Visibility Graphs

Joan Saló (u1953621)
Isaac de Palau (u1928988)

May 2023

El problema que intentem resoldre en aquest treball és el de trobar el camí més curt en el pla donats una sèrie d'obstacles o punts a evitar. A grans trets, necessitem primer construir un graf (el **graf de visibilitat**), i a partir d'aquí el problema es redueix a trobar el camí més curt entre dos punts d'aquest graf. El repositori amb el codi es pot consultar en aquest enllaç.

In Theory

0.1 *Visibility Graph*

Comencem per repassar la noció matemàtica de graf:

Definition 0.1 (Graph)**.** Un **graf** és una tripla (V, E, f) formada per un conjunt no buit V (anomenat el *conjunt de vèrtex*, o *conjunt de nodes*), un conjunt E anomenat el conjunt d'arestes, i una funció $f : E \longrightarrow V \times V$ que assigna una tupla de vèrtex a cada aresta de E. □

Si f assigna una tupla (v_1, v_2) a una aresta e, posarem $e = \overline{v_1\, v_2}$.

Definition 0.2 (Visibility graph)**.** Donat un conjunt disjunt d'obstacles poligonals S, podem definir graf de visibilitat $G_{vis}(S)$. Els nodes d'aquest graf són els vèrtexs de S, i hi ha un arc entre els vèrtexs v i w si es poden veure entre ells, és a dir, si el segment $\overline{v\, w}$ no interseca amb cap dels obstacles a S.□

Per tal de generar el graf de visibilitat hi ha tres algorismes, els quals explicarem breument a continuació.

0.1.1 Graf de visibilitat *naïve*

L'algorisme és senzill i es basa en el que ja hem explicat al punt anterior, i consisteix a afegir tots els vèrtexs d'S a G. Per cada parella de vèrtexs (v, w) de G, comprovem si v i w són visibles, és a dir, si $\overline{v\, w}$ no interseca amb cap altra aresta de S. Això fa que la complexitat de l'algorisme sigui de $O(n^3)$

0.1.2 Graf de visibilitat de D.T. Lee

L'algorisme de Lee fou la primera solució no trivial al problema de construïr grafs de visibilitat. La idea d'aquest algorisme consisteix a considerar únicament una part dels vèrtexs a l'hora de construir el graf de visibilitat, la qual cosa redueix la complexitat de l'algorisme a $O(n^2 log_2 n)$.

Abans de presentar el pseudocodi d'aquest algorisme, intentarem formular una idea intuïtiva de com funciona aquest algorisme. Considerem un espai amb la següent configuració d'obstacles, i suposem que volem construir el graf de visibilitat a partir del punt `s`:

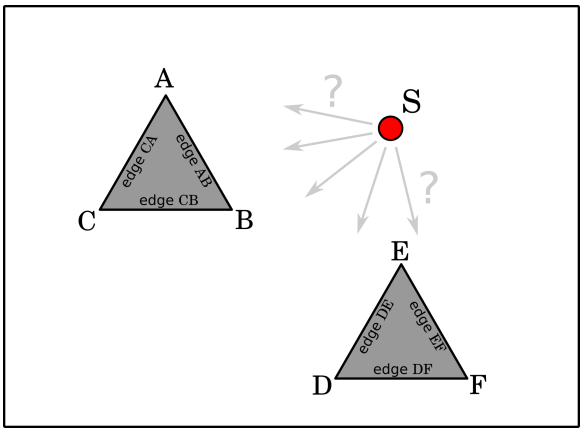


Figura 1: Tenim dos obstacles definits pels triangles $\widehat{ABC}$ i $\widehat{DEF}$ i volem construir el graf de visibilitat a partir del punt S.

Per tal de saber quins nodes es poden veure des d'`s`, cal visitar cada node `A`,`B`, `C`,...,`F`. La forma de fer-ho serà traçant un raig (*scanline*) que sorgirà de `s` i recorrerà tot l'espai en sentit contrari a les agulles del rellotge. Cada vegada que aquesta *scanline* toqui un node, comprovarà la visibilitat.

