Graph's representation

May 11, 2020

Academic Year: 2019/2020
Sztuczna Inteligencja - Wydział Informatyki - Semester 2
Jan Gruszczyński 145464
Subject: Algorithms and Data Structures

Part 1

Evaluation of different graph representations in terms of peformance w hen searching for an existance of an edge betw een tw o different vertices.

Imports:

In [1]:
```
import numpy as np
from random import randrange as rand_int
import os
import time
import multiprocessing as mp
import matplotlib.pyplot as plt
import sys
```

Graph generator:

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


def generate_graph(n):
    array = np.zeros((n, n), dtype=np.int8)
    number_of_edges_in_graph = number_of_edges(n=n, saturation=0.6)
    for i in range(0, n):
        array[i][i] = 1

    for i in range(number_of_edges_in_graph):
        x = rand_int(n)
        y = rand_int(n)
        while x == y or array[x][y]:
            x = rand_int(n)
            y = rand_int(n)
        array[x][y] = 1
        array[y][x] = 1
    return array


def save_graph_to_file(n):
    array = generate_graph(n)
    os.makedirs("data", exist_ok=True)
    np.savetxt("data/" + str(n) + ".txt", array, fmt="%d")
```

Data types:

Adjacency matrix:

In [3]:
```
def adjacency_matrix(n, generated_graph):
    return generated_graph


def find_edge_in_adjacency_matrix(generated_graph, x, y):
    return generated_graph[x][y]
```

Incidence matrix:

In [4]:
```
def incidence_matrix(n, array):
    number_of_edges_in_graph = number_of_edges(n=n, saturation=0.6)
    new_array = np.zeros((n, number_of_edges_in_graph), dtype=np.int8)
    g = 0
    for x in range(0, n):
        for y in range(x + 1, n):
            if array[x][y]:
                new_array[x][g] = 1
                new_array[y][g] = 1
                g = g + 1
    return new_array
def find_edge_in_incidence_matrix(generated_graph, x, y):
    return np.dot(generated_graph[x], generated_graph[y])  # returns 1 if
one edge connects two vertices, 0 if not
```

Edge list:

In [5]:
```
def edge_list(n, array):
    number_of_edges_in_graph = number_of_edges(n=n, saturation=0.6)
    new_array = np.zeros((number_of_edges_in_graph, 2))
    g = 0
    for a in range(0, n):
        for b in range(a + 1, n):
            if array[a][b]:
                new_array[g][0] = a
                new_array[g][1] = b
                g = g + 1
    return new_array


def find_edge_in_edge_list(array, x, y):
    x_length = array.shape[0]
    for a in range(0, x_length):
        if (array[a][0] == x and array[a][1] == y) or (array[a][0] == y an
d array[a][1] == x):
            return True
    return False
```

List of incidents:

In [6]:
```
def list_of_incidents(n, array):
    incidents_list = []
    for a in range(0, n):
        incidents = []
        for b in range(n):
            if a != b and array[a][b] == 1:
                incidents.append(b)
        incidents_list.append(incidents)
    return incidents_list


def find_edge_in_list_of_incidents(array, x, y):
    return y in array[x]
```

Time measuring function:

In [7]:
```
def measure_time(n, graph_representation, graph_representation_search_func
tion):
    array = np.loadtxt("data/" + str(n) + ".txt", dtype=np.int8)
    graph = graph_representation(n, array)

    test_number = 100
    time_elapsed_array = []
    for i in range(0, test_number):
        x = rand_int(n)
        y = rand_int(n)
        if y == x:
            while y == x:
                y = rand_int(n)

