Probabilistic-Roadmap-Solution

June 10, 2021

1 Probabilistic Roadmap

In this notebook you'll expand on previous random sampling exercises by creating a graph from the points and running A*.

1. Load the data
2. Sample nodes
3. Connect nodes
4. Visualize graph
5. Define heuristic
6. Define search method
7. Execute and visualize

We'll load the data and provide a template for visualization.

In [1]:
```
# Again, ugly but we need the latest version of networkx!
# This sometimes fails for unknown reasons, please just
# "reset and clear output" from the "Kernel" menu above
# and try again!
import sys
!{sys.executable} -m pip install -I networkx==2.1
import pkg_resources
pkg_resources.require("networkx==2.1")
import networkx as nx
```

```
Collecting networkx==2.1
  Downloading https://files.pythonhosted.org/packages/11/42/f951cc6838a4dff6ce57211c4d7f8444809c
    100% || 1.6MB 10.7MB/s ta 0:00:01   12% |                              | 204kB 4.4MB/s eta 0:0
Collecting decorator>=4.1.0 (from networkx==2.1)
  Downloading https://files.pythonhosted.org/packages/6a/36/b1b9bfdf28690ae01d9ca0aa5b0d07cb4448
Building wheels for collected packages: networkx
  Running setup.py bdist_wheel for networkx ... done
  Stored in directory: /root/.cache/pip/wheels/44/c0/34/6f98693a554301bdb405f8d65d95bbcd3e50180c
Successfully built networkx
scikit-image 0.14.2 has requirement dask[array]>=1.0.0, but you'll have dask 0.16.1 which is inc
pomegranate 0.9.0 has requirement networkx<2.0,>=1.8.1, but you'll have networkx 2.1 which is in
moviepy 0.2.3.2 has requirement decorator==4.0.11, but you'll have decorator 5.0.9 which is inco
```

```
jupyterlab-server 1.0.0 has requirement jsonschema>=3.0.1, but you'll have jsonschema 2.6.0 whic
ipywidgets 7.0.5 has requirement widgetsnbextension~=3.0.0, but you'll have widgetsnbextension 3
Installing collected packages: decorator, networkx
Successfully installed decorator-5.0.9 networkx-2.1
```

```
In [2]: nx.__version__ # should be 2.1
```

```
Out[2]: '2.1'
```

```
In [3]: import numpy as np
        import matplotlib.pyplot as plt
        from sampling import Sampler
        from shapely.geometry import Polygon, Point, LineString
        from queue import PriorityQueue

        %matplotlib inline
```

```
In [4]: plt.rcParams['figure.figsize'] = 12, 12
```

1.1 Step 1 - Load Data

```
In [5]: # This is the same obstacle data from the previous lesson.
        filename = 'colliders.csv'
        data = np.loadtxt(filename, delimiter=',', dtype='Float64', skiprows=2)
        print(data)
```

```
[[-310.2389   -439.2315     85.5          5.           5.          85.5     ]
 [-300.2389   -439.2315     85.5          5.           5.          85.5     ]
 [-290.2389   -439.2315     85.5          5.           5.          85.5     ]
 ...,
 [ 257.8061    425.1645      1.75852      1.292725     1.292725     1.944791]
 [ 293.9967    368.3391      3.557666     1.129456     1.129456     3.667319]
 [ 281.5162    354.4156      4.999351     1.053772     1.053772     4.950246]]
```

1.2 Step 2 - Sample Points

We've implemented a custom sampling class using a k-d tree.

```
In [6]: from sampling import Sampler
```

```
In [7]: sampler = Sampler(data)
        polygons = sampler._polygons
```

```
In [8]: # Example: sampling 100 points and removing
        # ones conflicting with obstacles.
        nodes = sampler.sample(300)
        print(len(nodes))
```

```
195
```

1.3 Step 3 - Connect Nodes

Now we have to connect the nodes. There are many ways they might be done, it's completely up to you. The only restriction being no edge connecting two nodes may pass through a polygon.

