioridades y propensiones. Tras ejecutarlo, se obtendrá la regla que debe ser ejecutada y el tiempo de espera hasta la siguiente reacción. A este último concepto, en adelante, se le denominará tau (τ).

Se procede a describir el algoritmo:

1. Se obtendrá la suma de las propensiones de las reglas proporcionadas. Sea dicho valor p_0.
2. Se generarán dos valores aleatorios uniformemente distribuidos en el rango (0,1). Sea este par a_1 y a_2 respectivamente.
3. Se calculará tau: $\tau = \frac{1}{p_0}\ln\frac{1}{a_1}$.
4. Se elegirá el índice j tal que $\sum_{k=1}^{j-1} p_k < a_2 * p_0 \leq \sum_{k=1}^{j} p_k$, siendo p_k la propensión de la regla de índice *k*.
5. Se devolverá el par (*j*, τ). Se modificará la tupla sumando el número de paso actual a τ.

Cuando el número de paso actual sea igual o mayor que la prioridad del elemento más prioritario de la cola, se ejecutará dicha regla y se volverá a aplicar el algoritmo.

6.2.3. Algoritmo K-Gillespie

El algoritmo K-Gillespie es una generalización del algoritmo de Gillespie de creación propia. Al igual que en el algoritmo de Gillespie, a cada regla se le asigna un tiempo de espera (τ). En este caso, habrá una cola de espera que contendrá *K* reglas. Cada una puede tener un tiempo de espera diferente. Cuando se agote el tiempo de espera de alguna de ellas, se aplicará y se elegirá una nueva regla que la sustituya en la cola de espera usando el algoritmo de Gillespie. Así, cada vez que se aplique una regla se realizará una ejecución independiente del algoritmo de Gillespie para elegir una regla.

Esta generalización puede aportar cierto grado de paralelismo sobre el algoritmo original de Gillespie. Aunque esto es una teoría, es también el objetivo de este trabajo el verificar esta ventaja y encontrar otras.

6.2.4. Algoritmo aleatorio

Es un algoritmo de creación propia que se usará para comprobar el funcionamiento del simulador. Sigue los siguientes pasos:

1. Crear los bloques de competencia explicados en el apartado del algoritmo de ARES (Campos, y otros, 2015). Después para cada bloque aplicar el paso 2.
2. Ordenar el bloque de forma aleatoria. Aplicar el paso 3 para cada regla en el orden obtenido.
3. Devolver la regla con el número de veces máximo que puede ser aplicada, multiplicada por la probabilidad de aplicar la regla.

6.3. Formato de escenarios de simulación

El escenario de simulación es la representación de un sistema P y servirá como entrada al simulador. Una vez especificado qué escenario debe simular, se realizarán pasos de ejecución en los que el sistema P evolucionará de acuerdo con lo especificado en el escenario.

El escenario de simulación debe constar de tres partes principales: la estructura inicial, el comportamiento de las membranas y el alfabeto de objetos. En la primera deberá especificarse el anidamiento de las membranas, los objetos que contiene cada una y el comportamiento que siguen. En el segundo, se explicitarán los diferentes comportamientos posibles. Esto es, los conjuntos de reglas que rigen las interacciones dentro de la membrana. No podrán crearse comportamientos dinámicamente, de modo que todos los comportamientos necesarios deberán ser descritos en el escenario de simulación. Finalmente, el alfabeto de objetos es una lista con los identificadores de los posibles objetos.

Nos proponemos los cuatro escenarios que se indican a continuación.

6.3.1. Chaining

Es un escenario sencillo que trabaja con reglas de objetos. Hay tres tipos de objetos que habitan el escenario *x*, *y* y *z*. La membrana entorno contiene tres tipos de membranas hijas: *A*, *B* y *C*. La membrana *A* se centra en la producción de objetos *x* y eliminación de *y*. La membrana *B* tendrá el comportamiento opuesto: generación de *y* y eliminación de *x*. Por último, la membrana *C* absorbe *x* e *y* para producir *z*. La membrana entorno será un corredor para los objetos.

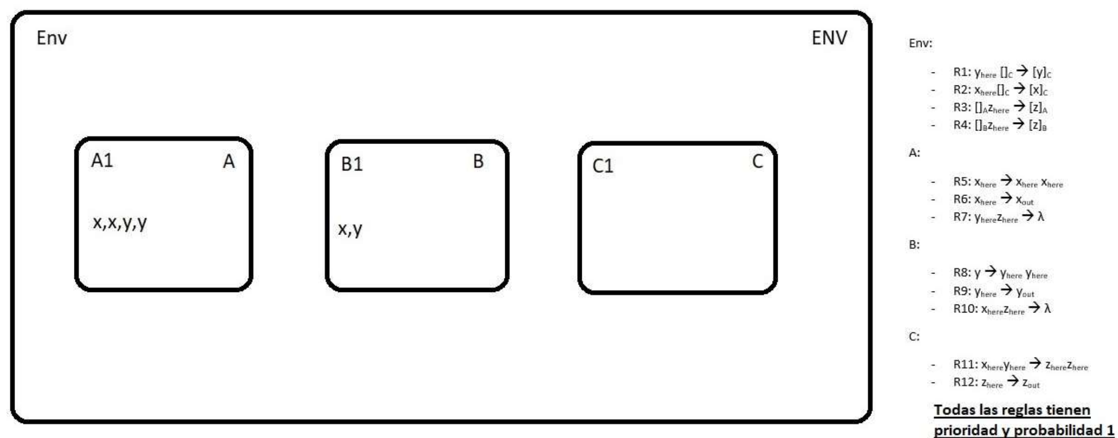
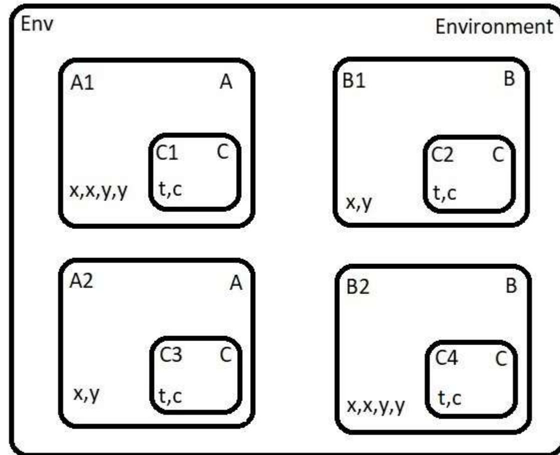


Figura 2: diagrama del escenario Chaining.

6.3.2. Dos hábitats

Describe un sistema P en el que existen dos tipos de membranas (*A* y *B*) que pueden contener objetos *x* e *y* junto con otro tipo de membrana (*C*). En este escenario entran en juego diferentes prioridades y probabilidades. Como resultado, *A* y *B* no se especializan en la producción de unos u otros objetos, no obstante, son más favorables a producir un tipo diferente. A su vez las membranas *C* se crean, anidan y diluyen de formas diferentes según estén en una u otra membrana. Cada membrana *C* contiene un objeto *c*, que cuando se libera a una membrana *A* o *B* se elimina. Hay, además otros objetos, *t* y *z*, cuyo destino está ligado a una membrana *C*. El objeto *t* es una suerte de catalizador que produce objetos *z* si está contenido en una membrana *C*. Es un objeto limitado, ya no pueden producirse más de las que hay inicialmente. Tampoco se pueden eliminar, por lo que su número permanece constante.



Environment:

- R1: $x_{here}y_{here}[]_A[]_B \rightarrow [x]_A[y]_B$ (prior=1, prob=0.75)
- R2: $x_{here}y_{here}[]_A[]_B \rightarrow [y]_A[x]_B$ (prior=1, prob=0.5)
- R3: $[[]_C]_A[]_B \rightarrow []_A[[]_C]_B$ (prior=2, prob=0.5)
- R4: $[]_A[[]_C]_B \rightarrow [[]_C]_A[]_B$ (prior=2, prob=0.5)

A:

- R5: $x_{here} \rightarrow x_{here}x_{here}y_{here}$ (prior=3, prob=0.75)
- R6: $y_{here} \rightarrow x_{here}y_{here}y_{here}$ (prior=3, prob=0.25)
- R7: $x_{here}x_{here}y_{here} \rightarrow x_{here}y_{out}$ (prior=2, prob=0.75)
- R8: $x_{here}y_{here}y_{here} \rightarrow x_{out}y_{here}$ (prior=1, prob=0.25)
- R9: $c_{here} \rightarrow \lambda$ (prior=1, prob=1)
- R10: $[c]_C \rightarrow [c]_C[c]_C$ (prior=2, prob=0.5)
- R11: $[]_C[]_C \rightarrow [[]_C]_C$ (prior=2, prob=0.3)
- R12: $[]_C \rightarrow \lambda$ (prior=2, prob=0.1)
- R13: $z[]_C \rightarrow [z]_C$ (prior=1, prob=0.5)

B:

- R14: $x_{here} \rightarrow x_{here}x_{here}y_{here}$ (prior=3, prob=0.25)
- R15: $y_{here} \rightarrow x_{here}y_{here}y_{here}$ (prior=3, prob=0.75)
- R16: $x_{here}x_{here}y_{here} \rightarrow x_{here}y_{out}$ (prior=1, prob=0.25)
- R17: $x_{here}y_{here}y_{here} \rightarrow x_{out}y_{here}$ (prior=2, prob=0.75)
- R18: $c_{here} \rightarrow \lambda$ (prior=1, prob=1)
- R19: $[c]_C \rightarrow [c]_C[c]_C$ (prior=2, prob=0.5)
- R20: $[]_C[]_C \rightarrow []_C$ (prior=1, prob=0.5)
- R21: $z[]_C \rightarrow [z]_C$ (prior=1, prob=0.5)

C:

- R22: $t \rightarrow t_{here}z_{here}$ (prior=1, prob=0.01)

Figura 3: diagrama del escenario Dos hábitats.

6.3.3. Nesting

El tercero de los escenarios de simulación juega con el anidamiento de membranas. En este escenario la membrana entorno es un simple contenedor, no tiene reglas asociadas. El papel que cumplía en escenarios anteriores es asumido por una membrana de tipo *Common area*. Es en ella donde se lleva a cabo el anidamiento progresivo de membranas *Individual*. Además, estas membranas *Individual* podrán ser absorbidas por una membrana *Absorption*. En esta membrana se diluyen las membranas *Individual*, de forma que por cada dilución se genera un objeto *d* que se expulsa como residuo. También cabe la posibilidad de que las membras *Individual* sean expulsadas antes de diluirse. Existe, además, en *Common area* una membrana *Generation* encargada de producir membranas *Individual*. La membrana *Witness* solo se usa como medio de crear membranas *Individual*.

Paralelamente, las membranas *Absorption* y *Generation* producen objetos *a*, *b* y *c* que, en *Common area* sufren una transformación cíclica.

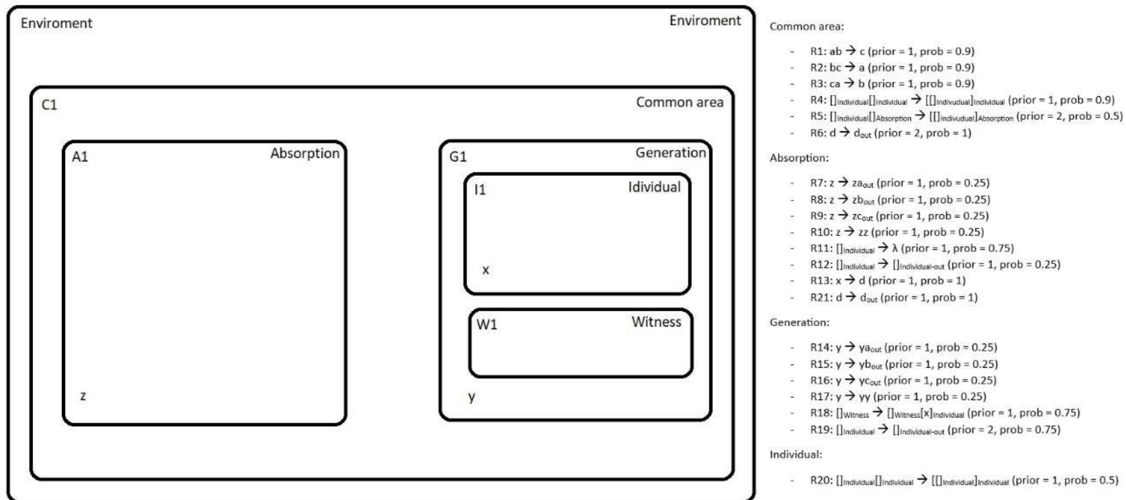


Figura 4: diagrama del escenario Nesting.

6.3.4. Anillo

En este escenario un tipo especial de membrana (*Passenger*) pasa del interior de unas membranas a otras en un ciclo que finaliza con la eliminación eventual de la membrana *Passenger*. Así, dependiendo de la membrana que la contenga en el momento de su disolución, su contenido evoluciona de forma diferente. De esta forma se puede saber cuántas membranas *Passenger* se han diluido en cada membrana del ciclo. Las membranas del ciclo pueden ser de tres tipos: *A, B* y *C*. Cada una de ellas contiene, también, una membrana *Seed* mediante la cual pueden generar nuevas membranas *Passenger*. Para terminar, el contenido de una membrana *Passenger* disuelta puede escapar de la membrana del ciclo antes de ser transformada. Si llega a la membrana Entorno antes de ser transformado, se convertirá en un objeto *u* (*unknown*), indicando que no se puede saber de qué membrana del ciclo proviene.

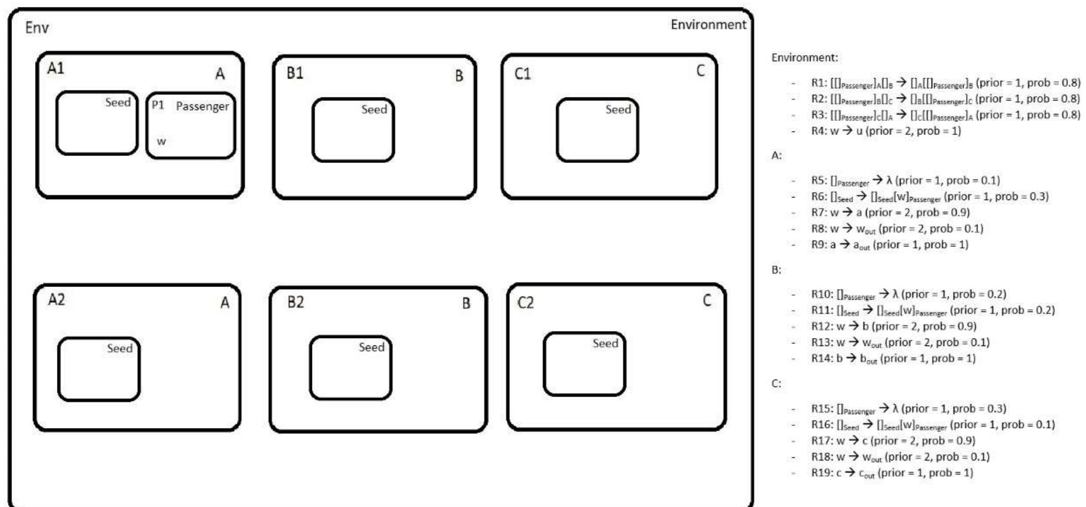


Figura 5: diagrama del escenario Anillo

7. Desarrollo de la solución

En las próximas líneas se describirá cómo se implementó el simulador de sistemas P que se ha descrito en apartados anteriores. Cabe destacar que la implementación se hizo en dos etapas claramente diferenciadas. Estas son, por orden cronológico, la implementación del motor de

simulación y la implementación de los algoritmos de elección de regla. Antes de empezar a implementar los algoritmos pareció conveniente tener un simulador plenamente funcional. El principal motivo fue que los errores de ejecución se localizarían únicamente en la implementación del algoritmo y podrían encontrarse fácilmente.

De esta forma, en su primera versión, el simulador disponía de un algoritmo de elección de regla rudimentario pero funcional. Este era una versión primitiva del algoritmo aleatorio. Entre sus limitaciones podemos destacar que no atendía a la propensión de las reglas (asumía que era en todos los casos 1). No obstante, era un algoritmo de elección exhaustivo, de forma que, si una regla podía ser aplicada, lo hacía. Esto era especialmente conveniente para el *testing*, fuera de este contexto carece de utilidad práctica.

Dividiremos la descripción de la implementación en tres grandes apartados: motor de simulación, escenarios de simulación e interfaz de usuario. En el primero se comenta cómo se implementó el simulador; en el segundo se describe el formato de creación de escenarios y en el último se apuntan unas instrucciones de uso del simulador. Todos los archivos fueron programados en lenguaje Python.

7.1. Motor de simulación

El motor de simulación se implementó en tres archivos Python: la librería *útil*, el archivo *Multiset*, la librería *algorithm* y el archivo *simulator*. En ellas se implementa el simulador mostrado en el siguiente diagrama:

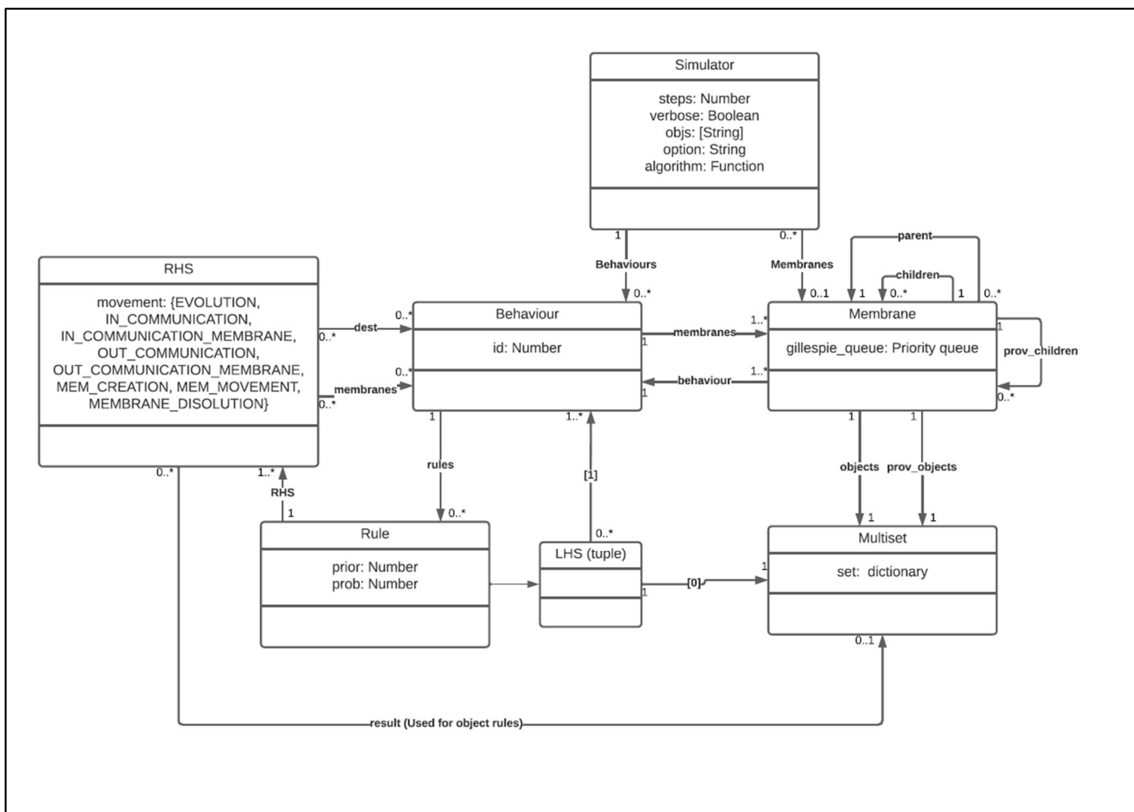


Figura 6: diagrama UML del simulador implementado

7.1.1. Librería *util*

Es un archivo que contiene la definición de las clases *Membrane*, *Behaviour*, *Rule* y *RHS*, denominado *util.py*.

En la clase *Membrane*, a parte de la implementación del diseño, se añadió una función para elegir una membrana hija con un comportamiento concreto de forma aleatoria. Se trata de la función *choose_child*.

En el caso de la clase *Rule*, se añadieron las funciones *objects_rule*, *times_appliable*, *compatible* y *remove_lhs*.

- La función *objects_rule* devuelve una respuesta afirmativa si ninguno de sus *RHS* involucra membranas. Queda exento el movimiento de comunicación hacia dentro.
- La función *times_appliable* devuelve el número de veces que podría aplicarse una regla en una membrana si no hubiese competencia.
- La función *remove_lhs* permite eliminar los objetos y membranas que serán usados en la aplicación de una regla.
- La función *compatible* devuelve una respuesta afirmativa si dos reglas no compiten por los elementos de su parte izquierda.

No se explicará con más detalle la implementación de las clases *Behaviour* y *RHS* puesto que se hizo una transposición directa de lo indicado en el diseño. La explicación de dicho proceso solo prolongaría el presente texto innecesariamente.

7.1.2. Archivo Multiset

Contiene la implementación de la clase *Multiset* para el manejo de multiconjuntos. Se ha implementado como un objeto que contiene un diccionario Python. Se implementaron funciones para obtener los objetos que contiene, en qué cantidades, añadir elementos y eliminarlos. También se implementaron las operaciones de unión (∪), intersección (∩), substracción (/), contención (⊆) e igualdad (≡) de conjuntos.

7.1.3. Librería *algorithm*

Esta librería contiene las funciones que implementan los algoritmos de elección de regla. Cabe destacar que, aunque la implementación de esta librería se hizo en una gran fase, esta se compuso de múltiples subfases. Cada una de dichas subfases son la implementación de un algoritmo y su prueba. Dichas funciones son *random, default*, *gillespie* y *k_gillespie*. Analicemos más a fondo estas funciones.

7.1.3.1. Función default

Para implementar el algoritmo por defecto de ARES (Campos, y otros, 2015) se generaron dos funciones auxiliares:

- La función *getBlock* devuelve el bloque más prioritario de las reglas de la membrana. Para ello hace un barrido de la lista de reglas del que extrae la regla más prioritaria y las que entran en conflicto con alguna de las reglas acumuladas para devolver.
- La función *refine_default* realiza el redondeo aludido en el paso final de la ejecución del algoritmo. Para ello, comprueba si la regla puede ejecutarse una vez más. Si es así genera un número aleatorio uniformemente distribuido en el intervalo (0,1). Si este número es menor que la parte decimal del número correspondiente a la regla en el vector de ejecución, dicha regla se ejecutará una vez más. En caso contrario se pasa a la siguiente regla. También elimina los elementos consumidos por la planificación de una membrana auxiliar.

La implementación, primero, realiza una criba de los casos en que la lista de reglas esté vacía o contenga una sola regla. Estos casos tienen solución inmediata. Después, adquiere una

naturaleza iterativa con el bucle más externo. Este bucle se ejecuta mientras quede alguna regla por analizar. Si es así realiza una ejecución del algoritmo con sus siete pasos. De esta forma, acumula los resultados de todas las ejecuciones que, finalmente, devuelve.

7.1.3.2. Función random

Implementa el algoritmo aleatorio de elección de regla. Para obtener los bloques de compatibilidad se usó la función auxiliar explicada en el apartado anterior. Para ordenar aleatoriamente estos bloques se usó la función *shuffle* de la librería *random* de Python.

7.1.3.3. Función gillespie

Esta función implementa el algoritmo de Gillespie. Antes de ejecutarlo realiza una criba en la lista de reglas, eliminando las que no pueden aplicarse. Después, si queda alguna regla activa, aplica el algoritmo de Gillespie. Si no puede aplicarse ninguna regla, se devuelve una respuesta nula en forma de tupla *(0,None)*.

El paso cuatro se realiza utilizando un bucle en el que se acumulan los valores de las propensiones de las reglas. Cuando estas superan $a_2 * p_0$ se sale del bucle, de forma que el último índice antes de rebasar el umbral es la solución.

El resultado de este algoritmo se almacena en la cola de prioridad de la membrana. Así, la prioridad será la suma del paso actual y τ, y el valor, la propia regla. En cada paso se aplican las reglas de la cola cuya prioridad en la cola sea igual o inferior al número de paso actual. La prioridad en la cola puede ser un número decimal.

En la ejecución de las reglas obtenidas con este algoritmo se ha habilitado la opción maximal o minimal. En modo maximal se ejecuta la regla tantas veces como sea posible. En el modo minimal la regla se ejecuta una sola vez.

7.1.3.4. Función k_gillespie

Es una generalización el algoritmo de Gillespie en la que se permite elegir más de una regla para aplicar. Esta función también utiliza la cola de prioridad aludida en la función *gillespie*. La cola de prioridad de cada membrana contendrá como máximo k reglas. En este caso es importante aclarar que la cola puede contener varias veces la misma regla, con distinta prioridad.

Primero se consulta la cola de prioridad de la membrana. Se aplicarán todas las reglas cuya prioridad en la cola sea menor o igual que el paso actual. Si se está ejecutando en modo minimal las reglas se aplican una vez por cada vez que se desapilen. En caso de que se esté ejecutando en modo maximal las reglas desapiladas tantas veces como sea posible. Si se desapila varias veces la misma regla en modo maximal, el efecto será el mismo que si sólo se hubiera desapilado una vez

Una vez aplicadas las reglas pertinentes, se calculará el número de "plazas libres" que quedan en la cola. Es decir, la cola puede contener como máximo k elecciones de regla y, tras haber ejecutado algunas (llamemos *n* a esta cantidad), puede haber espacio para almacenar nuevas reglas. Así, se elegirán $z = k - n$ reglas utilizando la siguiente variación del algoritmo de Gillespie:

1. Se eliminan las reglas que no puedan ser ejecutadas en el paso actual. El resto son llamadas reglas activas.
2. Se calcula el valor p_0 como la suma de las propensiones de las reglas activas
3. Se generan los números a_1 y a_2 aleatoriamente en el intervalo (0,1).

4. Se calcula el valor τ de la misma forma que en el algoritmo básico de Gillespie.
5. Se elige una regla de la misma forma que en el algoritmo básico de Gillespie.
6. Se introduce la regla escogida en la cola de forma que su prioridad en la cola sea igual a la suma del paso de ejecución actual más τ.
7. Se eliminan las reglas que no puedan ser ejecutadas en el paso actual teniendo en cuenta las reglas que han elegido aplicarse. Dependiendo de si estamos usando el modo maximal o minimal la última regla elegida puede eliminarse de las reglas activas o no.
8. Si todavía no se han elegido *z* reglas y sigue habiendo reglas activas se vuelve al paso 2. En caso contrario, se termina el algoritmo.

7.1.4. Archivo simulator

En este archivo se implementa el motor de simulación en sí mismo. Contiene las funciones de creación, *step* (para resolver un paso de ejecución) y funciones para visualizar el estado del sistema, además de funciones auxiliares.

El constructor de la clase toma un objeto que contiene el escenario de simulación directamente leído del archivo. Primero, se recorre la lista de comportamientos incorporándolos al nuevo objeto *Simulator*, incluyendo también todas las reglas que los definen. Después se recorre la estructura de membranas anidadas con los objetos que contienen y se hace lo propio. De esta forma, al terminar, el atributo *Membranes* apuntará a la membrana entorno. Es también en esta función dónde se indica qué algoritmo de selección de reglas va a utilizarse. No obstante, existe otra función para cambiar esta opción fuera del constructor.

Al aplicar un paso se hace un barrido sobre los comportamientos, y en cada comportamiento, de las membranas asociadas a ellos. En cada una, primero se llama al algoritmo de selección de regla, y después se aplican las instrucciones recibidas. Para este fin, se construyó una función *apply* que aplica una regla las veces indicadas sobre una membrana. Existe una función para aplicar cada tipo de parte derecha. Los resultados de todas ellas se guardan en el atributo *prob_objects* o *prov_membranes* de la membrana destino, a excepción de la disolución de membrana. En esta última sólo se elimina dicha membrana de la lista de hijas.

En este conjunto de funciones merece especial atención la aplicación del movimiento de membranas. En ella, se lleva a cabo un proceso de inferencia de la membrana origen. Primero, se filtran todas las membranas hijas de la actual que tienen un comportamiento señalado en la parte izquierda de la regla y se elimina la membrana destino de las posibles candidatas. Después se analizan las membranas hijas de las candidatas para saber cuál es la que contiene las membranas que se desean mover. Así, la membrana que contenga hijas de esos tipos será designada como membrana origen. Llevar a cabo el movimiento es trivial, simplemente se eliminan de la lista de hijas de la membrana origen y se añaden a la lista provisional de hijas de la membrana destino.

Hay además dos funciones para analizar el estado del sistema P simulado. Estas son *output* y *scann*. La primera devuelve un *Multiset* con los objetos activos en el sistema. También permite incluir los objetos contenidos en algunos tipos específicos de membrana. La segunda obtiene una representación en forma de texto de la estructura y el contenido de las membranas:

```
-Environment
 {}
  -Common area
   {}
    -Absorption
     {'z': 1}
    -Generation
     {'y': 1}
      -Witness
       {}
      -Individual
       {'x': 1}
```

Figura 7: resultado de aplicar la función scann sobre el escenario Nesting

El objeto *Simulator* tiene además una forma de ejecución verbosa que dará más información sobre la simulación.

7.2. Escenarios de simulación

Para ajustarse a estándares europeos, se decidió que los escenarios de simulación se especificarían en YAML. Así, el escenario tendría estructura de diccionario con tres elementos: uno de clave *structure*, que contendrá la estructura de membranas y objetos, otro de clave *objs* que almacena el alfabeto de objetos, y un último de clave *types*, que contiene la lista de comportamientos.

De esta forma, el valor de la clave *structure* será una lista anidada de la forma:

$$membrana = (contenido, IdComportamiento)$$

$$contenido = [elemento, \dots, elemento]$$

$$elemento = (objeto \mid membrana, multiplicidad)$$

Donde idComportamiento es el identificador del comportamiento que tendrá la membrana, objeto es una cadena de texto y multiplicidad es un número natural o cero.

El valor de la clave *objs* contiene una lista con las cadenas de texto que identifican los objetos:

$$alfabeto = [objeto, \dots, objeto]$$

Por último, el valor de la clave *types* será otra lista anidada que siga la estructura:

$$tipo, tipo \mid \lambda$$

$$tipo = [IdComportamiento, regla, \dots, regla]$$

$$regla = [prioridad, propensión, LHS, RHS, \dots, RHS]$$

$$LHS = ([(IdComportamiento, multiplicidad), \dots], [(objeto, multiplicidad), \dots])$$

$$RHS = \begin{cases} [0,(objeto,multiplicidad),\dots,(objeto,multiplicidad)]\ Evolución \\ [1,IdComportamiento,(objeto,multiplicidad),\dots,(objeto,multiplicidad)]\ Comunicacion\ dentro\ obj. \\ [2,IdComportamiento,IdComportamiento,\dots,IdComportamiento]\ Comunicación\ dentro\ mem. \\ [3,(objet\ \ ,multiplicidad),\dots,(objeto,multiplicidad)]\ Comunicación\ fuera\ obj. \\ [4,IdComportamiento,\dots,IdComportamiento]\ Comunicación\ fuera\ mem. \\ [5,IdComportamiento,(objeto,multplicidad),\dots,(objeto,multiplicidad)]\ Creación\ de\ membrana \\ [6,IdComportamiento,IdComportamiento,IdComportamiento]\ Movimiento\ de\ membranas \\ [7,IdComportamiento]\ Disolución\ de\ membrana \end{cases}$$

Así, cada tipo tiene asociadas una serie de reglas. Cada regla tiene su prioridad ($\mathbb{N} > 0$), propensión (en el intervalo [0,1]), parte izquierda (LHS) y partes derechas (RHS). Una regla será más prioritaria cuanto más cercana sea a 0 y más propensa cuanto más cercana esté a 1. La parte izquierda es una tupla en la que el primer elemento es una lista de comportamientos (para aludir membranas) y el segundo una lista de objetos. Finalmente, se pueden indicar tantas partes derechas en una regla como "movimientos" distintos quieran hacerse. Cada parte derecha es una lista cuyo primer valor es un natural en el intervalo [0,7] que identifica el tipo de "movimiento". El caso 0 es para las reglas de evolución, en el que, además, se incluirán los objetos, resultado de la aplicación de la regla. El caso 1 está reservado para la comunicación hacia dentro de objetos; así se indica el comportamiento de la membrana destino y el tipo y cantidad de objetos a ser absorbidos. El caso 2 hace lo propio con membranas, dejando claro que el primer comportamiento es el de la membrana destino. El caso 3 describe una expulsión de los objetos indicados, así como el caso 4 lo hace con las membranas de los comportamientos señalados. El caso 5 se reserva a la creación de membranas de comportamiento igual al segundo elemento de la lista, y que contendrá los objetos descritos en las posiciones restantes. El caso 6 se utiliza para codificar movimientos de membranas. Como se dijo en puntos anteriores la membrana origen se infiere, por lo que no es necesario indicarla en la parte derecha. Así el primer *IdComportamiento* es el de la membrana destino y los demás se refieren a las membranas que se moverán. Finalmente, el caso 7 expresa la disolución de la membrana con comportamiento igual al de la segunda posición.

7.3. Interfaz de usuario

Se ha implementado una interfaz por línea de comandos que contempla la ejecución en el terminal de dos archivos: *stepbystep.py* y *simulate.py*. Ambos están programados en Python.

El archivo *stetpbystep.py* ejecuta un paso de simulación y después pide aprobación para ejecutar el siguiente. Esta aprobación se realiza escribiendo *y* o *yes*. Seguirá en este bucle hasta que escribamos otra cosa o nada. Después de ejecutar cada paso imprime por pantalla el resultado de la función *scann* de *Simulator* explicada anteriormente. Pueden indicársele varias opciones además de la ruta del escenario de simulación, que es obligatoria. Estas opciones son el algoritmo que utilizar (precediéndolo de *-a*) y el habilitado de la función verbosa de *Simulator* (añadiendo al comando la cadena *-v*). Si se está ejecutando una variante de Gillespie, se usará la versión máximamente paralela por defecto. Si se quiere usar la variante mínimamente paralela ha de indicarse explícitamente agregando la cadena -mp.

El archivo *simulate* permite simular muchos pasos en una sola llamada. Cuando se complete la simulación de dichos pasos se pedirá confirmación para seguir igual que con el archivo anterior. Sin embargo, con este archivo se simulará nuevamente ese número de pasos de una vez. Al final de la ejecución también se muestra el tiempo invertido en ella. Nuevamente es obligatorio indicar la ruta al archivo que contiene el escenario de simulación. También las opciones del algoritmo y verbosa están disponibles. Además, se necesita la introducción del número de pasos a simular. Finalmente se puede introducir la ruta a un archivo dónde se guardará un registro de los objetos que contiene el sistema en cada paso de ejecución en formato *csv*. Se puede indicar que se vuelque en ese archivo un conteo de los objetos que contienen algunos tipos de membrana.

8. Pruebas

El testing se realizó en dos fases distintas: prueba del motor de simulación y prueba de la implementación de los algoritmos. La causa de esta diferenciación se expuso anteriormente y

consiste en la conveniencia de tener un simulador funcional antes de probar la implementación de los algoritmos.

8.1. Prueba del motor de simulación

Se usó la versión primitiva del algoritmo aleatorio explicad en el punto anterior, Desarrollo de la solución, ya que era sumamente útil para el *testing*.

Como banco de pruebas se utilizaron dos escenarios de simulación enfocados a mostrar todos los errores que pudieran ocultarse en la implementación. En ellos se intenta utilizar todos los de partes derecha en las situaciones más diversas posible. Estos escenarios no se utilizarán en la experimentación enfocada a comparar algoritmos, ya que carecen de interés más allá del propio *testing*.

El primer escenario "de prueba" es una primera criba que trata de sacar a relucir errores en reglas de objetos. También juega con el movimiento de membranas. Fue de gran utilidad para detectar fallos tanto en la lectura del escenario de simulación como en la aplicación reglas de objetos. Es el descrito por el siguiente diagrama:

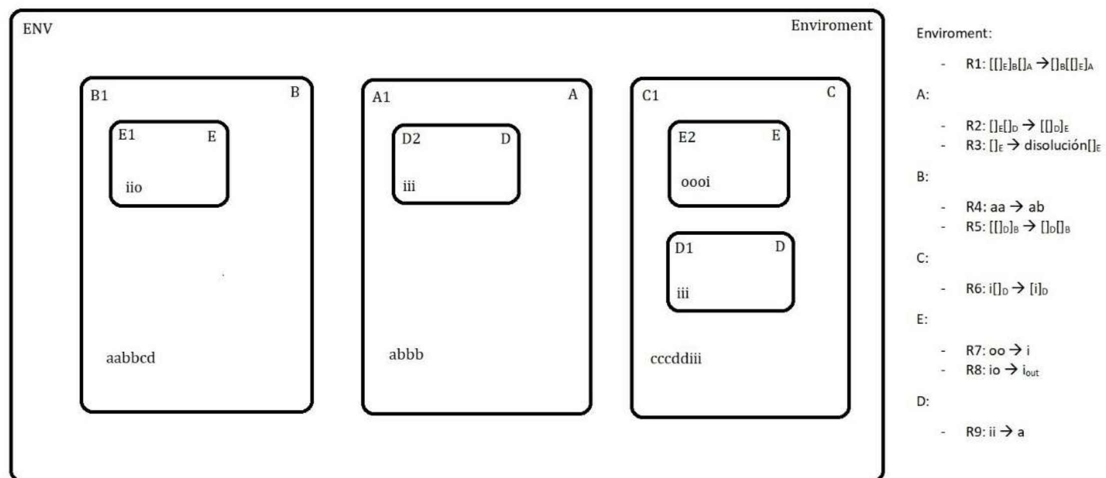


Figura 8: escenario de prueba básico

El segundo escenario "de prueba" se experimenta con la comunicación entre membranas, el anidamiento y la expulsión de objetos. También se observaron los errores de lectura de este tipo de reglas del escenario de simulación, junto con algunos fallos de compatibilidad entre partes derecha de distintas reglas. Es el mostrado a continuación:

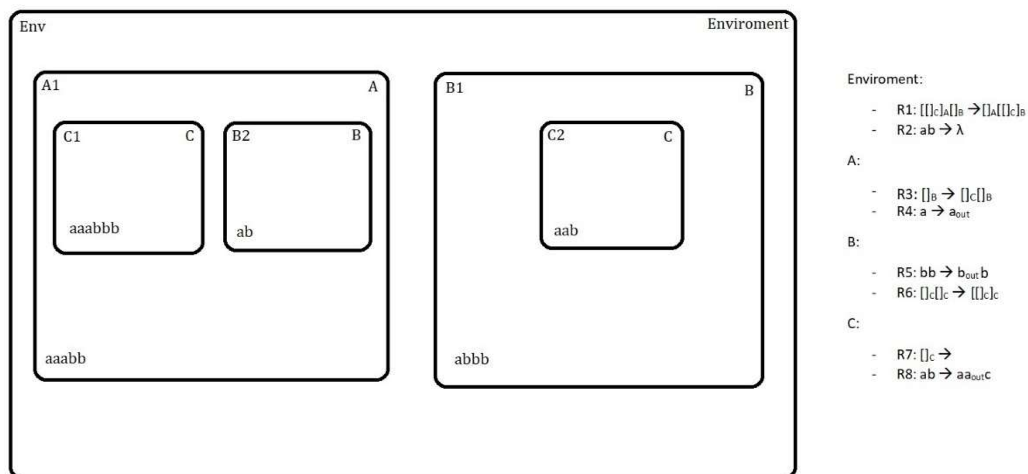


Figura 9: escenario de prueba medio

Cuando el simulador fue capaz de simular correctamente estos escenarios se comenzó con la implementación de los algoritmos.