        start = time.time()
        graph_representation_search_function(graph, x, y)
        end = time.time()
        time_elapsed = end - start
        time_elapsed_array.append(time_elapsed)
    return sum(time_elapsed_array) / test_number
```

In [8]:
```
def perform_one_test(X, graph_representation, graph_representation_search_
function):
    return [measure_time(n, graph_representation=graph_representation,
                         graph_representation_search_function=graph_repres
entation_search_function) for n in X]
```

Plotting function:

In [9]:
```
def plot_plot(X, bunch_of_Ys, labels, title):
    for a in range(len(bunch_of_Ys)):
        plt.plot(X, bunch_of_Ys[a], label=labels[a])
    plt.legend()
    plt.xlabel('Number of Elements [n]')
    plt.ylabel('Time [s]')
    plt.title(title)
    plt.show()
```

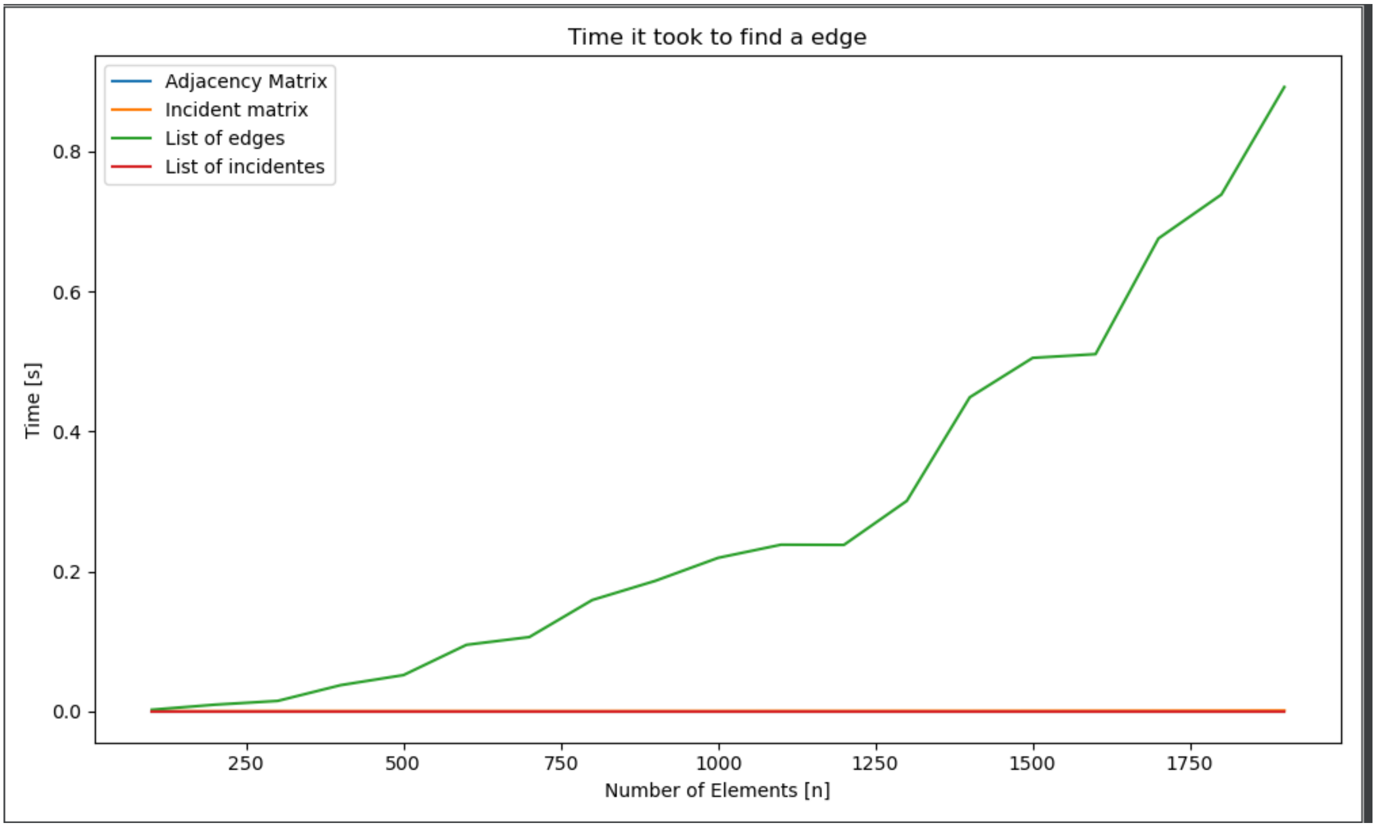
In [10]:
```
def make_tests(X, list_of_graphs_representations, list_of_graphs_represent
ations_search_functions, labels):
    pool = mp.Pool(mp.cpu_count() - 4)
    bunch_of_Ys = pool.starmap(perform_one_test,
                               [(X, graph_representation, graph_representa
tion_search_function) for
                                graph_representation, graph_representation
_search_function in
                                zip(list_of_graphs_representations,
                                    list_of_graphs_representations_search_
functions)])
    pool.terminate()

