Universidad de Zaragoza

Facultad de Ciencias

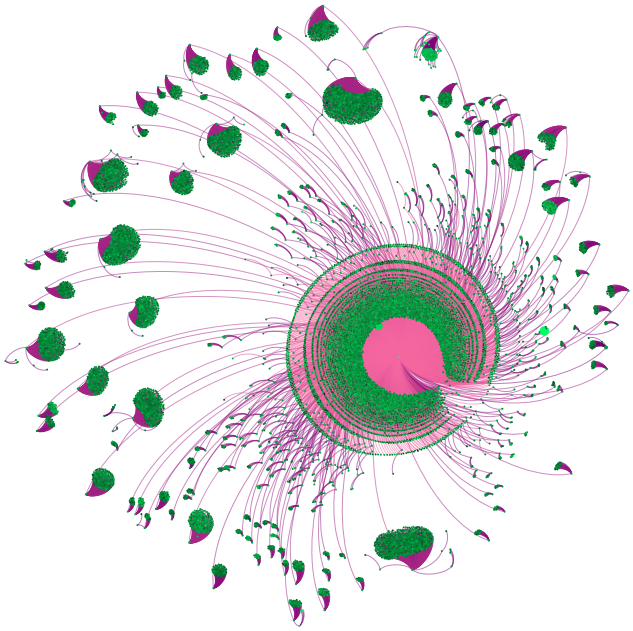
Departamento de Física Teórica
Departamento de Física de la Materia Condensada

Trabajo Fin de Máster

Modelización matemática de problemas biológicos

Autor:
Pablo Javier Blasco Hernández

Directores:
Dr. Pierpaolo Bruscolini
Dr. Fernando Falo



Índice

1. Introducción

Una gran cantidad de sistemas de naturaleza biológica o nanotecnológica se caracterizan por una dinámica dominada por eventos de baja frecuencia tales como el salto de una barrera de potencial. Es un gran reto caracterizar en este tipo de sistemas tanto el paisaje de energías como su dinámica, es decir, determinar los caminos óptimos entre los mínimos del potencial del sistema, la difusión en el tiempo de la distribución inicial, así como las configuraciones críticas que marcan los ritmos dinámicos por formar los estados de transición. Obtener toda esta información permite establecer parámetros de orden y coordenadas de reacción a posteriori, además de que forma la base de técnicas de diseño de sistemas moleculares con aplicaciones tecnológicas (por ejemplo, establecer las interacciones en un polímero al mutar su secuencia).

El espacio de configuraciones de un sistema de gran tamaño puede ser inabarcable para efectuar una descripción exhaustiva, pero un muestreo del mismo mediante técnicas de Montecarlo o de dinámica molecular permite extraer la información más relevante, que se centra en aquellos estados con una probabilidad de ocupación no despreciable. De este modo se obtienen los estados relevantes y las transiciones

entre ellos, despreciando los estados marginalmente poblados. Aquí surge de forma natural la descripción del sistema como una red de Markov. Un enfoque clásico consistiría en el cálculo de valores promedios de ciertos observables, mientras que una red de Markov permite dar una visión alternativa del paisaje de energías y la dinámica a la que da lugar.

Una vez el sistema se ha representado como una red de Markov, la información relevante se plasma en los autovalores y autovectores de la matriz que la caracteriza. Estos pueden en un principio ser calculados mediante técnicas tradicionales, pero su tamaño lo hace inabordable en un tiempo razonable. Afortunadamente, es posible desarrollar herramientas alternativas para obtener estimaciones de los mismos de forma mucho más rápida y de un modo tal que además se obtiene una interpretación física e intuitiva de los mismos en relación a las propiedades del sistema.

El método aquí empleado ha sido desarrollado por la doctora Cameron de la Universidad de Maryland [1] y calcula estimaciones de los autovalores y autovectores al tiempo que les otorga una interpretación como barreras de potencial entre estados y la formación de una jerarquía de cuencas de atracción. Para ello emplea ciertas propiedades comunes en redes de Markov que representen sistemas físicos y que formulan un problema análogo de optimización en grafos.

El método se concreta en el empleo de un algoritmo, denominado *Single Sweep Algorithm*, el cual ha sido aplicado originalmente a sistemas de átomos con interacciones de Lennard-Jones. Estos sistemas tienen propiedades análogas a las presentadas por el plegamiento de proteínas, como el disponer de una energía bien definida, por lo que usamos el mismo algoritmo en este nuevo tipo de sistemas.

Partimos de un modelo discreto para el plegamiento de proteínas aplicado a la proteína gpW. En este modelo, la proteína posee un espacio de configuraciones que comprende 2^{58} posibles configuraciones, si bien se ve reducido a un conjunto más limitado de estados característicos pues la red de Markov se extrae de simulaciones de Montecarlo, de modo que en la práctica solo encontramos aquellos estados que tengan una población apreciable. Estas simulaciones se efectúan a la temperatura de transición de la proteína, lo que implica que los estados plegado y desplegado tengan la misma probabilidad. Este sistema difiere sustancialmente del estudiado por Cameron, especialmente en que uno de los macroestados relevantes viene dado por un conjunto considerablemente grande de microestados escasamente poblados. Esto implica que la entropía tiene un papel relevante a la hora de caracterizar el espacio de configuraciones.

Nuestro objetivo es obtener información sobre el espacio de configuraciones de la proteína y el paisaje de energías subyacente empleando el *Single Sweep Algorithm*. Buscamos obtener una visión de las cuencas de atracción que presente, especialmente en el entorno de los estados nativo y desplegado, así como el camino maximal que une ambos estados. También es posible extraer a partir del algoritmo las autocorrientes que determinan la dinámica del plegamiento.

Para poder llevar a cabo estos objetivos, desarrollamos en primer lugar un código para obtener la red de Markov que representa el plegamiento de la proteína a partir de los datos de la simulación de Montecarlo, de forma que satisfaga las propiedades requeridas por el algoritmo. Este código también implementa un método de agrupamiento (*lumping*) de estados para reducir el tamaño de la red sin perder información relevante. A continuación, hacemos una implementación del algoritmo en C++. Finalmente, desarrollamos una serie de herramientas y códigos para analizar los datos obtenidos con el algoritmo.

La diferente naturaleza de nuestro sistema respecto al de Cameron ha dado lugar a algunas dificultades tales como la adecuada especificación de la energía de los estados y la probabilidad de transición o la simplificación del paisaje de energías para agrupar estados con información superflua.

Hemos analizado el proceso de plegamiento de la proteína gwp, elegida por su naturaleza adecuada para nuestro método, caracterizando el perfil de energías del camino de mínima energía, analizando las escalas temporales de los procesos de plegamiento mediante los autovalores de la matriz representativa

de la cadena de Markov y visualizando la estructura del paisaje de energías.

2. Formulación del problema

Para aplicar las técnicas que vamos a desarrollar, el problema debe poder ser formulado como una cadena de Markov. Esta cadena se trata como una red dirigida cuyos nodos representan los diferentes estados del sistema y los arcos, las transiciones entre ellos.

La red se caracteriza mediante una matriz generadora L y la distribución de probabilidad inicial, p_0. Las entradas de la matriz generadora L_{ij} marcan el ratio de transición entre los estados i y j,

$$L_{ij} = e^{-\beta U_{ij}}, \tag{1}$$

con $\beta = T^{-1}$ el parámetro de la temperatura y U_{ij} un coeficiente relacionado con la barrera de energía de la transición. Esto es así salvo en los elementos de la diagonal, que se definen de modo que se cumpla la condición

$$\sum_j L_{ij} = 0, \tag{2}$$

es decir, que la suma de los elementos de cada fila de la matriz generadora sea nula. El valor absoluto de estos elementos L_{ii} se corresponde a los ritmos de escape de los estados i.

Se asume que la red es irreducible (la probabilidad de alcanzar algún estado partiendo de cualquier otro es no nula, es decir, no hay más estados que los registrados en la propia red) y tiene un número finito de estados, n. Se deduce mediante el teorema de Perron-Frobenius que L tiene un autovalor único $\lambda_0 = 0$ y el resto de autovalores tienen parte real negativa.

Empleando la ecuación maestra o ecuación de Fokker-Planck se obtiene la evolución temporal de la distribución de probabilidad $p(t) = (p_1(t), \ldots, p_n(t))$, que representa la dinámica del sistema. Esta ecuación es

$$\frac{\mathrm{d}p}{\mathrm{d}t} = pL, \quad p(0) = p_0. \tag{3}$$

Es posible escribir la solución empleando la descomposición espectral de L, es decir, $L = \Phi Z \Psi$, siendo Φ y Ψ las matrices con los autovectores derechos e izquierdos como columnas, respectivamente, y Z la matriz diagonal dada por los autovalores. Esta solución es:

$$p(t) = p_0 \Phi e^{tZ} \Psi. \tag{4}$$

El objetivo es caracterizar la dinámica de esta red, para lo cual se requiere la obtención de sus autovalores y autovectores, los cuales además posibilitan obtener las autocorrientes y caracterizar los flujos en la red. Si bien para ello el procedimiento usual sería diagonalizar la matriz generadora L, su tamaño lo hace impracticable, así que vamos a emplear ciertas propiedades que permiten obtener estimaciones de los autovalores para temperatura $T = 0$. Esta temperatura no es la temperatura en que se realizan las simulaciones y no debe ser confundida con la temperatura física, sino que es una temperatura ficticia: suponiendo que fijamos el paisaje de energías a la temperatura de las simulaciones,

buscamos las trayectorias de menor energía, que serían las únicas posibles si la temperatura tendiese a cero. Aunque la verdadera temperatura a la que está el sistema es finita, hacemos los cálculos como si la temperatura tendiese a cero. Es esta aproximación la que permite trabajar con los pesos U_{ij}, que son los logaritmos de las entradas de la matriz L.

Vamos a considerar que la cadena de Markov es reversible en el tiempo, lo cual quiere decir que la matriz generadora cumple la condición de balance detallado,

$$\pi_i L_{ij} = \pi_j L_{ji}, \tag{5}$$

donde π es la distribución de probabilidad en el equilibrio. Gracias a esta condición se pueden extraer tres propiedades para la matriz L. Por un lado, se puede descomponer como $L = P^{-1}Q$ con Q una matriz traspuesta y P una matriz diagonal tal que cada elemento sea la probabilidad en el equilibrio correspondiente. Además, L es similar a la matriz simétrica

$$L_{\text{sim}} = P^{\frac{1}{2}} L P^{-\frac{1}{2}} = P^{-\frac{1}{2}} Q P^{-\frac{1}{2}}, \tag{6}$$

de modo que sus autovalores son reales y no positivos. Por último, las matrices formadas con los autovectores izquierdos (Ψ) y derechos (Φ) como columnas se pueden relacionar mediante la siguiente expresión:

$$\Psi = \Phi^T P. \tag{7}$$

Estas propiedades dan pie a usar el método iterativo del cociente de Rayleigh sobre la matriz L_{sim} para obtener estimaciones de los autovalores a temperatura diferente de cero a partir de las estimaciones a $T = 0$ que obtenemos en un primer momento, si bien en esta memoria únicamente tratamos este caso $T = 0$.

2.1. Propiedades de grafos para obtener estimaciones de los autovalores

El problema para obtener estimaciones de los autovalores de la matriz L a temperatura 0 se puede reducir a un problema de optimización en grafos. Para ello es conveniente extraer una serie de W-grafos, los cuales consisten en un conjunto de k nodos extraído del grafo principal, denominados como sumideros, y un conjunto de $n-k$ (siendo n el número de nodos en el grafo principal) arcos de forma que cada nodo que no sea sumidero tenga un arco saliente y el W-grafo sea un árbol (o un conjunto de árboles disconexos), es decir, no contiene ciclos. Además, un W-grafo se considera óptimo si la suma de los pesos de todos sus arcos es la mínima posible dado un número k de nodos. En adelante nos referiremos a estos W-grafos maximales como g_k^*, siendo k el número de sumideros que tenga.

En la figura 1 se muestra un grafo de ejemplo. Las figuras 1b) y 1c) muestran dos W-grafos óptimos resaltados en azul (corresponden a g_3^* y g_1^* respectivamente, dado el número de nodos y arcos que los forman). El grafo indicado en 1d) no es un W-grafo debido a que contiene un ciclo.

Asumiendo que todos los W-grafos óptimos son únicos para un dado grafo G, se pueden obtener las siguientes aproximaciones a orden exponencial para los autovalores:

$$\lambda_k = e^{-\Delta_k/\epsilon}(1 + o(1)), \quad \text{donde} \quad \Delta_k = \sum_{(i\to j)\in g_k^*} U_{ij} - \sum_{(i\to j)\in g_{k+1}^*} U_{ij}. \tag{8}$$

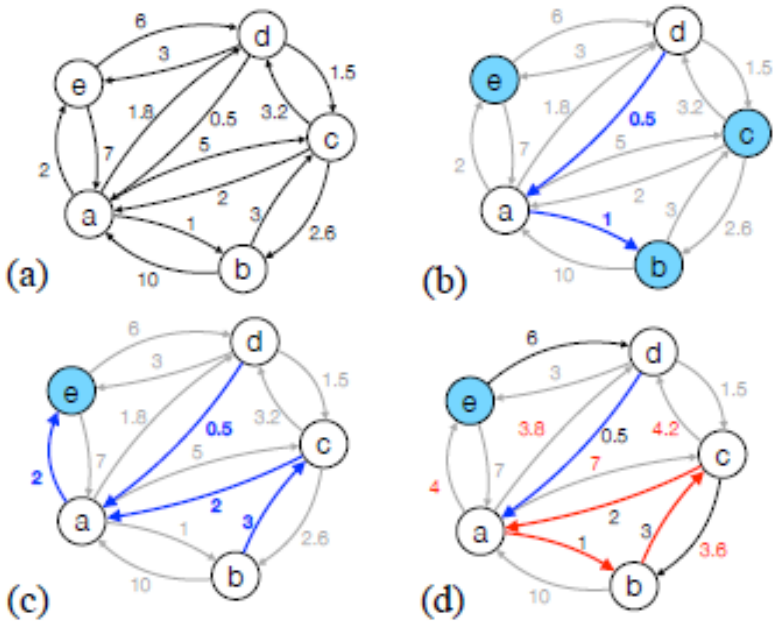


Figura 1: Ejemplo de W-grafos de un grafo dado. Figura adaptada de [1].

Esta expresión permite obtener una aproximación de los autovalores a partir del grafo que representa la cadena de Markov, pues cada sucesivo autovalor λ_k se obtiene a partir de la diferencia en la suma de los pesos de los respectivos W-grafos óptimos g_k^* y g_{k+1}^* extraídos de G. El problema reside ahora en la búsqueda de estos W-grafos óptimos.

El problema de buscar los W-grafos óptimos reside en, para cada valor $1 < k \leqslant n$, tomar k nodos de entre los n nodos del grafo G y a continuación elegir $n - k$ arcos tales que cumplan las condiciones para formar un W-grafo. Sin embargo, dada la gran cantidad de posibles W-grafos para cada valor de k, encontrar el óptimo puede ser muy costoso. Afortunadamente, ciertas propiedades de los W-grafos permiten simplificar el proceso.

Teorema 1 *Sea G un grafo dirigido con pesos U_{ij} con n nodos tal que sus W-grafos óptimos g_k^* con $1 < k \leqslant n$ sean únicos. Entonces se cumplen las siguientes propiedades:*

- *Cada sumidero de g_k^* es también un sumidero del W-grafo g_{k+1}^*.*
- *Sea S_k el conjunto de nodos que se encuentran en la componente conectada de g_{k+1}^* que no contiene ningún sumidero de g_k^* y S el conjunto de todos los nodos de G. Entonces todos los arcos que parten de los nodos de SnS_k en g_{k+1}^* coinciden con los arcos que parten de esos mismos nodos en g_k^*.*
- *Hay un único arco en g_k^* que parte de S_k y llega a SnS_k.*

A partir de este teorema, cuya demostración está dada en [1], se puede establecer una estrategia para encontrar todos los W-grafos óptimos comenzando con g_n^* de forma iterada, pues en cada paso la diferencia entre los W-grafos g_{k+1}^* y G_k^* es de un solo sumidero y las diferencias entre los arcos están localizadas en una sola componente conectada. Esta estrategia se ve realizada en el *Single Sweep Algorithm*, cuya lógica se expone en la sección siguiente.

3. Explicación del algoritmo

Este algoritmo, denominado *Single Sweep Algorithm* debido a que solo necesita recorrer una vez los valores $0 < k \leqslant n$, emplea las propiedades antes detalladas para obtener los valores Δ_k que dan una estimación de los autovalores de la matriz L.

El algoritmo comienza en el W-grafo g_n^*, el cual contiene todos los nodos del grafo G como sumideros y ningún arco. Este W-grafo es único y por lo tanto es óptimo. A continuación, se va disminuyendo el valor de k para encontrar en cada paso el W-grafo óptimo g_k^*, el cual se obtiene a partir de g_{k+1}^* eliminando un sumidero, reconformando la componente conectada del sumidero eliminado (si fuese necesario) y finalmente conectando esa componente conectada a alguna otra mediante un único arco.

Para cada nodo i del grafo G, definimos $\texttt{min\_arc}(i)$ como el arco de menor peso que parte de ese nodo. Tomamos a continuación todos los arcos $\texttt{min\_arc}(i)$ y los unimos en un conjunto M. En cada paso del algoritmo, tomamos el arco de menor peso de M y lo añadimos a un set G^* que contiene los arcos mediante los cuales se construyen los g_k^* a cada paso. Siempre y cuando no se formen ciclos en G^*, el W-grafo óptimo estará formado por los arcos que se han ido eliminando de M y como sumideros tomamos todos los nodos en G salvo aquellos que tienen un arco saliente de ellos en G^*. En tal caso, el valor de Δ_k viene dado directamente por el peso del arco que se ha eliminado en el paso k, pues es el único arco que difiere entre g_k^* y g_{k+1}^* (véase ecuación 8).

Sin embargo, en algún momento puede formarse un ciclo entre algunos arcos de G^*. Esto se produce si los dos nodos del arco retirado de M, sea $x \to y$, pertenecen a la misma componente conectada del último W-grafo óptimo encontrado. En tal caso, ahora todos los nodos de esa componente conectada tienen exactamente un arco saliente y no hay ningún arco en M que pueda unirla a otros nodos de G. Por tanto, es necesario añadir un nuevo arco a M tal que parta de alguno de los nodos del ciclo y vaya a algún nodo de G que no esté en él.

Al realizar esta operación, estamos sustituyendo un arco de G^* que ya pertenece a algún g_l^* con $l > k$ y que era el arco de menor peso que partía de un cierto nodo i, es decir, era $\texttt{min\_arc}(i)$, por el arco $(x \to y)$ que ha formado el ciclo, además de que hemos de añadir un nuevo arco $i \to j$ a M que posteriormente pasará a G^*. Al añadir este arco, la suma de los pesos aumenta con $U_{ij} + U_{xy} - U_{\texttt{min\_arc}(i)}$, ya que estamos retirando el arco $\texttt{min\_arc}(i)$ que daba lugar a la existencia de un ciclo pero añadimos los arcos $(x \to y)$ y $(i \to j)$. Para llevar la cuenta de estos cambios, aplicamos la siguiente regla de actualización a todos los arcos $i \to j$ que partan de todos los nodos del ciclo,

$$U_{ij} = U_{ij} + U_{xy} - U_{\texttt{min\_arc}(i)}. \tag{9}$$

Conforme avance el algoritmo, es posible que se generen nuevos ciclos que contengan nodos de un ciclo anterior. Para gestionar esto, cuando se genere un ciclo se conforma un conjunto con todos los nodos que hayan estado en un ciclo con los nodos del nuevo y se toma ese conjunto para realizar las operaciones antes comentadas.

Se presenta el algoritmo en forma de pseudocódigo como el algoritmo 1.

4. Plegamiento de proteínas y modelo WSME

Las proteínas llevan a cabo funciones clave en todo sistema biológico y es fundamental el papel de su estructura para ello. Están formadas por una cadena lineal de aminoácidos, la cual adquiere su estructura

Entrada: Grafo $G(S, A, U)$, con S el conjunto de nodos, A el conjunto de arcos y U el conjunto de pesos de los arcos.

Salida : Conjunto de valores Δ_k; sumideros que desaparecen s_k^*; sumideros absorbidos t_k^*; arcos de escape $(p_k^* \to q_k^*)$; colección de conjuntos cuasi invariantes S_k; colección de ciclos C_k y colección de W-grafos óptimos g_k^*.

Definir B_i como el conjunto de arcos que salen del nodo i, $\forall i$.
Definir `min_arc`(i) como el mínimo arco saliente de i, $\forall i$.
Eliminar `min_arc`(i) del conjunto B_i, $\forall i$.
Definir $M = \bigcup_{i \in S}$ `min_arc`(i).
Definir $C(i) = \{i\}$, $\forall i$.
Definir $G^* = \emptyset$.
Definir $k = n - 1$.
Definir $g_{k+1}^* = \emptyset$.
mientras $|M| > 1$ **hacer**
  Buscar el mínimo arco de M, $(x \to y)$.
  Eliminar $(x \to y)$ de M.
  Añadir $(x \to y)$ a G^*.
  si *x e y pertenecen a la misma componente conectada de* g_{k+1}^* **entonces**
    Establecer $(p_k^* \to q_k^*) = (x \to y)$.
    Establecer s_k^* como el sumidero de la componente conectada de g_{k+1}^* que contiene a x.
    Establecer t_k^* como el sumidero de la componente conectada de g_{k+1}^* que contiene a y.
    Establecer S_k como la componente conectada de g_{k+1}^* que contiene a s_k^*.
    Definir $C_k = C(s_k^*)$.
    Establecer $\Delta_k = U_{xy}$.
    Obtener el W-grafo óptimo g_k a partir de G^*.
    Establecer $k = k - 1$.
  en otro caso
    Detectar el ciclo C que se ha formado.
    $\forall i \in C$, excepto $i = x$, actualizar los pesos de los arcos en B_i.
    Definir $B = \bigcup_{i \in C} B_i$.
    Definir $C' = \bigcup_{i \in C} C(i)$.
    mientras *los vértices del arco* $(r \to t) = \arg\min_{(p \to q) \in B} U_{pq}$ *pertenecen a* C' **hacer**
      Eliminar el arco $(r \to t)$ de B.
    fin

    Establecer $B_i = B \forall i \in C'$.
    Establecer `min_arc`$(i) = \arg\min_{(p \to q) \in B} U_{pq} \forall i \in C'$.
    Establecer $C(i) = C' \forall i \in C'$.
    Eliminar el arco de mínimo peso de B.
    Añadir ese arco a M.
  fin
fin

Algoritmo 1: *Single Sweep Algorithm.*

funcional ("estructura nativa") a través de su plegamiento. Esta estructura puede ser muy compleja, con una organización tridimensional que en la mayoría de los casos es muy concreta: cada aminoácido toma una posición y una orientación muy específicas, al punto que es posible cristalizar tales estructuras. Un pequeño cambio en algún aminoácido de la proteína puede suponer que sea ineficaz en su función.

El proceso por el cual una proteína adquiere su estructura nativa (el "plegamiento de proteínas") es muy complejo y viene determinado por diversos factores, relacionados también con las condiciones del entorno en que se encuentra. En efecto, un cambio de las condiciones del entorno (variaciones de temperatura o pH, por ejemplo) puede producir modificaciones en la estructura y desencadenar el desplegamiento, con la consiguiente pérdida de la función de la proteína [2]. Los diferentes aminoácidos tienen una serie de propiedades que les hace tender a buscar una posición mutua que suponga una configuración energéticamente favorable, bien sea formando enlaces de hidrógeno o de Van der Waals, o maximizando los contactos hidrófobos. Esto, a fin de cuentas, es lo que da lugar a las estructuras nativas de las proteínas, la mayoría de las cuales sono capaces de plegarse "in vitro", encontrando su estructura, completamente determinada por la secuencia de aminoácidos, de forma autónoma y en tiempos relativamente cortos.

El estudio del plegamiento de proteínas trata de responder a cuestiones [3] tales como cuáles son los procesos físicos que permiten el plegamiento, cuál es la dinámica del proceso o cómo puede producirse tan rápido el plegamiento teniendo en cuenta la gran cantidad de posibles conformaciones a partir de una cadena de aminoácidos. Además, también se presentan retos que pueden tener aplicaciones importantes en cuanto a la predicción y el diseño de la conformación plegada de las proteínas.

Desde un punto de vista físico, el conjunto de conformaciones que adquiere la proteína (o más precisamente, el sistema proteína+disolvente), en contacto con un baño térmico, viene marcada por la energía libre, magnitud que determina la condición de equilibrio y espontaneidad en una reacción química. Cada estado de la proteína tiene asociada una energía libre que en conjunto forma el paisaje de energías de la proteína [5]. Durante el plegado, la proteína recorre este paisaje de energías dirigiéndose al mínimo. Este paisaje suele ser de dimensión muy alta y se compone de una serie de cuencas y mínimos separados por barreras de energía [6]. Para caracterizar el paisaje se deben encontrar sus diferentes mínimos y el modo en que se relacionan: cuáles son las barreras que los separan y en qué forma se agrupan en cuencas.

En la práctica, el estudio del sistema proteína+disolvente presenta dificultades relacionadas con el gran número de átomos, así como con las presencia de un gran abanico de escalas de tiempo de los procesos involucrados, desde la formación de un enlace hidrógeno ($10^{-15}s$), hasta el tiempo de plegamiento ($1ms\boldsymbol{\nabla} \cdot 1s$). Esto hace imposible un acercamiento verdaderamente fundamental (desde la mecánica cuántica) y deja paso a modelos más o menos realistas. Las simulaciones de dinámica molecular (MD) permiten estudiar el plegamiento con un alto grado de detalle, al simular todo el proceso a escala atómica o de pequeños grupos de átomos. Durante mucho tiempo ha sido imposible hacer estas simulaciones con tal grado de detalle debido al tamaño de los sistemas y a no disponer de una descripción adecuada de los campos de fuerzas presentes. Sin embargo, el avance de la tecnología y las técnicas de simulación ha posibilitado el desarrollo y aplicación de dinámica molecular [7]. Ahora bien, el resultado de las simulaciones es una enorme cantidad de datos y se hace necesario desarrollar técnicas para su análisis. Además, el gran problema de las simulaciones muy detalladas es su coste computacional, que limita el muestreo del espacio de las fases, haciendo más complicado el estudio del equilibrio. En cambio, modelos más sencillos se han demostrado útiles para estudiar de forma más extensa el equilibrio, aun al precio de un menor realismo.

En este trabajo aplicamos un algoritmo que ha sido usado en un principio en otro tipo de sistemas, pero que comparten ciertas propiedades fundamentales con el plegamiento de proteínas, para analizar los datos obtenidos a partir de simulaciones procedentes del modelo WSME-S [8]. Este modelo es una ampliación del modelo WSME [9], el cual considera a la proteína como una cadena de N aminoácidos cada

uno de los cuales tiene asociado un estado m_k, que puede ser plegado ($m_k = 1$) o desplegado ($m_k = 0$). El conjunto de los N valores m_k es un microestado del sistema.

A cada microestado en el modelo WSME se le asigna una energía libre efectiva (que se puede pensar como el resultado de integrar la función de partición sobre las variables del disolvente y de las cadenas laterales de los aminoácidos) que viene dada por

$$H = \sum_{i<j} \epsilon_{ij} \Delta_{ij} \prod_{k=1}^{j} m_k - RT \sum_{k=1}^{N} q_k (1 - m_k). \tag{10}$$

Aquí, ϵ_{ij} son las energías de contacto entre cada par de aminoácidos; Δ_{ij} representa la matriz de contactos, que toma valor 1 si los aminoácidos i y j están en contacto y en configuración nativa y 0 en otro caso; R y T son la constante de los gases y la temperatura y q_k son parámetros entrópicos que modelizan el hecho de que cada aminoácido tiene múltiples configuraciones no nativas posibles.

Este modelo recibe una ampliación en el modelo WSME-S, el cual toma la misma estructura pero permite que los parámetros sean dependientes con la temperatura,

$$H(\mathbf{m}, T) = \varphi(T) + \sum_{i<j} h_{ij}(T) \prod_{k=i}^{j} m_k. \tag{11}$$

Resta conocer la expresión de los parámetros dependientes de la temperatura, $\varphi(T)$ y $h_{ij}(T)$, pero al no ser la energía de cada interacción un observable, no se pueden determinar directamente. Para salvar este obstáculo, se relacionan los valores de $h_{ij}(T)$ con la estimación de la dependencia del calor específico con la temperatura hecha en Ref. [10]: esto permite reducir el número de parámetros libres de $N(N+1)/2$ a 4, y ajustarlos a la señal experimental del calor específico. De esta forma, quedan completamente determinadas las h_{ij} en 11.

Con esa expresión de la energía libre se puede estudiar una dinámica de Monte Carlo del sistema, como se detalla en la sección siguiente, que luego caracterizaremos utilizando las herramientas de los *Markov State Models*.

Los modelos de estados de Markov (MSM) constituyen una potente herramienta para el análisis de datos sobre el plegamiento de proteínas [11]. Estos modelos logran obtener simulaciones en las escalas de tiempo relevantes, con suficiente significancia estadística y con representaciones a resoluciones suficientemente grandes para que sea fácil de entender de forma intuitiva. Permiten pasar de simulaciones de trayectoria única a una aproximación estadística más completa y exhaustiva.

Para desarrollar un MSM hay que partir de unos datos iniciales, bien sean experimentales o bien procedan de una simulación. A partir de los datos hay que establecer los microestados y a continuación obtener la matriz de transiciones entre esos microestados. Se pueden tomar dos orientaciones: escalas temporales pequeñas (hacia los nanosegundos) y gran cantidad de estados para obtener resultados cuantitativos o bien escalas temporales más grandes (en torno a los milisegundos) y una menor cantidad de estados, de forma que sea más fácil obtener conclusiones cualitativas y una imagen intuitiva. Los MSM están siendo desarrollados como una herramienta cada vez más útil para estudiar el plegamiento [12].

5. Obtención y preparación de los datos

Los datos a emplear como entrada para el algoritmo proceden de simulaciones de Montecarlo realizadas sobre el modelo WSME-S, expuesto en la sección anterior, aplicado a la proteína gpW, una proteína

pequeña (58 aminoácidos) que ha sido caracterizada como *downhill folder*, es decir, que presenta barreras bastante pequeñas entre el estado nativo y el resto, fáciles de atravesar, haciendo posible un plegamiento de muy poca cooperatividad, que va poblando estados muy diferentes. Esta proteína ha sido elegida justamente porque la escasa barrera permite estudiar las transiciones de equilibrio con más estadística, en principio asegurando el balance detallado. Por otro lado, la ausencia de mínimos marcados, sobre todo en la región desplegada, hace que un conjunto grande de estados esté poblado, aunque escasamente. En el modelo cada aminoácido ("residuo") de la proteína toma el valor 0 si está en su estado desplegado y 1 si está en el estado plegado. En cada paso de las simulaciones se hace un barrido a todos los residuos y se propone un cambio en el mismo, el cual es aceptado o rechazado siguiendo el algoritmo de Metropolis, utilizando la variación de la energía efectiva (11) como criterio para la aceptación.

Las simulaciones se han realizado con una temperatura constante, igual a la temperatura de transición T_m donde supuestamente las poblaciones de estados nativo y desplegados son iguales. Los datos son obtenidos desde 6 trayectorias, y para cada una se efectúa un alto número de pasos de termalización (200000 *sweeps*, correspondiendo un *sweep* con 58 intentos de cambio de estado), que son desechados antes de comenzar a guardar los datos, durante otros 3000000 de *sweeps*. El estado de partida se extrae de la distribución de equilibrio a la temperatura dada.

Los datos resultantes consisten en una lista de estados que se han ido encontrando en cada iteración, en orden temporal. Cada estado consiste en una palabra binaria de 58 dígitos, pero al no encontrarse todos los estados posibles, se asignan índices enteros correlativos a los estados que sí se han encontrado, que son un total de 8745337 estados. A partir de estos datos se debe obtener la red de Markov que representa el sistema.

En primer lugar se obtienen las frecuencias de transición N_{ij} entre cada par de estados de la siguiente manera: se recorre la lista de estados encontrados y a cada paso se incrementa en una unidad la frecuencia entre el anterior estado leído y el actual.

A continuación se calculan las probabilidades en el equilibrio $\{\pi_i\}$,

$$\pi_i = \frac{1}{N_{\Delta t}} \sum_j N_{ij}, \tag{12}$$

donde $N_{\Delta t}$ es el número total de pasos de tiempo en las simulaciones originales, que en nuestro caso es $N_{\Delta t} = 1,8 \times 10^7$. La idea de este cálculo es contar el número de veces que se encuentra cada estado durante las simulaciones y en normalizarlo de modo que la suma $\sum_i \pi_i = 1$.

