Anexo A: Identificación de cúmulos en cualquier red. Generalización del algoritmo de Hoshen-Kopelman

A lo largo del trabajo se menciona en repetidas ocasiones el concepto de cúmulo, un grupo de partículas que se encuentran en contacto unas con otras aislado del resto de la red.

Para obtener los resultados de nuestro modelo y poder compararlos con los datos reales de cada país necesitamos una forma optimizada de poder distinguir los cúmulos que haya en cada momento en nuestra red, de forma que podamos estudiar su distribución por tamaño. Sin embargo, llevar a cabo esta tarea en redes complejas (como pueden ser la tridimensional o la red de Bethe) manteniendo los tiempos de ejecución de nuestro programa en cantidades razonables supone una gran complicación a nivel computacional.

En el caso de redes bidimensionales el algoritmo más optimizado para realizar esta tarea es el propuesto por Hoshen y Kopelman (1976). Tras ejecutar dicho algoritmo la red queda etiquetada, de forma que asigna a cada celda el número de cúmulo al que pertenece. Una vez etiquetada la red resulta muy sencillo realizar la estadística sobre ella.

El algoritmo realiza un barrido de la red comprobando en cada celda si las celdas situadas encima suyo o a su izquierda están ocupadas. De esta forma todos los elementos de la red quedan comprobados minimizando el número de comparaciones que hay que realizar (solo dos por celda).

Si ambas celdas están vacías a la celda que se está analizando, se le asigna un número una unidad superior al último número asignado (toma una etiqueta distinta a las ya empleadas anteriormente para designar otros cúmulos).

Si solo una de las celdas pertenece a un cúmulo, la celda a comprobar tomará su etiqueta, siendo parte de su mismo cúmulo.

Si ambas celdas están etiquetadas se asignará a todas las celdas en contacto el valor correcto que les corresponda según el cúmulo al que pertenezcan.

Esta última tarea mencionada se lleva a cabo en un segundo barrido que se hace a la red, en el que se reetiquetan las celdas de forma que el orden de etiquetado de los cúmulos sea el correcto.

Este algoritmo basa su optimización en minimizar el número de comprobaciones que se realizan en su ejecución, pero solo es aplicable a redes de dos dimensiones, en las que comprobando únicamente dos celdas por cada una que se quiere analizar se puede hacer un barrido completo de la red.

Para analizar casos más complejos tomamos una versión generalizada de este algoritmo (Futasi y Patzek, 2003; Popova, 2014).

El funcionamiento es similar a la versión original del algoritmo: se realiza un primer barrido de la red comprobando cuáles de los vecinos de la celda a analizar forman parte de un cúmulo. Si las celdas están vacías (o bien ninguna de ellas ha sido etiquetada todavía) la celda que estamos comprobando se marcará con una nueva etiqueta. Si alguna de las celdas está etiquetada se guarda la información de la menor de las etiquetas y posteriormente se procederá a un segundo barrido de reetiquetado. La diferencia con la versión original del algoritmo viene dada por el hecho de que ya no se comprueban únicamente las celdas superior e izquierda de la que estamos comprobando, sino que las celdas a comprobar vienen dadas por una matriz que llamaremos *matriz de vecinos*. Se trata de una matriz de tamaño V*N (V el número total de puntos de nuestra red y N es el número de vecinos de cada celda), en la que cada elemento (i,j) se

corresponde con el vecino j del elemento i de la red a analizar. Ésta matriz será la que determine la forma de nuestra red, definiendo el modo en el que sus elementos están conectados entre sí.

A continuación se encuentran las funciones de C correspondientes a la construcción de las diferentes matrices de vecinos en las redes que estudiamos.