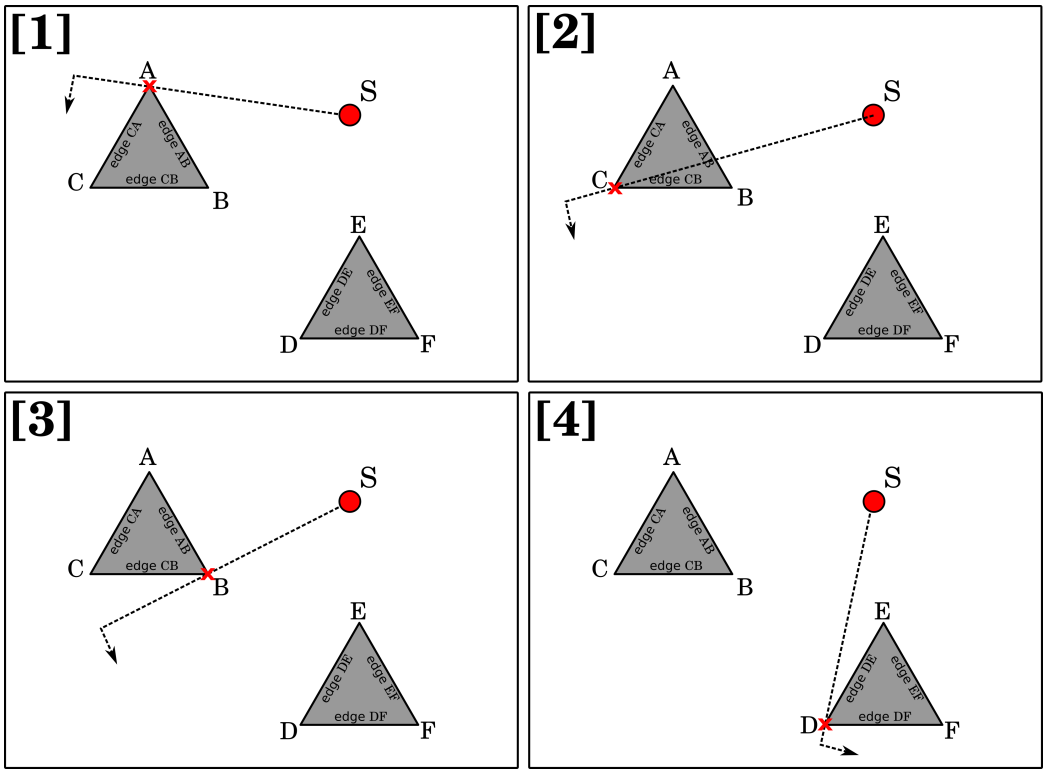


Figura 2: La *scanline* sorgeix de `s` i va recorrent els nodes en sentit contrari a les agulles del rellotge.

Fixem-nos en la primera imatge de la figura anterior, en què la *scanline* toca amb el punt A. Aquest punt té dues arestes incidents a ella ($\overline{AB}$ i $\overline{AC}$). El que farà l'algorisme és comprovar si hi ha alguna d'aquestes arestes en el sentit "contra rellotge" en què està circulant la scanline, en cas afirmatiu les afegirà a una llista anomenada la `OPEN_LIST`.

Continuant amb la figura anterior, fixem-nos en la segona imatge en què la *scanline* interseca amb el punt `C`. L'aresta `AC` queda ara en el sentit de les agulles del rellotge respecte de la *scanline*, per tant, mai tornarà a ser intersecada per aquesta i podem eliminar aquesta aresta de la `OPEN_LIST`. En canvi, l'aresta `BC` queda en el sentit contrari de les agulles del rellotge respecte de la *scanline*, per tant, l'afegim a la llista. Per cada punt, anem seguint aquest procediment: calculem les arestes incidents; si queden en sentit contrari a les agulles del rellotge, la posem a la `OPEN_LIST` (o la mantenim, si ja hi és). Altrament, l'eliminem de la `OPEN_LIST` (o directament no li afegim).

Anem ara a veure com es calcula la visibilitat. Quan la *scanline* visita el punt `A`, mira la llista `OPEN_LIST` per tal de saber si hi ha alguna aresta que potencialment pugui bloquejar la línia de visió; és evident que en el punt `A` no n'hi pot haver cap, per tant, l'algorisme continua. En canvi, quan la *scanline* es topa amb el punt `C`, veu que dins la `OPEN_LIST` hi ha l'aresta `AB` que és tallada per la *scanline*. Per tant, el punt `B` no és visible. Per tant, fixem-nos que la idea clau d'aquest algorisme és que només es fixa en les arestes que són rellevants i que, potencialment, poden bloquejar la visió, mentre que la implementació naïve comprovaria <u>*totes*</u> les arestes a cada punt.

