Algoritmos

Algoritmo ventana

Algorithm 1: Recorrido ventana. Complejidad: $O(x)$.

Datos de entrada: Num. renglones n; num. columnas m; cantidades de objetos por clase N_1, N_2; mallas 2×2 óptimas.

Resultado: Malla con objetos acomodados (representada por una matriz de enteros).

Declarar de mallas 2×2 óptimas.
$int\ N = N_1 + N_2$ `// Total de objetos.`

`// Verificar si es caso trivial.`
if $N < ceil\left(\frac{|n \times m|}{2}\right)$ **then**
  Caso trivial, resolver con el patrón establecido.
end

`// Condición para que haya al menos 1 caso imposible.`
if $max(\{N_1, N_2\}) > (|n \times m| - floor(n/2) * floor(m/2))$ **then**
  Hay casos imposibles en el arreglo.
end

Inicializar malla con todos sus valores iguales a -1, ya que si fueran 0's habría problemas a la hora de hacer calculos relacionados con el número de vacíos.

// Correr ventana en la malla. Los indices i y j recorren las celdas de la malla y corresponden a la esquina inferior derecha de la ventana.

for $i = 2,\ i \leq n,\ i++$ **do**

  for $j = 2,\ j \leq m,\ j++$ **do**

    // Declarar vector que contiene el número de objetos de cada clase que contendrá la ventana. Así como la variable que contiene el número de espacios a llenar de la ventana.

    $int\ v\_elem = \{0, 0\}$

    $int\ espacios\_a\_llenar$

    Contar el número de espacios a llenar de la ventana (contar los -1's) y poner el valor en $espacios\_a\_llenar$.

    Contar cuántos objetos de cada clase ya hay en la ventana y ponerlos en v_elem.

    while $espacios\_a\_llenar > 0$ **do**

      // Elegir los objetos a colocar en la ventana con el criterio basado en mayorias (tomar primero de los que hay más).

      Determinar de qué clase hay más objetos para colocar.

      Poner el objeto de tal clase en la ventana, es decir, hacer las actualizaciones correspondientes a v_elem, $espacios\_a\_llenar$, etc.

    end

    // El vector v_elem es la clave de un diccionario, al cual se le da el número de elementos de cada clase que tiene la ventana y te regresa una de las matrices de 2×2 óptimas con ese número de elementos.

    $matrizOptima = dic\_matOptimas(v\_elem)$

    // Inician casos para ver en qué zona de la malla se encuentra la ventana (las zonas están en la imagen del Word).

    case $i == 2 \wedge j == 2$ **do**

      Colocar los 4 elementos de $matrizOptima$ en la malla.

    end

case $i == 2 \wedge j > 2$ **do**

Rotar y reflejar la *matrizOptima*, hasta que los 2 elementos de su izquierda coincidan con los elementos ya colocados en la malla (aquí es mucho código, ya que hay que hacer operaciones para cada caso de rotación y reflexión).

Colocar los elementos de *matrizOptima* en la malla.

end

case $i > 2 \wedge j == 2$ **do**

Rotar y reflejar la *matrizOptima*, hasta que sus 2 elementos superiores coincidan con los elementos ya colocados en la malla (aquí es mucho código, ya que hay que hacer operaciones para cada caso de rotación y reflexión).

Colocar los elementos de *matrizOptima* en la malla.

end

case $i > 2 \wedge j > 2$ **do**

Rotar y reflejar la *matrizOptima*, hasta que coincida con los 3 elementos ya colocados en la malla (aquí es mucho código, ya que hay que hacer operaciones para cada caso de rotación y reflexión).

Colocar los elementos de *matrizOptima* en la malla.

end

end

end

// Una vez colocados todos los objetos en la malla, se verifica si hay casos imposibles, para esto se recorren los puntos clave de la malla (los que están en verde en el Word). Cada celda, y por lo tanto cada punto clave puede ser parte de 1 a 4 cuadros imposibles de 2×2. Si los ocho objetos que rodean al punto clave son iguales a este, se da el peor de los casos (4 cuadros imposibles).