NOTE: You can use `LineString` to create a line. Additionally, `shapely` geometry objects have a method `.crosses` which return `True` if the geometries cross paths.

```
In [9]: import numpy.linalg as LA
        from sklearn.neighbors import KDTree
```

```
In [10]: def can_connect(n1, n2):
             l = LineString([n1, n2])
             for p in polygons:
                 if p.crosses(l) and p.height >= min(n1[2], n2[2]):
                     return False
             return True

         def create_graph(nodes, k):
             g = nx.Graph()
             tree = KDTree(nodes)
             for n1 in nodes:
                 # for each node connect try to connect to k nearest nodes
                 idxs = tree.query([n1], k, return_distance=False)[0]

                 for idx in idxs:
                     n2 = nodes[idx]
                     if n2 == n1:
                         continue

                     if can_connect(n1, n2):
                         g.add_edge(n1, n2, weight=1)
             return g
```

```
In [11]: import time
         t0 = time.time()
         g = create_graph(nodes, 10)
         print('graph took {0} seconds to build'.format(time.time()-t0))
```

```
graph took 32.01547956466675 seconds to build
```

```
In [12]: print("Number of edges", len(g.edges))
```

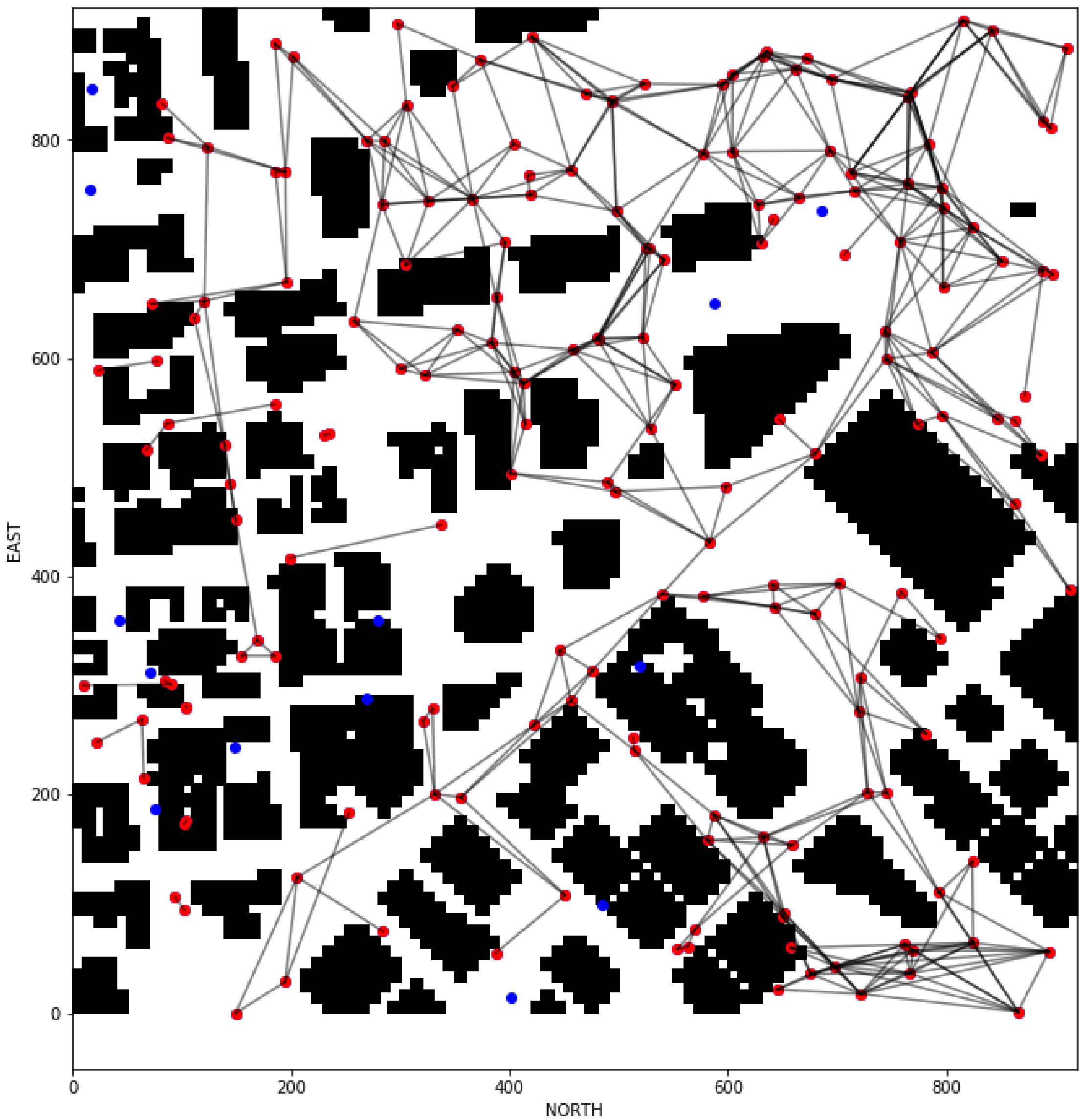
```
Number of edges 461
```

1.4 Step 4 - Visualize Graph

```
In [13]: from grid import create_grid
```

In [14]:
```
grid = create_grid(data, sampler._zmax, 1)
```

In [15]:
```
fig = plt.figure()

plt.imshow(grid, cmap='Greys', origin='lower')

nmin = np.min(data[:, 0])
emin = np.min(data[:, 1])

# draw edges
for (n1, n2) in g.edges:
    plt.plot([n1[1] - emin, n2[1] - emin], [n1[0] - nmin, n2[0] - nmin], 'black' , alph

# draw all nodes
for n1 in nodes:
    plt.scatter(n1[1] - emin, n1[0] - nmin, c='blue')