Anem a veure ara una formulació bàsica d'aquest algorisme en pseudocodi. Noti's que aquesta formulació no és del tot elemental, i acabarà depenent en bona part de les eines i estructures de dades que tinguem disponibles amb el llenguatge i entorn en què estiguem programant.

Algorithm 1: Algorisme D.T. de Lee

Suposem que tenim un punt d'origen a partir del que volem calcular el graf de visibilitat, i una sèrie d'obstacles poligonals.

Pas 1: ordenar els vèrtexs dels polígons dels obstacles en sentit contrari a les agulles del rellotge respecte al segment que uneix el punt d'origen amb cada vèrtex. (Nota: un canvi a coordenades polars converteix aquest pas en trivial). En cas d'empat, el vèrtex més proper a l'origen té preferència. Sigui $w_1, w_2, ..., w_n$ aquesta llista ordenada de vèrtex.

Pas 2: Sigui `s` la *scanline*. Definim també una llista `W` buida, on anirem guardant els vèrtexs visibles des de l'origen, i un arbre binari `T` buit, on anirem guardant les arestes que intersequin amb `s`.

Pas 3: Per cada vèrtex $i = 1, 2, ..., n$, fer:
- **Pas 3.1 :** Si `visible(w_i)`, afegir `w_i` a `W`.
- **Pas 3.2 :** Insertar a `T` les arestes incidents a `w_i` que estiguin en sentit contrari a les agulles del rellotge respecte `s`.
- **Pas 3.3 :** Eliminar de `T` les arestes incidents a `w_i` que estiguin en el sentit de les agulles del rellotge respecte `s`.

Pas 4: retornar `W`.

I a on la funció `visible(w_i)` es defineix pels següents passos:

Pas 1: Si `T` buit, retornar `True`.
Pas 2: Altrament, si l'aresta de l'origen a `w_i` no interseca l'aresta que estigui més a l'esquerra dins de `T`, retornar `True`.
Pas 3: Altrament, retornar `False`.

0.1.3 Graf de visibilitat de Gosh/Mount

Tot i que aquest algorisme s'executa en $O(e + nlog_2 n)$, i per tant és el més ràpid dels tres, la seva implementació és molt complexa i queda fora de l'abast d'aquesta pràctica. Per veure els detalls del funcionament i la implementació, vegeu [Rek16].

0.2 *Pathfinding*

Havent construït el graf de visibilitat, el problema de trobar la ruta entre dos punts arbitraris del pla es redueix a un simple problema de cerca de camins en un graf simple (o el que es coneix com a *graph pathfinding*, o *pathfinding*). En aquesta secció revisarem dos dels algorismes per excel · lència més utilitzats per fer pathfinding: Dijkstra i A-star.

0.2.1 Dijkstra

L'algorisme de Dijkstra (1959) fou un dels primers algorismes que va aconseguir resoldre de forma satisfactòria el problema de trobar el camí (**path**) més curt entre dos vèrtexs d'un graf. Aquest

algorisme és un dels dos que hem utilitzat en la nostra pràctica, i abans d'estudiar-lo introduirem algunes nocions teòriques addicionals.

Donat que la majoria de conceptes no tenen una traducció adequada al català, en molts casos usarem els seus noms en anglès per referir-nos-hi.

Definition 0.3 (Graf amb pesos (*Weighted graph*)**.** Sigui $G = (V, E, f)$ un graf connex simple. Sigui $w : E \longrightarrow \mathbb{R}^+$ una funció, anomenada **pes**, que assigna un valor real positiu (o zero) a cada aresta del graf G. En aquest cas, direm que (G, w) és un *graf amb pesos* (**weighted graph**). □

Definition 0.4 (Walk, trail i path)**.** Un **walk** de llargada k entre dos vèrtexs $v_1, v_k \in V$, $v_1 \neq v_k$ d'un graf $G = (V, E)$ és una seqüència de k arestes de la forma:

$$\overline{v_1\, v_2}, \overline{v_2\, v_3}, \overline{v_3\, v_4}, ..., \overline{v_{k-1}\, v_k}$$

Normalment, denotarem un *walk* únicament pels vèrtexs que el formen: $v_1v_2v_3...v_k$.
Un **trail** és un *walk* en el que no es repeteixen arestes. Un **path** és un trail en el qual tampoc es repeteixen vèrtex. □