// Basta con que el elemento del punto clave sea diferente de los objetos que lo rodean para que no haya ningún cuadro imposible en esas 9 celdas. Por lo que el proceso para arreglar los casos imposibles consiste en intercambiar el elemento del punto clave que pertenece a un cuadro imposible, con uno de su vecindad que sea diferente, o con uno de la vecindad de otro punto clave (en este caso, la vecindad de un punto clave son los 8 elementos que lo rodean).

```
   // Se declara una matriz de casos imposibles.  El número de elementos
   de esta matriz es el número de puntos claves en la malla (como si se
   construyera una matriz solo con los puntos verdes).  Cada elemento de
   la matriz indica el número de casos imposibles de los que es parte un
   punto clave (de 0 a 4).
38 int matrizImposibles[][]

39 Se recorre la ventana para contar los casos imposibles de cada punto clave y
   registrarlos en matrizImposibles

   // Correr indices en matriz de casos imposibles.
40 for i = 1, i ≤ renglonesMatrizImposibles, i++ do
41     for j = 1, j ≤ columnasMatrizImposibles, j++ do

42         if matrizImposibles[i][j] > 0 then        // Si hay un casos imposibles.
43
44             Verificar vecinos del punto clave correspondiente en busca de un elemento
               diferente.

45             if Todos los vecinos son iguales al punto clave. then
46                 Buscar en vecindades de los demás puntos claves.
47             end

48             Intercambiar elemento de punto clave con el elemento encontrado.

49             Actualizar matriz de casos imposibles.
50         end
51     end
52 end

   // Es en las líneas de la 44 a la 48 donde también se lleva gran
   cantidad de código, ya que para recorrer los 8 vecinos de cada punto
   clave se requieren varios if's.  Si bien al hacer el intercambio se
   liberan los cuadros imposibles de un lado, hay que verificar que del
   otro no se generen más, y de ser así hay que actualizar la matriz de
   casos imposibles, o sea, hay que ver a qué punto clave pertenece los
   imposibles generados al hacer el intercambio.

   // Y para todo lo anterior hay que estar verificando en qué zona de la
   malla se está, porque no es lo mismo una vecindad de la parte central
   de la malla que en un borde o una esquina (se dice fácil pero se lleva
   mucho código).
```

Algoritmo refuerzo

Algorithm 2: Aprendizaje por refuerzo. Complejidad: $O(3x + 24 * \#intercambios)$?.

Datos de entrada: Num. renglones n; num. columnas m; cantidades de objetos por clase N_1, N_2; diccionarios vecindades→puntajes.

Resultado: Malla con objetos acomodados (representada por una matriz de enteros).

$int\ N = N_1 + N_2$ `// Total de objetos.`

`// Declarar contador para llevar registro de las iteraciones seguidas en las que no mejoró la puntuación de la malla.`

$int\ contSinCambios$

`// Declarar diccionarios con la información vecindades→puntajes para cada zona de la malla (uno para esquinas, uno para bordes y uno para el centro). Un ejemplo de los puntajes está en la imágen de Word.`

$dicVecindadesPuntajesEsq$

$dicVecindadesPuntajesBordes$

$dicVecindadesPuntajesCentro$

`// Declarar matriz donde se guarda el puntaje de cada elemento, dado por los diccionarios.`

$matPunt[n][m]$

`// La explicación de las siguientes matrices es la parte más difícil de entender. Si no le entiende, tómelo solo como una matriz que contiene las puntuaciones de los elementos de la malla (similar a la anterior) y siga con lo demás.`

`// Declarar matrices de puntajes en cruz. Este es un arreglo de` $3 \times n \times m$ `(el 3 es por la cantidad de elementos diferentes que se pueden colocar en la malla: 1's, 2,s y 0's) donde, en cada celda de la primera matriz se guarda la suma de los siguientes puntajes de` $matPunt[n][m]$`: el del elemento en cuestión + el de sus 4 vecinos. Estos son los puntajes del estado actual de la malla, las otras 2 matrices se calculan haciendo lo mismo pero cambiando el elemento en cuestión o central por los otros 2 elementos (si hay un 1 en el estado actual, ahora se supondrá que hay un 2 para una matriz y un cero para la otra, los vecinos no cambian).`

$matricesPuntajes[3][n][m]$

```
// Función para calcular puntaje de un elemento de matPunt[n][m].
Verifica en qué parte de la malla se encuentra el elemento y llama al
diccionario adecuado.
function puntajeElemento(int renglon, int columna):

    int vectorVecindad[4]

    Obtener la vecindad del elemento y ponerla en vectorVecindad.

    case Elemento en esquinas do
        return dicVecindadesPuntajesEsq(vectorVecindad)
    end

    case Elemento en bordes do
        return dicVecindadesPuntajesBordes(vectorVecindad)
    end

    case Elemento en centro do
        return dicVecindadesPuntajesCentro(vectorVecindad)
    end
end

// Función para calcular puntaje de un elemento de matricesPuntajes.
function puntajeCruzElemento(int renglon, int columna):

    // Caso para los puntajes actuales. Para los otros 2 puntajes
    supuestos el código es similar. Son supuestos por que se supone
    que el elemento se cambia por otro distinto.
    puntaje = matPunt[renglon][columna]
    if renglon > 1 then puntaje+ = matPunt[renglon − 1][columna]
    if renglon < n then puntaje+ = matPunt[renglon + 1][columna]
    if columna > 1 then puntaje+ = matPunt[renglon][columna − 1]
    if columna < m then puntaje+ = matPunt[renglon][columna + 1]

    return puntaje
end

// Finalizan declaraciones y comienza ejecución del programa.

// Verificar si es caso trivial.
if N < ceil(|n×m| / 2) then
    Caso trivial, resolver con el patrón establecido.
end
```