Para obtener la matriz L hay que realizar una normalización sobre las frecuencias N_{ij}. Esta se hace como una normalización por filas,

$$L_{ij} = \frac{N_{ij}}{\sum_j N_{ij}}. \tag{13}$$

Además, se requiere que se cumpla la siguiente condición:

$$\sum_j L_{ij} = 0, \tag{14}$$

para lo cual se definen los elementos diagonales como el opuesto de la suma del resto de elementos de cada fila,

$$L_{ii} = -\sum_{j \neq i} L_{ij}. \tag{15}$$

Esta fórmula se puede adaptar teniendo en cuenta que la suma de un conjunto de valores normalizados es uno,

$$\sum_j \frac{N_{ij}}{\sum_j N_{ij}} = 1. \tag{16}$$

Extrayendo el elemento N_{ii},

$$\sum_{j \neq i} \frac{N_{ij}}{\sum_j N_{ij}} = 1 - \frac{N_{ii}}{\sum_j N_{ij}}. \tag{17}$$

Uniendo las ecuaciones 15 y 17, se obtiene

$$L_{ii} = \frac{N_{ii}}{\sum_j N_{ij}} - 1. \tag{18}$$

Por tanto, se determina la siguiente expresión general:

$$L_{ij} = \frac{N_{ij}}{\sum_j N_{ij}} - \delta_{ij}. \tag{19}$$

Tras aplicar esta última ecuación, ya se tiene la matriz L. Sin embargo y a pesar de que las condiciones de las simulaciones están a favor de que se cumpla el balance detallado (5), siendo simulaciones largas y con buena termalización, tal condición no se cumple por la matriz que obtenemos con los pasos anteriores. Debido a la naturaleza de la proteína en cuestión, que tiene un estado desplegado de mucha amplitud, muchos estados se exploran una sola vez. Debido a esto hemos tenido que desarrollar un método para imponer la condición de balance detallado perturbando el sistema lo menos posible.

El procedimiento consiste en comprobar para cada par de elementos L_{ij} y L_{ji} cuál es el menor y modificarlo de acuerdo a la expresión

$$L_{ij} = \frac{\pi_j}{\pi_i} L_{ji}, \tag{20}$$

si $L_{ij} < L_{ji}$ o con los índices invertidos en caso contrario. Tras aplicar esta ecuación, surge un nuevo problema debido a que en ocasiones la proporción entre las probabilidades pueden ser tales que surjan valores de L_{ij} mayores a la unidad. Esto supondría tener pesos negativos, lo cual carece de sentido físico. Para solucionar esto, procedemos a realizar una renormalización de toda la matriz L dividiendo todos sus elementos por el elemento máximo (en el caso de que sea mayor que la unidad).

Una vez se ha aplicado esta condición, la matriz L ya cumple las propiedades requeridas. Se puede obtener el grafo sobre el que trabaja el algoritmo como el conjunto de nodos dados por todos los estados representados en L y los arcos dados por todos los elementos L_{ij} no nulos y tales que $i \neq j$, con un peso

$$U_{ij} = -\log(L_{ij}). \tag{21}$$

Se observe que, de esta forma, la U_{ij} es adimensional, y representa en efecto el producto $\beta_{exp}E_{ij}$, donde β_{exp} es la temperatura experimental a la cual se han hecho las simulaciones Montecarlo. Esta expresión, por lo tanto, define un barrera de energía eficaz E_{ij} a partir de las probabilidades de saltos observadas.

6. Detalles de la implementación del algoritmo y rendimiento

6.1. Obtención de la red de Markov

Es preciso escribir un programa que tome como entrada los datos de las simulaciones y extraiga de ellos la red de Markov, en forma de grafo. Los datos proceden de varios archivos de texto con las trayectorias que se han obtenido mediante Montecarlo, sumando un total de 15 millones de transiciones. Estos datos se procesan de la forma que se ha comentado en la sección 5.

La estructura de datos escogida para almacenar el grafo y trabajar con él es un mapa cuyas claves son los índices enteros de los nodos del grafo y los valores asociados a cada una de estas claves son, de nuevo, mapas. Estos mapas *internos* tienen como claves los índices de los nodos tales que haya un arco que llegue a ellos procedente del nodo de la clave principal. El valor asociado es el peso de ese arco.

Esta estructura presenta varias ventajas: solo se almacenan los estados y transiciones que aparecen en la red de Markov; reconoce la posición en memoria de cada nodo directamente con el índice asignado a su estado, sin que sean necesariamente correlativos y se pueden hacer inserciones y eliminaciones con facilidad.

Una inspección del grafo formado por la red de Markov permite ver que frecuentemente se observan estados encadenados sin bifurcaciones, tal y como se muestra en la figura 2. Estos forman caminos que han de ser atravesados en su totalidad por una trayectoria y pueden agruparse como un único estado, considerando que el tiempo que una trayectoria permanece en tal estado tiene una correspondencia con el tiempo que una trayectoria estaría transitando entre los estados originales.

Este agrupamiento de estados o *lumping* permite reducir el número de nodos del grafo sobre el que hay que trabajar, de modo que el tiempo de computación requerido por el algoritmo será mucho menor. Además, mejora la calidad de la visualización de resultados al agrupar estados que aportan la misma información en conjunto y reduce el número de estados con el mismo peso. Esto último suponía un problema adicional que se daba con bastante frecuencia y procede del hecho de tener una cantidad finita de transiciones, que hace que las frecuencias $\{N_{ij}\}$ encontradas tengan valores concretos. El algoritmo va tomando en cada paso el arco de menor peso, pero en el caso de que haya dos arcos con el mismo peso no hay un criterio para escoger uno u otro, así que esta elección se hace de forma arbitraria y perjudica a la interpretación de los resultados.

El *lumping* se realiza siguiendo un procedimiento que no perturbe el sistema y tenga coherencia con la física subyacente. Sea Γ el conjunto de estados que forman una cadena, es decir, que cada estado está unido a otros dos estados que pertenezcan a Γ o a un estado de Γ y a un solo estado ajeno. Entonces se sustituye Γ por un único estado i con dos arcos que lo unan a los otros dos estados ajenos (sean k y l), de forma que el arco $(k \to i)$ tenga el mismo peso que el arco original que partía de k y el arco L_{il} sea:

$$L_{il} = \frac{\pi_m}{\sum_{j\in\Gamma}\pi_j} L_{ml}, \tag{22}$$

donde m es el estado perteneciente a Γ del arco original $(m \to l)$. Además, la probabilidad del nuevo

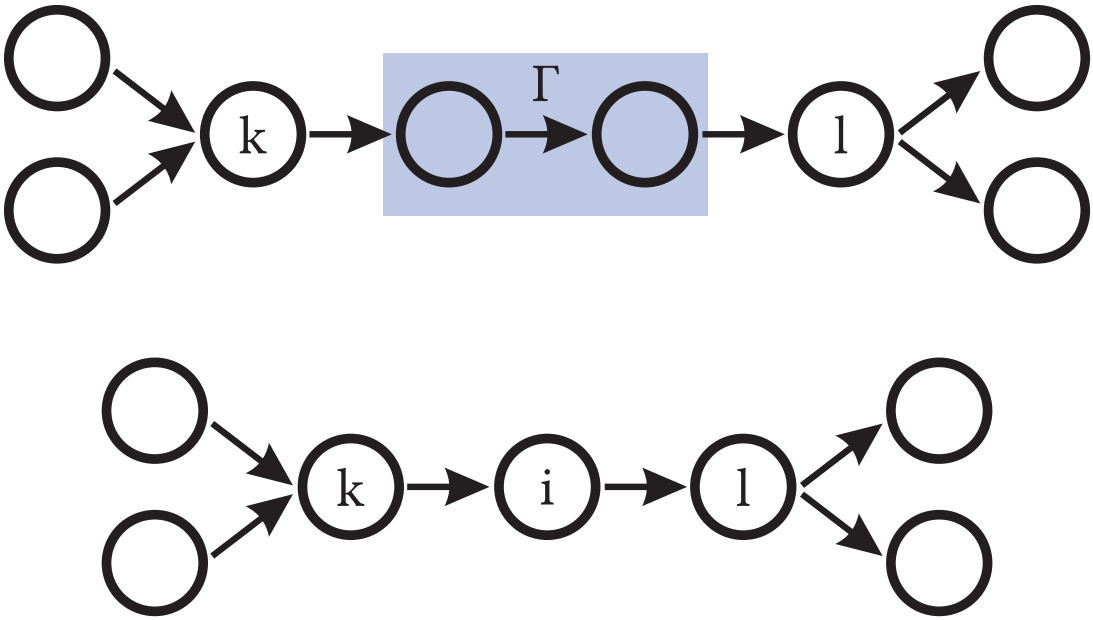


Figura 2: Arriba, representación de un grafo con una cadena Γ formada por dos nodos. Abajo, representación del mismo grafo en el cual la cadena ha sido reemplazada por un único nodo i.

estado es la suma de las probabilidades de los estados de Γ,

$$\pi_i = \sum_{j \in \Gamma} \pi_j. \tag{23}$$

Otro fenómeno frecuente que no proporciona información útil es cuando la trayectoria alcanza un estado e inmediatamente retorna al estado anterior, sin volver nunca a tal estado. Esto da lugar a estados aislados como el mostrado en la figura 3. Se puede eliminar estos estados fácilmente, asimilándolos al estado al que se unen:

$$\pi_{j'} = \pi_j + \pi_i. \tag{24}$$

Dado que el nuevo estado j' incrementa su población, también lo hará $|L_{j'j'}|$, por lo que hay que calcular los valores de $L_{j'k}$,

$$L_{j'k} = \frac{\pi_j}{\pi_i + \pi_j} L_{jk}. \tag{25}$$

En cualquier caso, en la práctica el número de estados eliminados de este modo es muy reducido y no supone una diferencia apreciable.

En cambio, el *lumping* de las cadenas permite reducir en casi un orden de magnitud el tamaño del grafo a analizar, además de que es una receta que puede ser usada de forma general para este tipo de problemas.

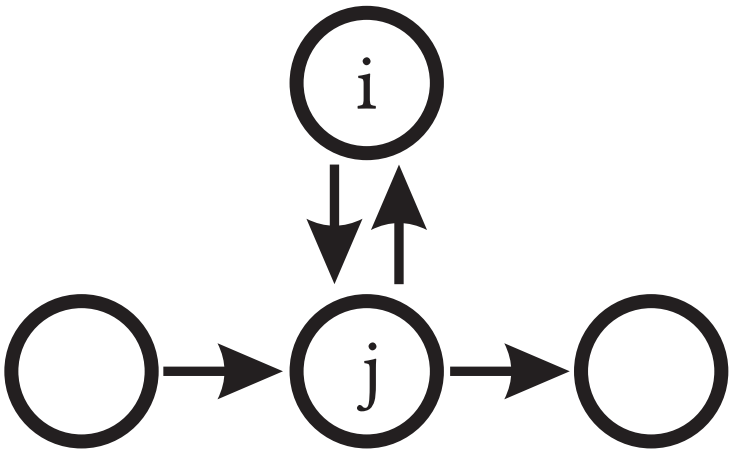


Figura 3: Representación de un grafo en el cual hay un nodo (i) que, al haber sido visitado una única vez en una trayectoria que ha regresado al estado anterior (j) después de visitarlo, ha quedado aislado del grafo salvo por un único camino.

Estados	Arcos	Tiempo de nuestro código (s)	Tiempo del código de Cameron (s)
3224	14664	20.443	0.736
5940	29506	84.298	0.703
10786	58194	495.499	0.902

Tabla 1: Varios valores del tiempo de computación requerido por ambos códigos para analizar un sistema de diferente número de estados y arcos.

6.2. Implementación del *Single Sweep Algorithm*

Una vez obtenido el grafo que representa la red de Markov, realizamos una implementación del *Single Sweep Algorithm* usando C++. El código utiliza la estructura de mapas comentada para representar los grafos y vectores para los conjuntos de nodos. Una comprobación mediante grafos de prueba (como el mostrado en la figura 1) permite asegurar que el código funciona correctamente.

Al realizar las simulaciones, comprobamos que los tiempos de ejecución no son óptimos en relación a las indicaciones de rendimiento teórico aportadas por la doctora Cameron [1]. Esto da pie a cuestionar nuestra implementación, por lo que hemos contactado con Cameron para resolver dudas sobre ciertos detalles del algoritmo y su implementación y nos ha ofrecido su código original. De este modo hemos podido comparar el rendimiento y funcionamiento de ambos códigos.

El código que hemos desarrollado realiza todas las operaciones indicadas en el algoritmo 1 de forma explícita, calculando en cada paso el w-grafo óptimo. El empleo de mapas implica que la búsqueda de arcos o nodos concretos sea relativamente lenta pues no son estructuras de datos con acceso aleatorio. Por otro lado, el código de Cameron realiza muchas de las operaciones implícitamente, sin calcular a cada paso el W-grafo óptimo y usando siempre variables con posibilidad de realizar accesos aleatorios. Esto, unido al uso de la búsqueda de árbol binario para encontrar arcos mínimos, permite que su ejecución sea mucho más rápida.

En la tabla 1 se muestran los tiempos requeridos para analizar sistemas de diferentes tamaños, todos

ellos tomados como un subconjunto de las simulaciones que analizamos. Se observa claramente la mayor eficiencia del código de Cameron.

No obstante, la mayor velocidad en tiempo de ejecución del código de Cameron se obtiene a costa de un código considerablemente más críptico y difícil de entender. Esto trae consigo la necesidad de realizar una documentación sobre el mismo para asegurar la correcta comprensión del código y poder emplearlo y modificarlo de acuerdo a las necesidades del análisis.

Para cerciorarnos de que el resultado del análisis es el mismo con ambos códigos, comprobamos que los resultados obtenidos por los dos son exactamente los mismos. Debido al mejor rendimiento computacional del código de Cameron, optamos por efectuar el análisis con él. Cuando ha sido necesario, hemos modificado el código para extraer resultados no contemplados en el programa original.

7. Exposición y análisis de los resultados

Los datos originales comprenden un total de 8745337 estados. Una vez se ha realizado el tratamiento inicial, incluyendo el *lumping*, el número de estados se ha visto reducido a 1562541. Se trata de una reducción considerable, de casi un orden de magnitud. Esto es debido a que una gran parte de estados han sido observados una única vez uno a continuación de otro, formando una cadena de estados en la red.

7.1. Estimaciones de los autovalores

El análisis que efectúa el *Single Sweep Algorithm* otorga una enorme cantidad de información sobre el sistema. El primer y más evidente parámetro a analizar es el conjunto de valores Δ_k, que son la base de la estimación de los autovalores dada por

$$\lambda_k = e^{-\Delta_k/T}. \tag{26}$$

En la figura 4 se muestran los valores Δ_k en función de k. En un primer momento, se observa que tiene una forma escalonada, con varias mesetas que se deben a que en las simulaciones originales, muchas transiciones han sido observadas una escasa cantidad de veces, lo que provoca que los pesos de los arcos del grafo de entrada al algoritmo tengan valores discretos. Se observe que en la mayoría del espectro los valores Δ_k forman prácticamente un continuo (incluso las partes que parecen verticales; en realidad están formadas por múltiples valores), resaltando que no hay una clara separación de escalas temporales, excepto en las k más bajas. Aquí, especialmente para $k = 14$, que representa la transición de plegamiento, resulta separada de la del resto de los procesos que tienen lugar durante el plegamiento.

A partir de los Δ_k se obtienen estimaciones de los autovalores de L mediante la ecuación (26). Estas estimaciones se han obtenido haciendo la asunción de $T = 0$, pero es posible tomarlas como conjetura inicial para el algoritmo iterativo del cociente de Rayleigh. No obstante, en este trabajo no se realiza esta exploración.