Construcción de matrices de vecinos

1. Red unidimensional:

```
void obten_vecinos_1D(int n, float *vecinos, int *nodos)
{
    vecinos[0] = ( n==V-1 ? nodos[0] : nodos[n+1] );
    //Delante. Si es el ultimo elemento tomamos el primero
    vecinos[1] = ( n==0 ? nodos[V-1] : nodos[n-1] );
    //Detrás. Si es el primer elemento tomamos el último
}
```

2. Red bidimensional:

```
void obten_vecinos_2D(int n, float *vecinos, int *nodos)
{
//Si el elemento no tiene celda en alguno de sus lados
//como ocurre en los extremos de la red, esa celda se marca con -1
    vecinos[0] = ( n<L ? -1 : nodos[n-L] );         //N
    vecinos[1] = ( n%L==0 ? -1 : nodos[n-1] );      //W
    vecinos[2] = ( n>=V-L ? -1 : nodos[n+L] );      //S
    vecinos[3] = ( (n+1)%L==0 ? -1 : nodos[n+1] );  //E
}
```

3. Red tridimensional:

```
void obten_vecinos_3D(int n, float *vecinos, int *nodos)
{
    vecinos[0] = ( n%(X*Y)<X ? -1 : nodos[n-X] );             //Above
    vecinos[1] = ( n%X==0 ? -1 : nodos[n-1] );                //Left
    vecinos[2] = ( (n+X)%(X*Y)<X ? -1 : nodos[n+X] );         //Below
    vecinos[3] = ( (n+1)%X==0 ? -1 : nodos[n+1] );            //Right
    vecinos[4] = ( n<(X*Y) ? -1 : nodos[n-(X*Y)] );           //Front
    vecinos[5] = ( n>=(V-(X*Y)) ? -1 : nodos[n+(X*Y)] );      //Back
}
```

4. Red de Bethe:

```
void obten_vecinos_Bethe(float *vecinos, int num_hojas,
float matriz_vecinos[][N], int V)
{
    int i, i_aux, j, vecino_superior, vecino_inferior, n;
```

```
    //VECINOS DE LAS HOJAS
    vecino_superior = num_hojas;
    for(i=0; i<num_hojas; i+=(N-1)){ //N-1 elementos comparten vecino superior
        for(i_aux = i; i_aux<(i+(N-1)); i_aux++){
            matriz_vecinos[i_aux][0] = vecino_superior;
            for(j=1; j<N; j++) { //En las hojas solo hay contacto con 1 celda
                matriz_vecinos[i_aux][j] = -1;
            }
        }
        vecino_superior++;
    }

    //VECINOS DE POSICIONES INTERMEDIAS
    vecino_inferior = 0;
    for(i=num_hojas; i<(V-N-1); i+=(N-1)){
        for(i_aux = i; i_aux < (i+N-1); i_aux++){
            for(j=0; j<(N-1); j++){
                matriz_vecinos[i_aux][j] = vecino_inferior;
                vecino_inferior++;
                }
            matriz_vecinos[i_aux][N-1] = vecino_superior;
        }
        vecino_superior++;
    }

    //VECINOS DE LA PRIMERA CAPA (comparten contacto con el nodo central)
    for(i=V-N-1; i<(V-1); i++){
        for(j=0; j<(N-1); j++){
            matriz_vecinos[i][j] = vecino_inferior;
                vecino_inferior++;
            }
            matriz_vecinos[i][N-1] = V-1;
    }

    //VECINOS DEL NODO CENTRAL
    vecino_inferior = V-N-1;
    for(i = 0; i<N; i++){
        matriz_vecinos[V-1][i] = vecino_inferior + i;
    }
}
```

Algoritmo Hoshen-Kopelman generalizado

Presentamos a continuación la implementación en C del algoritmo de Hoshen-Kopelman generalizado para redes complejas. Los vectores que se emplearán son los siguientes:

- NodeS: vector de ocupación de la red. Cada elemento i toma valor 0 si la celda i se encuentra vacía y valor 1 si está ocupada.
- NodeNext: matriz de vecinos. Establece las conexiones entre los puntos que forman la red.

- NodeL: vector tras aplicar el algoritmo Hoshen-Kopelman. Será el vector que devolvamos a nuestro programa en el que cada elemento de la red lleve la etiqueta del cúmulo al que pertenece.

