



Project Advanced Algorithms

https://github.com/GuidoBorrelli/AA_project

Guido Borrelli - 874451

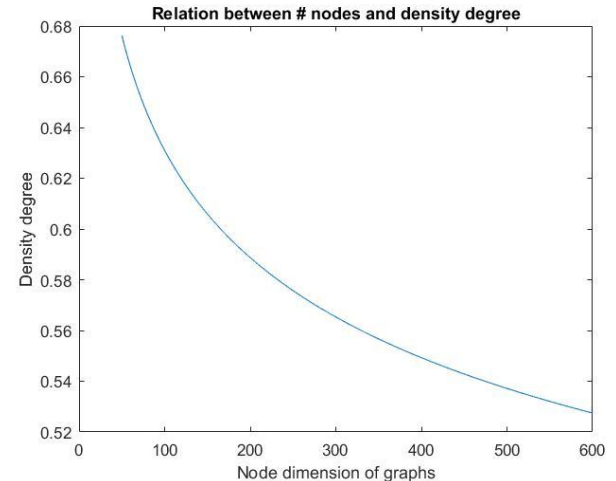


Project specification

1. Implementation of
 - a. Tarjan algorithm
 - b. Nuutila algorithm
 - c. Pearce algorithm
2. Comparison of performance to find strongly connected components within a graph

Implementation choices

- Which kind of graph to study?
 - Dimension
 - Small → 50 nodes
 - Small/Medium → 100
 - Medium/Big → 200
 - Big → 600
 - Density
 - Sparse → 1 edge per node (min nodes: 50)
 - Normal → 3 edge per node (min nodes: 50)
 - Dense → $N^{-1/10}$
(0.52 ÷ 0.67 of density degree ~ N^2)



Note: more nodes & density combination lead to maximum depth of recursion



Implementation choices / 2

- How many graph create to validate performance?

- ~~○ A function based on graph characteristics generate the feasible number~~

- 10000 (or possibly set SIZE_BENCHMARK global constant)

Performance results didn't change → Comparable mean

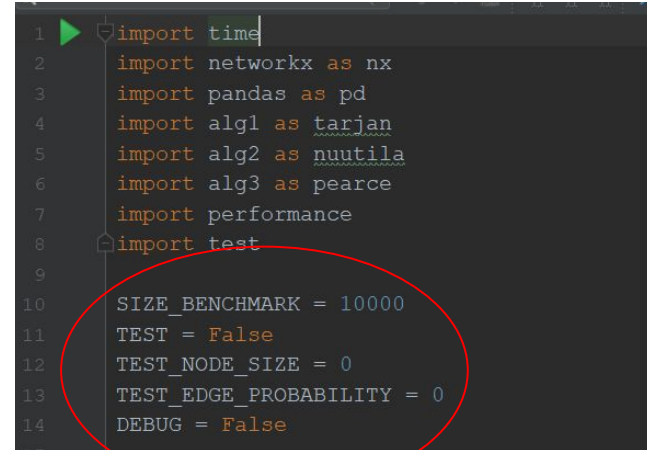
→ Fixed measure to capture also real variance differences among characteristics



Project run

2 ways to start the program

- TEST = False
 - Possibly change SIZE_BENCHMARK
 - Program execute a test set saving in img performance
- TEST = True
 - Program start the “test mode”
 - TEST_NODE_SIZE and TEST_EDGE_PROBABILITY constants not mandatory to be set
 - Graph is generated either randomly or with CONSTANT values
 - Validity of algorithms results is checked



```
1 import time
2 import networkx as nx
3 import pandas as pd
4 import alg1 as tarjan
5 import alg2 as nuutila
6 import alg3 as pearce
7 import performance
8 import test
9
10 SIZE_BENCHMARK = 10000
11 TEST = False
12 TEST_NODE_SIZE = 0
13 TEST_EDGE_PROBABILITY = 0
14 DEBUG = False
```



Implementation - main method

As aforementioned, the main method either:

1. call the function to create a test set:
 - `create_test_set()`
2. and the one to plot performance results:
 - `plot_result(dict_performance)`

or

1. call the function to test of all algorithms
 - a. `test_algorithms(TEST_NODE_SIZE, TEST_EDGE_PROBABILITY)`

```
def main():
    if not TEST:
        print("Start")
        dict_performance = create_test_set()
        if DEBUG:
            print("Tarjan performance: {}".format(dict_performance['Tarjan']))
            print("Nuutila performance: {}".format(dict_performance['Nuutila']))
            print("Pearce performance: {}".format(dict_performance['Pearce']))
        performance.plot_result(dict_performance)
        print("End")
    else:
        test.test_algorithms(TEST_NODE_SIZE, TEST_EDGE_PROBABILITY)
    return 0

if __name__ == "__main__":
    main()
```



