Task 6. Algorithms on graphs. Path search algorithms on weighted graphs.

Egor Turukhanov

Group C4134

04.12.19

I.

Graph Generation

Firstly, we generated weighted indirect graph with 100 vertices and 200 edges. I have used 100x200 weighed graph because of memory error for Bellman-Ford algorithm. I will show it in **Conclusion** section.

You can see code below:

```
graph = nx.gnm_random_graph(100,200, directed=False)
```

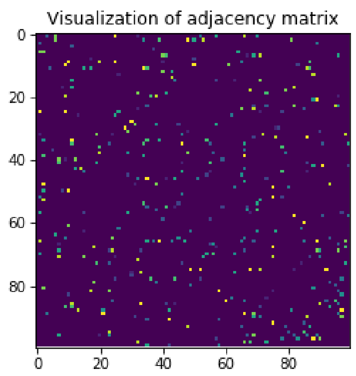
Generating weights for graph:

```
for (u, v) in graph.edges():
    graph.edges[u,v]['weight'] = random.randint(0,10)
```

After that we generated an adjacency matrix for our graph.

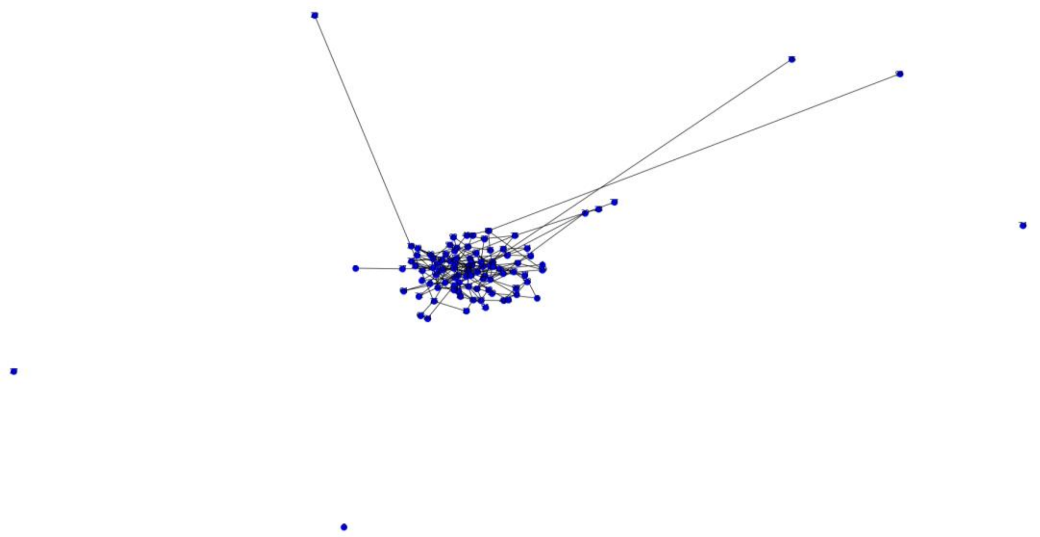
```
adj_matr = nx.to_numpy_matrix(graph)
print(adj_matr)
```

Below you can see plot of generated adjacency matrix.



Graph plotting.

Below you can see visualization of our simple undirected graph with 100 vertices(nodes) and 200 edges.



Now we implemented Dijkstra algorithm, repeat it 10 times and calculate average time:

```
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for i in range(0,10,1):
    t = timeit.default_timer()
    nx.single_source_dijkstra(graph, 1)
    elapsed_time = timeit.default_timer() - t
    total_time.append(elapsed_time)
vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

Average time of running for Dijkstra algorithm is 0.000299 seconds.

```
In [28]: vector_mean.Time
Out[28]:
0    0.000299
Name: Time, dtype: float64
```

Next, we have implemented Bellman-Ford algorithm also 10 times and measured average time of execution:

```
### Bellman Ford
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for i in range(0,10,1):
    t = timeit.default_timer()
    nx.single_source_bellman_ford(graph, 1)
    elapsed_time = timeit.default_timer() - t
    total_time.append(elapsed_time)
vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

Average time of Bellman-Ford algorithm is 0.001085 seconds.

```
In [30]: vector_mean.Time
Out[30]:
0    0.001085
Name: Time, dtype: float64
```

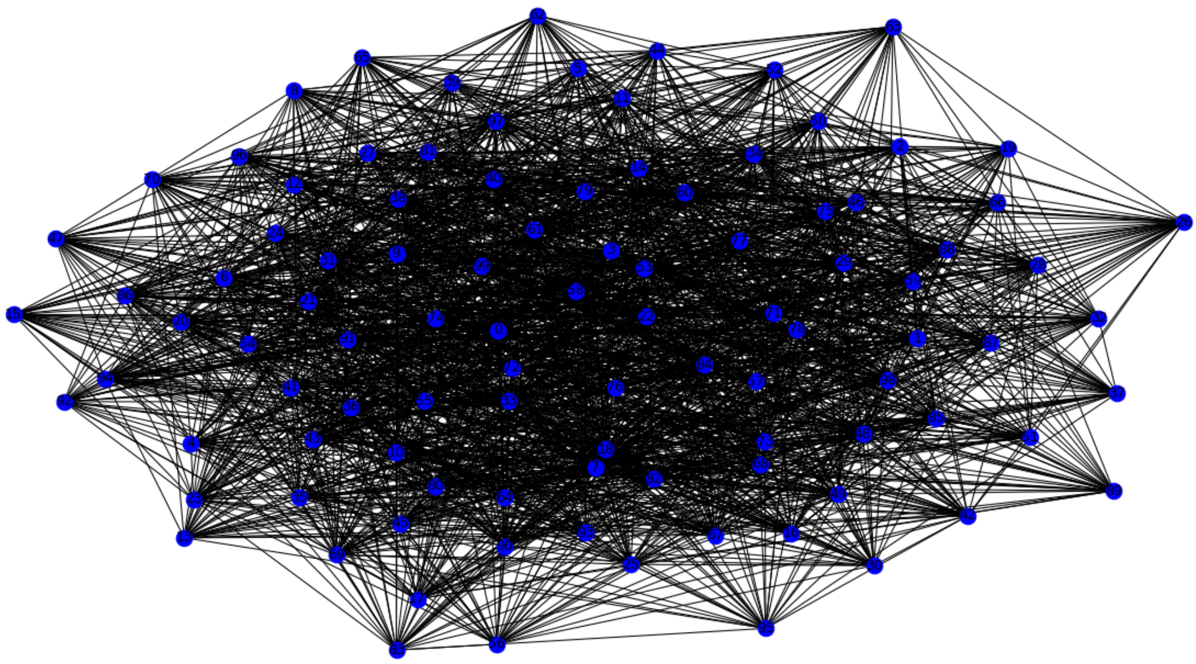
Conclusion

Dijkstra algorithm found shortest path from node 1 to all other nodes faster than Bellman-Ford by 0.00786 seconds. In our case Dijkstra algorithm faster because Bellman-Ford algorithms can deal with negative weights and it checks for check for negative-weight cycles, that is why it require more time to finish. Moreover, in case of 100x2000 graph Bellman-Ford algorithm failed to finish due to RAM error.

For 100x2000 weighted graph Dijkstra algorithm finished with average time of 0.0018 sec.

```
In [40]: vector_mean.Time
Out[40]:
0    0.0018
Name: Time, dtype: float64
```

Visualization of 100x2000 weighed graph:



Bellman-Ford algorithm with 100x2000 weighted graph raise a memory error:

```
Traceback (most recent call last):

  File "<ipython-input-43-9225c21af758>", line 5, in <module>
    nx.single_source_bellman_ford(graph, 1)

  File "C:\Users\fonma\Anaconda3\envs\tensorflow\lib\site-packages\networkx\algorithms
\shortest_paths\weighted.py", line 1603, in single_source_bellman_ford
    dist = _bellman_ford(G, [source], weight, paths=paths, target=target)

  File "C:\Users\fonma\Anaconda3\envs\tensorflow\lib\site-packages\networkx\algorithms
\shortest_paths\weighted.py", line 1321, in _bellman_ford
    path.append(cur)

MemoryError
```

II. A* Algorithm

First, we have generated 10x10 cell grid with 30 obstacle cells.

```
maze = [[0, 1, 1, 1, 0, 0, 0, 0, 0, 0],
        [0, 1, 1, 1, 0, 0, 1, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 0, 1, 1, 1],
        [0, 0, 0, 1, 0, 0, 0, 0, 0, 0],
        [0, 0, 0, 1, 0, 0, 1, 1, 0, 0],
        [0, 0, 0, 1, 0, 0, 1, 0, 0, 1],
        [0, 0, 0, 0, 0, 0, 1, 0, 0, 1],
        [1, 1, 1, 0, 0, 0, 1, 0, 0, 1]]
```

Next, we generated list with non-obstacle cell to random choose start and end points from this list.

```
[15]  non_blocked_cells = []
```

```
[16]  for i in range(0,10):
        for j in range(0,10):
          if maze[i][j] == 0:
            non_blocked_cells.append([i,j])
          else:
            next
```

Then, we have implemented A* algorithm to 10 random start and end point.

```
total_time = []
vector_mean = pd.DataFrame(columns=['Time'])
for i in range(0,10,1):
    t = timeit.default_timer()
    start_block = non_blocked_cells[random.randint(0,70)]
    start = (start_block[0],start_block[1])
    end_block = non_blocked_cells[random.randint(0,70)]
    end = (end_block[0],end_block[1])
    print("i is : ",i, "Start is ", start, "End is ", end)
    path = astar(maze, start, end)
    elapsed_time = timeit.default_timer() - t
    total_time.append(elapsed_time)
vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

A* algorithms had successfully found 10 random paths.

```
[36] total_time = []
     vector_mean = pd.DataFrame(columns=['Time'])
     for i in range(0,10,1):
         t = timeit.default_timer()
         start_block = non_blocked_cells[random.randint(0,70)]
         start = (start_block[0],start_block[1])
         end_block = non_blocked_cells[random.randint(0,70)]
         end = (end_block[0],end_block[1])
         print("i is : ",i, "Start is ", start, "End is ", end)
         path = astar(maze, start, end)
         elapsed_time = timeit.default_timer() - t
         total_time.append(elapsed_time)
     vector_mean = vector_mean.append({'Time': np.mean(total_time)}, ignore_index=True)
```

```
i is :  0 Start is  (7, 5) End is  (6, 9)
i is :  1 Start is  (8, 3) End is  (5, 8)
i is :  2 Start is  (2, 4) End is  (0, 8)
i is :  3 Start is  (5, 7) End is  (2, 5)
i is :  4 Start is  (7, 1) End is  (9, 4)
i is :  5 Start is  (6, 8) End is  (5, 9)
i is :  6 Start is  (6, 9) End is  (7, 8)
i is :  7 Start is  (2, 9) End is  (1, 5)
i is :  8 Start is  (0, 5) End is  (8, 7)
i is :  9 Start is  (8, 8) End is  (0, 5)
```

Average time of path computation is 0.001033 sec.

```
vector_mean.Time
```

```
0    0.001033
Name: Time, dtype: float64
```

This time is quite small. So, A* algorithm was originally created for path searching in graphs and grid spaces. Realization of A* algorithm is also can affect time of its execution.