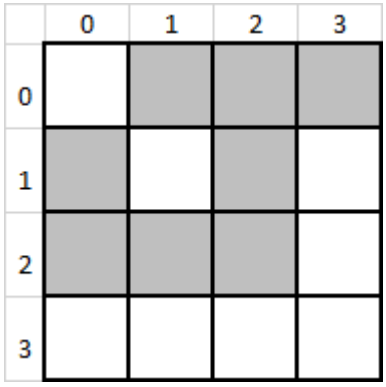
Algoritmo de Hoshen-Kopleman

El objetivo de estas notas es proporcionar una visión alternativa del algoritmo de Hoshen-Kopelman para el etiquetado de fragmentos para quienes aún le tengan desconfianza. Quizás algunas de las expresiones difieran respecto a las de la clase; esto es intencional.

Un fragmento es un conjunto de sitios ocupados conectados por primeros vecinos (arriba-abajo-izquierda-derecha para una red cuadrada). Por lo tanto, dada una red poblada cualquiera, queremos identificar los distintos fragmentos con etiquetas numéricas (comenzando en el 2).

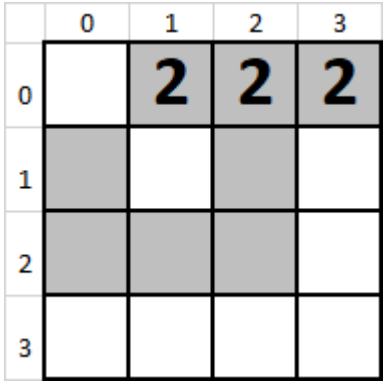
Un ejemplo a mano

En la figura de la derecha tenemos un ejemplo para una red de $L \times L = 4 \times 4$, donde los casilleros blancos están vacíos y los grises están ocupados. A simple vista vemos que hay un único fragmento en el sistema, por lo que felizmente le pondríamos la etiqueta "2" a todos los sitios ocupados. Pero como sabemos, la computadora no puede ver todo simultáneamente como nosotros.

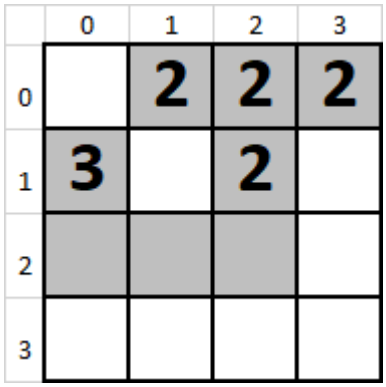


Intentemos entonces etiquetar este fragmento mirando un sitio a la vez, recorriendo cada fila de izquierda a derecha. Si el sitio está vacío, lo ignoramos y seguimos camino. Si el sitio está ocupado, vamos a mirar los vecinos *que ya hallamos recorrido*; el vecino superior y el izquierdo. Si estos vecinos están desocupados, entonces tenemos el primer sitio de un nuevo fragmento, al que le pondremos alguna etiqueta. Si estos vecinos están ocupados, ya pertenecerán a algún fragmento (dado que ya pasamos por ellos). Por lo tanto, el sitio actual (ocupado) deberá pertenecer al mismo fragmento que sus vecinos (superior y/o izquierdo) ocupados.

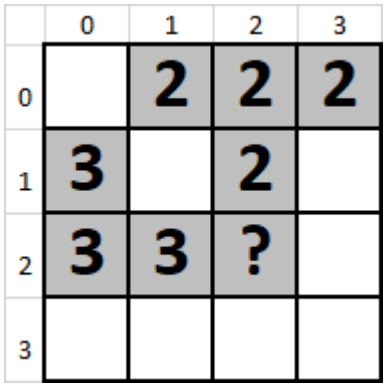
Apliquemos esta receta (llamarla algoritmo sería un insulto a Turing) para nuestro ejemplo. Comenzamos por la fila 0, donde salteamos el (0,0) porque está desocupado. Al llegar al ocupado (0,1), miramos sus vecinos más cercanos que ya hallamos recorrido; en este caso, el vecino izquierdo (0,0). Como este vecino está desocupado, el (0,1) se vuelve el fundador del primer fragmento de nuestra red, que etiquetaremos con un "2". Avanzamos al (0,2), donde al mirar a la izquierda vemos que (0,1) está ocupado. Por lo tanto, (0,2) debe pertenecer al mismo fragmento que (0,1), que ya etiquetamos como "2". Al (0,2) corresponde entonces otro "2". Con el (0,3) ocurre algo análogo, así que ponemos otro "2".