8.2. Prueba de la implementación de los algoritmos

Como ya se dijo anteriormente, esta fase debe verse como la unión de las pruebas de cada algoritmo, no como un periodo de pruebas. Después de implementar cada algoritmo se probó su ejecución con todos los escenarios disponibles. Esto incluye tanto los dos escenarios de prueba como a los escenarios de simulación creados para comparar los algoritmos.

9. Experimentación

Se ha ejecutado el simulador en todos los escenarios con los algoritmos ARES, aleatorio, Gillespie, 3-Gillespie y 5-Gillespie. Estos tres últimos fueron ejecutados en sus versiones maximal y minimal. También se probó el algoritmo 50-Gillespie, únicamente en su versión mínimamente paralela.

Para las combinaciones algoritmo-escenario más variables se realizaron 30 simulaciones, y para las más simples solo 10 con una duración acorde a cada combinación. Con esos datos se obtuvieron las ejecuciones "modelo" haciendo la media de objetos en cada paso, aludido en las gráficas con el nombre del objeto. También se obtuvo la desviación típica por paso, aludida como *Desv* seguido del nombre del objeto en forma de diagrama de áreas apiladas. Así se construyeron las gráficas expuestas a continuación.

9.1. Escenario *Chaining*

Recordemos que en este escenario se producen objetos *x* en la membrana *A* y objetos *y* en la membrana *B*. Estos objetos salen a la membrana entorno y de ahí pasan a la membrana *C*. En esta última membrana se combinan para formar objetos *z* que vuelven a salir a la membrana entorno. Estos vuelven a la membrana *A* o la *B* y se disuelven junto con los objetos allí producidos.

Con algunos algoritmos se llega a un punto en el que la ejecución se estabiliza sin que se produzcan más objetos de ningún tipo. Esto ocurre porque se expulsan todos los objetos que hacen de semilla en una membrana de producción.

9.1.1. ARES

El algoritmo de ARES (Campos, y otros, 2015) resultó en un aumento lineal de objetos x (en un 60% de las ocasiones) o de objetos z (en un 40% de las ocasiones). En ambos casos, la desviación típica es prácticamente nula. La cantidad de objetos *y* no parece incrementarse.

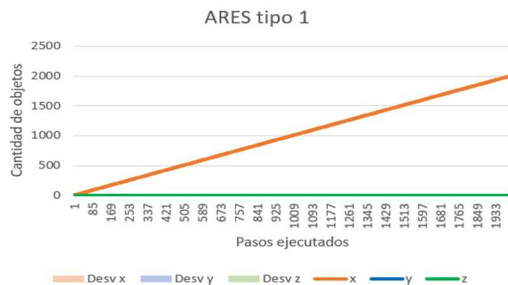


Figura 10: ejecución tipo 1 de ARES en Chaining

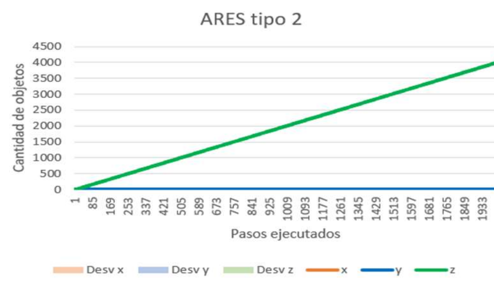


Figura 11: ejecución tipo 2 de ARES en Chaining

9.1.2. Aleatorio

El algoritmo aleatorio también muestra varios tipos de ejecuciones. En todos ellos la cantidad de objetos evoluciona en unos primeros pasos (habitualmente hasta 10 pasos), y después la evolución se paraliza. El caso más habitual es el uno (61%), seguido del dos (29%) y el caso tres que es muy poco frecuente (11%).

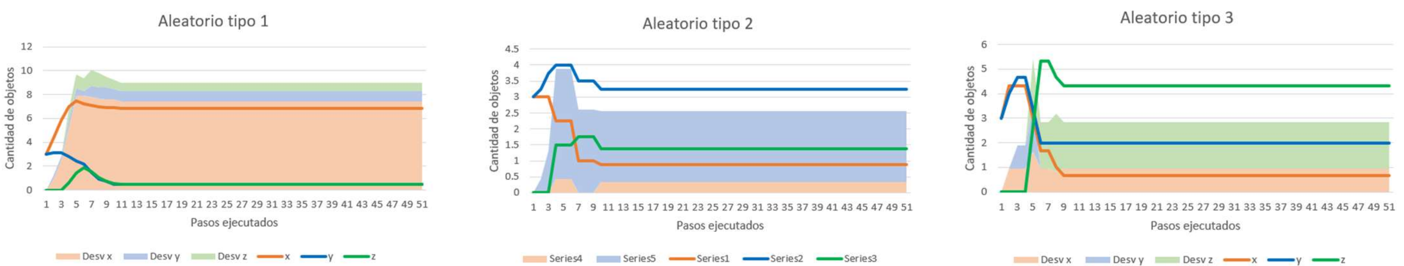
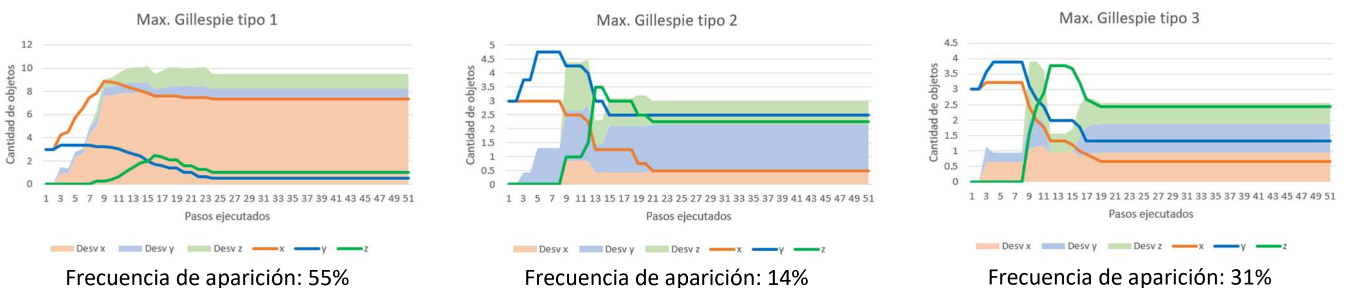


Figura 12: ejecuciones típicas del algoritmo Aleatorio en Chainig

9.1.3. Gillespie maximal

Las simulaciones con las variaciones de Gillespie en modo maximal tienen un comportamiento similar a las del algoritmo aleatorio. Suelen mostrar tres patrones de simulación en función de las cantidades en las que se estabilizan: el tipo uno con más *x*, el tipo dos con más *y* y el tipo tres con más *z*. En todas las variaciones el tipo uno es el más frecuente, aunque se puede apreciar que con el aumento de la k los diferentes patrones tienden a ser equiprobables.



Frecuencia de aparición: 55% Frecuencia de aparición: 14% Frecuencia de aparición: 31%

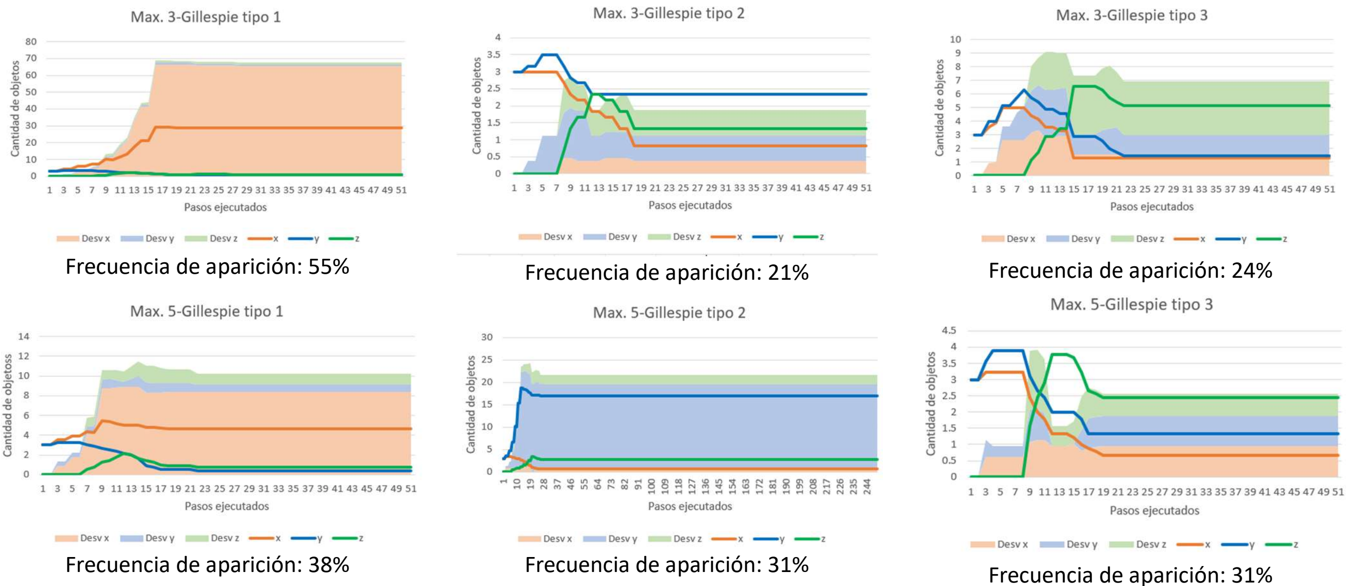
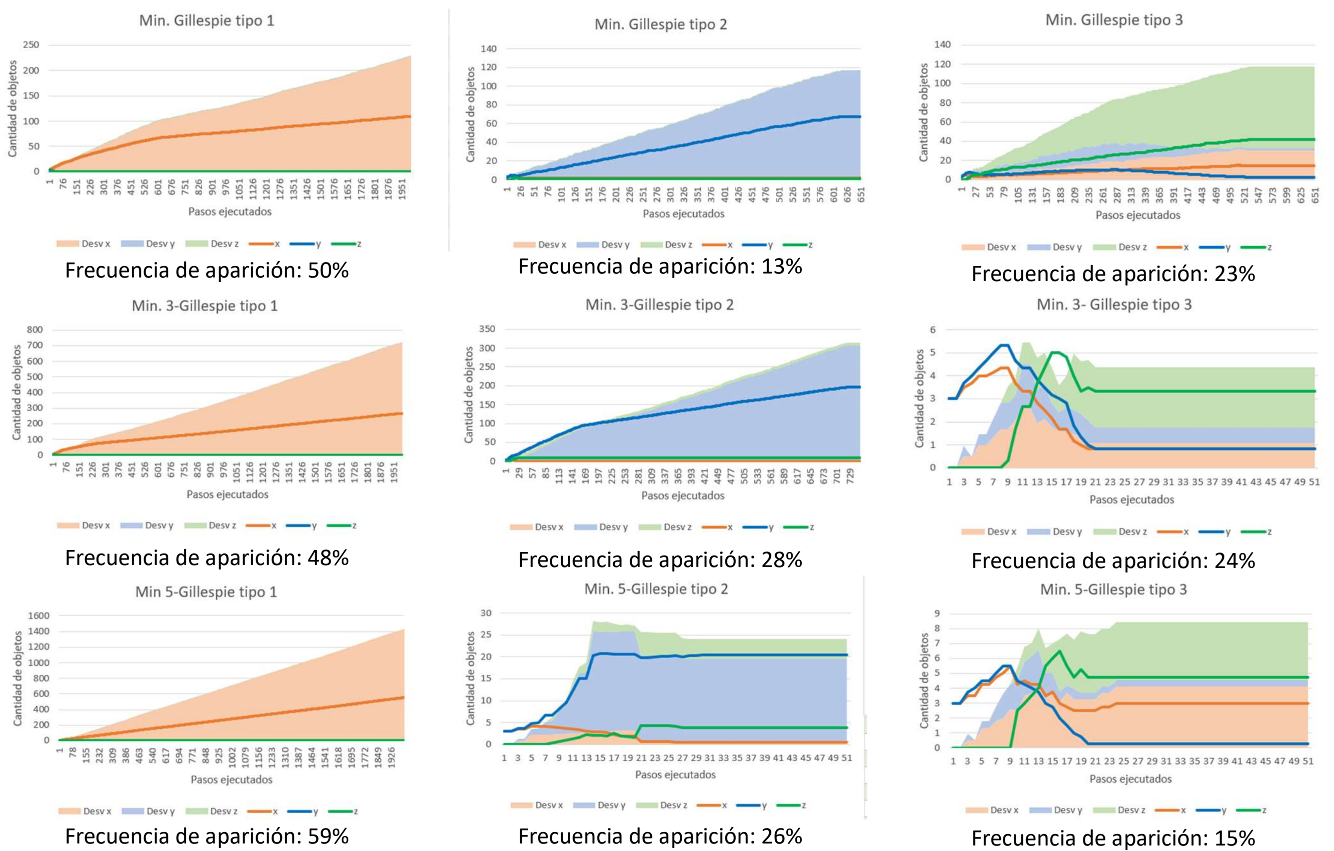


Figura 13: ejecuciones típicas de las variaciones maximales del algoritmo de Gillespie en Chaining

9.1.4. Gillespie minimal

Las ejecuciones de las versiones del algoritmo de Gillespie en su versión minimal muestran los tres mismos patrones que hemos tratado en la sección anterior. Sin embargo, puede observarse que, excepto en el tipo 3, la aproximación a la estabilización es más larga y estable. Son de especial interés las ejecuciones tipo uno de Gillespie, 3-Gillespie y 5-Gillespie. En ellas, hay ciertas simulaciones que no llegan a un punto de estabilización. Esas simulaciones (pertenecientes al tipo uno) tienen un comportamiento parecido al mostrado por el tipo uno de ARES.



