Universidad Veracruzana

Inteligencia Artificial

Métodos Probabilísticos

Actividad 6

Leonardo Flores Torres

13 de noviembre 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. El algoritmo de Dijkstra no fue implementado sino que se utilizó una librería 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.

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> 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);
```

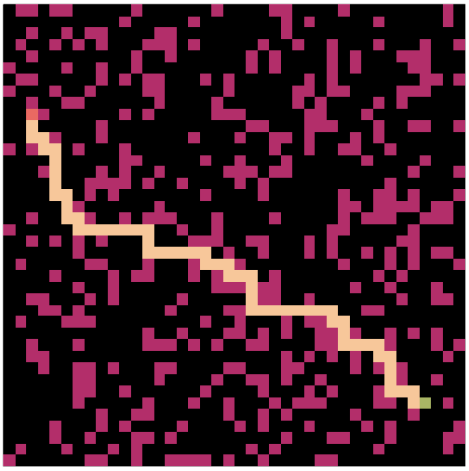
Las matrices de adyacencia guardadas en las variables `adjmat` y `adjmatdiag` requieren como argumentos la lista de obstaculos en `obstacles`, si las diagonales son permitidas o no en `allowdiags`, y el número de filas y de columnas. Estos dos primeros argumentos son posicionales mientras que los primeros 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.

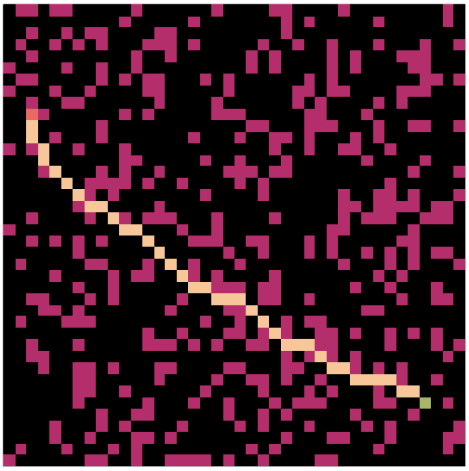
```
julia> path, dist = sp.findPath(adjmat, ncols; start=start, finish=finish);

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
```

En la figura 1 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 1a, y el caso en que el movimiento diagonal está permitido, figura 1b. Las distancias del recorrido en cada una de ellas son de 50 unidades sin diagonales en la figura 1a, y de 46.11 unidades considerando diagonales en la figura 1b.

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.



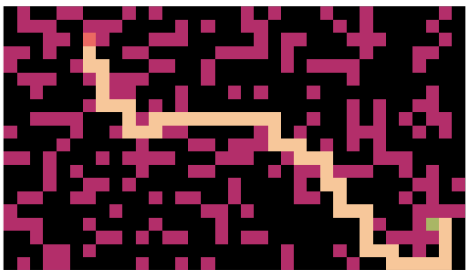
(a) 4 vecinos (diagonales no permitidas); distancia = $59u$.



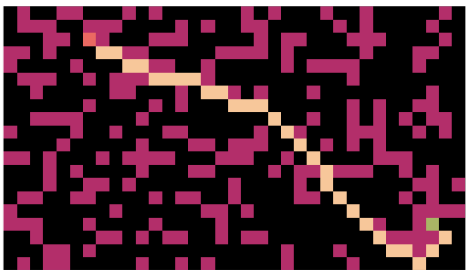
(b) 8 vecinos (diagonales permitidas); distancia = $46.1126u$.

Figura 1: Rejilla de 40×40 con obstaculos al 25 %.

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 2, 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 2: Rejilla de 20×35 con obstáculos del 30 %.

Por completez quisiera explicar la manera en como se computa la matriz de adyacencia, necesaria para pasarla como argumento a las funciones que se encargan de las operaciones para grafos. El problema se reduce a tener una rejilla de la forma

Apéndice

```
module ShortestPath

# NOTE: There are strange cases where findAll does not work.

#==============================
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,
        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)

    canvas[[obstacles...]] .= obscolor

    obstacles = obstacles .|> x -> id(x[1], x[2], ncols)
    sort!(obstacles)

    return canvas, obstacles
end


function updateneighborhood!(neighborhood, path)
    ncols = size(neighborhood)[2]
    pathcoordinates = posbyid.(path, ncols) .|> x -> CartesianIndex(x)

    neighborhood[pathcoordinates[1]] = RGB(169/255, 182/255, 101/255)
    neighborhood[pathcoordinates[end]] = RGB(234/255, 105/255, 98/255)
    neighborhood[pathcoordinates[begin+1:end-1]] .= RGB(247/255, 199/255, 154/255)
end


#==============================
Neighbors and path finding
==============================#

function adjacencyMatrix(nrows, ncols;
    obstacles=nothing, allowdiags=false
)
    ns = nrows * ncols          # neighborhood size
    ids = 1:ns                  # neighbors ids
    adjmat = zeros(ns, ns)
    diagstepsize = sqrt(2)      # diagonal step size

    for id in ids
        # neighbor to the right
        if !iszero(id % ncols)
            # adjmat[id, id + 1] = adjmat[id + 1, id] = 1
            adjmat[id, id + 1] = 1
        end
        # neighbor to the left
        if !iszero((id - 1) % ncols)
            adjmat[id, id - 1] = 1
        end
        # neighbor above
        if id > ncols
            adjmat[id, id - ncols] = 1
        end
        # neighbor below
        if id <= ncols * (nrows - 1)
            # adjmat[id, id + ncols] = adjmat[id + ncols, id] = 1
```

```
            adjmat[id, id + ncols] = 1
        end

        if allowdiags
            #neighbor up-right
            if !iszero(id % ncols) && id > ncols
                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


```

```
# DEPRECATED
function doit(nrows, ncols, start, finish, density, diags)
    # nrows = 7
    # ncols = 11

    # start = [6,1]
    # finish = [2,10]

    startid = id(start[1], start[2], ncols)
    finishid = id(finish[1], finish[2], ncols)

    idmat = [i for i in 1:(ncols*nrows)] ▷ x -> reshape(x, ncols, nrows) ▷ transpose
    nh, obs = neighborhood(nrows, ncols; start=start, finish=finish, obsdensity=density)

    adjmat = adjacencyMatrix(nrows, ncols; obstacles=obs, allowdiags=diags)
    dist, path = findAll(adjmat; start=startid, finish=finishid)

    updateneighborhood!(nh, path)
    visualize(nh)

    return dist, path, nh, idmat, adjmat, startid, finishid
end

# julia> dist, path, nh, idmat, adjmat, sid, fid
# = sp.doit(40, 40, [4,1], [35,37], 0.1, false); sp.plot(nh)

end # module ShortestPath
```

Referencias

[1] 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.

[2] 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.