El resto de elementos que intervienen en el algoritmo quedan explicados como comentarios en el mismo:

```
int hoshen_kopelman_gen(int *NodeS, float NodeNext[][N], int *NodeL){

    int i, i_cn, j, flag=0, aux=0;

    int cn, Num_cl[N+1], NodeLPmin, max_cluster;

    for(i=0; i<V; i++)  NodeL[i]=0;

    int NodeLP[V], cluster = 0;  //NodeLP sirve para reetiquetar NodeL
    for(i=0; i<V; i++)  NodeLP[i]=0;

    int NodeNextL[N]; //Array para comprobar los elementos alrededor de un punto

    //ESCANEO DE LA RED

    for(i=0; i<V; i++){
        flag = 0;
        aux = 0;
        if(NodeS[i]!=0){
            for(j=0; j<N; j++) {
                if(NodeNext[i][j]!=-1){
                    NodeNextL[j]=NodeL[(int)NodeNext[i][j]];
                    //Se guardan los cúmulos de los vecinos
                    if(NodeS[(int)NodeNext[i][j]]==0)  aux++;
                    //Se comprueba si no hay vecinos
                    //(cada vez que un vecino es 0 aumenta aux)
                    else if(NodeNextL[j]!=0)    flag = 1;
                    //Se comprueba si algún vecino pertenece a un cúmulo
                    //(si pasa, flag pasa a valer 1)
                }else NodeNextL[j]=0;
                //Si NodeNext es -1 significa que no hay vecino
                //El cúmulo de esa posición será 0
            }

            if(aux==N){  //CASO 1: no hay vecinos (todos los NodeS de los vecinos
                         //son 0, aux alcanza el valor N)-->NUEVO CÚMULO
                cluster++;
                NodeL[i]=cluster;
                NodeLP[cluster]=cluster;
            } else if(flag==0){    //CASO 2: ningún vecino pertenece a un cúmulo
                                   //(flag NO ha cambiado a 1)-->NUEVO CÚMULO
                cluster++;
                NodeL[i]=cluster;
```

```
                NodeLP[cluster]=cluster;
            }else{          //CASO 3: algún vecino pertenece a un cúmulo
                            //--> toma la menor etiqueta de los vecinos
                for (cn=0; cn<N+1; cn++) Num_cl[cn] = 0;
                cn = 0;       //Num_cl guarda las etiquetas de los vecinos
                for(j=0; j<N; j++){
                  if(NodeNextL[j]!=0) Num_cl[++cn] = NodeLP[NodeNextL[j]];
                }        //La primera casilla de Num_cl está vacía (++cn)
                if(cn == 0){    //Si no hay cúmulos-->NUEVO CÚMULO
                    cluster++;
                    NodeL[i]=cluster;
                    NodeLP[cluster]=cluster;
                }else{      //Si hay cúmulos tomamos el menor índice
                    NodeLPmin = 1000000000; //NodeLPmin guarda la menor etiqueta
                    for(j=1; j<=cn; j++){       //La primera casilla está vacía
                                                //Comenzamos leyendo Num_cl[1]
                        if(NodeLP[Num_cl[j]]<NodeLPmin)
                             NodeLPmin = NodeLP[Num_cl[j]];
                    }
                    NodeL[i] = NodeLPmin;
                    for (i_cn=1; i_cn<=cn; i_cn++)
                         NodeLP[Num_cl[i_cn]] = NodeLPmin;
                         //Asignamos a todos los vecinos la menor etiqueta
                }
            }
        }
    }
    //FIN DEL ESCANEO

    //Reenumeramos las etiquetas DE NodeLP

    j = 1;
    for (i=1; i<=cluster; i++)
      if (NodeLP[i] == i)
        NodeLP[i] = j++;
      else
        NodeLP[i] = NodeLP[NodeLP[i]];

    //Reetiquetamos NodeL

    NodeLP[0] = 0;
    for (i=0; i<V; i++) {
        NodeL[i] = NodeLP[NodeL[i]];
    }

    return 1;
}
```

Referencias

Hoshen, J. and Kopelman, R. (1976). Percolation and cluster distribution. I. Cluster multiple labeling technique and critical concentration algorithm. *Physical Review B* 14(8), p. 3438.

Futasi, A. and Patzek, T.W. (2003). Extension of Hoshen–Kopelman algorithm to non-lattice environments. *Physica A* 321, p. 665-678.

Popova, H. (2014). An adaptation of the Hoshen-Kopelman cluster counting algorithm for honeycomb networks. *Serdica J. Computing* 8(4), p. 363-388.

