Universidad Veracruzana

Inteligencia Artificial

Análisis de Algoritmos

Actividad 5

Leonardo Flores Torres

3 de diciembre de 2022

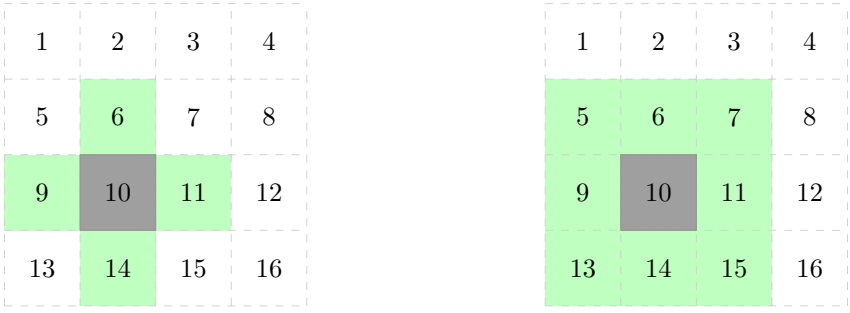
Adaptar el algoritmo de Dijkstra para que trabaje en una rejilla donde se defina el punto inicial (r_i, c_i) y el punto final (r_f, c_f) y encuentre la ruta óptima con las siguientes variantes,

1. usar 4 vecinos con distancias unitarias,
2. usar 8 vecinos,
3. incluir la posibilidad de encontrar obstáculos.

Solución:

El módulo desarrollado, `ShortestPath`, para resolver esta actividad se muestra en el apéndice de este trabajo, en él se pueden observar mejor las restricciones para detectar vecinos de los pixeles aunque se agregará una breve explicación de esto en pseudocódigo. El algoritmo de Dijkstra no fue implementado sino que se utilizó una librería [1] ya desarrollada en `julia` que permite el uso de grafos y además incluye la implementación de distintos algoritmos de recorrido y búsqueda de los caminos más cortos, como lo es el de Dijkstra.

Antes de comenzar a utilizar el módulo es importante pensar primero en cómo crear las relaciones que existen entre pixeles, en otras palabras, hay que idear de alguna manera como identificar los vecinos que un pixel tiene. El caso ideal es cuando el pixel del que se quiere encontrar los vecinos se encuentra en una posición como la que se muestra en la figura 1.



(a) Configuración de 4 vecinos. (b) Configuración de 8 vecinos.

Figura 1: Se muestra un cuadro en gris obscuro equivalente a un pixel y sus vecinos en gris claro en un arreglo representando a una imágen para ambas configuraciones de vecinos permitidas.

Se puede observar que no se está accediendo a los pixeles como si la imagen fuese un arreglo bidimensional, en realidad se hizo una transformación de los índices (row, col) de los pixeles de la imágen para relacionarlos con un identificador, id. Esta transformación es hecha a partir de

$$\text{id}(\text{row}, \text{col}) = \text{col} + n_c(\text{row} - 1),$$

donde n_c es el número de columnas. Si por algún motivo fuese necesario regresar a la representación de (row, col), esto puede hacerse como

$$\text{col} = \begin{cases} n_c & \text{si id mod } n_c = 0, \\ \text{id mod } n_c & \text{en cualquier otro caso.} \end{cases}$$

donde el primer caso es aquel cuando el id corresponde a pixeles ubicados en la útila columna, y el segundo caso es cuando el id corresponde a pixeles en cualquier otra columna diferente a la última. Por otro lado, calcular la fila es trivial si ya se conoce el identificador y la columna, solamente se despeja row de la expresión para $\text{id}(\text{row}, \text{col})$.

Independientemente de la configuración elegida, ya sea la de 4 u 8 vecinos, es importante reconocer todos los escenarios cuando el pixel se encuentra en una posición en la que no tiene acceso a todos los pixeles que tendría en el caso ideal, esto sucede cuando dicho pixel se encuentra en los bordes de la imágen. El primer escenario considerado sucede cuando se quieren encontrar los vecinos de un pixel a su derecha o a su izquierda, como en la figura 2. Para este caso hay dos restricciones evidentes, cuando el pixel se encuentra en la primera columna, y cuando lo está en la última columna.