Implementation - create_test_set

1. For each combination of both
 - a. Node dimensions
 - b. Node density
2. A benchmark of graphs with that characteristics is created
 - a. `create_benchmark(nodes_dimension, graph_density)`
3. Performance of each combination of characteristics are saved in a dataframe of dict
 - a. Organised per
 - i. Algorithm
 1. Density



Implementation - create_benchmark

1. SIZE_BENCHMARK graphs are created
 - create_direct_graph(nodes_dimension, graph_density)
2. After the creation of each graph, its result for each algorithm are saved

```
def create_benchmark(nodes_dimension, graph_density):
    print("Create benchmark : Nodes {} - Density type {}".format(nodes_dimension, graph_density))
    # List containing time records for each algorithm
    times1_list = []
    times2_list = []
    times3_list = []
    dict_times = {}
    for _ in range(0, SIZE_BENCHMARK):
        # print(i, end="\r")
        times1, times2, times3 = create_direct_graph(nodes_dimension, graph_density)
        times1_list.append(times1)
        times2_list.append(times2)
        times3_list.append(times3)
    dict_times['Pearce'] = times1_list
    dict_times['Nuutila'] = times2_list
    dict_times['Tarjan'] = times3_list
    return dict_times
```




Implementation - create_direct_graph

- A graph with the requested characteristics is created
 - If the graph is sparse, a special function of networkx is used optimized for creation of sparse graphs
- The function handling the execution of each algorithm is called
 - `apply_alg(graph)`
- Times performance are directly passed to caller

```
def create_direct_graph(n, d):  
    # An heuristic to build valid edge probabilities  
    def switch_density(argument):  
        switcher = {  
            0: 1 / n,  
            1: 3 / n,  
            2: 1 / (n ** 0.1),  
        }  
        value = switcher.get(argument, False)  
        if value is False:  
            raise "Invalid Density: " + argument  
        return switcher[argument]  
  
    p = switch_density(d)  
    if d == 0:  
        graph = nx.fast_gnp_random_graph(n, p, seed=None, directed=True)  
    else:  
        graph = nx.gnp_random_graph(n, p, seed=None, directed=True)  
    return apply_alg(graph)
```



Implementation - apply_alg

For each algorithm:

1. Start the time counter
2. Execute the algorithm
3. Stop the time counter

These times are passed to the caller


```
def apply_alg(graph):  
    # Keep track of time of each algorithm executed  
    t0 = time.time()  
    pearce.apply_alg(graph)  
    duration0 = time.time() - t0  
    t1 = time.time()  
    nuutila.apply_alg(graph)  
    duration1 = time.time() - t1  
    t2 = time.time()  
    tarjan.apply_alg(graph)  
    duration2 = time.time() - t2  
    return duration0, duration1, duration2
```



Implementation - main

Now the main method will call the plot function

- `plot_result(dict_performance)`



```
def main():
    if not TEST:
        print("Start")
        dict_performance = create_test_set()
        if DEBUG:
            print("Tarjan performance: {}".format(dict_performance['Tarjan']))
            print("Nuuttila performance: {}".format(dict_performance['Nuuttila']))
            print("Pearce performance: {}".format(dict_performance['Pearce']))
        performance.plot_result(dict_performance)
        print("End")
    else:
        test.test_algorithms(TEST_NODE_SIZE, TEST_EDGE_PROBABILITY)
    return 0
```



Implementation - plot_result

1. Get statistics values for each category
 - `get_values(performance_dict, "category")`
2. Plot each category
 - `plot_graph(x_sp, y_sp, e_sp, "Sparse")`

```
def plot_result(performance_dict):  
    # This function let get data in a format useful to plot  
    x_sp, y_sp, e_sp = get_values(performance_dict, 'Sparse')  
    x_md, y_md, e_md = get_values(performance_dict, 'Medium')  
    x_de, y_de, e_de = get_values(performance_dict, 'Dense')  
    # This function plot the data for each category  
    plot_graph(x_sp, y_sp, e_sp, 'Sparse')  
    plot_graph(x_md, y_md, e_md, 'Medium')  
    plot_graph(x_de, y_de, e_de, 'Dense')  
    return
```



Implementation - get_values

- Extraction from the dictionary of performance
 - Times per each algorithm for the provided category
- Extraction of statistics value
 - Per each algorithm
 - Mean
 - ~~Standard deviation~~
 - Variance



Original idea was to build an errorbar as plot graph BUT..