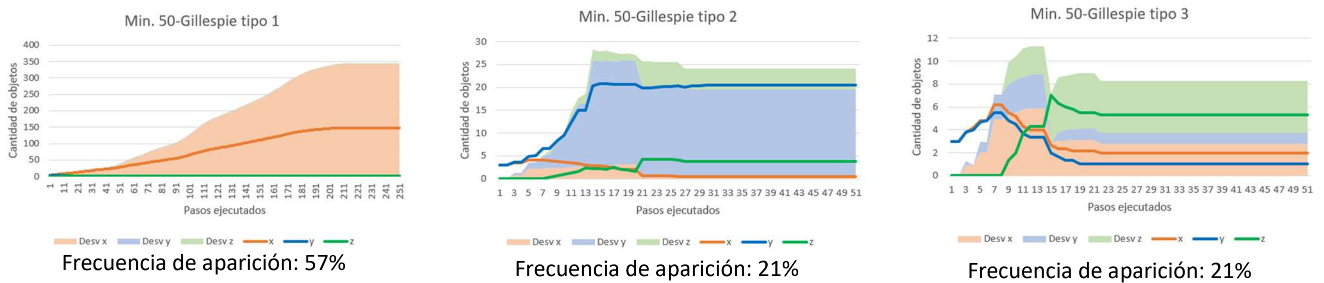


Figura 14: ejecuciones típicas de las versiones minimales del algoritmo de Gillespie en Chaining

El algoritmo de Gillespie minimal presenta un cuarto patrón importante que ha aparecido en el 13% de las simulaciones. En él, la simulación se estabiliza con la misma cantidad de *x* y de *y*, que es mayor que la cantidad de *z*. Conforme se aumenta la k, este patrón pasa a ser anecdótico.

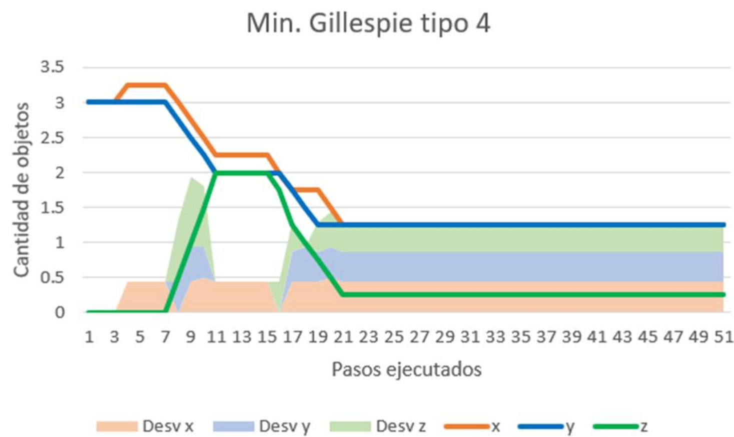


Figura 15: ejecución tipo 4 del algoritmo minimal de Gillespie en Chaining

9.2. Escenario Dos Hábitats

En este escenario los objetos *x* e *y* se producen y eliminan con diferente probabilidad en función de la membrana en la que estén contenidos. Los objetos *t* actúan de semillas y no se producen ni se eliminan. Si alguna *t* está contenida en una membrana *C*, se puede producir un objeto *z* con baja probabilidad. Cada membrana *C* contiene un objeto *c*. Cuando esta se disuelve en una membrana *A* o *B* el objeto c se elimina.

Para su estudio se empleó un análisis más profundo del estado del simulador en cada paso de ejecución. Así, se registró la cantidad total de objetos de cada tipo en el simulador y las cantidades de objetos inmediatamente contenidos en las membranas *A* y *B*. Estos objetos se referencian en las gráficas como por ejemplo *A.x* se refiere a la cantidad de objetos *x* contenidos en una membrana *A*.

9.2.1. ARES

La simulación del escenario Dos Hábitats con el algoritmo ARES (Campos, y otros, 2015), comporta un crecimiento exponencial de objetos *x* e *y* en la misma medida con una tasa similar en las membranas *A* y *B*. Por otra parte, no se ha registrado ninguna creación de un objeto *z*. Podemos ver que se crean y eliminan membranas *C* de forma exponencial tanto en las membranas *A* como en las *B*. Sin embargo, la tasa de eliminaciones en las membranas *B* es mucho mayor. Las desviaciones típicas de los objetos *x* e *y* parecen ser similares en todas las membranas.

Figura 16: ejecución típica del algoritmo ARES en Dos Hábitats

9.2.2. Aleatorio

El algoritmo aleatorio provoca también un crecimiento exponencial de objetos *x* e *y*. La tasa de creación de objetos x es mayor en las membranas *A*. Las *y* se crean en las membranas *B* en una cantidad ligeramente mayor. Con este algoritmo tampoco se ha registrado ninguna creación de objetos *z*. También podemos observar que se crean membranas *C* con una tasa exponencial. Las disoluciones de membranas *C* son menos habituales que con el algoritmo de ARES (Campos, y otros, 2015), como puede verse en la baja cantidad objetos *c* en las membranas *A* y *B*.

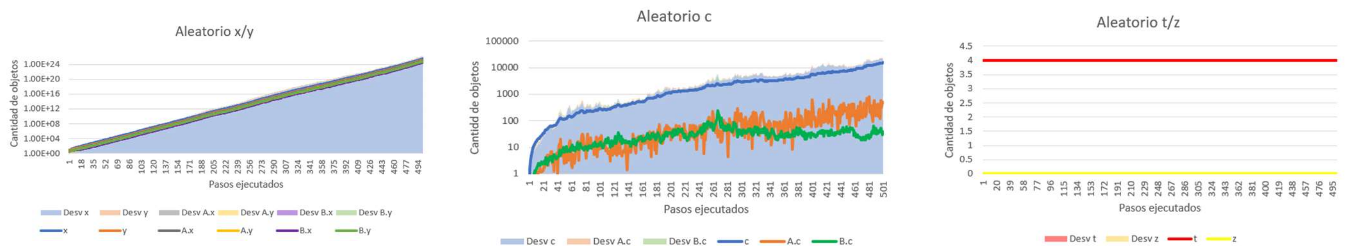
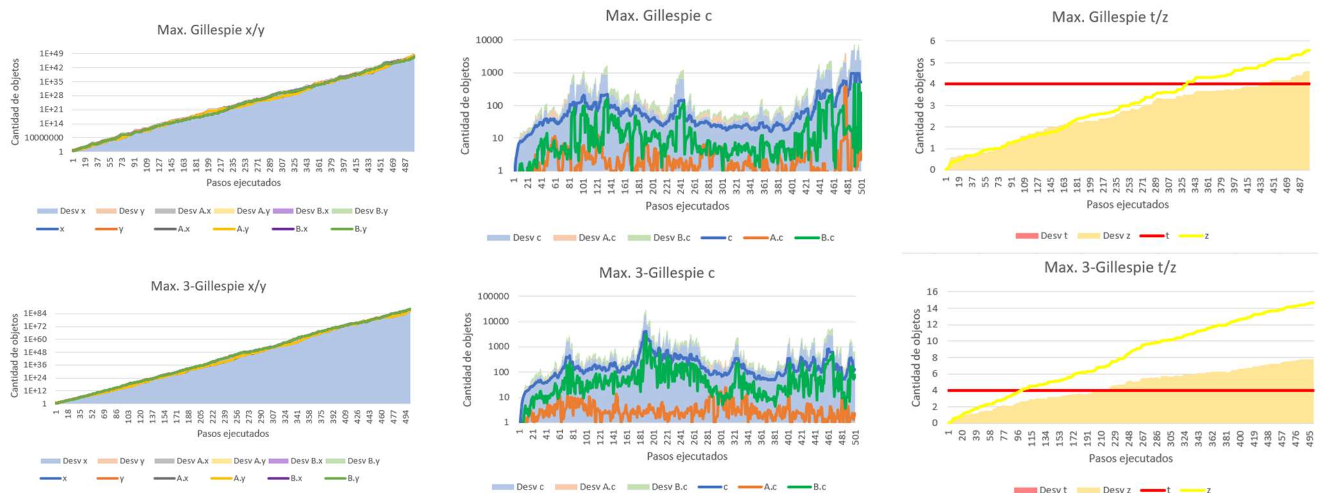


Figura 17: ejecución típica del algoritmo aleatorio en Dos Hábitats

9.2.3. Gillespie maximal

Las variaciones maximales del algoritmo de Gillespie muestran un crecimiento exponencial de objetos *x* e *y*. La producción de estos objetos se centra, en su mayoría en las membranas *B*, aunque se dan etapas en las que esta producción es mayor en las membranas *A*. Con este algoritmo si se crean objetos *z*, cuya tasa de producción aumenta con una mayor *K*. Respecto de las membranas *C*, se suceden etapas de creación con otras de disolución en masa. Las disoluciones suelen desarrollarse principalmente en el interior de las membranas *B*. La cantidad de este tipo de membranas es, por tanto, bastante inestable con este tipo de algoritmos.



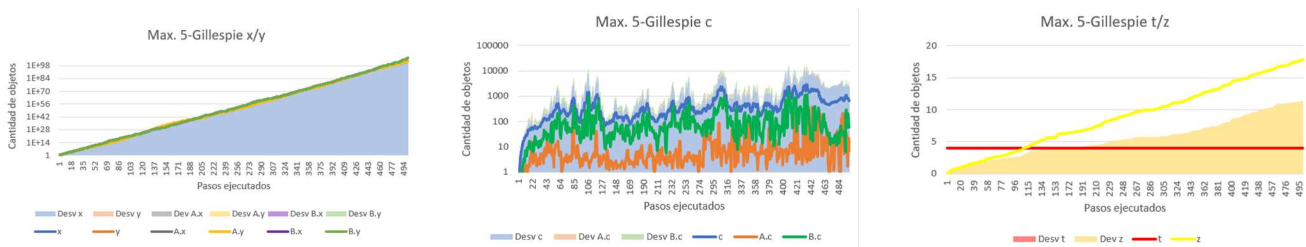


Figura 18: ejecuciones típicas de las variaciones maximales del algoritmo de Gillespie en Dos Hábitats

9.2.4. Gillespie minimal

Las variantes minimales del algoritmo de Gillespie tienden a un crecimiento lineal de todos los objetos. Es interesante apreciar como al aumentar la k, también se incrementa la cantidad de objetos *x* e *y* en la membrana entorno. Esto puede observarse en la diferencia entre la cantidad global de objetos x e y y las contenidas en membranas A o B. Ese aumento de la k también parece dar una mayor consistencia entre ejecuciones. Dicho incremento parece favorecer la producción de objetos *z*, pero puede apreciarse como con el 5-Gillespie alcanza un techo que no se supera con el algoritmo 50-Gillespie. Las disoluciones de membranas *C* parecen hacerse más frecuentes con una k mayor, y ocurren principalmente en las membranas *B*. Así, se pasa de un crecimiento lineal en el caso del Gillespie básico a una cantidad sostenida de membranas con el algoritmo 50-Gillespie

Figura 19: ejecuciones típicas de las versiones minimales del algoritmo de Gillespie

9.3. Escenario *Nesting*

El análisis de este escenario es algo más complejo debido a la variedad de objetos que pueden producirse. Así, por una parte, tenemos los objetos asociados a la creación y disolución de membranas y, por otra, los asociados con el ciclo de objetos en la membrana *Common Area*. Analizaremos, por tanto, las simulaciones en dos apartados diferentes, dedicados a cada uno de estos dos grupos.