    plot_plot(X, bunch_of_Ys, labels=labels, title="Time it took to find a
edge")
    return bunch_of_Ys
```

Generating files:

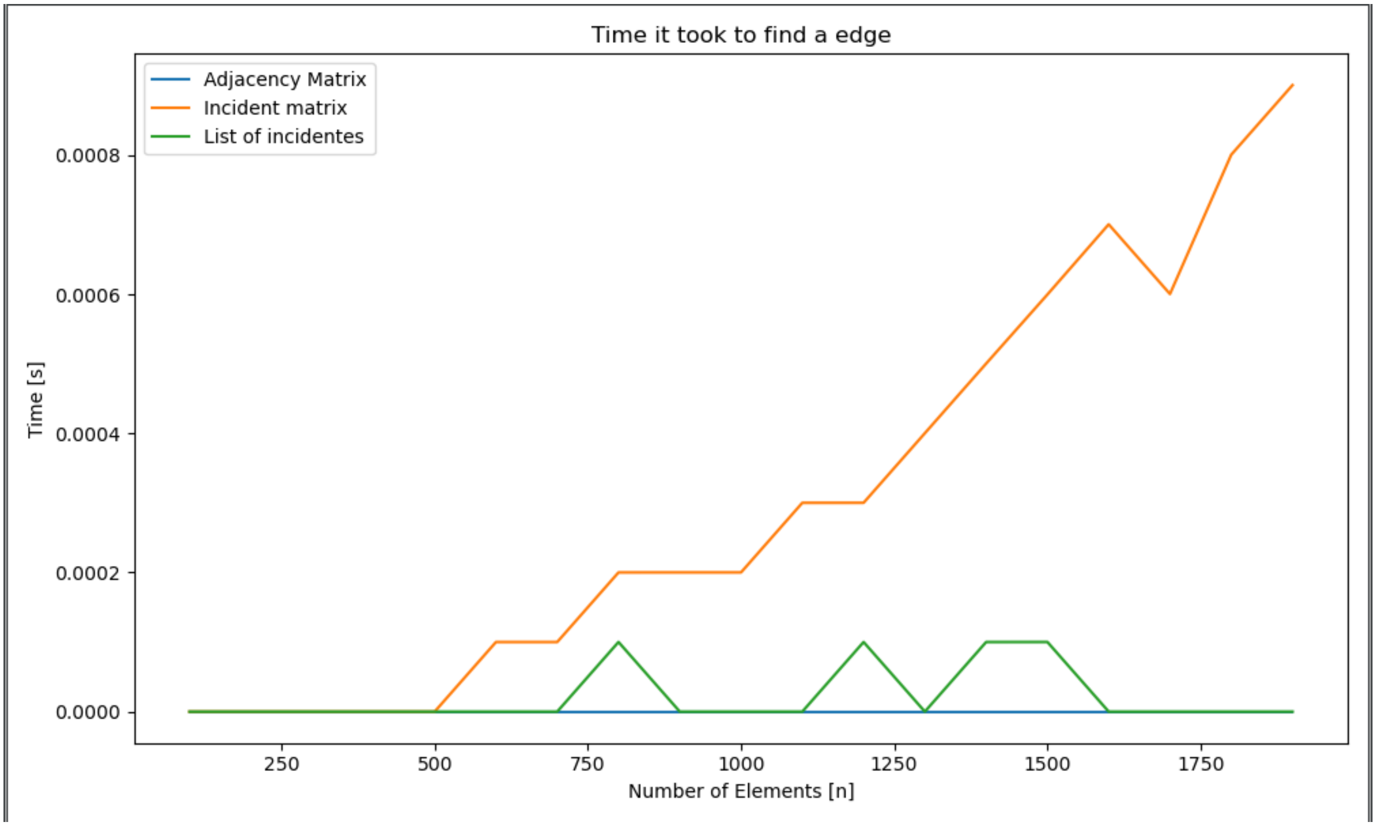
In [11]:
```
X = [n for n in range(100, 2000, 100)]
# for n in X:
#     save_graph_to_file(n)
```

Generating plots:

In [ ]:
```
X = [n for n in range(100, 2000, 100)]
list_of_graphs_representations = [adjacency_matrix, incidence_matrix, edge_list, list_of_incidents]
list_of_graphs_representations_search_functions = [find_edge_in_adjacency_matrix, find_edge_in_incidence_matrix,
                                                   find_edge_in_edge_list, find_edge_in_list_of_incidents]
labels = ["Adjacency Matrix", "Incident matrix", "List of edges", "List of incidentes"]