7.2. Camino maximal entre los estados desnaturalizado y nativo

Para estudiar el camino entre los estados desnaturalizado y el estado nativo, escogemos como ejemplo paradigmático de los primeros el estado de solo ceros, y del segundo el estado de todo unos. Por razones

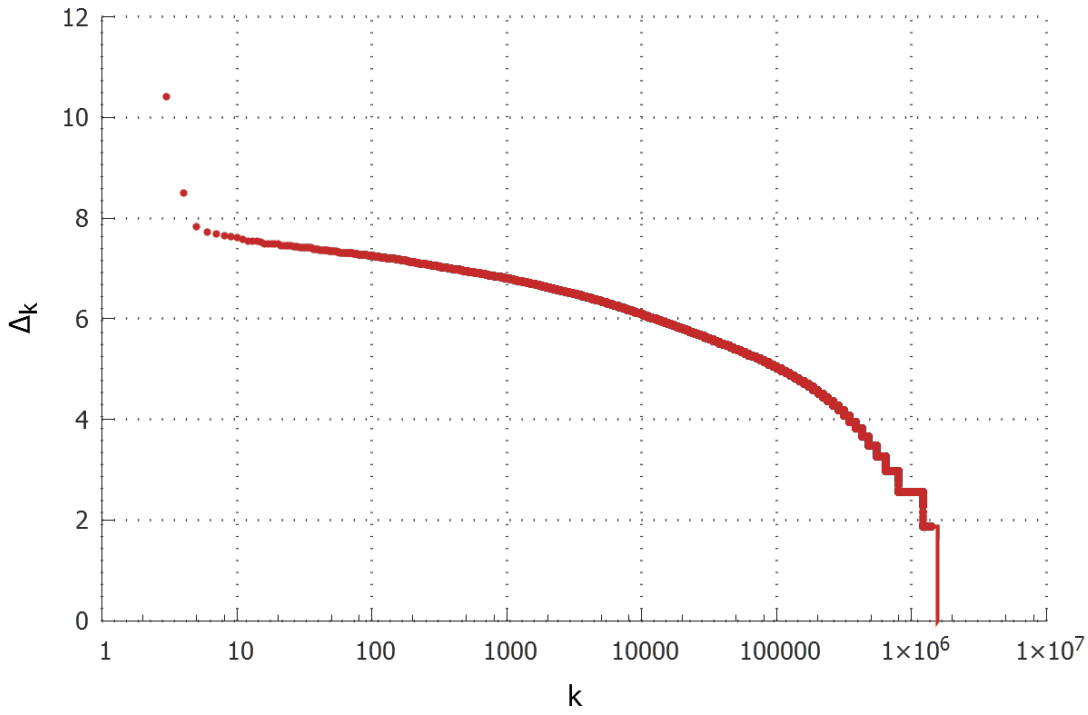


Figura 4: Valores Δ_k en función de k (escala semilogarítmica).

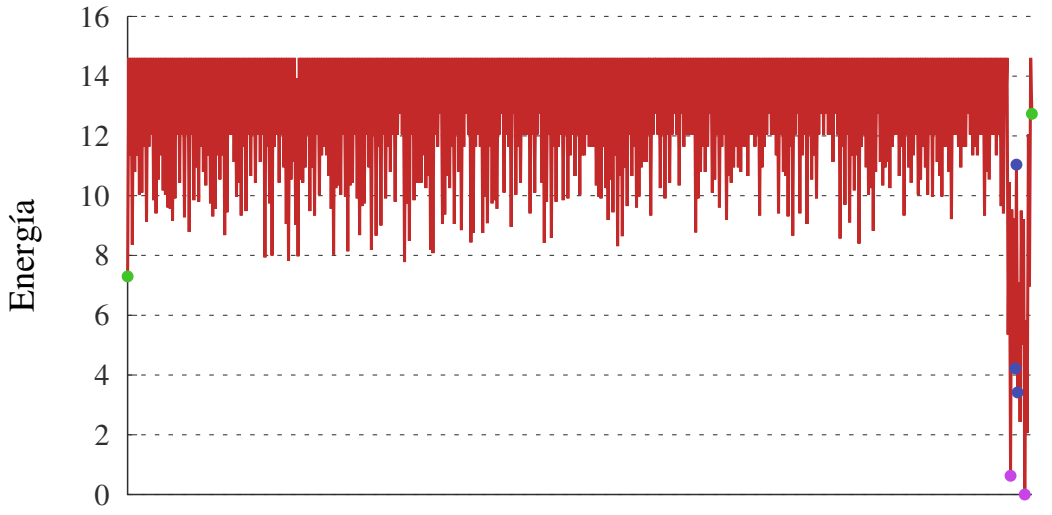


Figura 5: Perfil de energía del camino maximal entre el estado desnaturalizado, a la izquierda, y el nativo, a la derecha (ambos marcados con puntos verdes). La barrera que une ambas cuencas se muestra con puntos azules y los sumideros de ambas cuencas, en violeta.

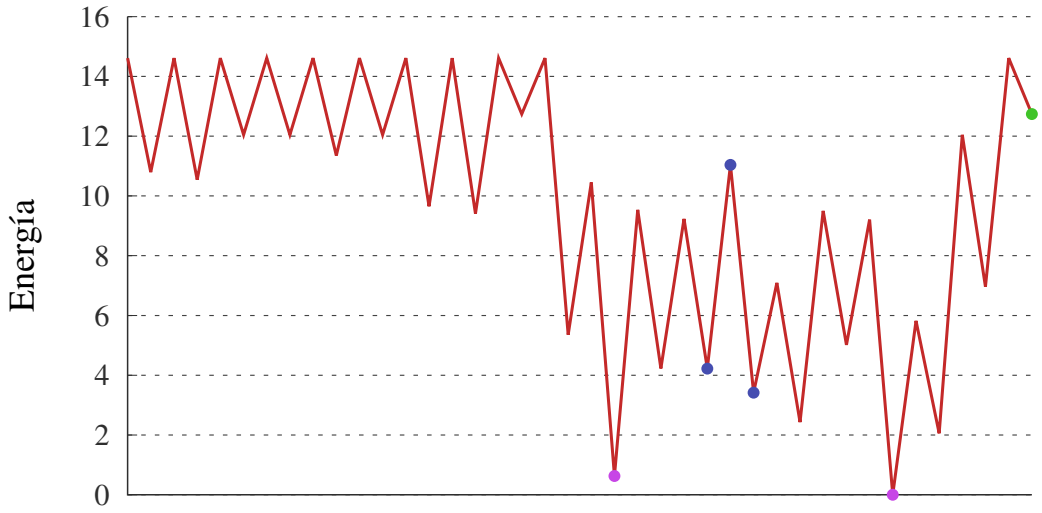


Figura 6: Detalle del perfil mostrado en la figura 5, mostrando la parte del camino que cruza la cuenca nativa.

entrópicas, no esperamos que ninguno de los dos sea un mínimo de energía, pero nuestra hipótesis es que el camino maximal entre ellos pase por los sumideros de la cuenca nativa y desnaturalizada y por la barrera entre ellas. Por tanto, nos dedicamos a identificar el camino maximal, es decir, de menor energía total, entre ambos estados.

El *Single Sweep Algorithm* se sitúa en el caso de temperatura cero y va tomando los arcos de menor peso, es decir, va buscando los caminos óptimos entre los diferentes estados. Con esos arcos construye los W-grafos óptimos g_k^*. Por tanto, desplazarse de un estado a otro recorriendo un W-grafo óptimo implica tomar el camino maximal, teniendo en cuenta siempre que el verdadero camino maximal estará incluido en el g_k^* tal que las componentes conectadas de ambos estados se hayan unido en ese paso de k. En pasos posteriores una posible reordenación de los arcos al formar un W-grafo óptimo implica una modificación de tales caminos.

Teniendo en cuenta lo anterior, resulta de inmediato interés encontrar cuál es el camino maximal entre los estados desnaturalizado y nativo. En concreto, lo interesante es encontrar el perfil de energías que se da entre ambos estados, proyectado sobre ese camino. Esto se muestra en la figura 5.

En esa figura están representados las energías de los estados que separan el estado desplegado y el nativo, así como las barreras entre cada par de estados. Las energías V_i se calculan a partir de las probabilidades en el equilibrio,

$$V_i = -\log\left(\frac{\pi_i}{\pi_f}\right), \tag{27}$$

donde π_f es la probabilidad del estado fundamental, es decir, la mayor de las probabilidades π_i para $1 \leqslant i \leqslant n$. Por su parte, las barreras de energía V_{ij} entre cada par de estados i y j se obtienen a partir de los pesos de los arcos,

$$V_{ij} = U_{ij} + V_i. \tag{28}$$

Para obtener la figura 5 se ha buscado el paso en que los estados nativo y desplegado caen en la misma componente conectada, que resulta ser el último, $k = 1$, y se ha extraído el W-grafo óptimo correspondiente, g_1^*. Partiendo de los nodos correspondientes a los dos estados, se ha recorrido el grafo hasta llegar a un nodo común, que es el sumidero s_0^*. La lista de nodos recorridos forma el camino maximal.

El perfil de energías encontrado muestra una gran región sin orden aparente y una pequeña región con un claro mínimo. La primera región corresponde a la cuenca del estado desplegado, que se encuentra a la izquierda de la figura 5, y la segunda región contiene la cuenca del estado nativo (a la derecha) y la zona en que se unen ambas cuencas.

La cuenca del estado desplegado tiene una gran amplitud, por lo que el camino maximal no se diferencia en exceso de otros caminos próximos. Esto está asociado a una mayor entropía. Está formada por estados con energía en general alta. Sin embargo, la cuenca del estado nativo es mucho más estrecha y sus estados tienen energías menores. La figura 6 muestra esta parte del perfil del camino maximal con más detalle. En esta figura se aprecia bien, marcada en azul, la barrera del arco con el cual se unen ambas cuencas. Esta es la mayor barrera que encuentra la proteína en el plegamiento y supone su cuello de botella. Se da entre los arcos 477729 (perteneciente a la cuenca del estado desplegado) y 719798 (de la cuenca nativa). El valor de esta barrera es $\Delta_1 = 10{,}41KT$.

La forma de este camino es coherente con la naturaleza de la proteína, ya que es un *downhill folder*, es decir, su plegamiento atraviesa una gran cantidad de estados escasamente poblados y separados por barreras de energía pequeñas.

7.3. Jerarquía de cuencas en el paisaje de energía

En cada paso del *Single Sweep Algorithm* se unen dos componentes conectadas del W-grafo óptimo del paso anterior. Los nodos de la componente conectada a la cual pertenece la cola del arco que se ha tomado en ese paso forman el conjunto S_k. De este modo, hay n conjuntos S_k, uno por cada paso k del algoritmo. Estos conjuntos tienen un significado físico destacable, pues los estados que los componen tienen barreras de energía menores entre ellos que con respecto a los estados externos, pues se han unido entre sí primero y después con un estado externo.

Conforme el algoritmo avanza, se van uniendo estos estados S_k entre sí hasta llegar al final, cuando todos quedan unidos. Cada conjunto representa una cuenca de energía y el avance del algoritmo muestra la jerarquía entre todas ellas.

Un conjunto S_k se considera maximal respecto a un estado s_l^* $(l < k)$ si no es subconjunto de ningún otro conjunto S_m para $l < m < k$. Estos conjuntos son las cuencas que se unen directamente a s_l^*, de forma que la barrera que tienen con este estado es menor que la que tienen con cualquier otro estado ajeno.

Es interesante, entonces, obtener la jerarquía de conjuntos S_k maximales respecto a los estados que permanecen como sinks hasta los últimos pasos del algoritmo. Esto permite caracterizar cualitativamente las cuencas del paisaje de energías.

Con el avance del algoritmo, los conjuntos S_k maximales se van uniendo al estado de referencia s_l^*, lo que implica que ese conjunto, que forma una componente conectada, se une a la componente conectada en la que yace el estado de referencia, mediante un cierto arco $(p_k^* \to q_k^*)$. En el proceso, además, el sumidero

k	Tamaño	Estado representante	M del representante
1	1553304	236283	0.66
257	6389	390914	0.78
480367	73	1026606	0.81

Tabla 2: Conjuntos S_k maximales más grandes respecto a $s_0^* = 1262694$.

s_k^* de la componente conectada de S_k deja de serlo. Consideraremos este como el estado representante del conjunto S_k.

El arco $(p_k^* \to q_k^*)$ que marca la unión del set S_k con s_l^* puede ser tal que $q_k^* = s_l^*$, pero no tiene por qué ser así. En muchos casos se da que q_k^* es un estado que pertenece a un set S_m anterior, es decir, $m < k$. Esto es lo que produce la *jerarquía* entre los conjuntos S_k, de forma que cada cuenca se une a la cuenca global con un enlace que la conecta a otra cuenca previamente unida a la cuenca global, pero sin relación directa con otras cuencas. Dado que los pesos de los arcos de unión decrecen con k, los arcos que conectan cuencas tienen un peso mayor conforme más alejados están del estado de referencia en la jerarquía.

Realizamos el análisis de los conjuntos S_k maximales respecto a $s_0^* = 1262694$, que es el último sumidero en desaparecer y lo hace, además, en el momento en que se unen las cuencas de los estados nativo y desplegado. Encontramos un total de 1451 conjuntos S_k maximales, pero de estos la mayoría contiene uno o muy pocos estados. En la figura 7 se muestran aquellos conjuntos cuyo tamaño es de al menos 5 estados. Para caracterizar a los estados representantes de cada conjunto, utilizamos el parámetro

$$M = \frac{1}{N} \sum_k m_k, \tag{29}$$

que toma valores próximos a 0 para estados muy desplegados o 1 para estados cercanos al nativo.

En tal figura se muestra la jerarquía entre los diferentes S_k maximales en forma de grafo. Cabe destacar que la mayoría de estos conjuntos contiene una muy pequeña cantidad de estados, habiendo solo dos con un tamaño verdaderamente notable. En la tabla 2 se muestran los tres S_k maximales más grandes. Los dos conjuntos más grandes están unidos entre sí y concretamente el más grande de ellos contiene la práctica totalidad de los estados del sistema. Este conjunto se forma en el penúltimo paso y merece un análisis similar al realizado para $s_0^* = 1262694$. Dado que su sumidero es el sumidero de la cuenca desnaturalizada, ese conjunto es en realidad esa cuenca. En cuanto al resto de conjuntos S_k maximales encontrados, su tamaño es mucho menor, como ya se ve en el tercer conjunto mostrado en la tabla 2.

La figura 7 representa la estructura interna de la cuenca del estado nativo, excluyendo al conjunto cuyo representante es el estado 236283, que es la cuenca desplegada. El grafo muestra las diferentes subcuencas y cómo se enlazan unas con otras, de tal forma que cada enlace representa la barrera de escape de la cuenca más baja. El análisis empleado para obtener la figura se puede repetir para descubrir la estructura interna de cada uno de estos conjuntos, tomando su sumidero como nueva referencia para considerar que un estado S_k sea maximal. Repetimos el proceso para encontrar la estructura de la cuenca desplegaada, que se muestra en la figura 8.