9.3.1. Creación y disolución de membranas

Cada membrana *Individual* contiene un objeto *x* durante toda su existencia. Estos objetos x solo cambian cuando son liberados en la membrana *Absorption*, tras la disolución de la membrana *Individual* que los contenía. Entonces, se transforman en objetos *d*, denotando que ha ocurrido una disolución.

9.3.1.1. ARES

El algoritmo de ARES (Campos, y otros, 2015) tiende a crear membranas de forma lineal (con objetos *x* en su interior), mientras que las disoluciones son prácticamente nulas. Las desviaciones típicas son también muy cercanas a 0. Puede apreciarse este comportamiento en la figura 20.

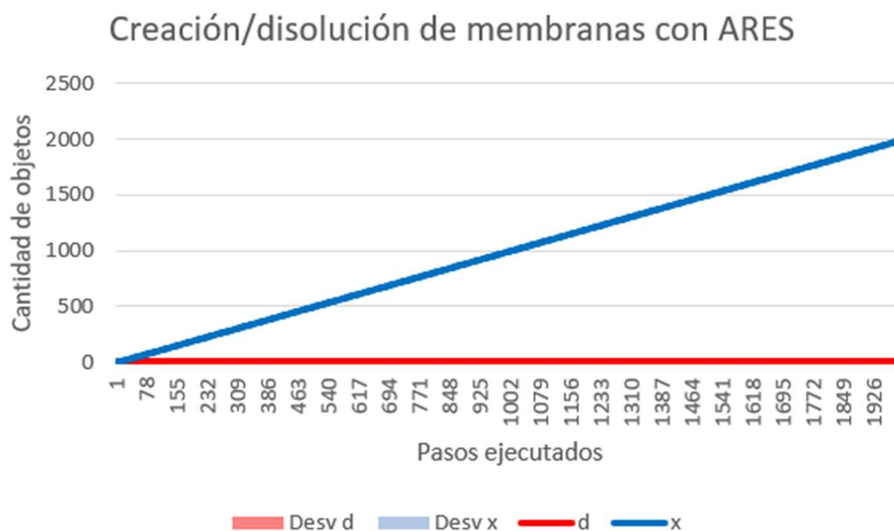


Figura 20: ejecución típica del algoritmo de ARES en Nesting

9.3.1.2. Aleatorio

Con el algoritmo aleatorio, por el contrario, las disoluciones son mucho más numerosas. Se siguen creando membranas de forma lineal, pero en este caso, las membranas creadas son eliminadas rápidamente. Nuevamente las desviaciones típicas son muy cercanas a 0.

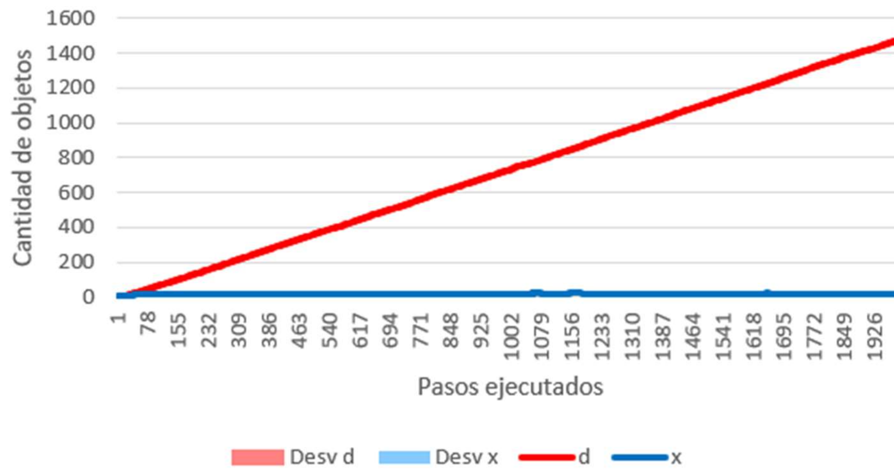


Figura 21: ejecución típica del algoritmo aleatorio en Nesting

9.3.1.3. Gillespie maximal

Las variaciones maximales de Gillespie han mostrado un comportamiento parecido al algoritmo aleatorio. Puede observarse que a medida que se aumenta la k, las ejecuciones son más parecidas entre sí.

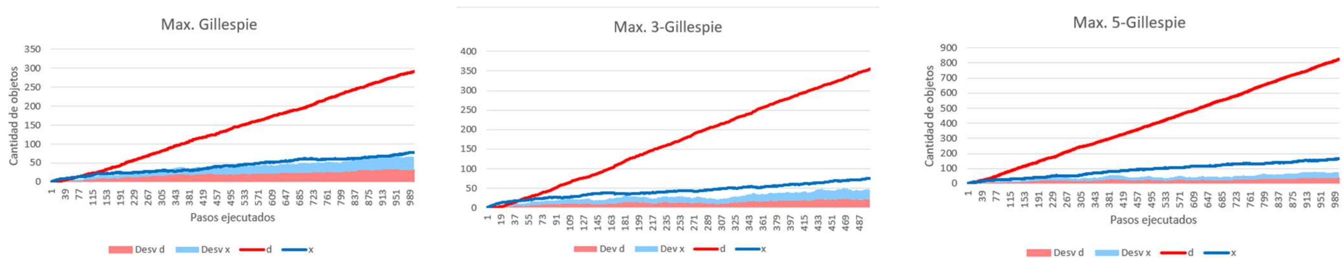
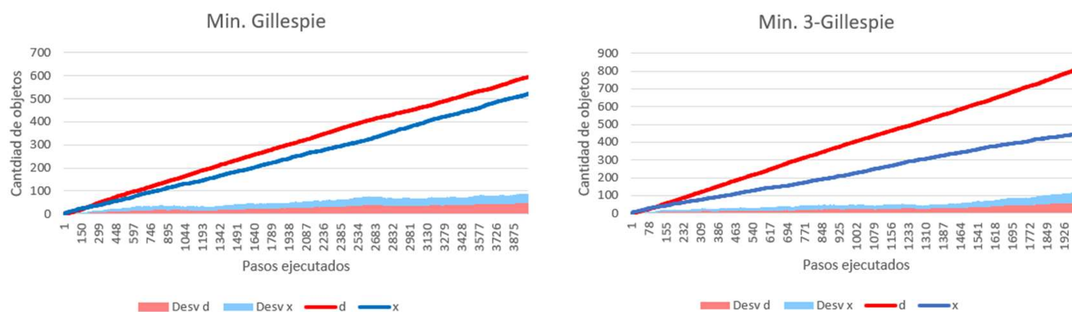


Figura 22: ejecuciones típicas de las variaciones maximales del algoritmo de Gillespie en Nesting

9.3.1.4. Gillespie minimal

Estos las ejecuciones de estos algoritmos muestran un crecimiento lineal de los objetos *x* y *d*. Sin embargo, puede observarse como, a medida que se aumenta la k, las membranas se eliminan más rápidamente.



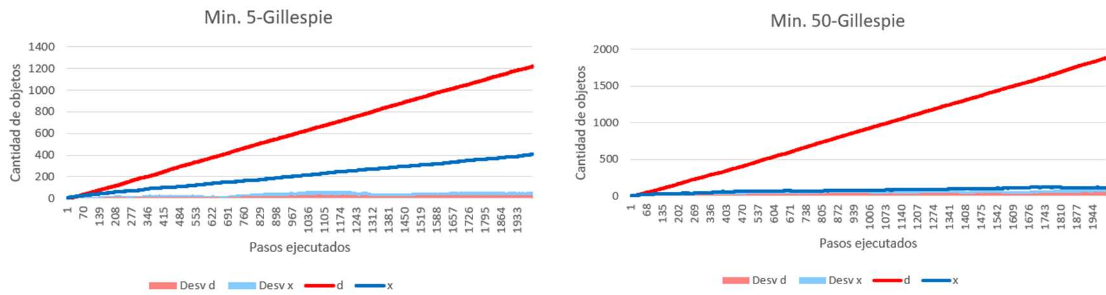


Figura 23: ejecuciones típicas de las variaciones minimales del algoritmo de Gillespie en Nesting

9.3.2. Ciclo de objetos

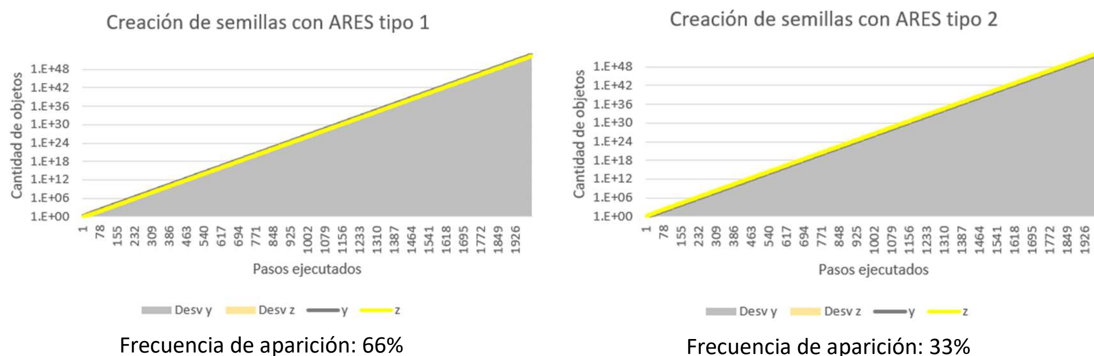
En las membranas *Absorption* y *Generation* hay objetos y *y z* que actúan de semillas. Pueden generar más objetos de su clase u objetos *a*, *b* o *c*. Estos últimos salen a la membrana *Common area* donde se combinan por pares de diferente tipo para generar el tercero.

9.3.2.1. ARES

La cantidad de objetos *a*, *b* y *c* conseguida con el algoritmo de ARES (Campos, y otros, 2015) tiene un comportamiento exponencial. Las cantidades de los tres tipos de objetos son sumamente parecidas a lo largo de toda la ejecución.

Figura 24: ejecución típica del algoritmo de ARES en Nesting

Respecto a la evolución de las semillas, ARES (Campos, y otros, 2015) muestra dos evoluciones diferentes. En ambas el crecimiento es exponencial, pero en el tipo uno la cantidad de objetos *y* es significativamente mayor. En el tipo dos ocurre lo contrario.



Frecuencia de aparición: 66%

Frecuencia de aparición: 33%

Figura 25: ejecución típica del algoritmo de ARES en Nesting

9.3.2.2. Aleatorio

El algoritmo aleatorio tiene un comportamiento parecido, aunque con un crecimiento más rápido e inestable:

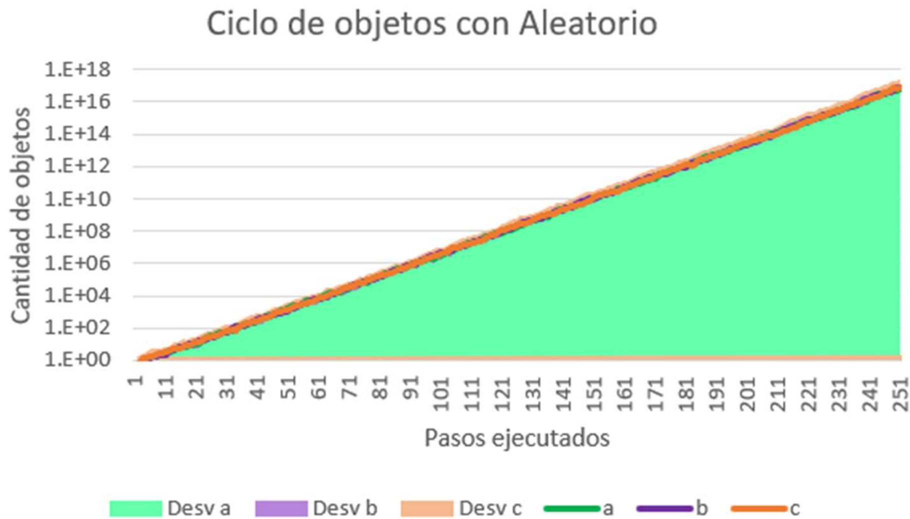


Figura 26: ejecución típica del algoritmo aleatorio en Nesting

Este algoritmo tiene los mismos dos comportamientos que mostraba el ARES (Campos, y otros, 2015) a la hora de crear semillas.