Anexo B: Cálculo de pendientes en regiones "ley de potencias"

Uno de los objetivos de este TFG es comparar los resultados del modelo de dinámica de empresas con los datos estadísticos que cada país recopila sobre el tamaño de sus empresas, entendiendo por "tamaño" el número de empleados de cada una de ellas. La distribución tamaño-frecuencia de las empresas resulta ser, en todos los casos analizados, una ley de potencias con un exponente de entre -2.1 y -2.8 aproximadamente.

El modelo de dinámica de empresas que hemos propuesto ha sido resuelto en diferentes tipos de redes y en cada una de ellas se han obtenido distribuciones tamaño-frecuencia diferentes, aunque todas ellas tienen en común (en la versión agresiva de las reglas del autómata celular) la presencia de una zona con un comportamiento de tipo ley de potencias. Al contrario que la distribución tamaño-frecuencia de las empresas reales, este comportamiento no se extiende en el modelo hasta las empresas más grandes sino solo hasta un valor máximo, k_{max}, menor que la mayor empresa que el autómata puede generar (del orden del número de celdas de la malla). Además, es habitual, tanto en el modelo como en la realidad, que las empresas más pequeñas, $k \lesssim 5$, se desvíen ligeramente de una ley de potencias. Por tanto, una dificultad con la que nos hemos encontrado ha sido calcular con precisión el exponente de la "zona ley de potencias" en las distribuciones tamaño-frecuencia del modelo.

La técnica más común de calcular el exponente α de una ley de potencias, $p(k) \propto k^{\alpha}$, es mediante un ajuste lineal por mínimos cuadrados al logaritmo de p(k), $log(p) = c - \alpha log(k)$, en cuyo caso el exponente α es la pendiente de la recta que resulta de representar la distribución tamaño-frecuencia en escala doble logarítmica. Sin embargo, se sabe que dicho ajuste tiene sesgo y produce resultados erróneos (Goldstein et al., 2004; Newman, 2006; Bauke, 2007; White et al., 2008; Clauset et al., 2009; Corral et al., 2012; Deluca y Corral, 2013) y que el método recomendado en todos los casos es el de máxima verosimilitud (MLE: *maximum likelihood estimation*).

Por tanto éste será el método con el que realizaremos las estimaciones del exponente α, pero como las distribuciones tamaño-frecuencia del modelo a estudiar en este trabajo solo siguen una ley de potencias dentro del rango acotado por los valores mínimo, k_{min}, y máximo, k_{max}, del tamaño de las empresas, primero ha habido que delimitar este rango con comportamiento ley de potencias, que aquí hemos denominado como "zona ley de potencias". Es decir, hemos tenido que realizar el ajuste MLE a una distribución con las siguientes particularidades importantes:

1. Es una distribución discreta, que solo está definida para valores enteros del argumento (el número de trabajadores de cada empresa).
2. Es una distribución con una cota inferior, $k_{min} > 1$, por debajo de la cual no se cumple la ley de potencias.
3. Es una distribución con una cota superior, k_{max}, por encima de la cual la ley tampoco se cumple.

Estas particularidades han hecho que el procedimiento de ajuste haya sido bastante elaborado, como se describe a continuación.

Con las condiciones que se dan en nuestros conjuntos de datos, según el MLE (Bauke, 2007) el exponente α de la distribución a estudiar será aquel que maximice la siguiente función:

$$\mathcal{L}(\alpha) = -\alpha \left(\sum_{i=1}^{N} \ln(k_i) \right) - N \; ln \left(\zeta(\alpha, k_{min}, k_{max}) \right)$$

Donde k_i son los N elementos del conjunto de datos que forman nuestra distribución, k_{min} y k_{max} son respectivamente los puntos inicial y final de la región que sigue una ley de potencias (se explicará más adelante el método empleado para obtener sus valores) y $\zeta(\alpha, k_{min}, k_{max})$ es una función definida como la diferencia entre las funciones zeta de Hurwitz correspondientes a los valores k_{min} y k_{max}:

$$\zeta(\alpha, k_{min}, k_{max}) = \zeta(\alpha, k_{min}) - \zeta(\alpha, k_{max})$$

A su vez, el valor de la zeta de Hurwitz para unos valores dados de α y k es el siguiente:

$$\zeta(\alpha,\ k) = \sum_{i=0}^{\infty} \frac{1}{(i+k)^{\alpha}}$$

Es decir, es una modificación de la función zeta de Riemann, de forma que el sumatorio en lugar de realizarse entre los valores 0 y ∞, se da entre los valores k y ∞.