La cuenca desplegada se compone de 15648 estados S_k maximales. Aquellos que tienen un tamaño superior a 50 estados se han representado en la figura 8, con su jerarquía correspondiente. En esta cuenca se encuentran dos subcuencas importantes; un resumen se muestra en la tabla 3. La cuenca S_2

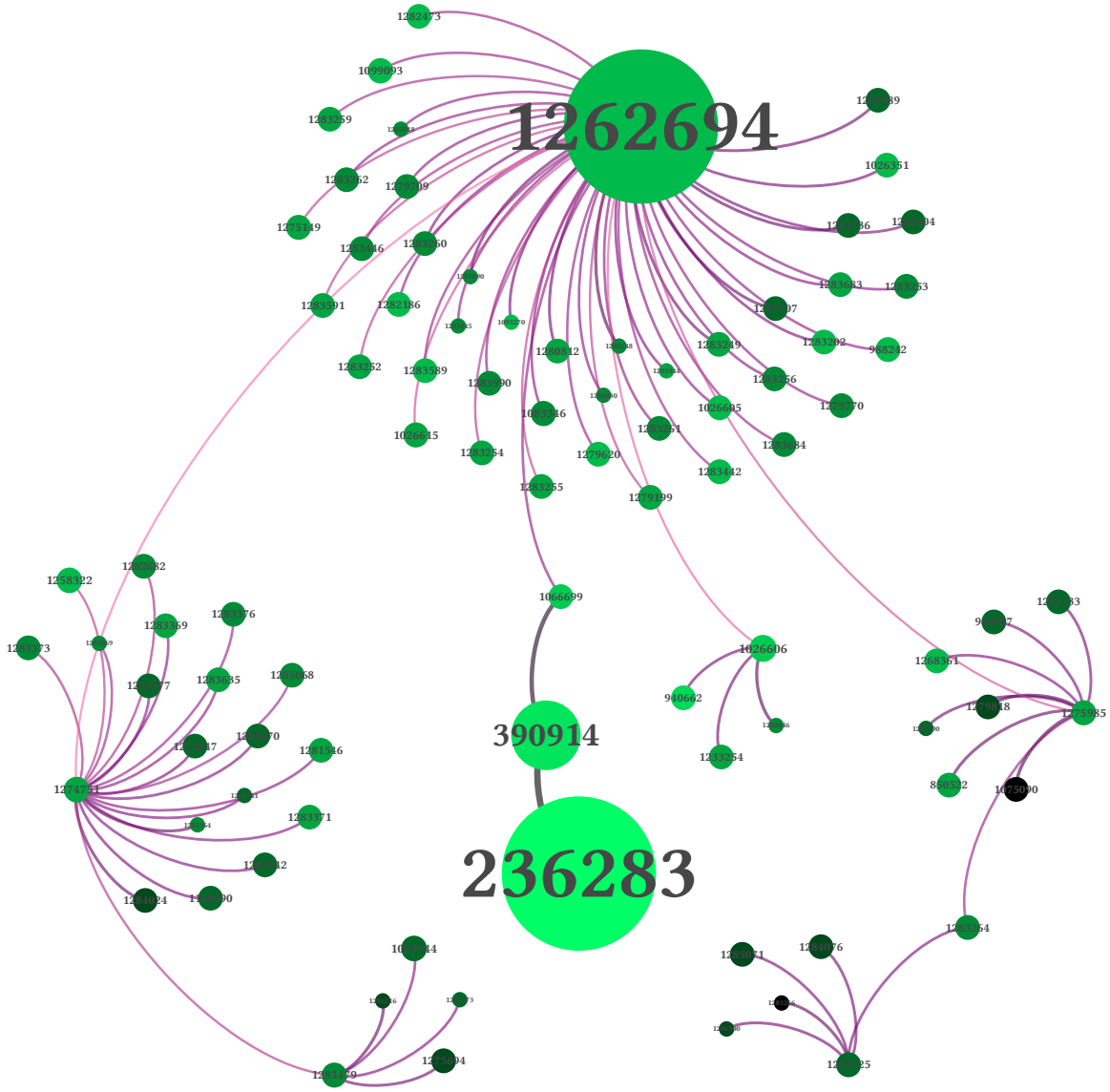


Figura 7: Conjuntos S_k maximales respecto de $s_0^* = 1262694$ con un tamaño de al menos 5 estados. Cada nodo es uno de estos conjuntos, representado con un radio que depende logarítmicamente del número de estados en el conjunto y un color que depende del parámetro $M = 1/N \sum_k m_k$ del sumidero representativo. Verde claro indica valores pequeños y negro, valores próximos a 1. Por otra parte, el color de los arcos va asociado a su peso, siendo el violeta claro para pesos bajos y negro para los altos. El nodo que representa a s_0^* tiene un tamaño arbitrario para facilitar la visualización (no representa a un estado S_k sino a un solo nodo).

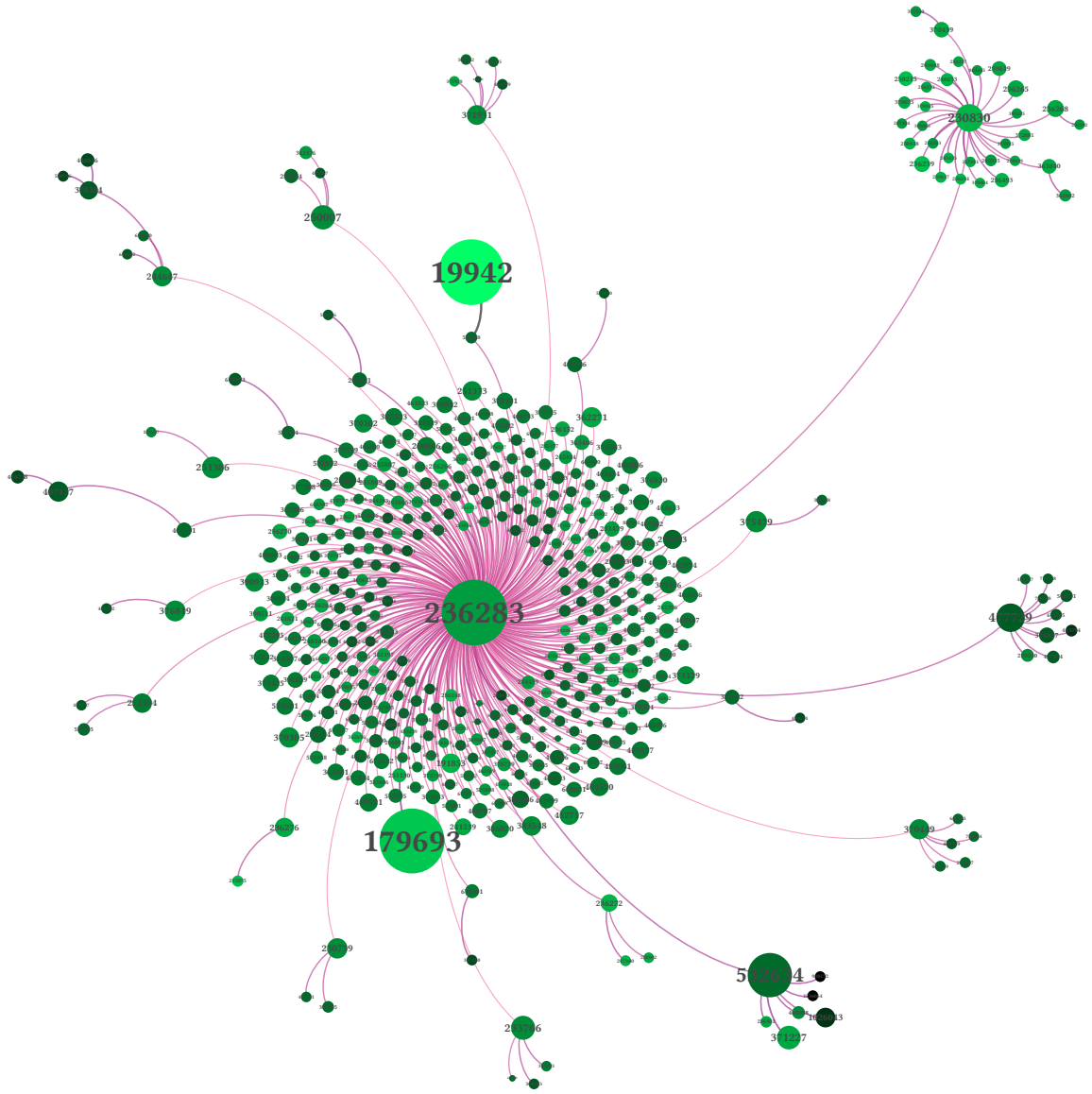


Figura 8: Conjuntos S_k maximales respecto de $s_1^* = 236283$ con un tamaño de al menos 50 estados. Cada nodo es uno de estos conjuntos, representado con un radio que depende logarítmicamente del número de estados en el conjunto y un color que depende del parámetro $M = 1/N \sum_k m_k$ del sumidero representativo. Verde claro indica valores pequeños y negro, valores más próximos a 1. Por otra parte, el color de los arcos va asociado a su peso, siendo el violeta claro para pesos bajos y negro para los altos. El nodo que representa a s_1^* tiene un tamaño arbitrario para facilitar la visualización (no representa a un estado S_k sino a un solo nodo). Las escalas de colores y tamaños no son cuantitativamente equivalentes a las empleadas en la figura 7.

k	Tamaño	Estado representante	M del representante
2	947400	19942	0.28
4034	426171	179693	0.59
33879	24254	532634	0.71

Tabla 3: Conjuntos S_k maximales más grandes respecto a $s_1^* = 236283$.

es la que contiene al estado con todo ceros, es la más grande y la que tiene un menor valor de M. Sin embargo, encontramos otra cuenca con aproximadamente la mitad de estados y un valor de M bastante mayor, $M = 0{,}59$, que indica que está a medio camino entre el estado desplegado y el nativo. Esta cuenca puede representar una trampa cinética: un conjunto de estados intermedios bastante desconectados energéticamente de los estados desplegado y nativo en el que la proteína puede permanecer un tiempo considerable durante el proceso de plegamiento. La barrera de escape de esta cuenca es $\Delta_{4034} = 6{,}42$, mientras que la barrera en el sentido inverso es de 8,54. Al ser estos valores del mismo orden, podemos decir que efectivamente actúa como una trampa cinética.

El resto de subcuencas de la cuenca desplegada tienen tamaños considerablemente menores. Como se ve en la tabla 3, hay un conjunto S_k maximal con 24254 estados y un valor de M bastante próximo al estado nativo; el resto de conjuntos tiene tamaños mucho menores.

8. Resumen y conclusiones

Hemos realizado una implementación en C++ del algoritmo *Single Sweep Algorithm*, el cual analiza una red de Markov para estudiar su dinámica obteniendo una interpretación física para sus resultados. Hemos comprobado el adecuado funcionamiento del código, empleando grafos de entrada especialmente diseñados para ponerlo a prueba. Una vez desarrollado completamente este código, lo hemos comparado con un código desarrollado por los autores del algoritmo y hemos comprobado que efectivamente su funcionamiento es correcto, si bien no resulta computacionalmente óptimo.

A continuación, hemos aplicado el algoritmo a una proteína; en concreto, a su proceso de plegado, centrándonos en el equilibrio entre estados desplegado y nativo. Esta es una aplicación novedosa del algoritmo, que en principio se ha usado con otro tipo de sistemas más sencillos. Gracias a los resultados que arroja, hemos podido analizar el camino maximal que une los estados desplegado y nativo. También hemos obtenido una visualización del paisaje de energías y su estructura de cuencas de atracción.

Los datos de la proteína empleados han sido obtenidos mediante simulaciones de Montecarlo sobre el modelo WSME-S. Estos datos han sido procesados y adaptados a un formato adecuado para el algoritmo mediante un código desarrollado para ello. Este código asegura que se cumplan las condiciones necesarias para la correcta aplicación del algoritmo.

Los datos aportados por este algoritmo son muy ricos y se puede ampliar el análisis que hemos realizado. Sería interesante realizar un cálculo exhaustivo de los autovectores de la matriz que representa a la red, a partir de los cuales se pueden obtener las autocorrientes que caracterizan la dinámica del sistema. También ha quedado fuera de nuestros objetivos la computación de estimaciones de los autovalores de esta matriz a temperatura finita.

Hemos podido comprobar que este algoritmo se puede aplicar al estudio del plegamiento de proteínas con éxito, abriendo las puertas a este tipo de análisis. El algoritmo se puede aplicar a otras proteínas

para obtener resultados similares a los presentados.

Referencias

[1] M. Cameron y T. Gan. "Spectral analysis and clustering of large stochastic networks. Application to the Lennard-Jones-75 cluster". En: *ArXiv e-prints* (nov. de 2015). arXiv: 1511.05269 [cond-mat.stat-mech].

[2] C. M. Dobson. "Protein folding and misfolding". En: *Nature* 426 (2003), págs. 884-890. DOI: 10.1038/nature02261.

[3] K. A. Dill y J. L. MacCallum. "The Protein-Folding Problem, 50 Years On". En: *Science* 338 (2012), págs. 1042-1046. DOI: 10.1126/science.1219021.

[4] S. W. Englander y L. Mayne. "The nature of protein folding pathways". En: *PNAS* 111.45 (2014), págs. 15873-15880. DOI: 10.1073/pnas.1411798111.

[5] A Schug y J. N. Onuchic. "From protein folding to protein function and biomolecular binding by energy landscape theory." En: *Current Opinion in Pharmacology* 10 (2010), págs. 709-714. DOI: 10.1016/j.coph.2010.09.012.

[6] J. Bryngelson y P. Wolynes. "Intermediates and barrier crossing in a random energy-model (with applications to protein folding)." En: *J. Phys. Chem.* 93.19 (1989), págs. 6902-6915. DOI: 10.1021/j100356a007.

[7] T. J. Lane y col. "To milliseconds and beyond: challenges in the simulation of protein folding". En: *Current Opinion in Structural Biology* 23 (1 2013), págs. 58-65. DOI: 10.1016/j.sbi.2012.11.002.

[8] P. Bruscolini y A. N. Naganathan. "Quantitative Prediction of Protein Folding Behaviors from a Simple Statistical Model". En: *Journal of the American Chemical Society* (133 2011), págs. 5372-5379. DOI: 10.1021/ja110884m.

[9] V. Muñoz y W. A. Eaton. "A simple model for calculating the kinetics of protein folding from three-dimensional structures". En: *Proc. Natl. Acad. Sci. USA* 96 (1999), págs. 11311-11316.

[10] E. Freire. En: *Protein Stability and Folding. Theory and Practice*. Ed. por B. A. Shirley. Vol. 40. Humana Press: Totowa, NJ: Methods in Molecular Biology, 1995, pág. 191.

[11] V. S. Pande, K. A. Beauchamp y Bowman G. R. "Everything you wanted to know about Markov State Models but were afraid to ask." En: *Methods* 52 (1 2010), págs. 99-105. DOI: 10.1016/j.ymeth.2010.06.002.

[12] A. Sirur, D. De Sancho y R. B. Best. "Markov state models of protein misfolding". En: *The Journal of Chemical Physics* 144.075101 (7 2016). DOI: https://doi.org/10.1063/1.4941579.