Definition 0.5 (Distància entre nodes)**.** Donat un *weighted graph* (G, w) podem definir la *llargada amb pesos* (**weighted length**) d'un *walk* $C = v_1v_2v_3...v_k$ com a

$$\text{length}_w(C) = \sum_{j=1}^{k-1} w(\overline{v_j\, v_{j+1}}) \ .$$

La *distància amb pesos* (**weighted distance**) $d_\omega(a, b)$ entre dos vèrtexs $a, b \in V$ es defineix com la menor *weighted length* d'entre tots els *paths* que uneixen els vèrtexs a i b. □

Sigui ara $S \subseteq V$, i definim $\overline{S} := V \setminus S$. Sigui també

$$d_\omega(u, \overline{S}) = min\big\{d_\omega(u, v) \,|\, v \in \overline{S}\big\} \ .$$

Aleshores, és trivial provar que $\forall u \in S$,

$$d_\omega(u, \overline{S}) = min\big\{d_\omega(u, v) + \omega(\overline{vz}) \,|\, v \in S, z \in \overline{S}\big\} \ .$$

Amb això ja tenim totes les eines per introduir l'algorisme de Dijkstra. El presentarem de dues maneres: la primera, com a algorisme teòric, i la segona com a algorisme en pseudocodi que podria ser implementat en una màquina.

Comencem amb l'algorisme teòric de Dijkstra. Aquest algorisme, com a tal, no ens dona verta-derament la distància més curta entre dos nodes ni el seu camí, sinó que dóna la distància més curta entre un node i tots els altres nodes del graf.

Algorithm 2: Algorisme de Dijkstra (versió teòrica)

Sigui (G, w) un graf amb pesos w, on $G = (V, E)$.

Pas 1: Elegim $u \in V$, i denotem $u_1 = u$ i $S_1 = \{u_1\}$. Busquem ara un $u_2 \in \overline{S_1}$ tal que

$$d_w(u_1, \overline{S_1}) = min\big\{d_w(u_1, v) + w(\overline{vz} \,|\, v \in S_1, z \in \overline{S_1}\big\} = w(\overline{u_1u_2}) \, .$$

Denotem $S_2 = S_1 \cup \{u_2\}$.

Passos 2+: Suposem que hem trobat la següent successió de conjunts de vèrtex:

$$S_1 \subset S_2 \subset ... \subset S_k$$

amb 1 < k < n, on n és el nombre de vèrtexs de V. Busquem ara un $u_{k+1} \in \overline{S_k}$ que ens satisfaci:

$$d_w(u_1, \overline{S_k}) = d_w(u_1, u_j) + w(\overline{u_j u_{k+1}}),$$

per algun $j \in \{1, ..., k\}$. Denotem per $S_{k+1} = S_k \cup \{u_{k+1}\}$.
Aturem aquest procés si $k = n$ on finalment obtenim $S_n = V(G)$.

Aquest plantejament de l'algorisme té poca utilitat fora de l'àmbit purament teòric. El segon plantejament que proposem és una implementació amb pseudocodi que pot ser adaptada i implementada (amb relativa facilitat) fent servir qualsevol llenguatge de programació genèric:

Algorithm 3: Algorisme de Dijkstra (pseudocodi)

Data: Conjunt de vèrtex V, arestes E. Dos arrays `prev` i `dist` . Vèrtex inicial `origen` $\in V$ i vèrtex final `desti` $\in V$ amb `origen` $\neq$ `desti`.

```
Crear Q llista buida;
for cada v ∈ V do
    dist[v] ← +∞;
    prev[v] ← NONE;
    Afegir v a Q;
end
dist[origen] ← 0;
while Q no buida do
    u ← vertex de Q amb menor dist[u];
    if u = desti then
        Sortir bucle;
    end
    Eliminar u de Q;
    for cada v ∈ Q i v veí de u do
        alt ← dist[u] + pes(E(u, v));
        if alt < dist[v] then
            dist[v] ← alt;
            prev[v] ← u
        end
    end
end
S ← array buit;
u ← desti;
if prev[u] ≠ NONE, o si u = origen then
    while u ≠ NONE do
        Insertar u al principi de S;
        u ← prev[u];
    end
end
```

0.2.2 A-star

L'algorisme A-star (també anomenat A* o A-estrella) es pot considerar una variant o extensió de l'algorisme de Dijkstra en el qual definim una funció heurística per fer una estimació de quins nodes tenen més possibilitats d'estar més a prop i donar-los així prioritat a l'hora d'expandir-los. En el cas mitjà, això es pot comprovar que fa que l'algorisme A-star sigui més ràpid que Dijsktra i, en el pitjor dels casos, l'iguala.