retrn = make_tests(X, list_of_graphs_representations, list_of_graphs_representations_search_functions, labels)
```

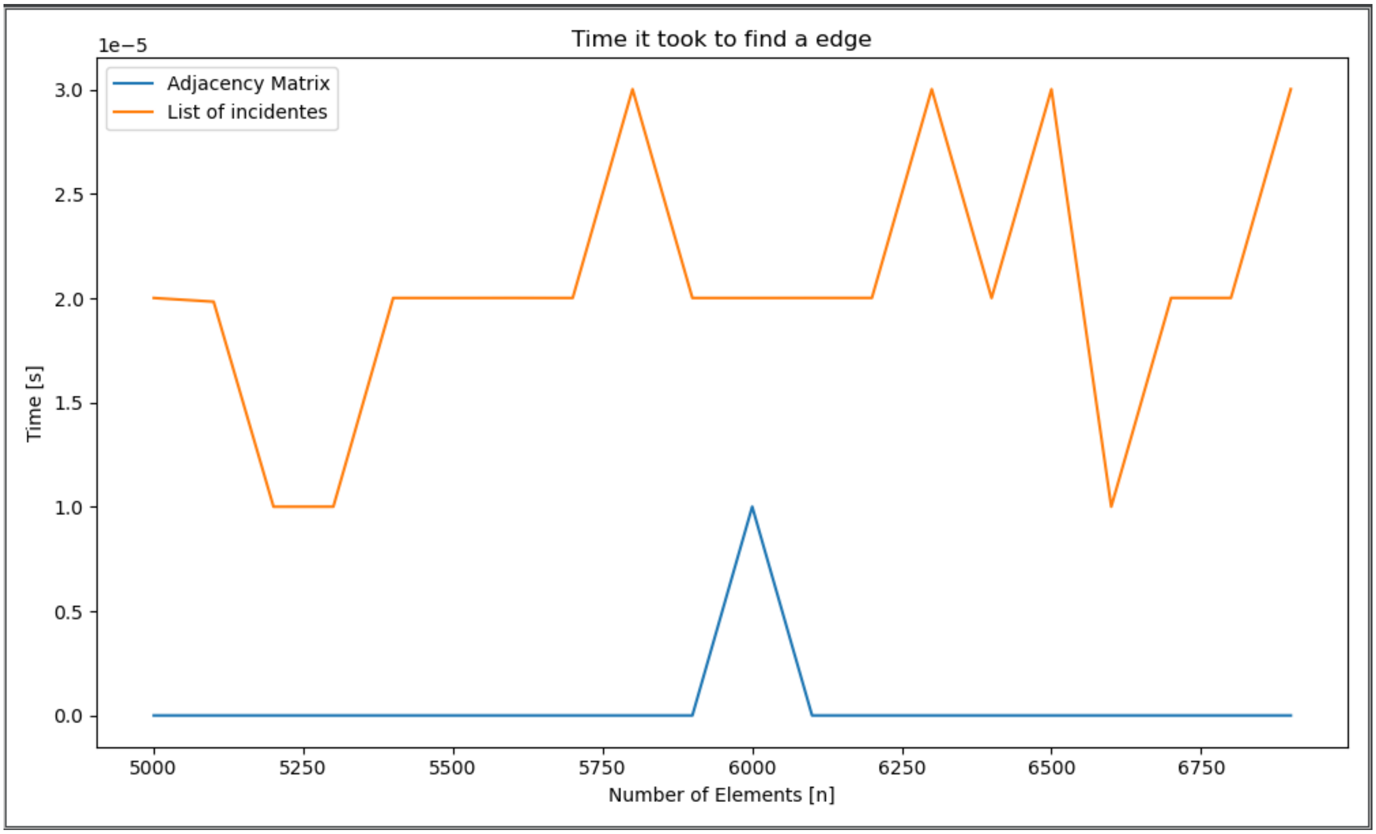


In [25]:
```
X = [n for n in range(100, 2000, 100)]
list_of_graphs_representations = [adjacency_matrix, incidence_matrix, list_of_incidents]
list_of_graphs_representations_search_functions = [find_edge_in_adjacency_matrix, find_edge_in_incidence_matrix,
                                                   find_edge_in_list_of_incidents]
labels = ["Adjacency Matrix", "Incident matrix", "List of incidentes"]
retrn = make_tests(X, list_of_graphs_representations, list_of_graphs_representations_search_functions, labels)
```



In [ ]:

```
X = [n for n in range(5000, 7000, 100)]
list_of_graphs_representations = [adjacency_matrix, list_of_incidents]
list_of_graphs_representations_search_functions = [find_edge_in_adjacency_
matrix,
                                                                       find_edge_in_list_of_in
cidents]
labels = ["Adjacency Matrix", "List of incidentes"]
retrn = make_tests(X, list_of_graphs_representations, list_of_graphs_repre
sentations_search_functions, labels)
```



Conclusion:

Computational difference betw een each grap representation is very sigificant. Edge list is the w orst of the bunch.
On the other hand, Adjacency Matrix appeared to be the best, w ith complexity equal to O(1), because w e access edges directly. Behind it w as list of incidents w ith time complexity of O(|V|) = O(n)
Incidence matrix requires dot product operation, w hich has time complexity of O(n^2), but as it is stored in array, it still is better than edge list stored in a list.

Part 2

Finding the best representation for topoligical order sorting and time performance evaluation.

I've chosen the List of Incident data structure, as the TOS algorithm doesn't need to perform any additional transformations to use this data type.

Generating DAG:

In [14]:
```
def generate_DAG(n):
    array = np.zeros((n, n), dtype=np.int8)
    number_of_edges_in_DAG = number_of_edges(n=n, saturation=0.3)