A. Código de la implementación del *Single Sweep Algorithm*

```
   ,
/*
    Coded by Pablo J Blasco(2017), Universidad de Zaragoza

    Implementation of Single Sweep Algorithm, as developed in M. Cameron and T. Gan,
    "Spectral analysis and clustering of large stochastic networks. Application
    to the Lennard-Jones-75 cluster". ArXiv e-prints (2015). arXiv: 1511.05269[cond-mat.
        stat-mech].
*/

#include <string>
#include <stdio.h>
#include <iostream>
#include <map>
#include <utility>
#include <fstream>
#include <sstream>
#include <cmath>
#include <vector>
#include <list>
#include <tuple>
#include <algorithm>
#include <queue>
#include <time.h>
#include <stdlib.h>

#define NUMBER_OF_TIME_STEPS 1.8e7

#define SAVE_AT_STATE 3000000000

//#define INFO //If active, the program will show a lot of info about what it's doing
#define MINIMAL_INFO //This will show only basic information, such as the current step.
//#define BIG_INFO //If active, and INFO is too, it will show big chunks of info, like M
    or G.
//#define LOAD_STATE //If active, the program will load its current state from files
    instead of starting from the beginning
//#define DEGENERACY_WARNING //If active, will show warnings if degeneracy found

using namespace std;
//With this typedef, it is unnecessary to write the double map type explicitly every time
typedef std::map< int, std::map<int,double> > map_of_transitions_type;

int minimal_arc(std::map<int,double> nodes);
int single_sweep_algorithm(map_of_transitions_type &nodes);
bool are_in_same_connected_component(const map_of_transitions_type &graph, int x, int y,
    int &sink_of_x, std::vector<int> &connected_component);
int sink_of_connected_component(map_of_transitions_type graph, int node);
void find_connected_component(map_of_transitions_type graph, int node, std::vector<int> &
    connected_component);
void detect_cycle(const map_of_transitions_type &graph, const int x, const int y, std::
    vector<int> &cycle);
void read_prepared_simulations(map_of_transitions_type &nodes);
void write_prepared_simulations(const map_of_transitions_type &nodes);
void create_testing_map(map_of_transitions_type &nodes);
int total_size_of_map(const map_of_transitions_type &nodes);
void save_current_step(const map_of_transitions_type &nodes,
                       const map_of_transitions_type &Minimum_outgoing_arcs,
                       const map_of_transitions_type &arcs_removed_from_M,
                       const map_of_transitions_type &optimal_W_graph,
```

```
                        const std::vector< std::vector<int> > &cycles_i,
                        const std::vector< std::vector<int> > &Bi_indexes,
                        const std::vector< std::tuple<int,int,double> > &min_arc,
                        const int k);
void load_current_step(map_of_transitions_type &nodes,
                        map_of_transitions_type &Minimum_outgoing_arcs,
                        map_of_transitions_type &arcs_removed_from_M,
                        map_of_transitions_type &optimal_W_graph,
                        std::vector< std::vector<int> > &cycles_i,
                        std::vector< std::vector<int> > &Bi_indexes,
                        std::vector< std::tuple<int,int,double> > &min_arc,
                        int &k);

//A function to convert any type into strings. There is a bug in MinGW so that gcc doesnt
     recognize the function std::to_string()
template <typename T>
std::string to_string(T value)
{
  //Create an output string stream
  std::ostringstream os ;

  //Throw the value into the string stream
  os << value ;

  //Convert the string stream into a string and return
  return os.str() ;
}

main()
{
    map_of_transitions_type nodes;

    #ifndef LOAD_STATE
    read_prepared_simulations(nodes);
    //create_testing_map(nodes);

    //Delete the autoloops, which make the algorithm go crazy
    for(map_of_transitions_type::iterator i = nodes.begin(); i!=nodes.end(); i++)
        if(i->second.count(i->first) != 0)
        {
            i->second.erase(i->first);
            if(i->second.size()==0)
            {
                int aux = i->first;
                i--;
                nodes.erase(aux);
            }
        }

    std::cout << "\nSize: " << nodes.size() << "\n";
    std::cout << "Simulations read\n";
    std::cout << "Map ready. Starting the algorithm...\n";
    #endif // LOAD_STATE

    single_sweep_algorithm(nodes);

    std::cout << "\nFinished\n";

    return 0;
}
```

```
113 int single_sweep_algorithm(map_of_transitions_type &nodes)  //The input: the double map
        which represents the infinitesimal generator L
114 {
115     #ifdef INFO
116     std::cout << "Starting Single Sweep Algorithm.\nAllocating memory...\n";
117     #endif // INFO
118
119     //We will need these variables as the outputs of the algorithm
120     double delta; //\Delta_k
121     int disappearing_sink; //s_k^*
122     int absorbing_sink; //t_k^*
123     std::pair<int,int> exit_arc; //arcs (p_k^* -> q_k^*)
124     std::vector<int> cycles; //Ck
125     map_of_transitions_type optimal_W_graph;
126     std::vector<int> quasi_invariant_set; //Sk. This is a vector in which we will store
            the vertices of
127                                           //every connected component of g_k^*
                                                containing s_k^*.
128
129     //After all these initializations of the outputs, we must declare and initialize the
            internal variables.
130     int k = nodes.size() - 1;   //The variable which takes into account in which
            iteration we are.
131
132     //We need a vector in which we will have the min_arc(i)
133     std::vector< std::tuple<int,int,double> > min_arc(nodes.size());
134
135     //We now create the graph of arcs of minimum weight, named M.
136     map_of_transitions_type Minimum_outgoing_arcs;
137
138     //We need to have record of the sets Bi. We do it by using a vector of vectors of
            ints.
139     std::vector< std::vector<int> > Bi_indexes; //In every ith vector, we list the
            vertexes which belong to set B(i).
140                                                 //For example, if B_a = B_abc, then
                                                        Bi_indexes.at(a) = {a, b, c}.
141
142     #ifdef LOAD_STATE
143     map_of_transitions_type arcs_removed_from_M;
144     std::vector< std::vector<int> > cycles_i = std::vector< std::vector<int> >(nodes.size
            ());
145     load_current_step(nodes, Minimum_outgoing_arcs, arcs_removed_from_M, optimal_W_graph,
             cycles_i, Bi_indexes, min_arc, k);
146     #else
147
148    //We build these sets by putting into them the corresponding index (initially, B(i) =
           {i}
149     for(int i = 0; i<nodes.size(); i++)               //A loop which makes N components
            in Bi_indexes
150         Bi_indexes.push_back(std::vector<int> (1,i+1)); //every one with the
                corresponding index of its vertex.
151
152     //We also create the output files as void files
153     {ofstream output_file; //With just one stream we will open and close all the files.
154             output_file.open("delta.txt");
155             output_file.close();
156
157             output_file.open("disappearing_sinks.txt");
158             output_file.close();
159
160             output_file.open("absorbing_sinks.txt");
161             output_file.close();
```

```
162
163             output_file.open("quasi_invariant_sets.txt");
164             output_file.close();
165
166             output_file.open("exit_arcs.txt");
167             output_file.close();
168
169             //output_file.open("W_graphs.txt");
170             //output_file.close();
171     }
172
173     #ifdef INFO
174     std::cout << "Memory Allocated.\nBuilding M...\n";
175     #endif // INFO
176
177     //We now proceed to the construction of the set of minimum outgoing arcs M.
178     //First, we declare an auxiliary variable
179     {int index_of_minimum_weighted_arc; //We limit the existence of this variable to the
            scope inside which it is going to be used.
180     //We sweep through all the map
181     for(map_of_transitions_type::iterator i=nodes.begin(); i!=nodes.end(); i++)
182     {
183         //For every node, we find first the minimal arc
184         index_of_minimum_weighted_arc = minimal_arc(i->second);
185
186         //We put this arc inside of the min_arc(i) vector
187         min_arc[i->first - 1] = std::make_tuple(i->first, index_of_minimum_weighted_arc,
                i->second.at(index_of_minimum_weighted_arc));
188
189         //We then create this arc in M
190         Minimum_outgoing_arcs[i->first][index_of_minimum_weighted_arc] = 0;
191         Minimum_outgoing_arcs[i->first][index_of_minimum_weighted_arc] = i->second.at(
                index_of_minimum_weighted_arc);
192
193         //And we erase it from the B_i set
194         (i->second).erase(index_of_minimum_weighted_arc);
195     }}//After this loop, we have correctly constructed the set M and we have updated the
            sets B_i so that those two don't have common members.
196
197     #ifdef INFO
198     #ifdef BIG_INFO
199     std::cout << "The set M is:\n";
200     for(map_of_transitions_type::iterator i=Minimum_outgoing_arcs.begin(); i!=
            Minimum_outgoing_arcs.end(); i++)
201     for(std::map<int,double>::iterator j=i->second.begin(); j!=i->second.end(); j++)
202         std::cout << "M has " << i->first << " to " << j->first << " with " << j->second
                << endl;
203     std::cout << "We now build the initial C(i) sets" << endl;
204     #endif // BIG_INFO
205     #endif // INFO
206
207     //Next, we must initialize the cycles as C(i) = {i}
208     std::vector< std::vector<int> > cycles_i = std::vector< std::vector<int> >(nodes.size
            ());
209     //cycles_i is a variable which we build inside this function and with which we will
            build cycles, which is an output.
210     for(unsigned int i=0; i<nodes.size(); i++)
211         cycles_i[i].push_back(i+1); //This adds element i as the last component (in this
                case, single component as it's a new vector)
212                                     //of the ith vector in cycles_i. (Remember cycles_i is a
                                        vector of vectors).
213
```

```
214     #ifdef INFO
215     std::cout << "Initial C(i) sets built." << endl;
216     #ifdef BIG_INFO
217     for(std::vector< std::vector<int> >::iterator i = cycles_i.begin(); i!=cycles_i.end()
            ; i++)
218         for(std::vector<int>::iterator j = (*i).begin(); j!=(*i).end(); j++)
219             std::cout << "The cycle " << (*i).at(0) << " has " << *j << endl;
220     #endif // BIG_INFO
221     #endif // INFO
222
223     //Last part of the initialization: we must declare the set G*
224     map_of_transitions_type arcs_removed_from_M;    //It starts void.
225
226     #ifdef INFO
227     std::cout << "We enter the main cycle.\n" << endl;
228     #endif // INFO
229
230     #endif // LOAD_STATE
231
232      //We will also use a variable to store the number of cycles found
233     int number_of_cycles_found = 0;
234
235     //The main cycle
236     while(total_size_of_map(Minimum_outgoing_arcs) > 1) //Do it until there is only one
            arc left in M
237     {
238         if(k==SAVE_AT_STATE)
239         {
240             save_current_step(nodes, Minimum_outgoing_arcs, arcs_removed_from_M,
                    optimal_W_graph, cycles_i, Bi_indexes, min_arc, k);
241             return 1;
242         }
243
244         std::pair<int,int> minimum_weight_arc(0,0);   //A pair of variables for the
                indexes of the arc of minimum weight
245         {//Find the minimum weight arc in M. We can't use minimal_arc here because we
                have a double map, not a simple one.
246             double minimum_weight = 9999999; //A variable to keep track of the ~
247
248             //We run through Minimum_outgoing_arcs and we search for the minimum
249             for(map_of_transitions_type::iterator i=Minimum_outgoing_arcs.begin(); i!=
                    Minimum_outgoing_arcs.end(); i++)
250                 for(std::map<int,double>::iterator j=i->second.begin(); j!=i->second.end
                        (); j++)
251                 {
252                     #ifdef DEGENERACY_WARNING
253                     if(j->second == minimum_weight)
254                         std::cout << "WARNING: Degeneracy found at searching for the
                                minimum outgoing arc." << endl;
255                     #endif // DEGENERACY_WARNING
256                     if(j->second < minimum_weight)
257                     {
258                         minimum_weight = j->second; //We store the minimum weight found
259                         minimum_weight_arc.first = i->first;  //We store the indexes of
                                the arc
260                         minimum_weight_arc.second = j->first;
261                     }
262                 }
263             //We now have to add this arc to G* and delete it from M
264             arcs_removed_from_M[minimum_weight_arc.first][minimum_weight_arc.second] =
                    Minimum_outgoing_arcs.at(minimum_weight_arc.first).at(minimum_weight_arc.
                    second);
```

```
265             Minimum_outgoing_arcs.at(minimum_weight_arc.first).erase(minimum_weight_arc.
                    second);
266
267             #ifdef MINIMAL_INFO
268             std::cout << "\nValue of k: " << k << endl;
269             #endif // MINIMAL_INFO
270
271             #ifdef INFO
272                 #ifndef MINIMAL_INFO
273                 std::cout << "\nValue of k: " << k << endl;
274                 #endif // MINIMAL_INFO
275                 std::cout << "Minimum arc: " << minimum_weight_arc.first << " to " <<
                        minimum_weight_arc.second << " with " << minimum_weight << endl;
276             #endif // INFO
277         }//End of the transferring of the minimum arc of M to G*
278
279         #ifdef INFO
280         #ifdef BIG_INFO
281             std::cout << "The G map is:" << endl;
282             for(map_of_transitions_type::iterator i =arcs_removed_from_M.begin(); i!=
                    arcs_removed_from_M.end(); i++)
283                 for(std::map<int,double>::iterator j=i->second.begin(); j!=i->second.end
                        (); j++)
284                     std::cout << "G has from " << i->first << " to " << j->first << "
                            with " << j->second << endl;
285
286             //std::cout << "The set M is: " << endl;
287             //for(map_of_transitions_type::iterator i=Minimum_outgoing_arcs.begin(); i!=
                    Minimum_outgoing_arcs.end(); i++)
288             //    for(std::map<int,double>::iterator j=i->second.begin(); j!=i->second.
                    end(); j++)
289             //        if(i->first % 100000 == 0) std::cout << "M has from " << i->first
                    << " to " << j->first << " with " << j->second << endl;
290         #endif // BIG_INFO
291         std::cout << "We test if the new arc connects two different connected components
                of g*k+1\n";
292         #endif // INFO
293
294         //We now proceed depending on whether the two endpoints of the minimal arc are on
                the same connected component
295         if(!are_in_same_connected_component(optimal_W_graph, minimum_weight_arc.first,
                minimum_weight_arc.second, disappearing_sink, quasi_invariant_set))
296         {
297             #ifdef INFO
298                 std::cout << "It doesn't! We proceed normally\n";
299                 std::cout << "Saving minimal arc, absorbing and disappearing sinks and
                        finding connected component" << endl;
300             #endif // INFO
301             //We store the minimal arc as the arc (p_k^* -> q_k^*)
302             exit_arc = minimum_weight_arc;
303
304             //We store the sink of the connected component of q_k^* (for p_k^* it has
                    already been made).
305             absorbing_sink = sink_of_connected_component(optimal_W_graph,
                    minimum_weight_arc.second);
306
307             #ifdef INFO
308                 std::cout << "Disappearing sink: " << disappearing_sink << " Absorbing
                        sink: " << absorbing_sink << endl;
309                 std::cout << "The connected component of the " << k+1 << " optimal W-
                        graph is:" << endl;
```

```
310                 for(vector<int>::iterator i = quasi_invariant_set.begin(); i!=
                        quasi_invariant_set.end(); i++)
311                     std::cout << *i << "\t";
312                 std::cout << endl;
313
314             #ifdef BIG_INFO
315                 std::cout << "The " << k << " cycle is:" << endl;
316                 for(std::vector<int>::iterator i=cycles_i.at(disappearing_sink - 1).begin
                        (); i!=cycles_i.at(disappearing_sink - 1).end(); i++ )
317                     std::cout << "Cycle has vertex " << *i << endl;
318             #endif // BIG_INFO
319             #endif // INFO
320
321             //We build the kth cycle
322             cycles = cycles_i.at(disappearing_sink - 1);
323
324             //We now have to store the optimal W-graph
325             //In this snippet we search for the k optimal W-graph
326             {
327                 map_of_transitions_type inversed_G; //A inversed graph of G will make it
                        faster to go through it
328
329                 //We have to remove the elements in the connected component which has
                        been joined because there may be cycles
330                 for(vector<int>::iterator i=quasi_invariant_set.begin(); i!=
                        quasi_invariant_set.end(); i++)
331                     if(optimal_W_graph.count(*i)==1)
332                         optimal_W_graph.erase(*i);
333
334                 //We build the inversed_G map
335                 for(map_of_transitions_type::iterator i = arcs_removed_from_M.begin(); i
                        !=arcs_removed_from_M.end(); i++)
336                     for(std::map<int,double>::iterator j = i->second.begin(); j!=i->
                            second.end(); j++)
337                         inversed_G[j->first][i->first] = j->second;
338
339                 //Now we have to trace from the joined arc backwards in the connected
                        component S_k
340                 //In a list we will have record of bifurcations we find. It shall be a
                        list because we want to erase it in constant time.
341                 std::list<int> bifurcations, bifurcations_next; //bifurcations_next will
                        have the nodes which are a bit further than those which we are taking
                         into account at the moment.
342                 bifurcations.push_back(absorbing_sink);
343
344                 //We need a vector to know at every moment which nodes we have visited.
345                 std::map<int, bool> already_visited;
346                 for(map_of_transitions_type::iterator i = arcs_removed_from_M.begin(); i
                        !=arcs_removed_from_M.end(); i++)
347                     already_visited[i->first] = 0;
348                 already_visited[absorbing_sink] = 1;
349
350                 while(1!=0)
351                 {
352                     if(bifurcations.size() == 0)
353                         break; //If there aren't any bifurcations left, we have ended.
354
355                     //We make a loop through every bifurcation at this level
356                     for(std::list<int>::iterator i = bifurcations.begin(); i!=
                            bifurcations.end(); i++)
357                         //We only search for the node if it has any incoming arcs