`// Condición para que haya al menos 1 caso imposible.`

if $max(\{N_1, N_2\}) > (|n \times m| - floor(n/2) * floor(m/2))$ **then**

  Hay casos imposibles en el arreglo.

end

Inicializar la malla aleatoriamente con la cantidad N_1 de cubos (1's) y N_2 de prismas (2's, los vacíos aquí si son 0's).

Llenar los puntajes de $matPunt[n][m]$ mediante la función $puntajeElemento()$.

A partir de los puntajes anteriores llenar la información de $matricesPuntajes[1][n][m]$ y calcular lo puntajes para las otras 2 matrices, mediante la función $puntajeCruzElemento()$.

`// Hasta aquí llevamos los` $3x$ `de complejidad (se supone que cada vez que se llena una de las 3 matrices es una` x`).`

`// Bucle para mejorar el puntaje de la malla. Uso este formato de indices para no poner 4` for`'s.`

for $indice1$ **in** $matricesPuntajes[1]$ **do**

  for $indice2$ **in** $matricesPuntajes[1]$ **do**

    `// Si los objetos son iguales, no hacer nada.`

    if $malla[indice1] == malla[indice2]$ **then**

      Saltar a la sig. iteración.

    end

    $puntajeActual = matricesPuntajes[1][indice1] + matricesPuntajes[1][indice2]$

    $puntajeIntercambio = matricesPuntajes[2][indice1] + matricesPuntajes[3][indice2]$ `// Se cambia a las matrices de puntajes supuestos, ya sea la 2 o la 3.`

    if $puntajeIntercambio > puntajeActual$ **then**

      $contSinCambios = 0$

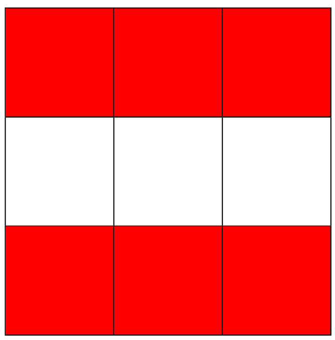
      Hacer el intercambio de elementos. Esto es: intercambiar los elementos en la $malla$, cambiar los puntajes actuales por los de intercambio en $matricesPuntajes$ y actualizar $matPunt$ de los 2 elementos en cuestión.

```
48          Actualizar los puntajes de los vecinos de los elementos intercambiados en
            matricesPuntajes y en matPunt (los puntajes de los elementos
            intercambiados ya están actualizados con la línea anterior).

            // Es en la línea anterior donde aparece el segundo término de
            complejidad:  24 ∗ #intercambios.  Ya que en total son 8
            vecinos y hay que actualizar sus 3 puntajes en
            matricesPuntajes (el actual y los supuestos), de ahí el 24
            (8*3).
49        else
50          contSinCambios + +
51        end

52        if contSinCambios > ciertoLimite then Terminar.  // Condición de paro
53
          // Es en las líneas de la 40 a la 45 donde sí se realizan esas
          operaciones con complejidad cuadrática.  Pero si las ve, solo
          son 2 sumas y 2 comparaciones, que comparadas con las
          operaciones a las que sí tomo en cuenta para la complejidad, son
          muy pocas, por lo que no las tomo en cuenta, ¿estoy bien en no
          tomarlas en cuenta?.  Esa es la duda por la que no sé si ponerle
          complejidad lineal + el otro término al algoritmo o no.
54    end
55  end
```

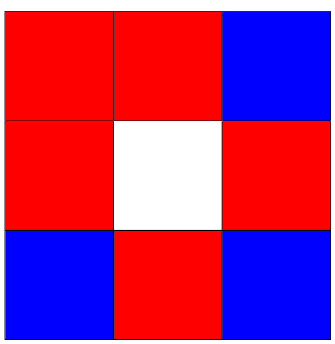
Algunos ejemplos

Se muestran ejemplos del acomodo de los 2 algoritmos y la suma de movimientos para acceder a todos los objetos.