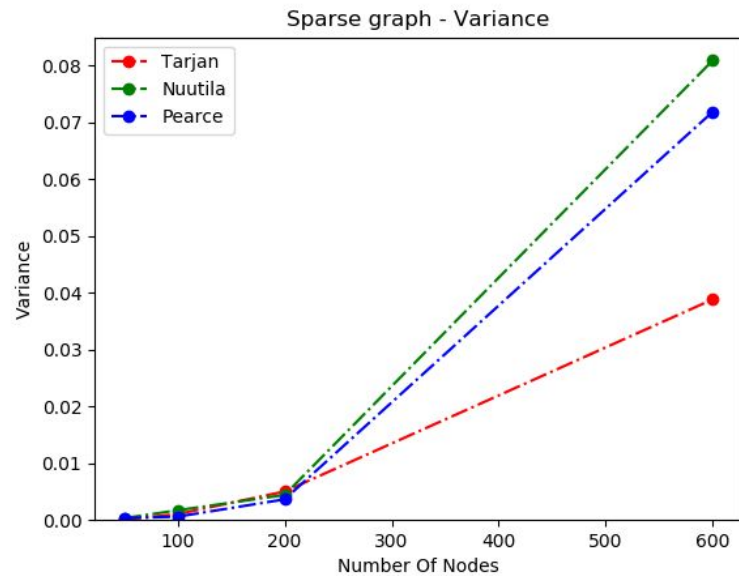
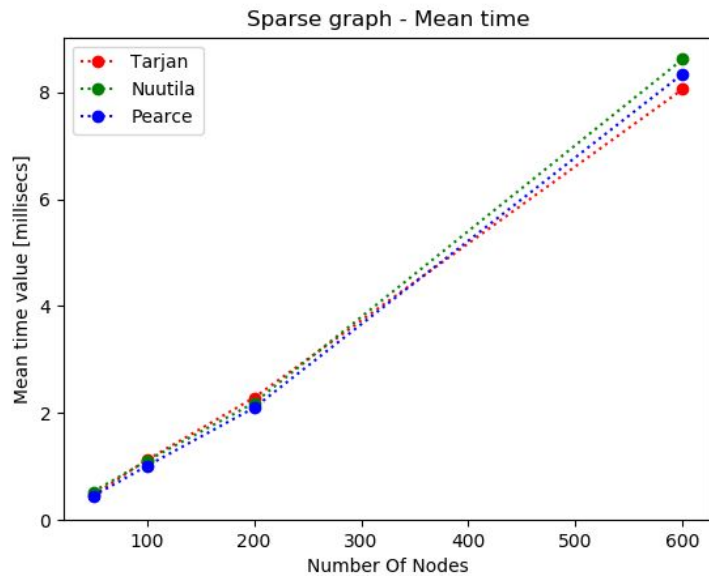


Implementation - plot_graph

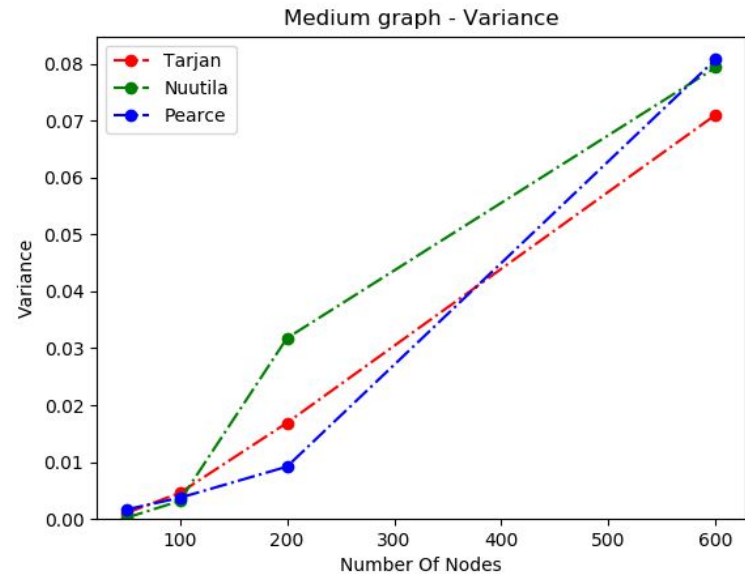
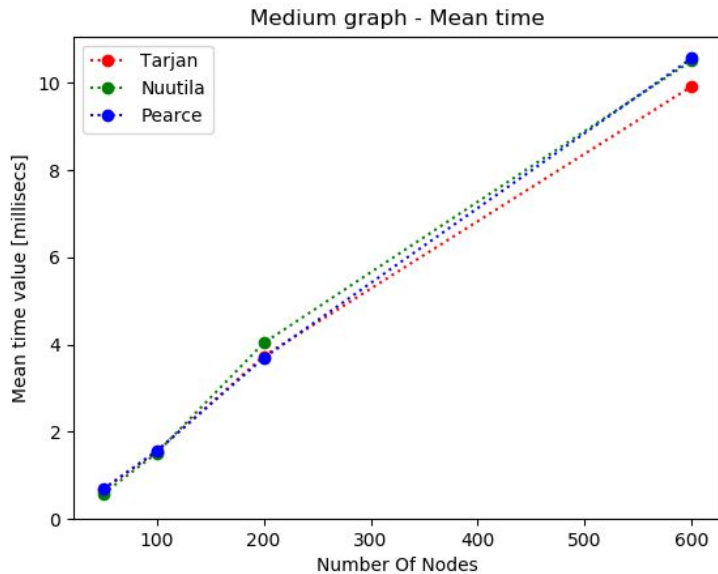
Per each category: Sparse, Medium, Dense