358                         if(inversed_G.count(*i) != 0)
```

```
359                         //Then we go to the list of incoming arcs
360                         for(std::map<int,double>::iterator j=inversed_G.at(*i).begin
                                ();  j!=inversed_G.at(*i).end();  j++)
361                         {
362                             //We check if this node has already been visited
363                             if(already_visited.at(j->first) == 0)
364                             {//If we haven't found this node
365                                 if(optimal_W_graph.count(j->first) == 0)    //If it
                                        has no outgoing arc, we create it
366                                     optimal_W_graph[j->first][*i] = j->second;
367                                 //We also include the node in this two containers.
368                                 bifurcations_next.push_back(j->first);
369                                 already_visited[j->first] = 1;
370                             }
371                         }
372
373                 //We copy the nodes of bifurcations found in this step and start
                        again with them.
374                 bifurcations = bifurcations_next;
375                 bifurcations_next.clear();
376             }
377         }//End of snippet to find the optimal W_graph
378
379         #ifdef INFO
380         #ifdef BIG_INFO
381             std::cout << "The k=" << k << " optimal W graph is: " << endl;
382             for(map_of_transitions_type::iterator i = optimal_W_graph.begin(); i!=
                    optimal_W_graph.end(); i++)
383                 for(std::map<int,double>::iterator j = i->second.begin(); j!=i->
                        second.end(); j++)
384                     std::cout << i->first << " to " << j->first << " with " << j->
                            second << endl;
385         #endif // BIG_INFO
386         #endif // INFO
387
388         //We can take now the value of delta[k]
389         delta = arcs_removed_from_M.at(minimum_weight_arc.first).at(
                minimum_weight_arc.second);
390
391         //Here we would also take the value of A_k. But we set it as 1 for all k.
392
393         #ifdef INFO
394         std::cout << "The found value of Delta_" << k << " is " << delta << ".\tEnd
                of the step. Increasing k." << endl;
395         #endif // INFO
396
397         //Finally, we write in files all the computed data
398         ofstream output_file; //With just one stream we will open and close all the
                files.
399
400         output_file.open("delta.txt", std::ofstream::app);
401         output_file << delta << "\n";
402         output_file.close();
403
404         output_file.open("disappearing_sinks.txt", std::ofstream::app);
405         output_file << disappearing_sink << "\n";
406         output_file.close();
407
408         output_file.open("absorbing_sinks.txt", std::ofstream::app);
409         output_file << absorbing_sink << "\n";
410         output_file.close();
411
```

```
412             output_file.open("quasi_invariant_sets.txt", std::ofstream::app);
413             for(vector<int>::iterator i=quasi_invariant_set.begin(); i!=
                    quasi_invariant_set.end(); i++)
414                 output_file << *i << "\n";
415             output_file << "\n";
416             output_file.close();
417 
418             output_file.open("exit_arcs.txt", std::ofstream::app);
419             output_file << exit_arc.first << "\t" << exit_arc.second << "\n";
420             output_file.close();
421 
422             /*output_file.open("W_graphs.txt", std::ofstream::app);
423             for(map_of_transitions_type::iterator i = optimal_W_graph.begin(); i!=
                    optimal_W_graph.end(); i++)
424                 for(map<int,double>::iterator j = i->second.begin(); j!=i->second.end();
                        j++)
425                     output_file << i->first << "\t" << j->first << "\t" << j->second <<
                            "\n";
426             output_file << "\n";
427             output_file.close();*/
428 
429             //We decrease the value of k and go to the next iteration.
430             k = k-1;
431         }
432         else
433         {
434             //We increase the variable which stores the number of cycles found
435             number_of_cycles_found++;
436 
437             #ifdef MINIMAL_INFO
438             #ifndef INFO
439                 std::cout << "Number of cycles found: " << number_of_cycles_found << "."
                        << endl;
440             #endif // INFO
441             #endif // MINIMAL_INFO
442 
443             #ifdef INFO
444                 std::cout << "It does! We look for the cycle which has been formed." <<
                        endl;
445                 std::cout << "Number of cycles found: " << number_of_cycles_found << "."
                        << endl;
446             #endif // INFO
447 
448             //A cycle is created. We have to detect it. We take a vector to store it.
449             std::vector<int> detected_cycle;
450 
451             //Now we detect it
452             detect_cycle(arcs_removed_from_M, minimum_weight_arc.first,
                    minimum_weight_arc.second, detected_cycle);
453 
454             #ifdef INFO
455             std::cout << "The detected cycle has the following vertexes: \n";
456             for(std::vector<int>::iterator i = detected_cycle.begin(); i!=detected_cycle.
                    end(); i++)
457                 std::cout << "Detected cycle: " << *i << endl;
458             std::cout << "Starting the weight update process." << endl;
459             #endif // INFO
460 
461             //We update the weights of the arcs which go out of every vertex in the cycle
462             //We make a loop through the cycle
463             for(std::vector<int>::iterator i = detected_cycle.begin(); i!=detected_cycle.
                    end(); i++)
```

```
464                 if(*i != minimum_weight_arc.first) //We don't take the case i = x
465                 {
466                     //we search for the minimal arc in every set B_i
467                     double weight_minimum_arc_Bi = std::get<2>(min_arc.at(*i-1)); //We
                            are going to use a variable to store the minimum weight for every
                             set B_i
468
469                     //We proceed to the update
470                     #ifdef BIG_INFO
471                         std::cout << "The values before the update are the following:" <<
                                 endl;
472                         for(std::vector<int>::iterator i_Bi=Bi_indexes.at(*i - 1).begin()
                            ; i_Bi!=Bi_indexes.at(*i - 1).end(); i_Bi++)
473                             for(std::map<int,double>::iterator j = nodes.at(*i_Bi).begin
                                (); j!=nodes.at(*i_Bi).end(); j++)
474                                 std::cout << "Before update: " << *i_Bi << " to " << j->
                                    first << " has " << nodes[*i_Bi][j->first] << endl;
475                         std::cout << "Updating by: " << arcs_removed_from_M.at(
                            minimum_weight_arc.first).at(minimum_weight_arc.second)
476                                                     -  weight_minimum_arc_Bi <<
477                                                     "\t= XY: " << arcs_removed_from_M.at(
                                                        minimum_weight_arc.first).at(
                                                        minimum_weight_arc.second) <<
478                                                     "\t- Min: " << weight_minimum_arc_Bi
                                                        << endl;
479                     #endif // BIG_INFO
480
481                     //We make a loop through every subset of B_i
482                     for(std::vector<int>::iterator i_Bi=Bi_indexes.at(*i - 1).begin();
                            i_Bi!=Bi_indexes.at(*i - 1).end(); i_Bi++)
483                         for(std::map<int,double>::iterator j = nodes.at(*i_Bi).begin(); j
                            !=nodes.at(*i_Bi).end(); j++)
484                             nodes[*i_Bi][j->first] += arcs_removed_from_M.at(
                                minimum_weight_arc.first).at(minimum_weight_arc.second)
485                                                     -  weight_minimum_arc_Bi;
486
487                     #ifdef BIG_INFO
488                         std::cout << "The values after the update are the following:" <<
                                endl;
489                         for(std::vector<int>::iterator i_Bi=Bi_indexes.at(*i - 1).begin()
                            ; i_Bi!=Bi_indexes.at(*i - 1).end(); i_Bi++)
490                             for(std::map<int,double>::iterator j = nodes.at(*i_Bi).begin
                                (); j!=nodes.at(*i_Bi).end(); j++)
491                                 std::cout << "After update: " << *i_Bi << " to " << j->
                                    first << " has " << nodes[*i_Bi][j->first] << endl;
492                     #endif // BIG_INFO
493
494                 }//Here we have ended the update
495
496             #ifdef INFO
497                 std::cout << "Update finished. Merging Bi sets..." << endl;
498             #endif // INFO
499
500             //We have to merge the B_i sets for all i in C
501             {std::vector<int> merged_Bi; //We are going to construct the merged Bi set
502             for(std::vector<int>::iterator i = detected_cycle.begin(); i!=detected_cycle.
                    end(); i++)
503                 for(std::vector<int>::iterator i_Bi = Bi_indexes.at(*i - 1).begin(); i_Bi
                        !=Bi_indexes.at(*i - 1).end(); i_Bi++)
504                     if(std::find(merged_Bi.begin(), merged_Bi.end(), *i_Bi) == merged_Bi.
                            end())
505                         merged_Bi.push_back(*i_Bi);
```

```
506
507             //Now it is constructed. We have to make every Bi set for i in C equal to
                    this set.
508             for(std::vector<int>::iterator i = detected_cycle.begin(); i!=detected_cycle.
                    end(); i++)
509             {
510                 //Bi_indexes.at(*i - 1).clear();
511                 Bi_indexes[*i - 1] = merged_Bi;
512             }}//End of merging Bi sets.
513
514             #ifdef INFO
515             #ifdef BIG_INFO
516                 std::cout << "The Bi sets are the following:" << endl;
517                 for(std::vector< vector<int> >::iterator i = Bi_indexes.begin(); i!=
                        Bi_indexes.end(); i++)
518                 {
519                     std::cout << "Set B_" << (*i).at(0) <<  " has\t";
520                     for(std::vector<int>::iterator j = (*i).begin(); j!=(*i).end(); j++)
521                         std::cout << *j << "\t";
522                     std::cout << endl;
523                 }
524             #endif // BIG_INFO
525                 std::cout << "Merging the cycles\n";
526             #endif // INFO
527
528             //We need a vector in which we will merge the sets C(i)
529             std::vector<int> merged_cycles;
530
531             //We make a loop to get every vertex of the detected_cycle
532             for(std::vector<int>::iterator i=detected_cycle.begin(); i!=detected_cycle.
                    end(); i++)
533                 //We go to the (vector) component of cycles of that vertex and copy it
                        into merged_cycles.
534                 for(std::vector<int>::iterator j=cycles_i.at(*i - 1).begin(); j!=cycles_i
                        .at(*i - 1).end(); j++)
535                     if(std::find(merged_cycles.begin(), merged_cycles.end(), *j) ==
                            merged_cycles.end()) //We assure that there are no repeated
                            elements in merged_cycles
536                         merged_cycles.push_back(*j);
537
538             #ifdef INFO
539             std::cout << "The merged cycle is:" << endl;
540             for(std::vector<int>::iterator i=merged_cycles.begin(); i!=merged_cycles.end
                    (); i++)
541                 std::cout << *i << "\t";
542             std::cout << endl;
543             std::cout << "Deleting inadequate arcs..." << endl;
544             #endif // INFO
545
546             //Here we will delete those arcs whose vertexes are in C' and are minimal in
                    B.
547             //We need a variable for the minimum weight found
548             double minimum_weight = 999999;
549
550             //And another variable for the arc with that weight
551             std::pair<int,int> minimum_weight_arc_Cprime;
552
553             //We have a loop which breaks if the minimum arc outgoing from a vertex in B
                    is not in C'
554             while(1!=0)
555             {
556                 //We loop through the detected_cycle to get the minimum arc
```

```
557                 for(std::vector<int>::iterator i=detected_cycle.begin(); i!=
                        detected_cycle.end(); i++)
558                     //For every vertex in the detected_cycle, we go to the set B_i (this
                            is equivalent to B)
559                     for(std::vector<int>::iterator i_Bi = Bi_indexes.at(*i - 1).begin();
                            i_Bi != Bi_indexes.at(*i - 1).end(); i_Bi++)
560                         //For every i in the set Bi_indexes, we go to the set of outgoing
                                arcs from it
561                         if(nodes.count(*i_Bi) >= 1)
562                         for(std::map<int,double>::iterator j=nodes.at(*i_Bi).begin(); j!=
                            nodes.at(*i_Bi).end(); j++)
563                         {
564                             #ifdef DEGENERACY_WARNING
565                             if(j->second == minimum_weight)
566                                 std::cout << "WARNING: Degeneracy found at searching for
                                        the minimum outgoing arc not in C'." << endl;
567                             #endif // DEGENERACY_WARNING
568
569                             if(j->second < minimum_weight) //If a lesser value is found
570                             {
571                                 //We update the minimum value
572                                 minimum_weight = j->second;
573                                 //And the found minimum arc
574                                 minimum_weight_arc_Cprime.first = *i_Bi;
575                                 minimum_weight_arc_Cprime.second = j->first;
576                             }
577                         }
578
579                 //We now have to check whether the two vertexes of the
                        minimum_weight_arc_Cprime are on C'
580                 int check = 0; //This variable serves as a value for checking
581
582                 //We go through the set C'. If we find the vertexes of the
                        minimum_weight_arc_Cprime, we sum 1 to check
583                 for(std::vector<int>::iterator i=merged_cycles.begin(); i!=merged_cycles.
                        end() && check < 2; i++)
584                     if(minimum_weight_arc_Cprime.first == *i || minimum_weight_arc_Cprime
                            .second == *i)
585                         check ++;   //If the two vertexes are found, check will take
                                value 2
586
587                 if(check==2)    //If the two vertexes of the arc belong to C'
588                 {               //We remove the arc from B
589                     nodes.at(minimum_weight_arc_Cprime.first).erase(
                            minimum_weight_arc_Cprime.second);
590
591                     //We reset the variable which stores the minimum weight
592                     minimum_weight = 999999;
593
594                     #ifdef INFO
595                     std::cout << "Erasing arc " << minimum_weight_arc_Cprime.first << "
                            to " << minimum_weight_arc_Cprime.second << endl;
596                     #endif // INFO
597                 }
598                 else    //If they do not, we exit from the loop
599                     break;
600             }
601
602             #ifdef INFO
603             std::cout << "Making cycles C(i) equal to C'..." << endl;
604             #endif // INFO
605
```

```
606             //We have to make all the C(i) cycles as C', for all i in C'
607             for(std::vector<int>::iterator i=merged_cycles.begin(); i!=merged_cycles.end
                    (); i++)
608             //For every C(i) set with i in C'
609                 cycles_i.at(*i-1) = merged_cycles; //Copy the merged_cycles into every
                        cycles_i
610
611
612             #ifdef INFO
613             std::cout << "Removing the minimum arc from B and adding it to M..." << endl;
614             #endif // INFO
615
616             //Finally we delete the minimum arc from B and add it to M
617
618             //we search for the minimal arc in every set B_i
619             double weight_minimum_arc_Bi = 9999999; //We are going to use a variable to
                    store the minimum weight for every set B_i
620             int index_minimum_arc_Bi; //This will be the second index of the minimum arc
                    outgoing from i
621             pair<int,int> indexes_absolute_minimum_arc_Bi; //This will be the minimum arc
                     from B
622
623             //We make a loop through the different subsets of B_i
624             for(std::vector<int>::iterator i=merged_cycles.begin(); i!=merged_cycles.end
                    (); i++)
625             for(std::vector<int>::iterator j=Bi_indexes.at(*i - 1).begin(); j!=Bi_indexes
                    .at(*i - 1).end(); j++)
626             {
627                 if(nodes.count(*j) >= 1)
628                 if(nodes.at(*j).size() != 0)
629                 {
630                     #ifdef DEGENERACY_WARNING
631                     if(nodes.at(*j).at(index_minimum_arc_Bi) < weight_minimum_arc_Bi)
632                         std::cout << "WARNING: Degeneracy found at searching for the
                                minimum outgoing arc in B_i." << endl;
633                     #endif // DEGENERACY_WARNING
634
635                     //We search for the index of the minimal arc in the subset of B_i
                            which is the set of arcs outgoing from i
636                     index_minimum_arc_Bi = minimal_arc(nodes.at(*j));
637
638                     //If the minimal arc for this subset of B_i is lesser than the by now
                             found minimal weight of B_i
639                     if(nodes.at(*j).at(index_minimum_arc_Bi) < weight_minimum_arc_Bi)
640                     {
641                         //We save this new minimum.
642                         weight_minimum_arc_Bi = nodes.at(*j).at(index_minimum_arc_Bi);
643                         indexes_absolute_minimum_arc_Bi.first = *j;
644                         indexes_absolute_minimum_arc_Bi.second = index_minimum_arc_Bi;
645                     }
646                 }
647             }
648
649             #ifdef INFO
650                 std::cout << "The arc added to M is: " << indexes_absolute_minimum_arc_Bi
                        .first << " to " << indexes_absolute_minimum_arc_Bi.second << " with
                        " << weight_minimum_arc_Bi << endl;
651             #endif // INFO
652
653             //We also insert this arc in min_arc(i) for all i in C'
654             for(std::vector<int>::iterator i=merged_cycles.begin(); i!=merged_cycles.end
                    (); i++)
```

```
655                 min_arc.at(*i - 1) = std::make_tuple(indexes_absolute_minimum_arc_Bi.