    for i in range(number_of_edges_in_DAG):
        x = rand_int(n)
        y = rand_int(n)
        while x >= y or array[x][y]:
            x = rand_int(n)
            y = rand_int(n)
        array[x][y] = 1
    return array


def save_DAG(n):
    array = generate_DAG(n=n)
    os.makedirs("data/DAG", exist_ok=True)
    np.savetxt("data/DAG/" + str(n) + ".txt", array, fmt="%d")
```

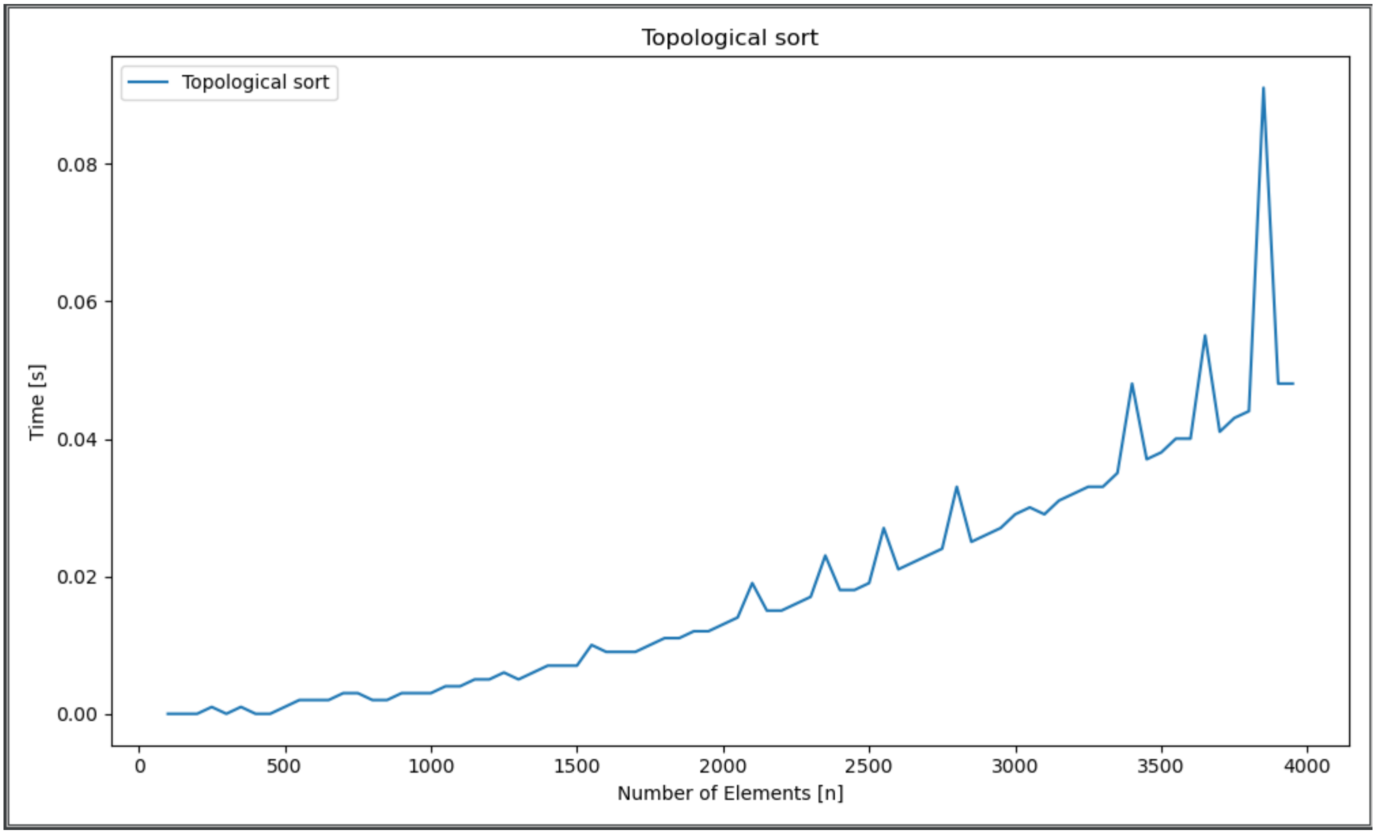
Sorting functions:

In [15]:
```
def sort_topologically_recursive(variable, incident_list, visited_list, st
ack):
    visited_list[variable] = 1

    for a in incident_list[variable]:
        if not visited_list[a]:
            sort_topologically_recursive(a, incident_list, visited_list, s
tack)
    stack.append(variable)


def sort_topologically(n, incident_list):
    visited = [0 for n in range(0, n)]
    stack = list()
    for a in range(0, n):
        if not visited[a]:
            sort_topologically_recursive(a, incident_list, visited, stack)
    return stack.reverse()
```

Measuring functions:

In [20]:
```
def measure_time_2(n):
    array = np.loadtxt("data/DAG/" + str(n) + ".txt", dtype=np.int8)
    incidence_list = list_of_incidents(n, array)
    start = time.time()
    sort_topologically(n, incidence_list)
    end = time.time()
    return end - start


def perform_test2():
    X = [i for i in range(100, 4000, 50)]
    Ys = [[measure_time_2(n) for n in X]]
    plot_plot(X, Ys, ["Topological sort"], "Topological sort")
```

In [22]:
```
for i in range(100, 4000, 50):
    save_DAG(i)
perform_test2()
```



Conclusion:

Theoretical complexity of topological sorting algorithm is O(|E| + |V|).
Where V is the number of vertices and E is the number of edges (defined by our number_of_edges_function).
Representations other than list of incidents w ould requier additional trasformations w hich w ould take more time. (Only if the graph w ould have very small amount of edges compared to the amount of vertices, the edge list w ould be more effective)