De este modo, realizamos un barrido de todos los posibles valores de α (en nuestro caso entre los valores -1.5 y -3 para asegurarnos de que el rango de interés queda cubierto) y se toma como correcto aquel que maximice la función $\mathcal{L}(\alpha)$ para nuestro conjunto de datos. El resultado es como el que se muestra en la Figura 1, siendo el valor que maximiza la curva el que tomaremos como valor correcto de α.

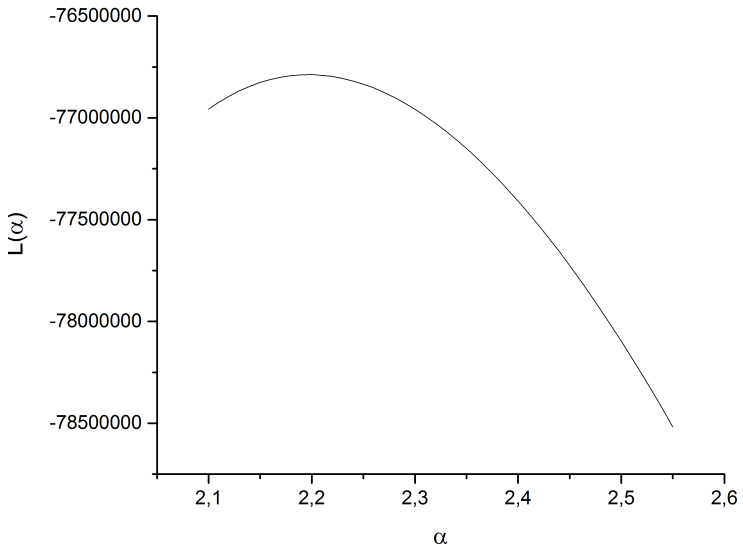


Figura 1: Representación de $\mathcal{L}(\alpha)$ en el MLE.

Para acotar la región que sigue una ley de potencias comenzamos hallando el límite superior de ésta. Para ello fijamos el valor de k_{min} en 1 (el punto inicial de la distribución de datos) y vemos cómo varía el valor de la pendiente en función de cuál sea el valor de k_{max}. El resultado obtenido en todos los modelos es de la forma representada en la Figura 2.

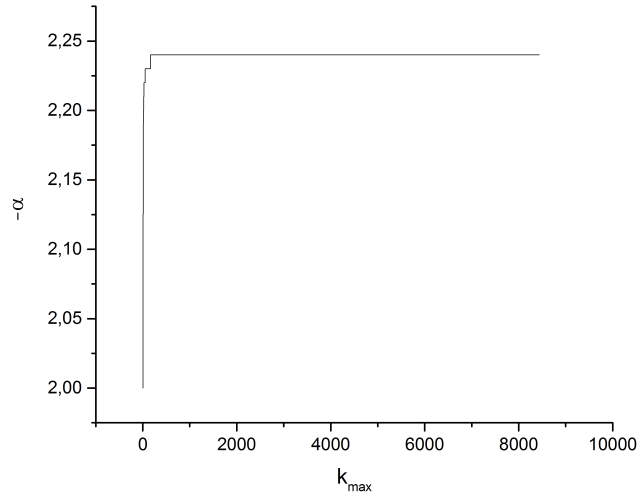


Figura 2: Variación del valor de la pendiente α en función del valor de k_{max} con k_{min}=1. Los datos representados son para la red 3D de lado 75, pero sigue la misma evolución en todas las redes.

Como se puede ver en la figura la tendencia de α es aumentar hasta alcanzar un máximo a partir del cual no se modifica. Tomaremos como valor correcto de k_{max} el primer valor en el que la pendiente haya alcanzado su máximo. En el caso de la Figura 2, el valor resulta ser 166.

A continuación repetimos el proceso al contrario: fijamos el valor de k_{max} como el obtenido en el paso anterior y estudiamos cómo cambia el valor de α en función del valor que tome k_{min} (que variará desde 1 hasta k_{max}-1). De este modo se obtienen datos como los recogidos en la Figura 3.