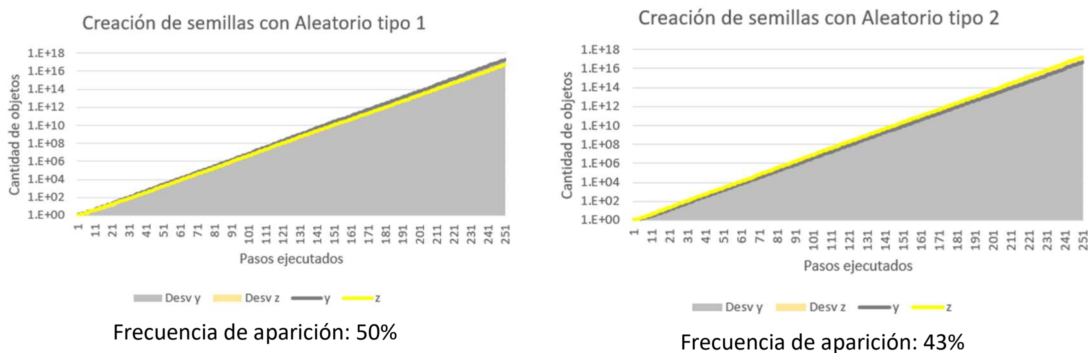


Figura 27: ejecución típica del algoritmo aleatorio en Nesting

9.3.2.3. Gillespie maximal

Las versiones maximales del algoritmo de Gillespie también reflejan este crecimiento exponencial de objetos *a, b* y *c*, que, a medida que crece la k, se hace más inestable. Como puede observarse, la producción de objetos es significativamente mayor con 3-Gillespie y 5-Gillespie que con el Gillespie básico. Sin embargo, no parece haber demasiada diferencia entre los propios 3-Gillespie y 5-Gillespie.

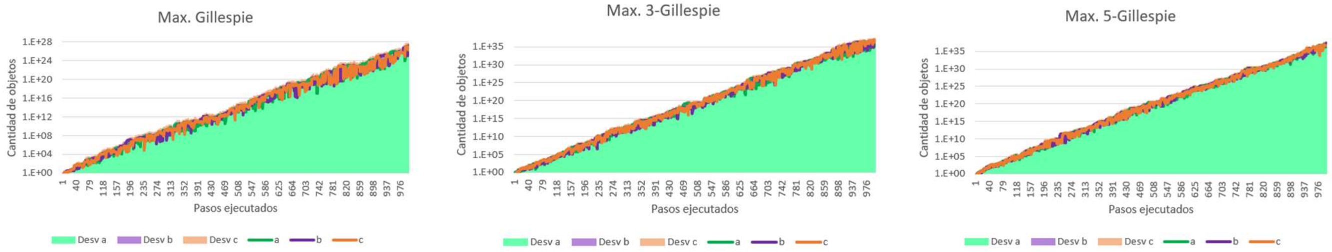


Figura 28: ejecución típica de las versiones maximales del algoritmo de Gillespie en Nesting

Respecto de los objetos *y* y *z* puede decirse que en las maximales tienen un crecimiento exponencial también. La cantidad de objetos *z* suele ser mayor que la de objetos *y*. Sin embargo, esta diferencia disminuye con la k. La diferencia en las desviaciones típicas de la producción de las semillas disminuye a su vez. También pude verse que la producción de semillas de 3-Gillespie y 5-Gillespie es muy similar y significativamente mayor que la de Gillespie básico.

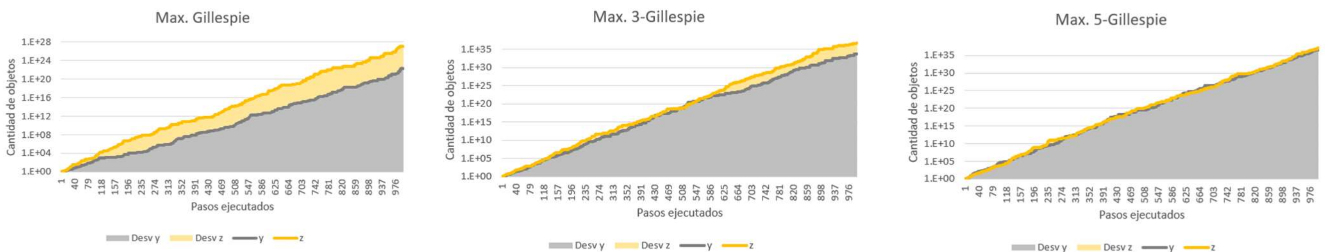


Figura 29: ejecución típica de las variaciones del algoritmo de Gillespie en Nesting

9.3.2.4. Gillespie minimal

Las versiones minimales del algoritmo de Gillespie generan objetos *a*, *b* y *c* de forma lineal. Los objetos más frecuentes van variando, pero todas las cantidades tienen el mismo orden. Puede verse como la desviación típica entre ejecuciones disminuye ligeramente con el aumento de la k.

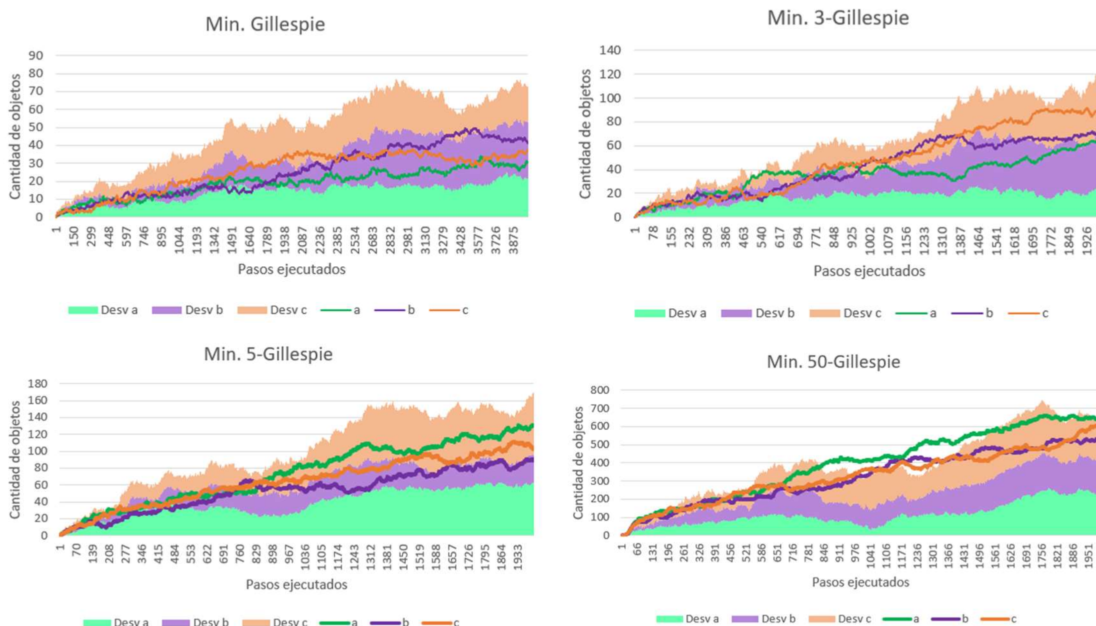


Figura 30: ejecuciones típicas de las variaciones minimales del algoritmo de Gillespie en Nesting

Respecto de la producción de semillas, las variaciones minimales de Gillespie tienden a un crecimiento lineal con las mismas cantidades de objetos *y* y *z*. También se aprecia una ganancia de coherencia entre ejecuciones.

Figura 31: ejecuciones típicas de las variaciones minimales de Gillespie en Nesting

9.4. Escenario Anillo

En este escenario las membranas *Passenger* contienen un objeto *w* y están contenidas en una membrana *A, B* o *C*. Van cambiando de membrana cíclicamente y, dependiendo de en cuál se disuelvan, su objeto *w* se transforma en un objeto *a*, *b* o *c* que se libera a la membrana entorno. Si el objeto *w* se libera a la ventana entorno antes de transformarse, se convierte en un objeto *u*.

9.4.1. ARES

La ejecución típica del algoritmo ARES (Campos, y otros, 2015) en este escenario contempla un crecimiento lineal de todos los objetos con pendiente proporcional a la probabilidad asociada a la disolución de la membrana *Passenger* en cada membrana del ciclo. Al igual que en el resto de ejecuciones de este algoritmo observamos nuevamente unas desviaciones típicas casi nulas.

Por la cantidad de objetos w presentes a lo largo de la ejecución, puede verse que las membranas *Passenger* se eliminan rápidamente o, al menos, a un ritmo comparable al de su producción.

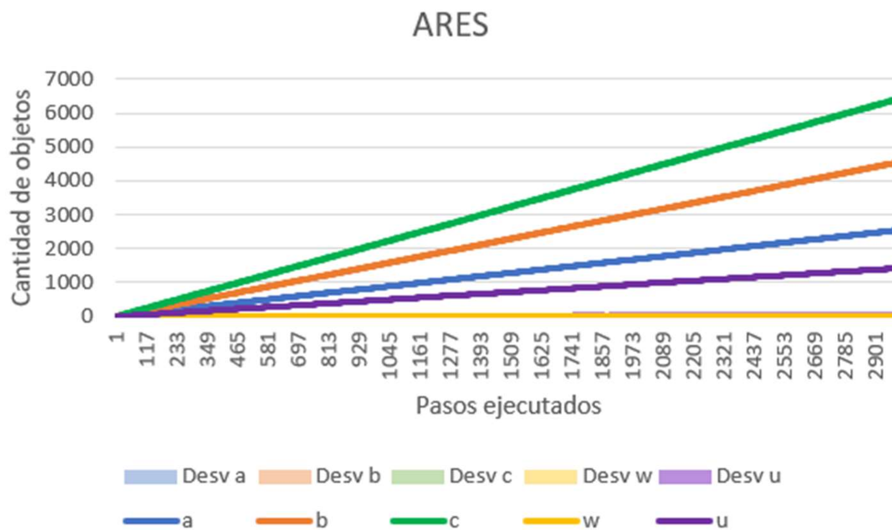


Figura 32: ejecución típica del algoritmo de ARES en Anillo

9.4.2. Aleatorio

En las ejecuciones del algoritmo aleatorio, los objetos *u* tienen una pendiente de crecimiento más pronunciada que con el algoritmo ARES. Aun así, sigue siendo un crecimiento lineal. Las membranas *Passenger* también se eliminan rápidamente. Notamos, a su vez, unas desviaciones típicas mínimas.

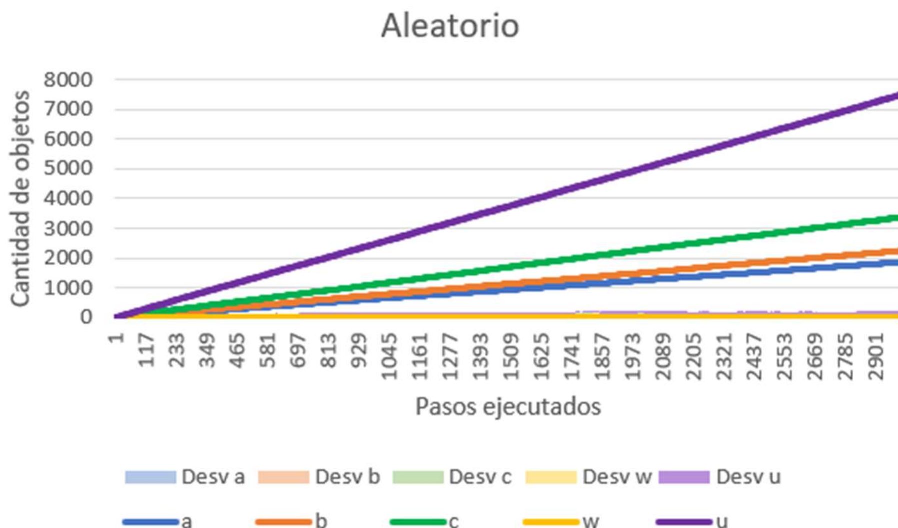


Figura 33: ejecución típica del algoritmo aleatorio en Anillo

9.4.3. Gillespie maximal

Los resultados obtenidos con las versiones maximales de Gillespie son bastante parecidas a las de ARES. No parece haber una diferencia sustancial entre el algoritmo básico de Gillespie y la generalización K-Gillespie.

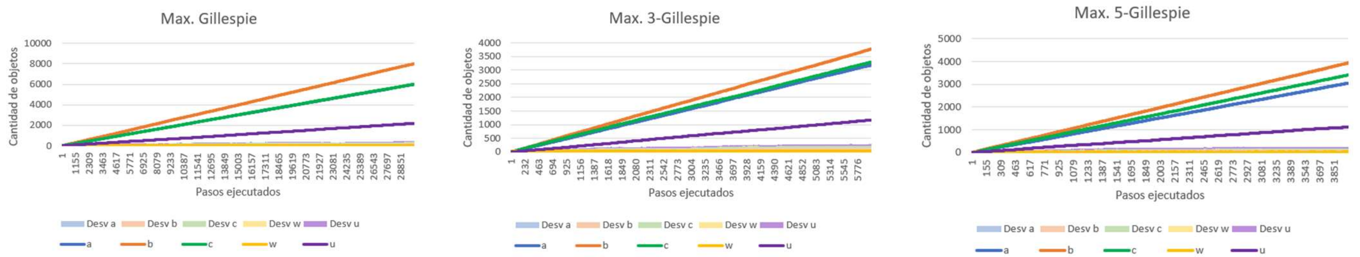


Figura 34: ejecución típica de las versiones maximales del algoritmo de Gillespie en Anillo

9.4.4. Gillespie minimal

Con las versiones minimales seguimos obteniendo un crecimiento lineal de todos los objetos. Es interesante apreciar que a medida que aumentamos la k la cantidad de objetos *w*, y por tanto las membranas *Passenger*, es menor. También puede apreciarse una disminución de la variabilidad con el aumento de la k.