Algorithm 4: Algorisme A-star (pseudocodi)

Data: Conjunt de vèrtex V, arestes E. Quatre arrays `OPEN`, `CLOSE`
, `dist`, `prev`. Vèrtex inicial `origen` $\in V$ i vèrtex final `desti` $\in V$ amb `origen` $\neq$ `desti`.
Funció heurística $h(v)$ que, donat un node v, retorna la seva distància estimada a `desti`.

```
Afegir desti a OPEN;
dist[origen] ← 0);
while OPEN no estigui buida do
    u ← node de OPEN amb menor dist[u] + h(u);
    if u = desti then
        Sortir del bucle;
    end
    for cada node v veí de u do
    end
end
S ← array buit;
u ← desti;
if prev[u] ≠ NONE, o si u = origen then
    while u ≠ NONE do
        Insertar u al principi de S;
        u ← prev[u];
    end
end
```

In praxis

En aquesta part del document utilitzarem la tècnica dels grafs de visibilitat per a trobar el camí més curt entre dos punts en un espai bidimensional donats una sèrie d'obstacles definits en forma de polígon. El fet del canvi de format en aquesta part del document és degut al fet que aquesta s'ha creat a partir de `Jupyter Notebook`, i adjuntada al fitxer original posteriorment, degut a la difucultat d'integrar-ho tot des del principi.

Tot i que la versió original de `pyvisgraph` (la llibreria usada), només implementa l'algorisme de `Dijkstra` per tal de trobar el camí més curt entre dos punts en un graf, nosaltres usarem una versió modificada que inclou l'algorisme `A*`, per comparar el seu rendiment.

Hem partit dels exemples existents a la llibreria per tal de realitzar aquest notebook, tot i que amb algunes modificacions.

Comencem carregant les llibreries necessàries:

```
import geopandas as gpd
import geoviews as gv
import geoviews.feature as gf
import hvplot.pandas
import shapefile
import pyvisgraph as vg
import holoviews as hv
from holoviews import opts
from pyvisgraph.visible_vertices import visible_vertices
import time
import matplotlib.pyplot as plt
from PIL import Image
from IPython.display import display_png

renderer = hv.renderer('bokeh')
```

El primer exemple consisteix en definir dos obstacles simples, per després poder trobar el camí més curt donats dos punts.

Així doncs, primer comencem definint aquests obstacles:

```
polys = [[vg.Point(0.0,1.0), vg.Point(3.0,1.0), vg.Point(1.5,4.0)],
         [vg.Point(4.0,4.0), vg.Point(7.0,4.0), vg.Point(5.5,8.0)]]
```

I en construïm el seu graf de visibilitat:

```
g = vg.VisGraph()
g.build(polys)
```

```
100%|
 1/1 [00:00<?, ?it/s]
```

A partir d'aquí, podem calcular el camí més curt entre dos punts donats:

```
shortest = g.shortest_path(vg.Point(1.5,0.0), vg.Point(8.0, 8.0))
shortest
```

```
[Point(1.50, 0.00), Point(3.00, 1.00), Point(7.00, 4.00), Point(8.00, 8.00)]
```

La sèrie de passos següents són necessaris per tal de representar gràficament el graf de visibilitat i el camí més curt obtinguts.

Primer, canviem el format dels punts del camí més curt (ho convertim a llista de tuples d'enters):

```
shortest_path = [(point.x, point.y) for point in shortest]
```

Fem el mateix amb els punts dels polígons. Fins ara estàvem representant punts en l'espai, però ara ens interessa construir un camí que uneixi aquests punts. Per això, dupliquem el punt inicial (per 'tancar la línia').

```
paths = []
for poly in polys:
    poly_aux = []
    for point in poly:
        poly_aux.append((point.x, point.y))
    poly_aux.append(poly_aux[0])
    paths.append(poly_aux)
paths
```

```
[[(0.0, 1.0), (3.0, 1.0), (1.5, 4.0), (0.0, 1.0)],
 [(4.0, 4.0), (7.0, 4.0), (5.5, 8.0), (4.0, 4.0)]]
```

Els ajuntem en una sola llista:

```
all_points = set([point for point in poly for poly in polys] + shortest)
all_points
```