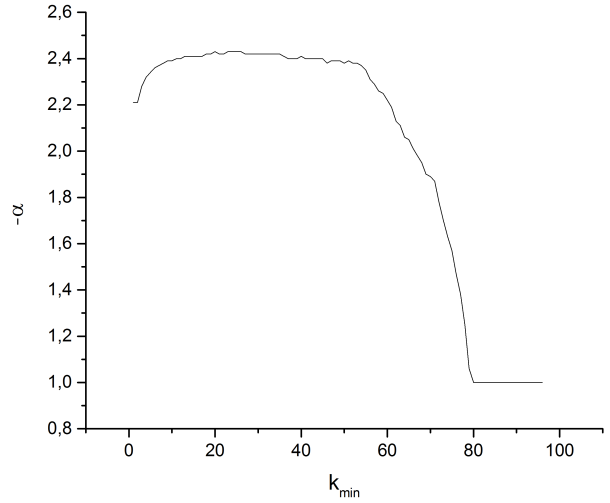


Figura 3: Variación del valor de la pendiente α en función del valor de k_{min}. Los datos representados son para la red 3D de lado 20. Para todas las redes se presenta un comportamiento similar, pudiendo empeorarse y presentar más oscilaciones si la calidad de la estadística de los datos a analizar no es lo suficientemente buena.

En este caso el valor de α comienza aumentando ligeramente hasta mantenerse en torno a un máximo, a partir del cual comienza a disminuir de nuevo. Como la tendencia de la pendiente es a disminuir cuando consideramos las regiones incorrectas (los puntos iniciales y finales de la Figura 3) consideraremos como el valor correcto de k_{min} el valor medio en la región en forma de "meseta", donde no se observa la influencia de datos incorrectos que disminuyan su valor. En el caso de la Figura 3, $k_{min} = -2{,}422 \pm 0{,}006$.

Considerando como pendiente correcta la obtenida con las cotas marcadas por los valores elegidos de k_{min} y k_{max}, maximizando con esos valores la función $\mathcal{L}(\alpha)$, obtenemos ajustes como el mostrado en la Figura 4.

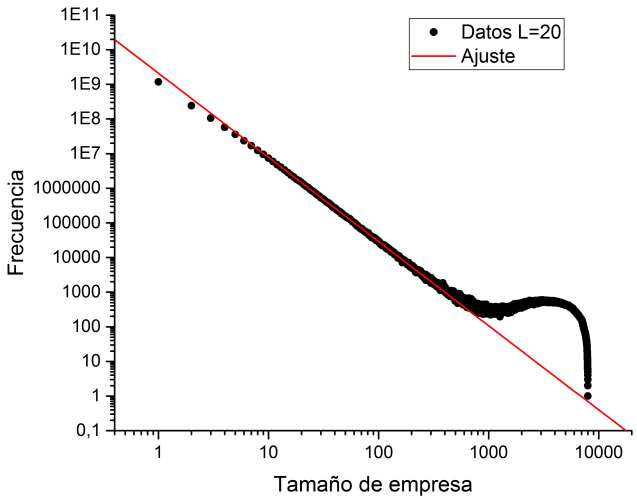


Figura 4: Representación de la distribución de empresas por tamaño en la red 3D de lado 20 sobre la que representamos el ajuste a una recta obtenido con el método descrito en este Apéndice.

Se puede ver que el ajuste a una recta no se adapta a todos los puntos de la región inicial (los primeros puntos quedan descartados por presentar una desviación respecto a la "zona ley de potencias") sino que se ajusta a la región que sigue puramente una ley de potencias.

Referencias

Bauke, H. (2007). Parameter estimation for power-law distributions by maximum likelihood methods. *Eur. Phys. J. B* 58, 381.

Clauset, A., Shalizi, C.R. y Newman, M.E.J. (2009). Power-law distributions in empirical data. *SIAM Review*, 51(4), 661-703.

Corral, A., Deluca, A. y Ferrer-i-Cancho, R. (2012). A practical recipe to fit discrete power-law distributions. arXiv:1209.1270v1 [stat.AP].