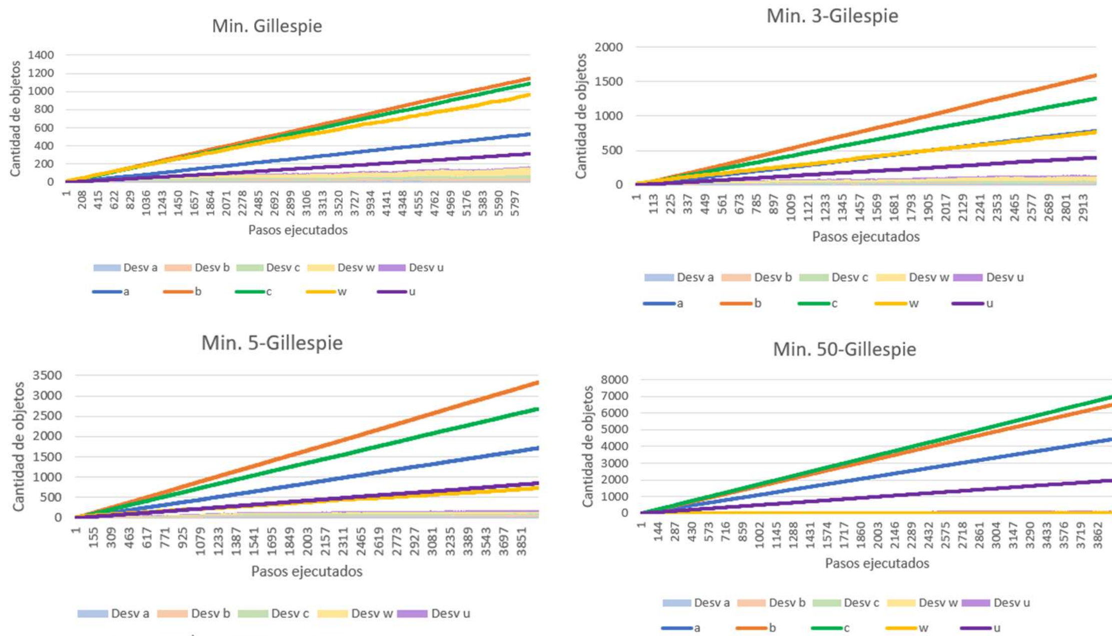


Figura 35: ejecuciones típicas de las versiones minimales del algoritmo de Gillespie en Anillo

10. Conclusiones

Tras haber realizado las simulaciones explicadas en el apartado anterior, puede deducirse que no todos los algoritmos de elección de regla propician un comportamiento similar del escenario. A la vista de las pruebas podemos agrupar los algoritmos estudiados en 3 bloques: algoritmo de ARES (Campos, y otros, 2015), las versiones maximales del algoritmo de Gillespie, el algoritmo aleatorio y, finalmente, las versiones minimales del algoritmo de Gillespie.

10.1. ARES

El algoritmo de ARES (Campos, y otros, 2015) tiende a repartir los recursos entre las reglas que compiten entre sí. Es decir, tiende a ejecutar el mayor número de reglas diferentes posible. Cuando has recursos suficientes para repartir entre todas las reglas tiene un comportamiento bastante determinista. En esas circunstancias solo deja al azar el redondeo alto o bajo del número de veces que se ha de aplicar una regla. No obstante, como hemos podido observar en el escenario C*haining*, cuando los recursos son escasos la ejecución puede cambiar completamente entre unas y otras simulaciones. En el escenario *Chaining* se podían crear

objetos *x* o *z* de forma lineal mientras que los otros permanecían constantes. Esto se debe a que en función de si se aplicaba una regla u otra se podían agotar los recursos en una membrana y paralizarse su proceso de producción.

En situaciones de escasez de recursos el algoritmo de ARES (Campos, y otros, 2015) premia más que el resto de los algoritmos estudiados a las reglas con más propensión. En estos casos, como pasaba en el escenario *Nesting* el algoritmo tiende a ejecutar sólo las reglas con más propensión. En dicho escenario, pudimos ver cómo cuando sólo había una o dos membranas *Individual* inmediatamente contenidas en *Common Area* el algoritmo tendía a ejecutar siempre la regla de anidamiento en lugar de la que la introducía en la membrana *Absorption*. En la línea de estas reglas poco probables, en el escenario *Dos Habitats* queda a la luz que, con pocos recursos (objetos *t*), no se aplican reglas atípicas (regla 22, que producía objetos *z* a partir de objetos *t* en membranas *Individual*).

Podemos concluir que este algoritmo y las versiones minimales del algoritmo de Gillespie tienen un comportamiento bastante similar puesto que tienden a repartir los objetos disponibles entre todas las reglas aplicables.

10.2. Gillespie maximal

En el ámbito de aplicación maximal, el algoritmo básico de Gillespie y el K-Gillespie no muestran grandes diferencias. Como las reglas se aplican de forma exhaustiva, si dos reglas que compiten han de ser ejecutadas sucesivamente solamente se ejecutará la primera. El algoritmo K-Gillespie puede aportar un cierto grado de paralelismo entre bloques de reglas que no compiten entre sí. No obstante, en la práctica este grado de paralelismo ha resultado ser mínimo. Ahora bien, en los escenarios con un crecimiento continuo de la cantidad de objetos, una *K* mayor parece fomentar ligeramente ese crecimiento. Este factor habrá de tenerse en cuenta a la hora de extraer conclusiones de la simulación. Por otra parte, este aumento de la *K* parece conferir de mayor determinismo a las ejecuciones. Así, suelen ser necesarias menos ejecuciones para poder extraer patrones. Esta ganancia de determinismo es una consecuencia de la cantidad de reglas que se eligen: al haber un número limitado de reglas disponibles, si queremos elegir ese número de reglas o más, siempre estaremos eligiendo todas las reglas. La variabilidad observada en estos casos viene del orden de aplicación de las reglas elegidas.

10.3. Aleatorio

El algoritmo aleatorio muestra las consecuencias de una aplicación exhaustiva de reglas. En situaciones de abundancia de recursos, los reparte entre las reglas aplicables, no siempre de forma proporcional a su propensión. Ejemplo de ello es el escenario Anillo, en el que el objeto más producido es el *u*. Sin embargo, la regla que lo produce es la que menor propensión tiene. En este caso, la producción de dicho objeto parece favorecida porque puede llevarse a cabo en varias membranas. En cambio, la producción de los objetos *a*, *b* o *c* solo puede darse en membranas específicas.

No parece que este algoritmo favorezca la aplicación de reglas extremadamente poco propensas. Así lo demuestra el uso de este algoritmo en el escenario Dos Hábitats, donde no se produce ningún objeto *z*.

10.4. Gillespie minimal

Las versiones minimales del algoritmo de Gillespie presentan siempre crecimientos lineales, o constantes por compensación. Esto se da porque en cada turno pueden aplicarse *K* reglas como máximo. Así, cuando hay abundancia de recursos para aplicar reglas, la tasa de

crecimiento del sistema es proporcional a la *K*, no a la cantidad de recursos. Es por ello, que el valor de la *K* influye firmemente en la evolución del simulador. Una mayor *K* comporta casi siempre una mayor tasa de aplicación de reglas.

Otra característica de este algoritmo es que tiende a repartir los recursos disponibles entre las reglas aplicables. No sigue el reparto perfectamente proporcional a la propensión que sí realizaba el algoritmo de ARES (Campos, y otros, 2015); pero aún así, reparte más los recursos que las versiones maximales. Tanto es así, que en ocasiones puede promover la sobreaplicación de reglas menos propensas. Podemos observar un ejemplo de este comportamiento con el algoritmo 50-Gillespie sobre el escenario *Nesting*. En el apartado de experimentación hemos podido ver, que el número de membranas *C* crecía hasta una cantidad en la que permanecía estable. Llegada esa cantidad, se producía un patrón de disolución-creación de membranas que hacía que el número de membranas *C* oscilase en torno a dicha cantidad. Esto se producía porque, a medio plazo se aplicaba el mismo número de creaciones que de disoluciones. Sin embargo, las reglas de disoluciones tienen una propensión menor.

Si la *K* es demasiado baja puede ocurrir lo contrario: puede que no se apliquen suficientes reglas para asumir la transformación de un flujo continuo de recursos. Esto ocurría en el escenario Anillo, en el que la cantidad de objetos *w* crecía con una tasa comparable a la de los objetos en los que debía transformarse. Aun así este comportamiento puede ser útil para poner limitaciones a la capacidad de ciertas membranas.

11. Bibliografía

Baquero, F., Campos, M., Llorens, C., & Sempere, J. M. (2021). P systems in the time of COVID-19. *Journal of Membrane Computing*.

Campos, M., Capilla, R., Naya, F., Futami, R., Coque, T., Moya, A., . . . Baquero, F. (2019). Simulating Multilevel Dynamics of Antimicrobial Resistance in a Membrane Computing Model. *ASM Journals, mBio*.

Campos, M., Llorens, C., Sempere, J. M., Futami, R., Rodríguez, I., Carrasco, P., . . . Baquero, F. (2015). A membrane computing simulator of trans-hierarchical antibiotic resistance evolution dynamics in nested ecological compartments (ARES). *Biology Direct*.

Gibson., M. A., & J. Bruck. (2000). Efficient Exact Stochastic Simulation of Chemical. *Journal of Physical Chemistry*, 1876-1889.

Gillespie, T. (22 (1976)). D.T. A general method for numerically simulating the. *J. Comp. Phys.*, 403-434.

Gillespie, T. (81 (1977)). Exact stochastic simulation of coupled chemical reactions. *J. Phys. Chem.*, 2340-2361.

Paun, G. (1998). Computing with membranes. *TUCS Report 208*.

ANEXO

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**	X			
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**			X	
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**			X	
ODS 12. **Producción y consumo responsables.**		X		
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Escola Tècnica
Superior d'Enginyeria
Informàtica

ETS Enginyeria Informàtica
Camí de Vera, s/n. 46022. València
T +34 963 877 210
F +34 963 877 219
etsinf@upvnet.upv.es - www.inf.upv.es

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Todo trabajo de fin de grado está inevitablemente relacionado con el desarrollo. Bien como la puesta en práctica de una investigación o como la investigación misma, estos trabajos buscan el desarrollo de la sociedad. Ahora bien, es necesario, sobre todo últimamente, que este desarrollo sea sostenible. No podemos seguir fomentando un desarrollo desigual, irresponsable y, en definitiva, imposible de mantener indefinidamente. Toda la sociedad debemos favorecer este cambio, y desde este trabajo se han tenido en cuenta los objetivos de desarrollo sostenible.
El objetivo de este trabajo consiste en averiguar si el uso de diferentes algoritmos de elección de regla puede ser un factor determinante en el comportamiento de un simulador de epidemias. Es decir, en función del algoritmo escogido, cómo podría variar la simulación de una misma enfermedad a nivel epidemiológico. Esta información puede ser crucial a la hora de desarrollar un simulador que prediga la evolución de una epidemia. Si se desarrollan simuladores apropiados, podríamos adelantarnos a los movimientos de futuras epidemias. Esto puede cambiar de forma determinante la forma en la que la humanidad se enfrenta a nuevas enfermedades, porque este tipo de simuladores pueden adaptarse fácilmente a las evidencias científicas disponibles. No es necesario disponer de numerosos datos para realizar una predicción, como en las aproximaciones basadas en el aprendizaje automático. Por todo ello, este trabajo está íntimamente relacionado con el objetivo de desarrollo sostenible dedicado a la salud y el bienestar.
Los simuladores de epidemias nos permiten comprender mejor la forma de extenderse de una enfermedad. De ese conocimiento podrían surgir avances que nos permitan adaptarnos mejor a futuras epidemias. Esa adaptación puede darse tanto a nivel de infraestructuras sanitarias, como en la propia forma de las ciudades. Las ciudades del futuro deberán estar preparadas para soportar oleadas epidémicas, y los simuladores de epidemias pueden ser un factor estratégico a la hora de dirigir el cambio.
En otras aproximaciones de simuladores basadas en el aprendizaje automático, no sólo se necesita una cantidad ingente de datos para conseguir que dicho simulador desempeñe su función. También es necesario un tiempo de entrenamiento del modelo, durante el cual un computador estará haciendo cálculos costosos. Este proceder comporta un gran consumo de energía por parte del procesador, la tarjeta gráfica y los sistemas de refrigeración. Sin embargo, los simuladores basados en sistemas P, no necesitan este periodo de entrenamiento, sino que se nutren directamente de la investigación científica. El entrenamiento del modelo se sustituye por la traducción de la investigación científica a un sistema P que pueda imitar el comportamiento de la epidemia simulada. Así, este trabajo favorece también el consumo energético responsable.
Es por las razones expuestas que este trabajo no sólo no pierde de vista los Objetivos de Desarrollo Sostenible, sino que favorece su satisfacción. En el nuevo entorno en el que nos estamos empezando a encontrar con el cambio climático el ahorro energético y la preparación contra las epidemias están tomando un papel primordial. Así, esperamos que este trabajo de fin de grado ayude a mejorar la preparación de la sociedad a esos dos respectos.