                        first, indexes_absolute_minimum_arc_Bi.second, weight_minimum_arc_Bi)
                        ;
656
657             //We add it
658             Minimum_outgoing_arcs[indexes_absolute_minimum_arc_Bi.first][
                    indexes_absolute_minimum_arc_Bi.second] = weight_minimum_arc_Bi;
659
660             //And remove it from B
661             nodes.at(indexes_absolute_minimum_arc_Bi.first).erase(
                    indexes_absolute_minimum_arc_Bi.second);
662
663             #ifdef INFO
664                 std::cout << "End of the step." << endl;
665             #endif // INFO
666         } //End of if-else
667     }//End of main cycle (end while)
668     return 1; //Return a success.
669 }
670
671 //This function searches for the arc of minimum weight for a given node.
672 int minimal_arc(std::map<int,double> nodes)
673 {
674     double minimum_weight = 9999999; //A variable to keep track of the ~
675     int found_index = 0;    //The index found at every step. If it's 0 at the end, that
            means there was no minimal arc. If that makes any sense at all.
676
677     //We make a loop through the given map
678     for(std::map<int,double>::iterator i=nodes.begin(); i!=nodes.end(); i++)
679     {
680         #ifdef DEGENERACY_WARNING
681         if(i->second == minimum_weight)
682             std::cout << "WARNING: Degeneracy found at searching for the minimal arc." <<
                    endl;
683         #endif // DEGENERACY_WARNING
684         if(i->second < minimum_weight)  //For every entry we check whether it is lesser
                than the last minimum found.
685         {
686             //If it is, we change our control variables accordingly
687             minimum_weight = i->second;
688             found_index = i->first;
689         }
690     }
691
692
693     if(found_index == 0) cout << "\n\nThere is a zero index for the minimal_weight\n\n";
694     //After the loop, we should have the index of the minimal arc, which is then returned
            .
695     return found_index;
696 }
697
698 //A function which returns 1 if the vertexes x and y are in the same connected component
        of graph, 0 otherwise.
699 bool are_in_same_connected_component(const map_of_transitions_type &graph, int x, int y,
        int &sink_of_x, std::vector<int> &connected_component)
700 {
701     //First we assure that the vector is empty
702     connected_component.clear();
703
704     //In a list we will have record of the bifurcations we find. It shall be a list
            because we want to erase it in constant time.
705     std::list<int> bifurcations;
```

```
706
707     //The sink of the connected component of x.
708     sink_of_x = sink_of_connected_component(graph, x);
709
710     //(The first element in the connected component as well as) the first bifurcation
            will be the sink.
711     bifurcations.push_back(sink_of_x);
712
713     //We have to insert the sink in the connected_component vector
714     connected_component.push_back(sink_of_x);
715
716     //A variable to know at every moment which node we are analyzing
717     int current_node;
718
719     while(1!=0)
720     {
721         if(bifurcations.size() != 0)
722         //We put the current_node in the last bifurcation we have noted
723             current_node = bifurcations.back();
724         else
725             break; //If there aren't any bifurcations left, we have ended.
726
727         //We make a sweep through the graph
728         for(map_of_transitions_type::const_iterator i = graph.begin(); i!=graph.end(); i
                ++)
729             for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.
                    end(); j++)
730                 if(j->first == current_node)    //If the endpoint of an arc is the
                        current_node
731                 {
732                     //We note the found arc in our two containers.
733                     connected_component.push_back(i->first);
734                     bifurcations.push_back(i->first);
735                 }
736
737         //We end the step by erasing the track of the analyzed node from the list of
                bifurcations.
738         bifurcations.remove(current_node);
739     }
740
741     //Finally we detect if y is in the connectad_component of x.
742     for(std::vector<int>::iterator i = connected_component.begin(); i!=
            connected_component.end(); i++)
743         if(*i == y)
744             return 1;
745
746     return 0;   //If we have not found y in the connected component of x, then we return
            0.
747 }
748
749 //Searches for the sink of the connected component which node belongs to.
750 int sink_of_connected_component(map_of_transitions_type graph, int node)
751 {
752     //The sink of the connected component of node. Initialized in node
753     int sink = node;
754
755     do //We will find the sink of the connected component of node here
756     {
757         if(graph.count(sink) == 0)  //If there aren't any outgoing arcs from sink
758             break;  //We exit from the loop, because we have already found the sink
759         else
```

```
760             sink = graph.at(sink).begin()->first; //We now consider as the sink the first
                    (and we hope that unique) outgoing arc from sink
761     }while(1!=0); //We can only get out of the loop if we find the sink. There MUST be a
            sink...
762
763     return sink;
764 }
765
766 //This functions gives a vector with all the vertexes in the connected component of node
767 void find_connected_component(map_of_transitions_type graph, int node, vector<int> &
        connected_component)
768 {
769     //First we assure that the vector is empty
770     connected_component.clear();
771
772     //We now find the sink
773     int sink = sink_of_connected_component(graph, node);
774
775     //In a list we will have record of bifurcations we find. It shall be a list because
            we want to erase it in constant time.
776     std::list<int> bifurcations;
777     bifurcations.push_back(sink);
778
779     //We have to insert the sink in the connected_component vector
780     connected_component.push_back(sink);
781
782     //A variable to know at every moment which node we are analyzing
783     int current_node;
784
785     while(1!=0)
786     {
787         if(bifurcations.size() != 0)
788         //We put the current_node in the last bifurcation we have noted
789             current_node = bifurcations.back();
790         else
791             break; //If there aren't any bifurcations left, we have ended.
792
793         //We make a sweep through the graph
794         for(map_of_transitions_type::iterator i = graph.begin(); i!=graph.end(); i++)
795             for(std::map<int,double>::iterator j = i->second.begin(); j!=i->second.end();
                    j++)
796                 if(j->first == current_node)    //If the endpoint of an arc is the
                        current_node
797                 {
798                     //We note the found arc in our two containers.
799                     connected_component.push_back(i->first);
800                     bifurcations.push_back(i->first);
801                 }
802
803         //We end the step by erasing the track of the analyzed node from the list of
                bifurcations.
804         bifurcations.remove(current_node);
805     }
806 }
807
808 void detect_cycle(const map_of_transitions_type &graph, const int x, const int y, std::
        vector<int> &cycle)
809 {
810     int current_node = y;
811
812     //Make sure that the cycle starts void
813     cycle.clear();
```

```
814
815     //We need to keep track of the deadpoints we find as well as the already visited
            nodes in every iteration
816     std::vector<int> deadpoints;
817
818     do  //Make this loop until we have reached the end
819     {
820         //If there are no outgoing arcs from here,
821         if(graph.count(current_node) == 0)
822         {   //We put this node in the deadpoints vector and start over.
823             if(std::find(deadpoints.begin(), deadpoints.end(), current_node) ==
                    deadpoints.end())
824                 deadpoints.push_back(current_node);
825             current_node = y;
826             cycle.clear();
827         }
828         else //In any other case,
829         {
830             //We get an iterator to the current_node
831             map_of_transitions_type::const_iterator i = graph.find(current_node);
832             std::map<int,double>::const_iterator j;
833             //We go through the inner map
834             for(j = i->second.begin(); j!=i->second.end(); j++)
835                 //If it is not a deadpoint nor an already visited node
836                 if(std::find(deadpoints.begin(), deadpoints.end(), j->first) ==
                        deadpoints.end() &&
837                    std::find(cycle.begin(), cycle.end(), j->first) == cycle.end())
838                 {   //We get the current_node there.
839                     current_node = j->first;
840                     cycle.push_back(current_node);
841                     break; //And exit the loop.
842                 }
843
844             if(j == i->second.end()) //If we reached the end of the inner map, we start
                    over.
845             {
846                 if(std::find(deadpoints.begin(), deadpoints.end(), i->first) ==
                        deadpoints.end())
847                     deadpoints.push_back(i->first);
848                 cycle.clear();
849                 current_node = y;
850             }
851         }
852     }while(current_node!=x);
853     //We have to insert the very last component of the cycle, which closes it.
854     cycle.push_back(y);
855 }
856
857 void read_prepared_simulations(map_of_transitions_type &nodes)
858 {
859     FILE* infile;
860
861     infile = fopen("nodes_map.bin", "rb");
862
863     double mapsize;
864
865     fread(&mapsize, sizeof(mapsize), 1, infile);
866
867     for(int i = 0; i<mapsize; i++)
868     {
869         int first_element;
870         fread(&first_element, sizeof(int), 1, infile);
```

```
871         double inner_mapsize;
872         fread(&inner_mapsize, sizeof(inner_mapsize), 1,  infile);
873         for(int j = 0; j<inner_mapsize; j++)
874         {
875             int second_element;
876             double third_element;
877             fread(&second_element, sizeof(int), 1, infile);
878             fread(&third_element, sizeof(double), 1, infile);
879             nodes[first_element][second_element] = third_element;
880         }
881         if(i%1000==0) std::cout << "Completed " << (i/mapsize)*100 << "%%\r";
882     }
883     fclose(infile);
884 }
885
886 void write_prepared_simulations(const map_of_transitions_type &nodes)
887 {
888     FILE* outfile;
889
890     outfile = fopen("nodes_map.bin", "wb");
891
892     double mapsize = nodes.size();
893
894     fwrite(&mapsize, sizeof(mapsize), 1, outfile);
895
896     for(map_of_transitions_type::const_iterator i = nodes.begin(); i!=nodes.end(); i++)
897     {
898         int first_element = i->first;
899         fwrite(&first_element, sizeof(int), 1, outfile);
900         double inner_mapsize = i->second.size();
901         fwrite(&inner_mapsize, sizeof(inner_mapsize), 1,  outfile);
902         for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.end
                ); j++)
903         {
904             int second_element = j->first;
905             double third_element = j->second;
906             fwrite(&second_element, sizeof(int), 1, outfile);
907             fwrite(&third_element, sizeof(double), 1, outfile);
908         }
909     }
910     fclose(outfile);
911 }
912
913 //This function creates a simple map for testing purposes.
914 void create_testing_map(map_of_transitions_type &nodes)
915 {//Fig 5 Cameron(2017).
916     nodes[1][2] = 1;
917     nodes[1][3] = 5;
918     nodes[1][4] = 2.5;
919     nodes[1][5] = 2.7;
920
921     nodes[2][3] = 3;
922     nodes[2][1] = 10;
923
924     nodes[3][1] = 2;
925     nodes[3][2] = 2.6;
926     nodes[3][4] = 3.2;
927
928     nodes[4][1] = 0.5;
929     nodes[4][3] = 1.5;
930     nodes[4][5] = 3;
931
```

```
932     nodes[5][1] = 7;
933     nodes[5][4] = 6;
934 }
935 
936 //A simple function to compute the total size of a map, not only that of its outer part.
937 //Needed to a good computation of the size of a map, since by erasing its inner elements
        the outer size is not decreased.
938 int total_size_of_map(const map_of_transitions_type &nodes)
939 {
940     int total_size = 0;
941     for(map_of_transitions_type::const_iterator i=nodes.begin(); i!=nodes.end(); i++)
942         total_size += i->second.size();
943     return total_size;
944 }
945 
946 void save_current_step(const map_of_transitions_type &nodes,
947                        const map_of_transitions_type &Minimum_outgoing_arcs,
948                        const map_of_transitions_type &arcs_removed_from_M,
949                        const map_of_transitions_type &optimal_W_graph,
950                        const std::vector< std::vector<int> > &cycles_i,
951                        const std::vector< std::vector<int> > &Bi_indexes,
952                        const std::vector< std::tuple<int,int,double> > &min_arc,
953                        const int k)
954 {
955     ofstream output_file;
956     output_file.open("saved_nodes.txt");
957 
958     for(map_of_transitions_type::const_iterator i = nodes.begin(); i!=nodes.end(); i++)
959         for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.end
                ); j++)
960             output_file << i->first << "\t" << j->first << "\t" << j->second << "\n";
961 
962     output_file.close();
963 
964 
965     output_file.open("saved_M.txt");
966 
967     for(map_of_transitions_type::const_iterator i = Minimum_outgoing_arcs.begin(); i!=
            Minimum_outgoing_arcs.end(); i++)
968         for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.end
                ); j++)
969             output_file << i->first << "\t" << j->first << "\t" << j->second << "\n";
970 
971     output_file.close();
972 
973 
974     output_file.open("saved_G.txt");
975 
976     for(map_of_transitions_type::const_iterator i = arcs_removed_from_M.begin(); i!=
            arcs_removed_from_M.end(); i++)
977         for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.end
                ); j++)
978             output_file << i->first << "\t" << j->first << "\t" << j->second << "\n";
979 
980     output_file.close();
981 
982 
983     output_file.open("saved_W_graph.txt");
984 
985     for(map_of_transitions_type::const_iterator i = optimal_W_graph.begin(); i!=
            optimal_W_graph.end(); i++)
```

```
986         for(std::map<int,double>::const_iterator j = i->second.begin(); j!=i->second.end
                ); j++)
987             output_file << i->first << "\t" << j->first << "\t" << j->second << "\n";
988
989     output_file.close();
990
991
992     output_file.open("saved_cycles.txt");
993
994     for(std::vector< std::vector<int> >::const_iterator i=cycles_i.begin(); i!=cycles_i.
            end(); i++)
995     {
996         output_file << (*i).size() << "\n";
997         for(std::vector<int>::const_iterator j = (*i).begin(); j!=(*i).end(); j++)
998             output_file << (*j) << "\n";
999     }
1000
1001    output_file.close();
1002
1003
1004    output_file.open("saved_Bi.txt");
1005
1006    for(std::vector< std::vector<int> >::const_iterator i=Bi_indexes.begin(); i!=
            Bi_indexes.end(); i++)
1007    {
1008        output_file << (*i).size() << "\n";
1009        for(std::vector<int>::const_iterator j = (*i).begin(); j!=(*i).end(); j++)
1010            output_file << (*j) << "\n";
1011    }
1012
1013    output_file.close();
1014
1015
1016    output_file.open("saved_min_arc.txt");
1017
1018    for(std::vector< std::tuple<int,int,double> >::const_iterator i=min_arc.begin(); i!=
            min_arc.end(); i++)
1019    {
1020        output_file << std::get<0>(*i) << "\t" << std::get<1>(*i) << "\t" << std::get
                <2>(*i) << "\n";
1021    }
1022
1023    output_file.close();
1024
1025
1026    output_file.open("saved_k.txt");
1027
1028    output_file << k;
1029
1030    output_file.close();
1031 }
1032
1033 void load_current_step(map_of_transitions_type &nodes,
1034                        map_of_transitions_type &Minimum_outgoing_arcs,
1035                        map_of_transitions_type &arcs_removed_from_M,
1036                        map_of_transitions_type &optimal_W_graph,
1037                        std::vector< std::vector<int> > &cycles_i,
1038                        std::vector< std::vector<int> > &Bi_indexes,
1039                        std::vector< std::tuple<int,int,double> > &min_arc,
1040                        int &k)
1041 {
1042     ifstream input_file;
```

```
1043     input_file.open("saved_nodes.txt");
1044 
1045     int first_state;
1046     int second_state;
1047     double weight;
1048 
1049     nodes.clear();
1050     while(!input_file.eof())
1051     {
1052         input_file >> first_state >> second_state >> weight;
1053         nodes[first_state][second_state] = weight;
1054     }
1055     input_file.close();
1056 std::cout << "nodes read\n";
1057 
1058     input_file.open("saved_M.txt");
1059 
1060     Minimum_outgoing_arcs.clear();
1061     while(!input_file.eof())
1062     {
1063         input_file >> first_state >> second_state >> weight;
1064         Minimum_outgoing_arcs[first_state][second_state] = weight;
1065     }
1066 
1067     input_file.close();
1068 std::cout << "M read\n";
1069 
1070     input_file.open("saved_G.txt");
1071 
1072     arcs_removed_from_M.clear();
1073     while(!input_file.eof())
1074     {
1075         input_file >> first_state >> second_state >> weight;
1076         arcs_removed_from_M[first_state][second_state] = weight;
1077     }
1078 
1079     input_file.close();
1080 std::cout << "G read\n";
1081 
1082     input_file.open("saved_W_graph.txt");
1083 
1084     optimal_W_graph.clear();
1085     while(!input_file.eof())
1086     {
1087         input_file >> first_state >> second_state >> weight;
1088         optimal_W_graph[first_state][second_state] = weight;
1089     }
1090 
1091     input_file.close();
1092 std::cout << "W_graph read\n";
1093 
1094     input_file.open("saved_cycles.txt");
1095 
1096     int size_of_vector = 0;
1097     int element;
1098 
1099     cycles_i.clear();
1100 
1101     while(!input_file.eof())
1102     {
1103         vector<int> aux_vector;
1104         input_file >> size_of_vector;
```

```
1105         for(int i=0; i<size_of_vector; i++)
1106         {
1107             input_file >> element;
1108             aux_vector.push_back(element);
1109         }
1110         cycles_i.push_back(aux_vector);
1111     }
1112 
1113     input_file.close();
1114 std::cout << "cycles read\n";
1115 
1116     input_file.open("saved_Bi.txt");
1117     size_of_vector = 0;
1118 
1119     Bi_indexes.clear();
1120     while(!input_file.eof())
1121     {
1122         vector<int> aux_vector;
1123         input_file >> size_of_vector;
1124         for(int i=0; i<size_of_vector; i++)
1125         {
1126             input_file >> element;
1127             aux_vector.push_back(element);
1128         }
1129         Bi_indexes.push_back(aux_vector);
1130     }
1131 
1132     input_file.close();
1133 std::cout << "Bi_indexes read\n";
1134 
1135     input_file.open("saved_min_arc.txt");
1136 
1137     min_arc.clear();
1138     while(!input_file.eof())
1139     {
1140         input_file >> first_state >> second_state >> weight;
1141         min_arc.push_back(std::make_tuple(first_state, second_state, weight));
1142     }
1143 
1144     input_file.close();
1145 std::cout << "min_arc read\n";
1146 
1147     input_file.open("saved_k.txt");
1148 
1149     input_file >> k;
1150 
1151     input_file.close();
1152 std::cout << "k read\n";
1153 }
```