Deluca, A. y Corral, A. (2013). Fitting and Goodness-of- Fit Test of Non-Truncated and Truncated Power-Law Distributions. *Acta Geophysica* 61(6), 1351-1394.

Goldstein, M.L., Morris, S.A. y Yena, G.G. (2004). Problems with fitting to the power-law distribution. *Eur. Phys. J. B* 41, 255-258.

Newman, M.E.J. (2006). Power laws, Pareto distributions and Zipf's law. arXiv:cond-mat/0412004v3 [cond-mat.stat- mech].

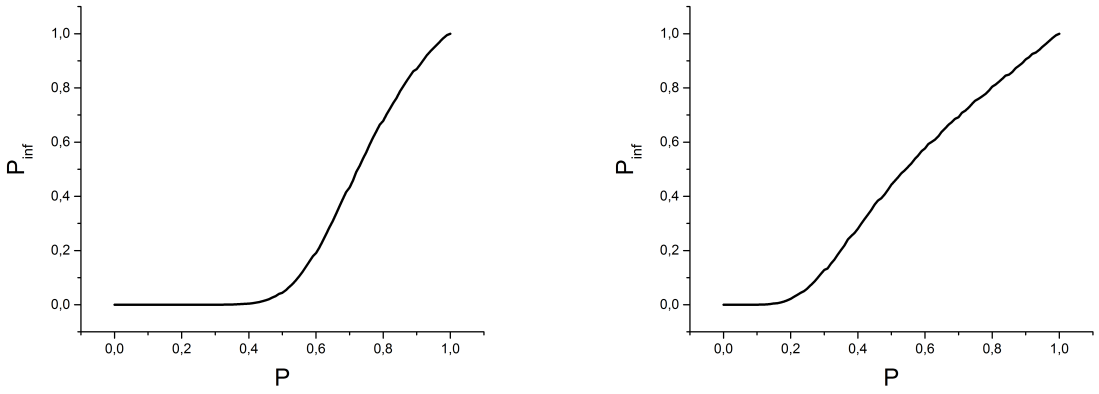
White, E.P., Enquist, B.J. y Green, J.L. (2008). On estimating the exponent of power-law frecuency distributions. *Ecology* 89(4), 905-912.

Anexo C: Comprobación de la construcción correcta de la red de Bethe

Al plantearnos simular el modelo en la red de Bethe surge un problema que en casos anteriores se había podido resolver con facilidad y es que la creación de esta red es más compleja. Mientras que en los modelos 1D, 2D y 3D la comprobación de que la red se estaba construyendo de forma correcta resultaba relativamente sencilla, para la red de Bethe tuvimos que establecer un modo de comprobación diferente que nos asegurara que se hubiera creado correctamente.

En su artículo Christensen (2002) define un valor que emplearemos en la comprobación, el umbral de percolación (*percolation threshold*), p_C. Se trata de la densidad de ocupación en la red a partir de la cual se observan cúmulos de percolación (cúmulos que unen dos extremos de la red conectando al menos dos vecinos diferentes de la primera capa). En el artículo se demuestra que en la red de Bethe el valor de p_C debe coincidir con el inverso del número de coordinación, $1/(Z-1)$.

De este modo planteamos la siguiente simulación: llenamos las redes creadas con diferentes valores de Z hasta una cierta densidad de ocupación p. Una vez alcanzada dicha ocupación detenemos el llenado y comprobamos si se ha formado un cúmulo de percolación. El proceso se repite muchas ocasiones, calculando la fracción de esas veces en las que se ha formado un cúmulo de percolación (p_∞). Los resultados para esta simulación son los recogidos en la Figura 1. Las simulaciones se realizan para los valores $Z = 3$ y $Z = 6$ ya que serán éstos los que se empleen en el modelo.



(a) Simulación en una red con $Z = 3$.

(b) Simulación en una red con $Z = 6$.

Figura 1: Fracción de cúmulos de percolación, p_∞, frente a densidad de ocupación, p.

Como se puede observar en la Figura anterior en ambos casos se comienzan a observar cúmulos de percolación para un valor de densidad de ocupación que coincide con $1/(Z-1)$, por lo que consideramos que las redes se construyen de forma correcta.

Referencias

Christensen, K. (2002). Percolation theory. Imperial College London, p. 1-2.