# draw connected nodes
for n1 in g.nodes:
    plt.scatter(n1[1] - emin, n1[0] - nmin, c='red')



plt.xlabel('NORTH')
plt.ylabel('EAST')

plt.show()
```



1.5 Step 5 - Define Heuristic

```
In [16]: def heuristic(n1, n2):
             # TODO: finish
             return LA.norm(np.array(n2) - np.array(n1))
```

1.6 Step 6 - Complete A*

```
In [17]: def a_star(graph, heuristic, start, goal):
             """Modified A* to work with NetworkX graphs."""

             # TODO: complete
```

```
    path = []
    queue = PriorityQueue()
    queue.put((0, start))
    visited = set(start)

    branch = {}
    found = False

    while not queue.empty():
        item = queue.get()
        current_cost = item[0]
        current_node = item[1]

        if current_node == goal:
            print('Found a path.')
            found = True
            break
        else:
            for next_node in graph[current_node]:
                cost = graph.edges[current_node, next_node]['weight']
                new_cost = current_cost + cost + heuristic(next_node, goal)

                if next_node not in visited:
                    visited.add(next_node)
                    queue.put((new_cost, next_node))

                    branch[next_node] = (new_cost, current_node)

    path = []
    path_cost = 0
    if found:

        # retrace steps
        path = []
        n = goal
        path_cost = branch[n][0]
        while branch[n][1] != start:
            path.append(branch[n][1])
            n = branch[n][1]
        path.append(branch[n][1])

    return path[::-1], path_cost
```

In [18]:
```
start = list(g.nodes)[0]
k = np.random.randint(len(g.nodes))
print(k, len(g.nodes))
goal = list(g.nodes)[k]
```

```
39 182
```

```
In [19]: path, cost = a_star(g, heuristic, start, goal)
         print(len(path), path)
```