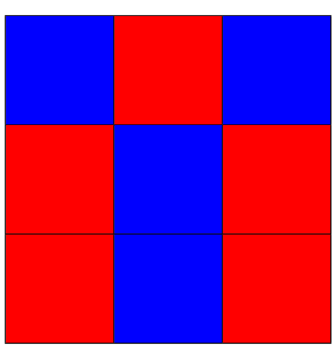
Acomodo Refuerzo



$\equiv 6$

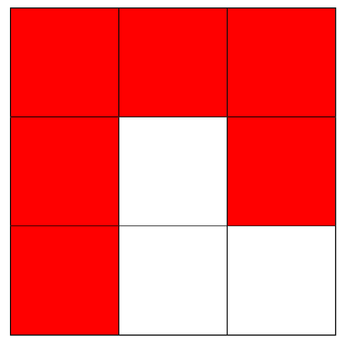


$\equiv 9$

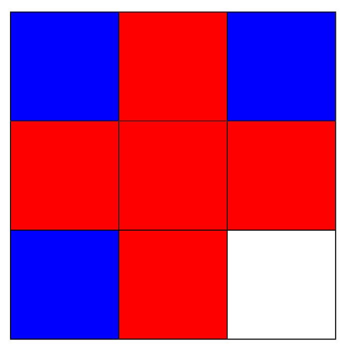


$\equiv 14$

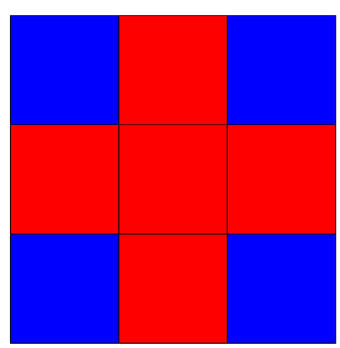
Acomodo Ventana



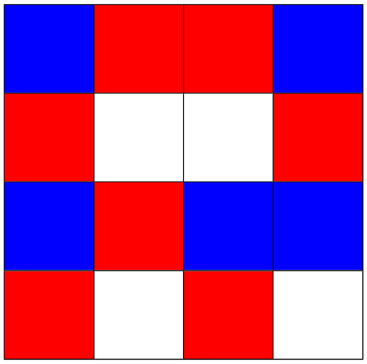
$\equiv 8$



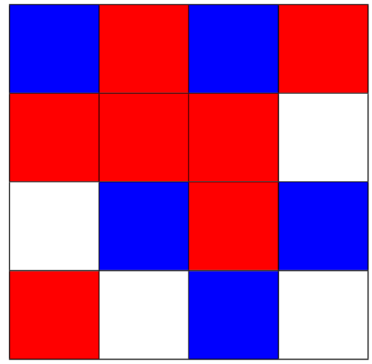
$\equiv 19$



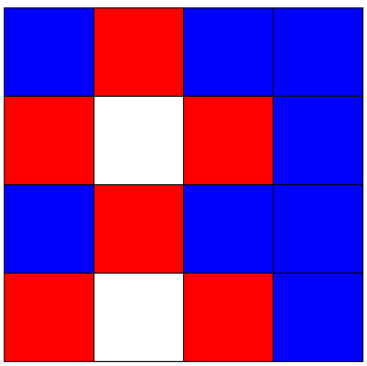
$\equiv 23$



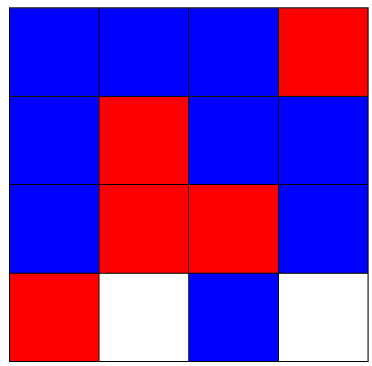
$\equiv 12$



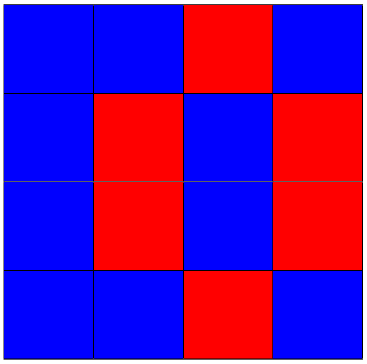
$\equiv 25$



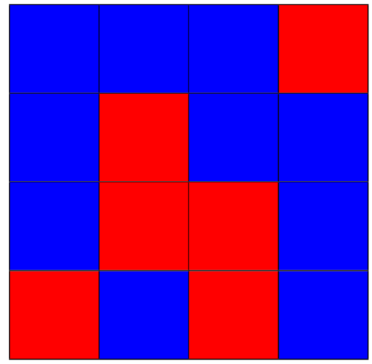
$\equiv 18$



$\equiv 34$



$\equiv 26$



$\equiv 42$