Pasando a la fila 1, vemos de entrada que el (1,0) está ocupado. Como su único vecino ya recorrido es el (0,0) y está desocupado, no queda otra más que tomarlo como el primer elemento de un nuevo fragmento con etiqueta “3”. Luego de saltear el sitio desocupado (1,1), llegamos al (1,2). El (1,2) es el primer caso de un sitio ocupado con dos vecinos ya recorridos (izquierdo (1,1) y superior (0,2)). Dado que su sitio izquierdo (1,1) está desocupado, es inmediato ver que el (1,2) pertenece al fragmento de su vecino superior (0,2), por lo que le corresponde una etiqueta “2”. Salteando el (1,3) desocupado, terminamos esta fila.



A esta altura ya vemos como viene la mano, así que podemos agilizar el análisis de la fila siguiente. Al sitio (2,0) le corresponde el “3” al tener un vecino superior con esa etiqueta. Al sitio (2,1) le corresponde la misma etiqueta que al (2,0), dado que su vecino superior (1,1) está desocupado. Hasta ahora, vemos dos etiquetas desconectadas; por un lado tenemos “2” y por el otro “3”. El conflicto aparece cuando pasamos al elemento (2,2), que tiene a ambos vecinos ocupados (tanto el izquierdo (2,1) como el superior (1,2)) y con distinta etiqueta. Como dijimos, (2,2) debe pertenecer al mismo fragmento que (1,2) y que (2,1), lo que por transitividad nos dice que todos los elementos del fragmento “2” son elementos del fragmento “3” y viceversa. En criollo, nos dice que son todos elementos de un único fragmento, por lo que las etiquetas nos están quedando confusas.



La solución fuerza bruta (o cabeza de termo) sería decir que la etiqueta “3” está mal, por lo que tenemos que volver a pasar por los sitios que ya recorrimos corrigiendo ese “3” por un “2” antes de seguir avanzando. No hay que ser ningún experto en complejidad computacional para darse cuenta que eso puede ser muuuy lento, teniendo un costo computacional que escala como $O(L^4)$ (en peor caso, tengo que volver a recorrer todo con cada sitio nuevo que visito).

La solución Hoshen-Kopelman (o procrastinación) es recordar que la etiqueta “3” es equivalente a la “2” y dejar la corrección para después de terminar esta primera pasada. Recién cuando terminemos de recorrer la red, volvemos a pasar por cada sitio, cambiando las etiquetas equivalentes por una misma etiqueta común (la “etiqueta verdadera”). Es justamente el objetivo del `historial` (mencionado en clase) mantener el seguimiento de estas equivalencias entre etiquetas. Básicamente, si tenemos que “t” y “s” son etiquetas equivalentes, podemos por ejemplo poner `historial[s]=-t`. Con el signo negativo simbolizamos que “s” no es la “etiqueta verdadera”, pero el módulo nos dice que la etiqueta es la misma que t. Con esto en mente, resulta razonable definir una función que dado un `historial` y una etiqueta `s`, nos devuelva la “etiqueta verdadera”

```
int etiqueta_verdadera(int* historial, int s){
  while(historial[s]<0){ // Es negativo, no es la etiqueta verdadera
    s = -historial[s]; // Tomo la etiqueta equivalente
  }
  return historial[s]; // Primer etiqueta equivalente positiva -> verdadera
}
```

Es por esto que, en general, cuando tengamos etiquetas equivalentes "t" y "s", elegiremos aquella con menor etiqueta verdadera a la hora de elegir que elemento de `historial` modificamos. Por ejemplo, si

$$\texttt{etiqueta\_verdadera(historial, t)} < \texttt{etiqueta\_verdadera(historial, s)}$$

entonces vamos a setear `historial[s]=-t`. Esto es una convención con el único objetivo de asegurarnos que los números de las etiquetas sean los más chicos posibles.

La moraleja de esto es que **`historial` guarda información sobre las etiquetas, NO sobre los sitios**. Esta confusión puede darse simplemente porque definimos `historial` como un array de L^2 elementos, para cubrirnos en el peor caso en que haya L^2 etiquetas distintas. Para pensar, ¿realmente es posible que haya una red que nos obligue a utilizar hasta L^2 etiquetas en la primer pasada? Si no es así, ¿cual es el máximo real de etiquetas y para que red?

Pseudo-código

En la sección anterior introdujimos la receta de identificación de fragmentos y mostramos la necesidad de mantener un seguimiento de las equivalencias entre etiquetas (vía `historial`). Ahora vamos a armar el pseudo-código de Hoshen-Kopelman, haciendo hincapié en las distintas sub-funciones del problema. Antes de empezar, aclaro que siendo un pseudo-código voy a tomarme múltiples libertades que ciertamente `C` no les va a tolerar. Esto es intencional; el proceso de "traducir" los va a ayudar a entender-que-corno-esta-haciendo-esto.

Lo primero que notamos, es que la primer fila y la primer columna son más sencillas al tener (como máximo) un vecino ya recorrido. En estos sitios jamás tendremos un conflicto entre etiquetas, por lo que en todo caso copiaremos la etiqueta del vecino ocupado. Sin embargo, queremos copiar la "etiqueta verdadera" de nuestro vecino, con la ilusión de evitar en el futuro actualizar este sitio o, al menos, ahorrarnos parte del recorrido entre etiquetas. Para esto usaremos la función `actualizar`. En caso de que nuestro vecino no tenga etiqueta, tenemos que crear una nueva para nuestro sitio. El entero `frag` mantiene seguimiento de las etiquetas ya utilzadas (todas las menores).

```
int actualizar(int* sitio, int* historial, int s, int frag){
  if (*sitio){ // Si el sitio esta desocupado, no hago nada
    if (s!=0){
      *sitio = etiqueta_verdadera(historial, s);
    }else{
      ... // Creo nueva etiqueta, actualizo maxima etiqueta (frag)
    }
  }
}
```

Esta función nos va a servir para la primer fila y la primer columna, excepto para el elemento (0,0) que resolvemos a mano de forma trivial. También nos va a servir en cualquier otro sitio que NO tenga ambos vecinos recorridos como ocupados. Si ambos vecinos están ocupados, puede surgir el problema de etiquetas de la sección anterior. Este problema lo resolveremos mediante la función `etiqueta_falsa`

```
int etiqueta_falsa(int* sitio, int* historial, int sup, int izq){
  if(*sitio!=0){       // Si el sitio esta desocupado, no hago nada
    sup_v = etiqueta_verdadera(historial, sup); // Busco las etiquetas
    izq_v = etiqueta_verdadera(historial, izq); // verdaderas de ambos
    if (sup_v==izq_v){
      *sitio = sup_v;  // Si tienen la misma etiqueta verdadera, safo
    }else{
      historial[max(sup_v,izq_v)] = -min(sup_v,izq_v);
      *sitio = min(sup_v,izq_v);
    } // La etiqueta verdadera de mayor valor pasa a ser falsa; son equivalentes
  }
}
```

Con estas herramientas en nuestro arsenal, podemos encarar el pseudo-código de la funcion `clasificar` para etiquetar los distintos fragmentos según el algoritmo de Hoshen-Kopelman. Para hacer un paralelismo con nuestra notación matricial, decimos que el elemento (k,h) de la red está asociado al elemento $k*L+h$ del array `red`.

Es importante recordar que al final de la primera pasada, tendremos en `historial` todas las equivalencias entre etiquetas, por lo que deberemos hacer una segunda pasada para corregir todas las etiquetas por sus etiquetas verdaderas.

```
int clasificar(int* red, int L){
  ... // Creo historial con valores no negativos (todas las etiquetas arrancan
     verdaderas)
  ... // Resuelvo a mano el sitio (0,0) correspondiente a red[0]
  for(i=1;i<L;i++){ // Recorro primera fila
    s = red[i-1];
    actualizar(red+i, historial, s, frag);
  }
  for(j=1;j<L;j++){ // Paso a las siguientes filas
  // Trato aparte el caso de la primer columna
    s = red[j*L-L];
    actualizar(red+j*L, historial, s, frag);
    for(i=1;i<L;i++){ // Recorro la fila j
      sup = ... // Miro los vecinos
      izq = ... // ya recorridos
      if (sup!=0 && izq!=0){
      // Ambos ocupados, posible conflicto de etiquetas
        etiqueta_falsa(...);
      }else{
        if (sup!=0){ // Solo superior ocupado
          actualizar(red+j*L+i, historial, sup, frag);
        }else{ // A lo sumo izquierdo ocupado
          actualizar(...)
        }
      }
    }
  }
  for(i=1;i<L*L;i++){ // Terminada la pasada, corrijo las etiquetas
    red[i] = etiqueta_verdadera(historial, red[i]);
  }
}
```