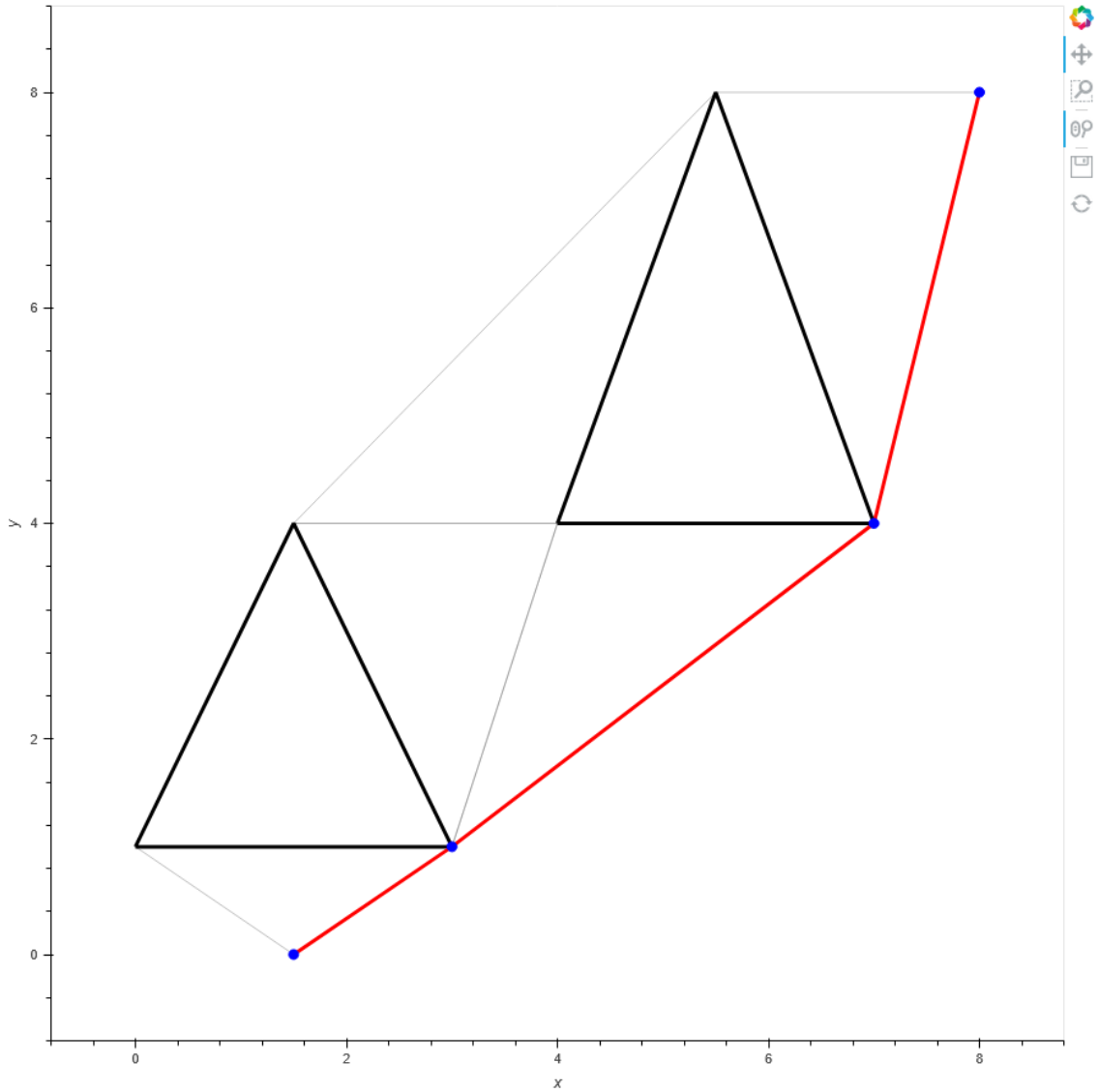
```
{Point(4.00, 4.00),
 Point(8.00, 8.00),
 Point(3.00, 1.00),
 Point(7.00, 4.00),
 Point(1.50, 0.00),
 Point(5.50, 8.00)}
```

I calculem, per cada punt del mapa, quins són els punts visibles:

```
visible_points = []

for start_point in all_points:
    visible = visible_vertices(start_point, g.graph, None, None)
    for point in visible:
        visible_points.append([(start_point.x, start_point.y), (point.x, point.y)])
```