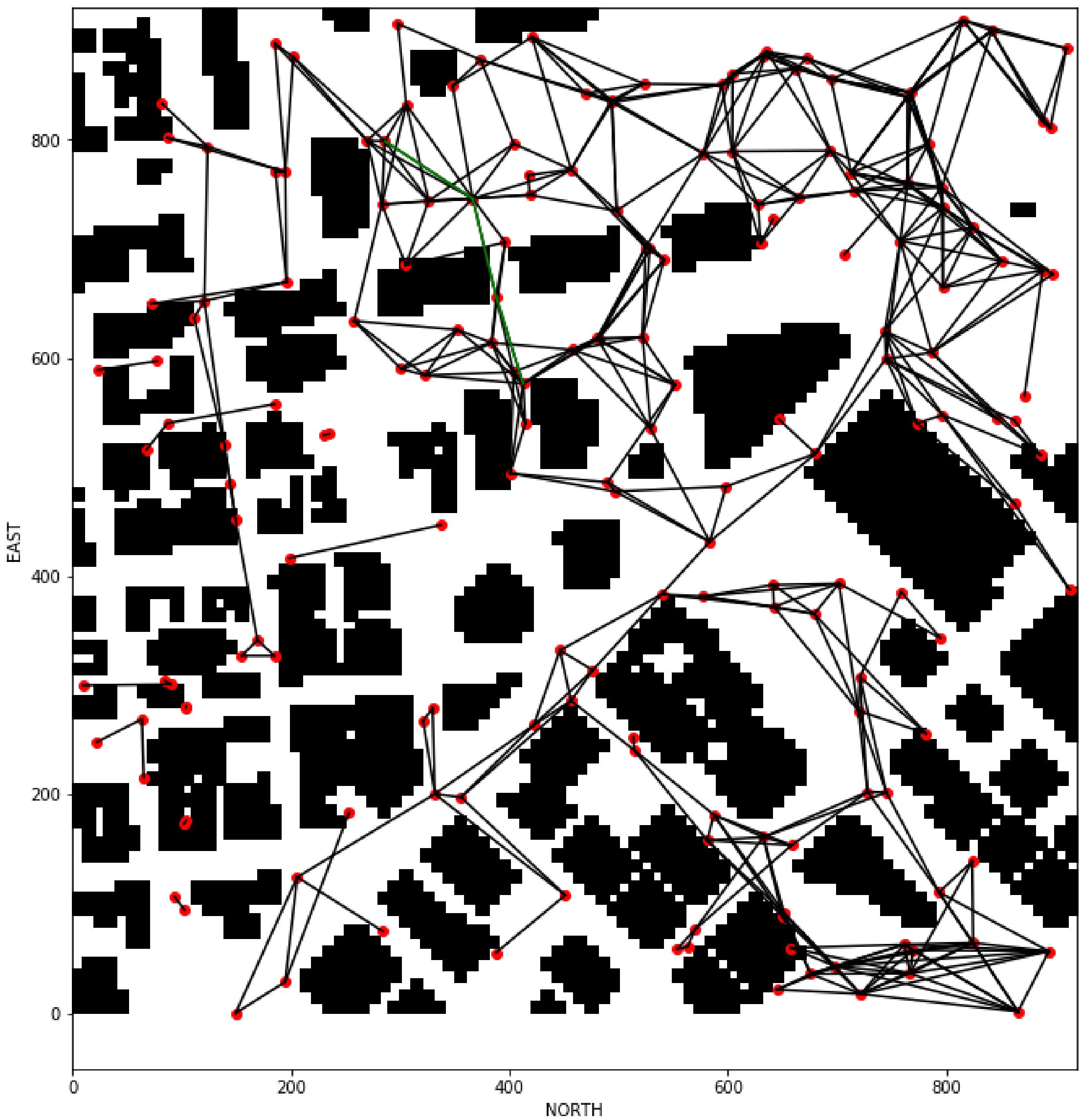
```
Found a path.
4 [(488.43611219052406, -153.23721914336522, 15.937696731153679), (435.85881789353112, -72.78266
```

```
In [20]: path_pairs = zip(path[:-1], path[1:])
         for (n1, n2) in path_pairs:
             print(n1, n2)
```

```
(488.43611219052406, -153.23721914336522, 15.937696731153679) (435.85881789353112, -72.782661833
(435.85881789353112, -72.782661833930547, 8.3975872188879475) (345.92301220374264, -51.391784138
(345.92301220374264, -51.391784138277671, 0.64022412438051512) (266.34072041983012, -26.50234374
```

1.7 Step 7 - Visualize Path

```
In [21]: fig = plt.figure()

         plt.imshow(grid, cmap='Greys', origin='lower')

         nmin = np.min(data[:, 0])
         emin = np.min(data[:, 1])

         # draw nodes
         for n1 in g.nodes:
             plt.scatter(n1[1] - emin, n1[0] - nmin, c='red')

         # draw edges
         for (n1, n2) in g.edges:
             plt.plot([n1[1] - emin, n2[1] - emin], [n1[0] - nmin, n2[0] - nmin], 'black')

         # TODO: add code to visualize the path
         path_pairs = zip(path[:-1], path[1:])
         for (n1, n2) in path_pairs:
             plt.plot([n1[1] - emin, n2[1] - emin], [n1[0] - nmin, n2[0] - nmin], 'green')


         plt.xlabel('NORTH')
         plt.ylabel('EAST')

         plt.show()
```



In [ ]:

In [ ]: