Backtracking Algorithms

June 15, 2020

1 Backtracking Algorithms

[468]:
```
import pandas as pd
import numpy as np
from random import randint,sample, randrange,shuffle
import itertools
import time
import matplotlib.pyplot as plt
import sys
sys.setrecursionlimit(10**9)
```

1.0.1 Function returning number of edges

[141]:
```
def number_of_edges(n,saturation):
    return int(n*(n-1)*saturation/2)
```

1.0.2 Generating random connected graph

[259]:
```
def generate_connected_graph(n,saturation):
    matrix = np.zeros((n,n), dtype=int)
    n_edges = number_of_edges(n,saturation)
    not_connected =sample(list(range(n)),n)
    connected = [not_connected.pop()]
    for i in range(n-1):
        currVertex = not_connected.pop()
        adjVertex = random.choice(connected)
        matrix[currVertex][adjVertex]=1
        matrix[adjVertex][currVertex]=1
        connected.append(currVertex)
    for i in range(n_edges+1-n):
        x,y = randrange(n), randrange(n)
        while x==y or matrix[x][y]:
            x,y = randrange(n), randrange(n)
        matrix[x][y],matrix[y][x] = 1,1
```

```
    return matrix
x = generate_connected_graph(10,0.6)
print(x)
```

```
[[0 1 1 1 1 1 1 1 1 1]
 [1 0 0 1 0 1 1 0 0 1]
 [1 0 0 1 1 0 0 1 0 1]
 [1 1 1 0 1 0 0 0 0 1]
 [1 0 1 1 0 0 0 1 1 1]
 [1 1 0 0 0 0 1 0 1 1]
 [1 1 0 0 0 1 0 0 0 1]
 [1 0 1 0 1 0 0 0 0 1]
 [1 0 0 0 1 1 0 0 0 0]
 [1 1 1 1 1 1 1 1 0 0]]
```

1.0.3 Checking whether graph is connected or not

[260]:
```
def _is_connected(graph,n,ver_id,visited):
    visited[ver_id] = True
    if all(visited):
        return True
    for i in range(n):
        if graph[ver_id][i] and not visited[i]:
            if _is_connected(graph,n,i,visited):
                return True
def is_connected(graph,n):
    visited = [False]*n
    return _is_connected(graph,n,0,visited)
print(is_connected(x,len(x)))
```

```
True
```

1.0.4 Generating graph with eulerian cycle

[443]:
```
def generate_eulerian(n, saturation):
    graph = generate_connected_graph(n, saturation)
    cur_num_edges = number_of_edges(n,saturation)
    even,odd = [],[]
    for i in range(n):
        num_od_edges = sum(graph[i])
        even.append(i) if num_od_edges % 2 == 0 else odd.append(i)
    while len(odd):
        for v1, v2 in itertools.combinations(odd,2):
            if graph[v1][v2]:
                graph[v1][v2], graph[v2][v1] = 0,0
```

```
                if is_connected(graph, n):
                    cur_num_edges-=1
                    del odd[odd.index(v1)],odd[odd.index(v2)]
                    break
                graph[v1][v2], graph[v2][v1] = 1,1
                if len(odd) == 2:
                    edge = 0
                    while edge == v1 or edge == v2 or graph[edge][v1] or␣
↪graph[edge][v2]:
                        edge+=1
                    ␣
↪graph[edge][v1],graph[edge][v2],graph[v1][edge],graph[v2][edge] = 1,1,1,1
                    del odd[odd.index(v1)],odd[odd.index(v2)]
                    break
        for v1, v2 in itertools.combinations(odd,2):
                if not graph[v1][v2]:
                    cur_num_edges+=1
                    graph[v1][v2],graph[v2][v1] = 1,1
                    del odd[odd.index(v1)],odd[odd.index(v2)]
                    break
    return graph
x = generate_eulerian(100,0.95)
```

1.0.5 Converting adjacency matrix to adjacency list

[444]:
```
def convert(matrix):
    n = len(matrix)
    adjlist = {}
    for i in range(n):
        incidents = []
        for j in range(n):
            if i !=j and matrix[i][j] == 1:
                incidents.append(j)
        adjlist[i] = incidents
    return adjlist
graph = convert(x)
```

1.0.6 Hierholzer's algorithm for finding eulerian graph

[476]:
```
def Eulerian(graph):
    for i in graph.items():
        if len(i)%2 ==1: return False
    stack,temp = [random.randint(0,len(graph)-1)],[]
    while len(stack) != 0:
```

```
        _eulerian(graph,stack[-1],stack)
        temp.append(stack.pop(len(stack)-1))
    return temp
def _eulerian(graph,k,stack):
#     print(k,graph[k])
    if len(graph[k]):
        n = graph[k][0]
        graph[k].remove(n)
        graph[n].remove(k)
        stack.append(n)
        _eulerian(graph,n,stack)
# print(graph)
# print(Eulerian(graph))
```

1.0.7 Finding hamiltonian cycle in a graph

[419]:
```
def same(lst1,lst2):
    return True if sorted(lst1) == sorted(lst2) else False
```

[420]:
```
def investigate_vertex(graph, v, pos, path):
    if v not in graph[path[pos-1]]:
        return False
    if v in path:
        return False
    return True
```

[475]:
```
def _hamiltonian_cycle(graph, path, pos,start):
    if time.time() - start >= 30:
#         print("Timeout has occured")
        raise TimeoutError
    if same(path,graph.keys()):
        return True if path[0] in graph[path[pos-1]] else False

    for v in range(1,len(graph)):
        if investigate_vertex(graph,v, pos, path):
            path[pos] = v
            if _hamiltonian_cycle(graph,path, pos+1,start):
                return True
            path[pos] = -1

    return False
def Hamiltonian_cycle(graph):
    path = [-1] * len(graph)
    path[0] = 0
    try:
        start = time.time()
```

```
        if not _hamiltonian_cycle(graph,path,1,start):
#             print ("Solution does not exist\n")
            return False
        return path
    except TimeoutError:
        return False
```

[462]:
```
graph = {0:[1,3],1:[0,2],2:[1,3,4],3:[0,2,5],4:[2,5],5:[3,4]}
```

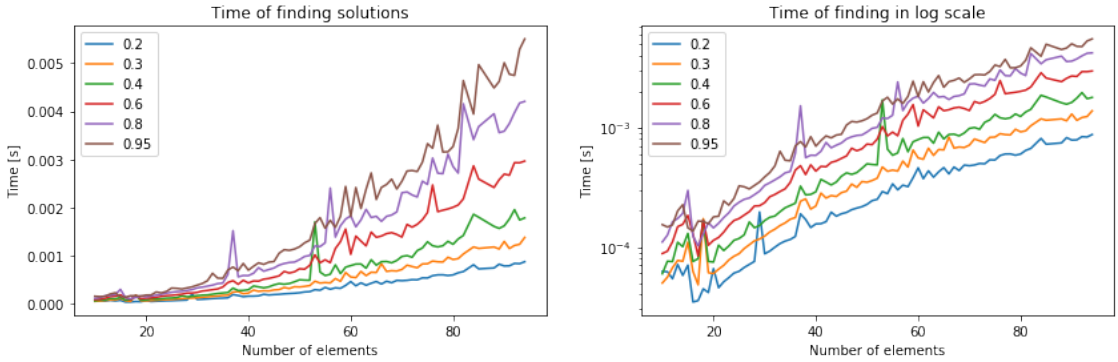
[464]:
```
Hamiltonian_cycle(graph)
```

[464]:
```
[0, 1, 2, 4, 5, 3]
```

[427]:
```
def Evaluate_Eulerian():
    df = []
#     print(df)
    saturations = [0.2,0.3,0.4,0.6,0.8,0.95]
    for v in range(10,100):
        lst = []
        for saturation in saturations:
            graph = convert(generate_eulerian(v, saturation))
            time1 = time.time()
            Eulerian(graph)
            time2 = time.time() - time1
            lst.append(time2)
        df.append(lst)
    return df
```

[474]:
```
x = Evaluate_Eulerian()
df = pd.DataFrame(x, columns = ["0.2","0.3","0.4","0.6","0.8","0.95"],index =␣
↪[i for i in range(10,100)])
```

[470]:
```
def graphs(df):
    fig, (ax1,ax2) = plt.subplots(1,2, figsize=(14,4))
    ax1.set_xlabel("Number of elements")
    ax1.set_ylabel("Time [s]")
    ax2.set_xlabel("Number of elements")
    ax2.set_ylabel("Time [s]")
    df.plot(kind = "line",ax=ax1,title = "Time of finding solutions")
    df.plot(logy=True,ax=ax2,title = "Time of finding in log scale")
    plt.show()
graphs(df)
```



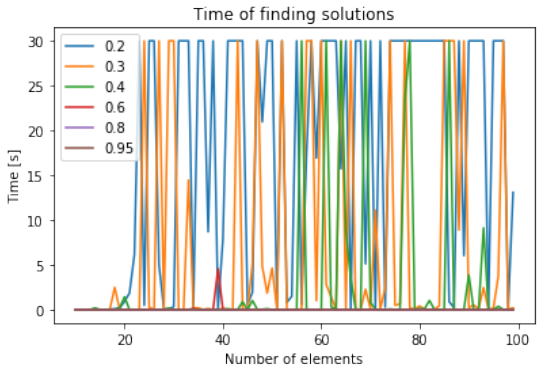
[472]:
```
def Evaluate_Hamiltonian():
    df = []
    saturations = [0.2,0.3,0.4,0.6,0.8,0.95]
    for v in range(10,100):
        lst = []
        for idx,saturation in enumerate(saturations):
            graph = convert(generate_eulerian(v, saturation))
            time1 = time.time()
            Hamiltonian_cycle(graph)
            time2 = time.time() - time1
            lst.append(time2)
        df.append(lst)
    return df
```

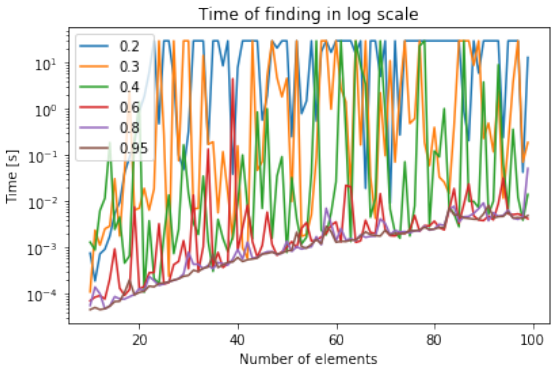
[473]:
```
x = Evaluate_Hamiltonian()
df = pd.DataFrame(x, columns = ["0.2","0.3","0.4","0.6","0.8","0.95"],index =␣
 ↪[i for i in range(10,100)])
graphs(df)
```

```
Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist
```

```
Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Solution does not exist

Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
```

```
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
Timeout has occured
```





1.0.8 Conclusions

As a graph representation for searching algorithm I chose adjacency list because it allows to access directly all the adjacent vertices to the one we are interested in. Adjacency list, as showed it one of the previous task has one of the best balance between time complexity O(|V|) (best complexity has adjacenc matrix with O(1)) and space complexity O(|V|+|E|) (where adjacency matrix has O(|V|^2)).

Graph representing time of finding eulerian cycle in the graph shows the strong dependance between time and the saturation. Complexity is linear O(|E|) because saturation ~ number of edges. Finding the eulerian cycle is not in NP class.

Although graphs showing the relation between search time of hamiltonian cycle and number of elements at first glance may seem chaotic it can be observed that the bigger the saturation the more stable the function gets. We can also draw a conclusion that search time is usually greater for graphs with lower saturation. Moreover, one can state that the bigger graph saturation the easier it is for an algorithm to find a cycle. Those conclusion are no suprise as finding hamiltonian cycle belongs to the NP-Complete class.

Created by: Maciej Filanowicz, AI, 145462