Finalment, representem els resultats obtinguts del graf de visibilitat. En blau, els vèrtexs del camí mínim, i les arestes en vermell. En color negre, les arestes del graf que representen els obstacles, i la resta d'arestes del graf, en gris.

```
overlay = hv.Path(visible_points).opts(opts.Path(color='grey', line_width=0.5)) \
    * hv.Path(paths).opts(opts.Path(color='black', line_width=3)) \
        * hv.Path(shortest_path).opts(opts.Path(color='red', line_width=3)) \
            * hv.Points(shortest_path).opts(opts.Points(color='blue', size=8))

plot = overlay.opts(width=900, height=900)

png, info = renderer(plot, fmt='png')
display_png(png, raw=True)
```

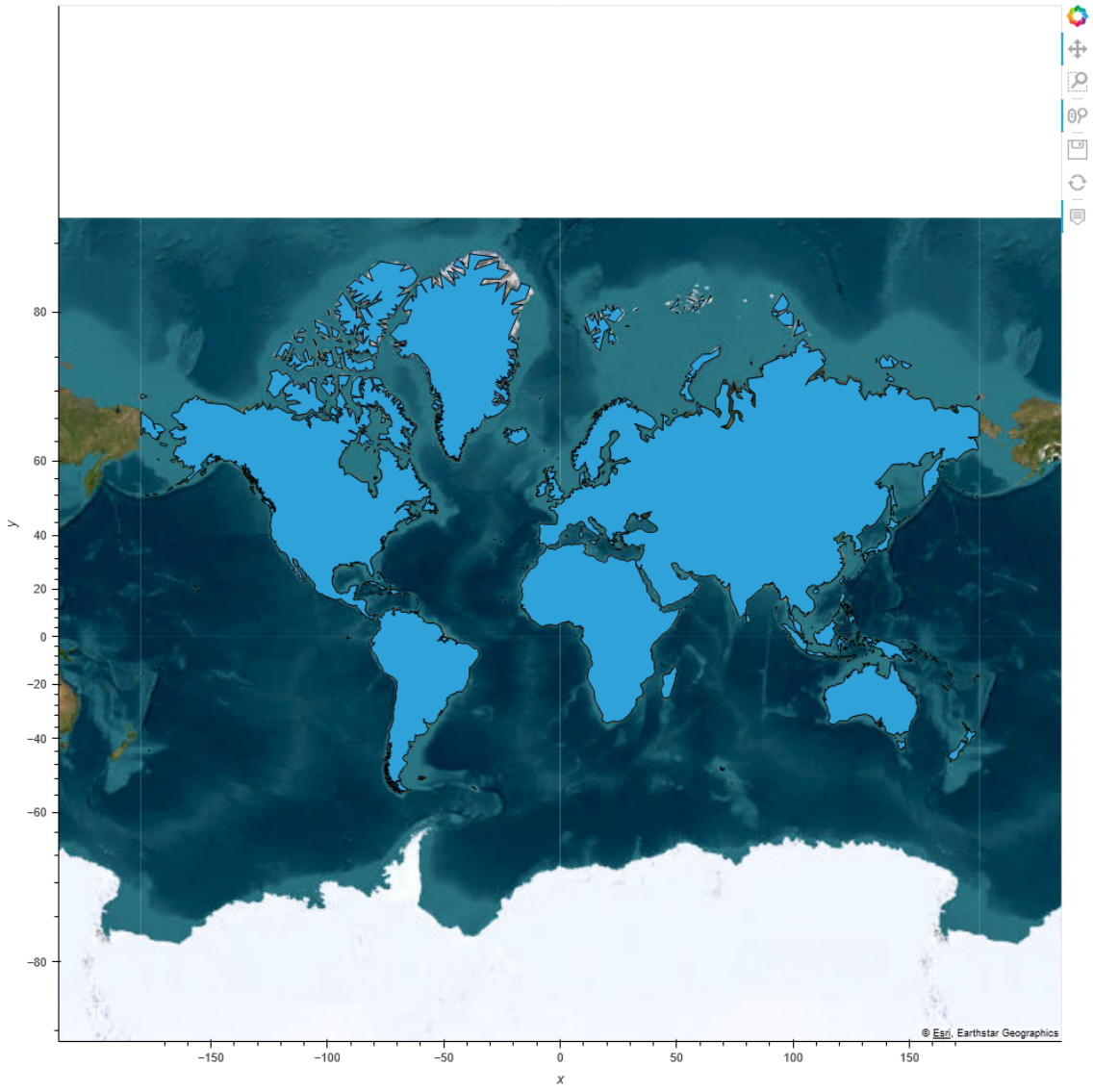


El segon exemple és semblant al primer, però trobant el camí més curt per mar donats dues coordenades qualsevols de la superficie terrestre. El primer que hem de fer és importar el fitxer `shoreline/GSHHS_c_L1`. Aquest fitxer és de tipus `shapefile`, i conté una capa poligonal dels continents (els obstacles):

```
input_shapefile = shapefile.Reader('shoreline/GSHHS_c_L1')
```