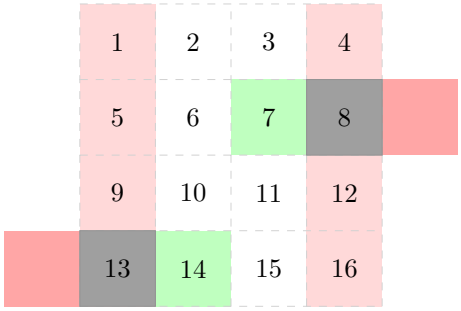


Figura 2: Vecinos no permitidos, en dirección horizontal, de un pixel dado.

De esta manera podemos decir que si el módulo del identificador de un pixel respecto al número total de columnas n_c es igual a cero no tiene vecino a la derecha, o si el módulo del identificador menos 1 respecto n_c es cero entonces no tiene vecino a la izquierda. Esto se puede ver expresado en el pseudocódigo mostrado a continuación (para todos los snippets de pseudocódigo `nc` es el número de columnas y `nr` el de filas):

```
# Pseudocodigo
# Vecino a la derecha
if id % nc != 0
    agrega (id + 1) como vecino
end if
# Vecino a la izquierda
if (id - 1) % nc != 0
    agrega (id - 1) como vecino
end if
```

El siguiente escenario se muestra en la figura 3 donde la búsqueda de vecinos para un pixel dado se realiza verticalmente, esto es, se buscan los vecinos que queden inmediatanente arriba y abajo de un cierto pixel. Para este caso se observa que hay dos restricciones, una de ellas es cuando el pixel se encuentra en la primera fila y la otra es cuando lo está en la última fila ya que los pixeles en la primera fila no tienen vecinos arriba de ellos, mientras que los que están en la última fila no tienen vecinos abajo.

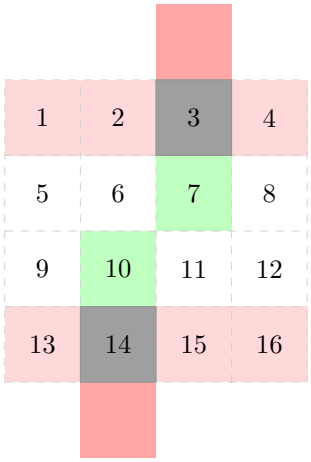


Figura 3: Vecinos no permitidos, en dirección vertical, de un pixel dado.

Así, se puede decir que los pixeles con identificadores que tengan valor menor o igual al número total de columnas no tendrán vecinos sobre de ellos, mientras que aquellos pixeles con identificadores en la última fila no tendrán vecinos debajo de ellos. Lo interesante es ver que si un pixel tiene un vecino arriba, este vecino tendrá como identificador el identificador del pixel menos el número de columnas de la imágen, $\text{id}_{\text{pixel}} - n_c$. Algo similar sucede cuando se buscan vecinos abajo, el identificador del vecino es el del pixel agregandole el número de columnas, $\text{id}_{\text{pixel}} + n_c$. Esto se puede representar de la siguiente manera en pseudocódigo:

```
# Pseudocodigo
# Vecino arriba
if id > nc
    agrega (id - nc) como vecino
end if
# Vecino abajo
if id <= nc * (nr - 1)
    agrega (id + nc) como vecino
end if
```

Un penúltimo escenario es el mostrado en la figura 4 donde se muestra la cuadrícula dos veces debido a que se puede dar el caso de caer en una de ellas. Por ejemplo, se puede dar el caso de que el pixel del que se quieren obtener sus vecinos se encuentre en la primera columna o en la última fila, en ambos casos no hay vecino de dicho pixel abajo a la izquierda. El otro caso es si el pixel se encuentra en la primera fila o en la última columna, donde no habría disponible un vecino arriba a la derecha.

Las restricciones que surgen a causa de esto son combinaciones de las ya definidas anteriormente.