Ara el tornem a llegir però amb la llibreria `geopandas`, per tal de poder representar-lo gràficament:

```
df = gpd.read_file('shoreline/GSHHS_c_L1.dbf')
plot = df.hvplot(width=1000, height=1000, geo=True, tiles='ESRI')

png, info = renderer(plot, fmt='png')
display_png(png, raw=True)
```



Mirem quants polígons diferents hi ha:

```
shapes = input_shapefile.shapes()
print('El shapefile conté {} polígons.'.format(len(shapes)))
```

```
El shapefile conté 742 polígons.
```

I també quants punts diferents formen aquests polígons:

```
points = sum([len(shape.points) for shape in shapes])

print(f"El shapefile conté {points} punts.")
```

```
El shapefile conté 7282 punts.
```

Construïm el graf de visibilitat executant la següent cel·la de codi. És un procés costós, que tarda uns 15 minuts. Si el paral·lelitzem, en el nostre cas en `10` nuclis, aleshores tarda uns segons. També el guardem a disc, per després poder-lo recuperar si ens interessa.

```
# Crear llista de polígons
polygons = []
for shape in shapes:
    polygon = []
    for point in shape.points:
        polygon.append(vg.Point(point[0], point[1]))
    polygons.append(polygon)

# Construïr el graf de visibilitat
output_graphfile = 'GSHHS_c_L1.graph'
graph = vg.VisGraph()
graph.build(polygons, workers = 10)

graph.save(output_graphfile)
```

```
100%|
```

```
 654/654 [02:29<00:00,  4.37it/s]
```

Ara passem a definir dos punts qualssevol (en coordenades GPS), que representaran el punt inicial i final del camí:

```
start_point = vg.Point(12.568337, 55.676098) # Copenhagen
end_point = vg.Point(103.851959, 1.290270) # Singapur
```

En calculem el camí més curt, primer utilitzant l'algorisme de `Dijkstra`:

```
startTime = time.time()
shortest_path_dijsktra  = graph.shortest_path(start_point, end_point)
print("Camí més curt amb algorisme de Dijkstra's trobat en {}
segons.".format(time.time() - startTime))
```

```
Camí més curt amb algorisme de Dijkstra's trobat en 0.9575605392456055 segons.
```

I altra vegada, però utilizant l'algorisme l'algorisme de `A*`:

```
startTime = time.time()
shortest_path_astar  = graph.shortest_path(start_point, end_point, solver = "astar")
print("Camí més curt amb algorisme de A*'s trobat en {} segons.".format(time.time() -
startTime))
```

```
Camí més curt amb algorisme de A*'s trobat en 0.5655708312988281 segons.
```

Veiem que el **temps d'execució és considerablement menor** (la meitat aproximadament).

Ara, tornem a fer un canvi de format per poder representar gràficament la solució:

```
visible_points = []

for start_point in shortest_path_astar:
    visible = visible_vertices(start_point, graph.graph, None, None)
    for point in visible:
        visible_points.append([(start_point.x, start_point.y), (point.x, point.y)])
```

I representem. Enlloc de tot el graf, simplement mostrem el camí més curt, i les arestes que porten als punts visibles des dels punts del camí més curt (ja que altrament tindríem massa arestes):

```
points = [(point.x, point.y) for point in shortest_path_dijsktra]
path = gv.Path(points)

# Customize the appearance of the path and add a basemap
plot = df.hvplot(width=1500, height=1000, geo=True, tiles='ESRI') \
    * gv.Path(visible_points).opts(opts.Path(color='grey', line_width=1)) \
        * path.opts(opts.Path(color='red', line_width=3)) \
            * gv.Points(path).opts(opts.Points(color='k', size=8))

png, info = renderer(plot, fmt='png')
display_png(png, raw=True)
```

2 Fonts

Fonts

[Rek16] Christian Reksten-Monsen. *Distance Tables Part 2: Lee's Visibility Graph Algorithm*. 2016. URL: https://taipanrex.github.io/2016/10/19/Distance-Tables-Part-2-Lees-Visibility-Graph-Algorithm.html.