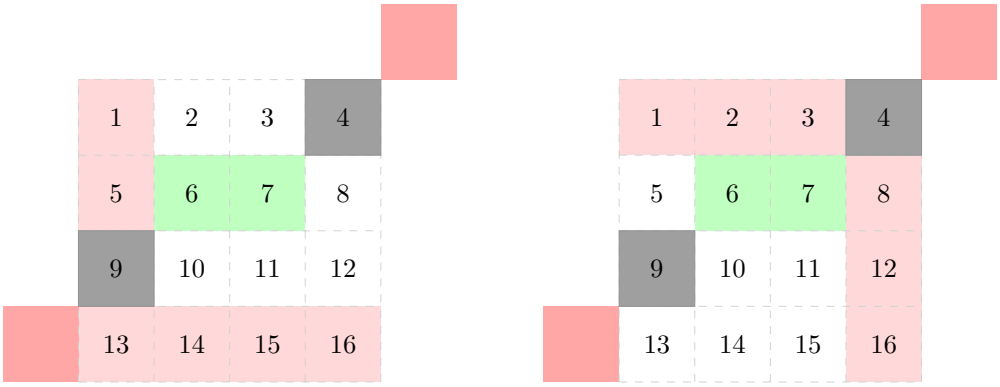


Figura 4: Vecinos no permitidos, en una posible dirección vertical, de un pixel dado.

Todos los pixeles en la zona roja de la primera columna y la última fila no pueden tener vecinos abajo a la izquierda, y todos los pixeles en la zona roja de la primera fila y última columna no pueden tener vecinos arriba a la derecha, así que se combinan las restricciones pertinentes. Esto se puede escribir en pseudocódigo como se muestra a continuación:

```
# Pseudocodigo
# Vecino arriba a la derecha
if id % nc !=0 && id > nc
    agrega (id - nc + 1) como vecino
end if
# Vecino abajo a la izquierda
if (id - 1) % nc != 0 && id <= nc * (nr - 1)
    agrega (id + nc - 1) como vecino
end if
```

El último escenario es aquel en que el pixel del que se quieren saber sus vecinos se encuentra en la primera columna o en la primera fila, y también se puede dar que dicho pixel se encuentre en la última fila o última columna. Para ambos casos se hace una combinación de las reglas más simples ya definidas.

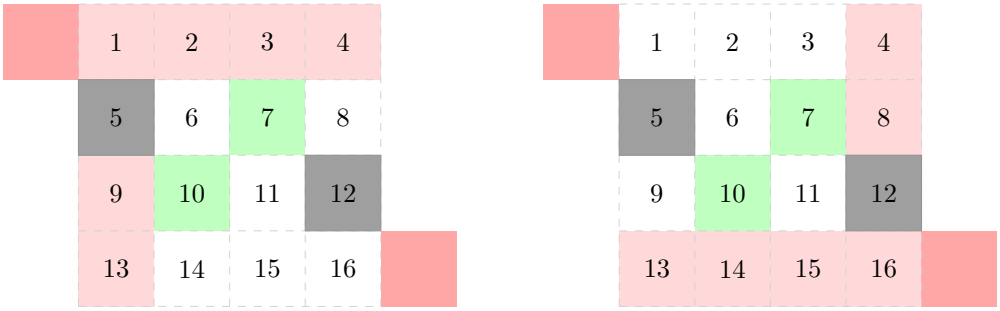


Figura 5: Vecinos no permitidos, en una segunda posible dirección vertical, de un pixel dado.

El pseudocódigo es similar al anterior, y se muestra a continuación:

```
# Pseudocodigo
# Vecino arriba a la izquierda
if (id - 1) % nc != 0 && id > nc
    agrega (id - nc - 1) como vecino
end if
# Vecino abajo a la derecha
if id % nc != 0 && id <= nc * (nr - 1)
    agrega (id + nc + 1) como vecino
end if
```

No se especifíca en el pseudocódigo cómo o en dónde agregar a los vecinos, lo importante de esto es saber reconocer cuales son las restricciones para hacerlo, y el lector decidirá la mejor manera de realizar su propia implementación con la estructura de datos que desee. La razón por la que se eligió cambiar la representación del problema fue debido a que si no se hacía se tendría que haber utilizado dos `for-loop`'s anidados para iterar sobre todas las columnas y todas las filas. Aunque no representa una carga computacional pesada hacer esto para este problema en especifíco, lo consideré pertinente tomando en cuenta que se trata del curso de *Análisis de Algoritmos* y quisiera poner en práctica lo aprendido siempre que me sea posible.

La manera de utilizar el módulo, `ShortestPath`, en el `REPL` de `julia` se muestra a continuación. Primero se define el tamaño de la rejilla en `nrows` y `ncols`, y con estas variables se computa la representación de índices de los elementos de la rejilla, aunque resulta ser impractica de visualizar mientras más el número más vecinos hay en total dentro de la vecindad.

```
julia> using ShortestPath; sp = ShortestPath;
julia> nrows = 40; ncols = 40;
julia> idarray = [i for i in 1:(ncols * nrows)] ▷ x -> reshape(x, ncols, nrows) ▷ transpose;
julia> start = [10, 3]; finish = [35, 37];
```

Una vecindad es el nombre dado a la rejilla tomando en cuenta los puntos libres por los que se puede transitar, y los obstáculos, que son puntos imposibles de alcanzar. Además, los obstáculos se eligen de pixeles de manera aleatoria de la rejilla inicialmente libre, esto quiere decir que todos los pixeles son alzancables al inicio, se hace un muestreo aleatorio y aquellos elegidos cambian su estado a obstáculos. El argumento `obsdensity` de la función `neighborhood` indica la densidad de obstáculos deseada entre $[0, 1]$.

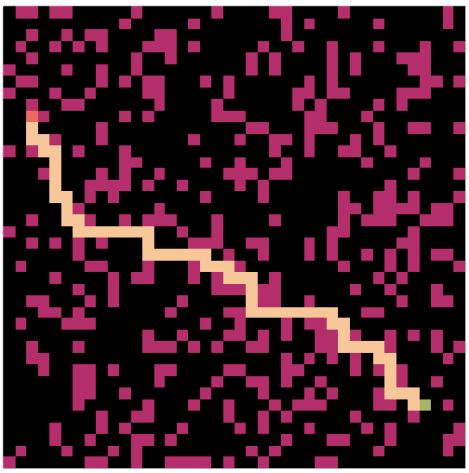
Se computó una vecindad con los puntos de partida inicial y final mostrados en `start` y `finish`, respectivamente, y una densidad de obstáculos del 0.25. También se hizo una copia profunda de la vecindad `nh` en `nhdiag` ya que se obtuvieron las matrices de adyacencia para los casos donde el movimiento se puede dar sin y con diagonales en la misma rejilla, para 4 y 8 vecinos, respectivamente.

```
julia> nh, obs = sp.neighborhood(nrows, ncols; start=start, finish=finish, obsdensity=0.25);
↪  sp.visualize(nh)
julia> nhdiag = deepcopy(nh);
julia> adjmat = sp.adjacencyMatrix(nrows, ncols; obstacles=obs, allowdiags=false);
julia> adjmatdiag = sp.adjacencyMatrix(nrows, ncols; obstacles=obs, allowdiags=true);
```

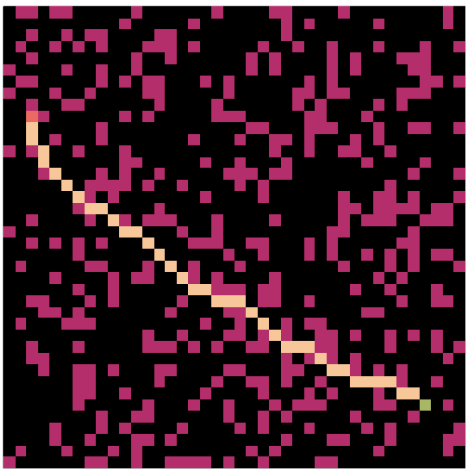
Las matrices de adyacencia guardadas en las variables `adjmat` y `adjmatdiag` requieren como argumentos la lista de obstáculos en `obstacles`, si las diagonales son permitidas o no en `allowdiags`, y el número de filas y de columnas. Estos dos primeros argumentos son posicionales mientras que los primeros dos mencionados son argumentos de palabra clave, se observa la división entre un tipo de argumentos y otro con un `;` dentro de los argumentos de una función.

Posteriormente se obtienen los caminos y las distancias de esos caminos con la función `findPath`; se computaron los caminos sin diagonales en `path` y con diagonales en `pathdiag`, con sus repectivas matrices de adyacencia. Para terminar, los caminos se usan para cambiar el estado de los pixeles en las vecindades `nh` y `nhdiag` con la función `updateneighborhood!`. Nótese que la función lleva un signo de admiración `!` al final de su nombre para denotar que es una función que modifica alguno de sus argumentos.

```
julia> path, dist = sp.findPath(adjmat, ncols; start=start, finish=finish);
julia> pathdiag, distdiag = sp.findPath(adjmatdiag, ncols; start=start, finish=finish);
julia> sp.updateneighborhood!(nhdiag, pathdiag); sp.visualize(nhdiag)
julia> sp.updateneighborhood!(nh, path); sp.visualize(nh)
julia> dist
59.0
julia> distdiag
46.11269837220809
```



(a) 4 vecinos (diagonales no permitidas); distancia = $59u$.



(b) 8 vecinos (diagonales permitidas); distancia = $46.1126u$.

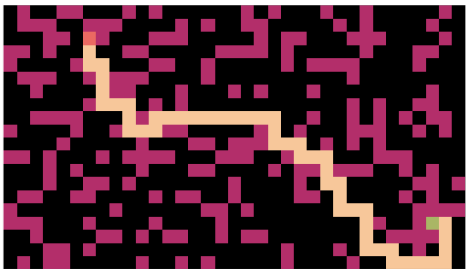
Figura 6: Rejilla de 40×40 con una densidad de obstáculos al 25 %.

En la figura 6 se muestran las imágenes para una rejilla de 40×40 incluyendo el caso en que el movimiento hacia vecinos solamente se da horizontal y verticalmente, figura 6a, y el caso en que el movimiento diagonal está permitido, figura 6b. Las distancias del recorrido en cada una de ellas son de 50 unidades sin diagonales en la figura 6a, y de 46.11 unidades considerando diagonales en la figura 6b.

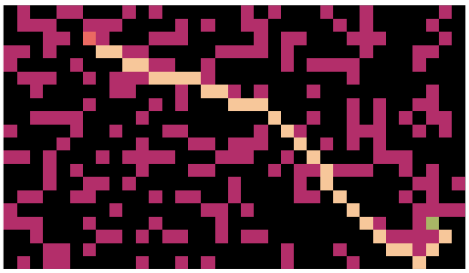
Los pixeles mostrados en color ■ son los pixeles libres por los que se puede transitar mientras que los mostrados de color los pixeles de color ■ son los obstáculos; el pixel de color ■ es el punto

de partida definido en `start`, el pixel de color es el punto de llegada definido en `finish`, y los pixeles de color muestran el camino atravesado para llegar a la meta.

Esta implementación permite utilizar dimensiones para la rejilla diferentes de $n \times n$, esto quiere decir que se pueden elegir un número diferente de filas que de columnas, y viceversa. Un ejemplo de esto se muestra en la figura 7, con 20 filas y 35 columnas. El camino sin diagonales permitidas es de 50 unidades, mientras que aquel en que se permiten las diagonales tiene longitud de 36.87 unidades.



(a) 4 vecinos (diagonales no permitidas); distancia $= 50u$.



(b) 8 vecinos (diagonales no permitidas); distancia $= 36.87u$.

Figura 7: Rejilla de 20×35 con una densidad de obstáculos del 30 %.

Quisiera agregar un par de comentarios extras acerca de la matriz de adyacencia, hablar acerca de porqué la representación elegida es útil. Lo primero es que la matriz de adyacencia es una matriz cuadrada de $n \times n$ donde n es el número de pixeles dentro de la imágen independientemente de si son o no obstáculos. La representación elegida a partir de identificadores permite fácilmente acceder a un arreglo bidimensional tomando los identificadores como índices de una matriz, la matriz de adyacencia. Por ejemplo,

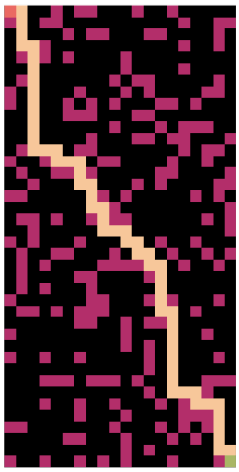
$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ n \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & \dots & n \end{array} \\ \begin{pmatrix} v(1,1) & v(1,2) & v(1,3) & v(1,4) & \dots & v(1,n) \\ v(2,1) & v(2,2) & v(2,3) & v(2,4) & \dots & v(2,n) \\ v(3,1) & v(3,2) & v(3,3) & v(3,4) & \dots & v(3,n) \\ v(4,1) & v(4,2) & v(4,3) & v(4,4) & \dots & v(4,n) \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ v(n,1) & v(n,2) & v(n,3) & v(n,4) & \dots & v(n,n) \end{pmatrix} \end{array}$$

donde $v(i,j)$ en realidad denota el $j-$ésimo vecino del pixel con identificador i. De esta manera, se puede acceder al arreglo bidimensional equivalente a la matriz de adyacencia en `julia` como $v(\text{row}, \text{column})$ y así agregar o quitar vecinos sin realizar transformaciones extras a los índices de los pixeles en la imágen original. Si se quisieran agregar los vecinos del pixel con identificador $i = 1$ bastaría con computarlos de acuerdo a las restricciones explicadas al inicio de la actividad y agre-

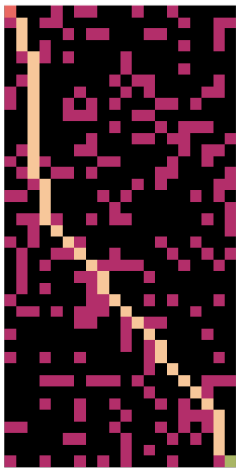
garlos a la matriz de adyacencia como $v(1, j)$ para todo j dentro del conjunto de identificadores, $j = (1, \ldots, n_{\text{total}})$. Lo segundo es que gracias a la representación usada se pueden fácilmente generar mapas de pixeles de diferentes dimensiones $n \times m$ donde $n \neq m$, aunque seguimos restringidos a tener mapas rectangulares pero no solamente cuadrados. Una idea interesante podría ser la implementación de mapas diferentes al caso rectangular como podría serlo el movimiento dentro de un espacio tridimensional.

De lo anteriormente mencionado, si se conocen los identificadores de los obstáculos, entonces las entradas de la matriz $v(k, j)$ serían igual a cero para todo j dentro del conjunto de identificadores y para todo k dentro del conjunto de los identificadores de los obstáculos ya que esas entradas corresponden a las filas de las interacciones de los obstáculos con otros. Lo mismo se aplicaría para las interacciones de otros pixeles con los obstáculos representadas por $v(i, k)$, también serían iguales a cero. De esta manera se tendrían filas y columnas enteras iguales a cero las cuales las cuales son aquellas en las k-ésimas posiciones dentro de la matriz para ambos casos. Por ejemplo, si el conjunto de los obstáculos fuera igual a obs $= \{2, 4\}$, entonces la matriz mostrada anteriormente se veria como,

$$\begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \\ \vdots \\ n \end{array} \begin{array}{c} \begin{array}{cccccc} 1 & 2 & 3 & 4 & \ldots & n \end{array} \\ \begin{pmatrix} v(1,1) & 0 & v(1,3) & 0 & \ldots & v(1,n) \\ 0 & 0 & 0 & 0 & \ldots & 0 \\ v(3,1) & 0 & v(3,3) & 0 & \ldots & v(3,n) \\ 0 & 0 & 0 & 0 & \ldots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \vdots \\ v(n,1) & 0 & v(n,3) & 0 & \ldots & v(n,n) \end{pmatrix} \end{array}.$$



(a) 4 vecinos (diagonales no permitidas); distancia $= 58u$.



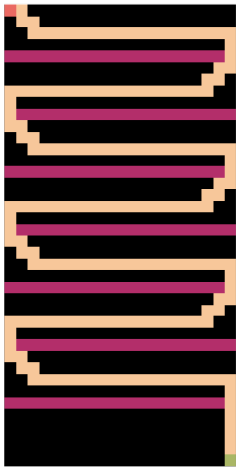
(b) 8 vecinos (diagonales no permitidas); distancia $= 46.87u$.

Figura 8: Rejilla de 40×20 con una densidad de obstáculos del 25 %.

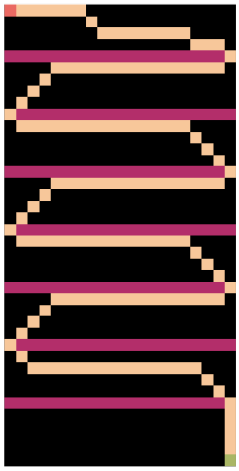
Para finalizar quisiera mostrar un caso en el que los obstáculos no se generan de manera aleatoria. No es el propósito explicar a detalle los pasos a continuación pero se hará lo mejor posible para que el lector pueda seguir los pasos. Primero, definimos el número de columnas y de filas que vamos a considerar para la imágen, también declaramos manualmente un color guardado en la variable

`color_` para identificar a los obstáculos[1] y se debe crear el vecindario, esto es, la imágen que contiene a todos los pixeles inicializados en negro `nh`:

```
julia> color_ = sp.RGB(0.70196, 0.18039, 0.41568);
julia> nh = zeros(sp.RGB, nrows, ncols);
```



(a) 4 vecinos (diagonales no permitidas); distancia $= 172u$.



(b) 8 vecinos (diagonales no permitidas); distancia $= 152.08u$.

Figura 9: Rejilla de 40×20 con una densidad de obstáculos del 16 %.

Se está considerando un arreglo similar al mostrado en la figura 8 con 40 filas y 20 columnas. Se crearan obstáculos en forma de escalera, el color de estos pixeles se cambiará al color que corresponde a los obstáculos:

```
julia> nh[5,begin:end-1] .= color_;
julia> nh[10,begin+1:end] .= color_;
julia> nh[15,begin:end-1] .= color_;
julia> nh[20,begin+1:end] .= color_;
julia> nh[25,begin:end-1] .= color_;
julia> nh[30,begin+1:end] .= color_;
julia> nh[35,begin:end-1] .= color_;
```

Los obstaculos se encuentran preguntando cuáles no son de color negro, esto se hace con `findall`, este comando da una lista de índices cumpliendo cierta condición. Para este caso, la condición es aquellos pixeles que no son negros:

```
julia> obs = findall(x -> x != sp.RGB(0,0,0), nh) .▷ x -> sp.id(x[1], x[2], ncols);
```

Nótese que se convirtieron los índices que se obtendrían con `findall` a la representación de los índices de la que se habló anteriormente ya que la función `adhacencyMatrix` requiere que los obstáculos se pasen como argumento en esta representación. Habiéndo dicho esto, se procede a calcular la matriz de adyacencia para ambos casos, cuando se permite moverse en diagonales y cuando no:

[1] Corresponde al mismo color mostrado en las imágenes anteriores.

```
julia> adjmat = sp.adjacencyMatrix(nrows, ncols; obstacles=obs, allowdiags=false);
julia> adjmatdiag = sp.adjacencyMatrix(nrows, ncols; obstacles=obs, allowdiags=true);
julia> path, dist = sp.findPath(adjmat, ncols; start=start, finish=finish);
julia> pathdiag, distdiag = sp.findPath(adjmatdiag, ncols; start=start, finish=finish);
```

Finalmente, se actualiza los vecindarios originales para incluir los caminos, y se guardan las imágenes:

```
julia> sp.updateneighborhood!(nh, path)
julia> sp.updateneighborhood!(nhdiag, pathdiag)
julia> sp.savefig(fig, "../figures/escalera_nodiag.png");
julia> sp.savefig(figdiag, "../figures/escalera_diag.png");
```

En la figura 9 se muestran ambos casos, cuando no se está permitido moverse en diagonales y cuando sí lo está permitido. A pesar de que no esté implementado en el módulo escrito elegir pixeles de manera interactiva, se pueden definir obstáculos de esta manera para generar estructuras más interesantes para las cuales encontrar sus caminos más cortos. Se invita al lector, estudiante o no, que juegue con esta funcionalidad para generar alún tipo de laberinto sencillo.

Apéndice

```
module ShortestPath

#==============================
Required libraries
==============================#

using Images
using Plots
using StatsBase: sample
using Graphs: dijkstra_shortest_paths
using SimpleWeightedGraphs: SimpleWeightedGraph

#==============================
Image operations
==============================#

function visualize(neighborhood; size=(600,600))
    fig = plot(neighborhood,
        ticks = false,
        axis = false,
        background_color = :transparent,
        foreground_color = :black,
        gridalpha = 1,   # temporal fix to grid over plot
        size = size,
    )
    return fig
end

#==============================
Map generation and operations
==============================#

id(row, col, ncols) = col + ncols * (row - 1)

function posbyid(id::Int, ncols::Int)
    col = iszero(id % ncols) ? ncols : id % ncols
    row = 1 + (id - col) / ncols |> Int
    return row, col
end

function neighborhood(nrows::Int, ncols::Int;
    start=nothing, finish=nothing, obsdensity=0
)
    obsdensity = obsdensity * nrows * ncols |> round |> Int
    obscolor = RGB(0.70196, 0.18039, 0.41568)
    canvas = zeros(RGB, nrows, ncols)

    start = !isnothing(start) ? CartesianIndex(start[1], start[2]) : []
    finish = !isnothing(finish) ? CartesianIndex(finish[1], finish[2]) : []

    available = setdiff(CartesianIndices(canvas), [start, finish])
    obstacles = sample(available, obsdensity, replace=false)
```

```
53
54      canvas[[obstacles...]] .= obscolor
55
56      obstacles = obstacles .▷ x -> id(x[1], x[2], ncols)
57      sort!(obstacles)
58
59      return canvas, obstacles
60  end
61
62  function updateneighborhood!(neighborhood, path)
63      ncols = size(neighborhood)[2]
64      pathcoordinates = posbyid.(path, ncols) .▷ x -> CartesianIndex(x)
65
66      neighborhood[pathcoordinates[1]] = RGB(169/255, 182/255, 101/255)
67      neighborhood[pathcoordinates[end]] = RGB(234/255, 105/255, 98/255)
68      neighborhood[pathcoordinates[begin+1:end-1]] .= RGB(247/255, 199/255, 154/255)
69  end
70
71  #==============================
72  Neighbors and path finding
73  ==============================#
74
75  function adjacencyMatrix(nrows, ncols;
76      obstacles=nothing, allowdiags=false
77  )
78      ns = nrows * ncols          # neighborhood size
79      ids = 1:ns                  # neighbors ids
80      adjmat = zeros(ns, ns)
81      diagstepsize = sqrt(2)      # diagonal step size
82
83      for id in ids
84          # neighbor to the right
85          if !iszero(id % ncols)
86              # adjmat[id, id + 1] = adjmat[id + 1, id] = 1
87              adjmat[id, id + 1] = 1
88          end
89          # neighbor to the left
90          if !iszero((id - 1) % ncols)
91              adjmat[id, id - 1] = 1
92          end
93          # neighbor above
94          if id > ncols
95              adjmat[id, id - ncols] = 1
96          end
97          # neighbor below
98          if id <= ncols * (nrows - 1)
99              # adjmat[id, id + ncols] = adjmat[id + ncols, id] = 1
100             adjmat[id, id + ncols] = 1
101         end
102
103         if allowdiags
104             #neighbor up-right
105             if !iszero(id % ncols) && id > ncols
106                 adjmat[id, id - ncols + 1] = diagstepsize
```

```
            end
            # neighbor up-left
            if !iszero((id - 1) % ncols) && id > ncols
                adjmat[id, id - ncols - 1] = diagstepsize
            end
            # neighbor down-right
            if !iszero(id % ncols) && id <= ncols * (nrows - 1)
                # adjmat[id, id + ncols + 1] = adjmat[id + ncols + 1, id] = diagstepsize
                adjmat[id, id + ncols + 1] = diagstepsize
            end
            # neighbor down-left
            if !iszero((id - 1) % ncols) && id <= ncols * (nrows - 1)
                # adjmat[id, id + ncols - 1] = adjmat[id + ncols - 1, id] = diagstepsize
                adjmat[id, id + ncols - 1] = diagstepsize
            end
        end
    end

    # Removing obstacles from being part of the neighborhood
    for id in obstacles
        adjmat[:, id] .= 0.0
        adjmat[id, :] .= 0.0
    end

    return adjmat
end

function findPath(adjmat, ncols; start=start, finish=finish)
    startid = start |> x -> id(x[1], x[2], ncols)
    finishid = finish |> x -> id(x[1], x[2], ncols)

    graph = SimpleWeightedGraph(adjmat) # A weighted graph is needed
    dijkstra = dijkstra_shortest_paths(graph, startid)

    path_nodes = []
    node = finishid
    distance = dijkstra.dists[finishid]
    while node != 0
        append!(path_nodes, node)
        node = dijkstra.parents[node]
    end

    return path_nodes, distance
end

end # module ShortestPath
```

Referencias

[1] James Fairbanks, Mathieu Besançon, Schölly Simon, Júlio Hoffiman, Nick Eubank, and Stefan Karpinski. Juliagraphs/graphs.jl: an optimized graphs package for the julia programming language, 2021.

[2] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B. Shah. Julia: A fresh approach to numerical computing. *SIAM Review*, 59(1):65–98, 9 2017.