- Mean plot
 - For each algorithm
 - Mean values per node dimension
- Variance plot
 - For each algorithm
 - Variance values per node dimension

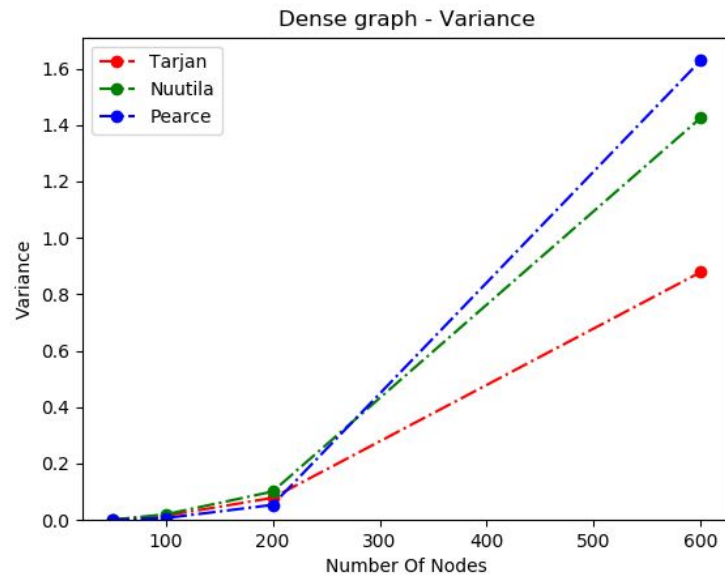
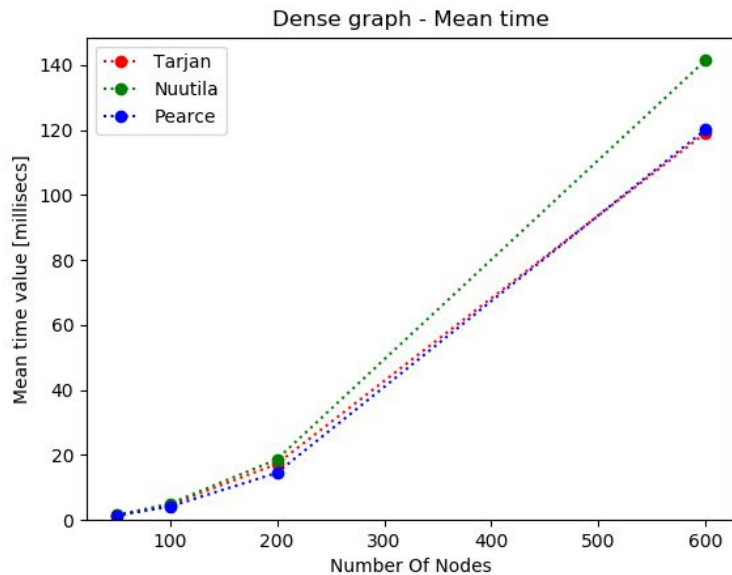
Results: Sparse



Results: Normal (Medium)



Results: Dense





Comments

Regardless of the category, the Tarjan algorithm performs substantially better on big node dimension (600 nodes graph) by both perspective of mean and variance.

The Pearce algorithm performs slightly better till medium/big node dimension (up to 200) in particular by the perspective of the variance. This means that the high variability of graph that can be generated doesn't affect its performance the same way of the other algorithms.



Algorithms implemented

- Tarjan algorithm (so called in paper)
- Nuutila algorithm 2 (so called in paper)
- PEA_FIND_SSC2 (so called in paper)




Implementation - test

1. Call the function to test of all algorithms
 - a. test_algorithms(TEST_NODE_SIZE,
 - b. TEST_EDGE_PROBABILITY)

```
def main():
    if not TEST:
        print("Start")
        dict_performance = create_test_set()
        if DEBUG:
            print("Tarjan performance: {}".format(dict_performance['Tarjan']))
            print("Nuutila performance: {}".format(dict_performance['Nuutila']))
            print("Pearce performance: {}".format(dict_performance['Pearce']))
        performance.plot_result(dict_performance)
        print("End")
    else:
        test.test_algorithms(TEST_NODE_SIZE, TEST_EDGE_PROBABILITY)
    return 0

if __name__ == "__main__":
    main()
```





Implementation - test_algorithms

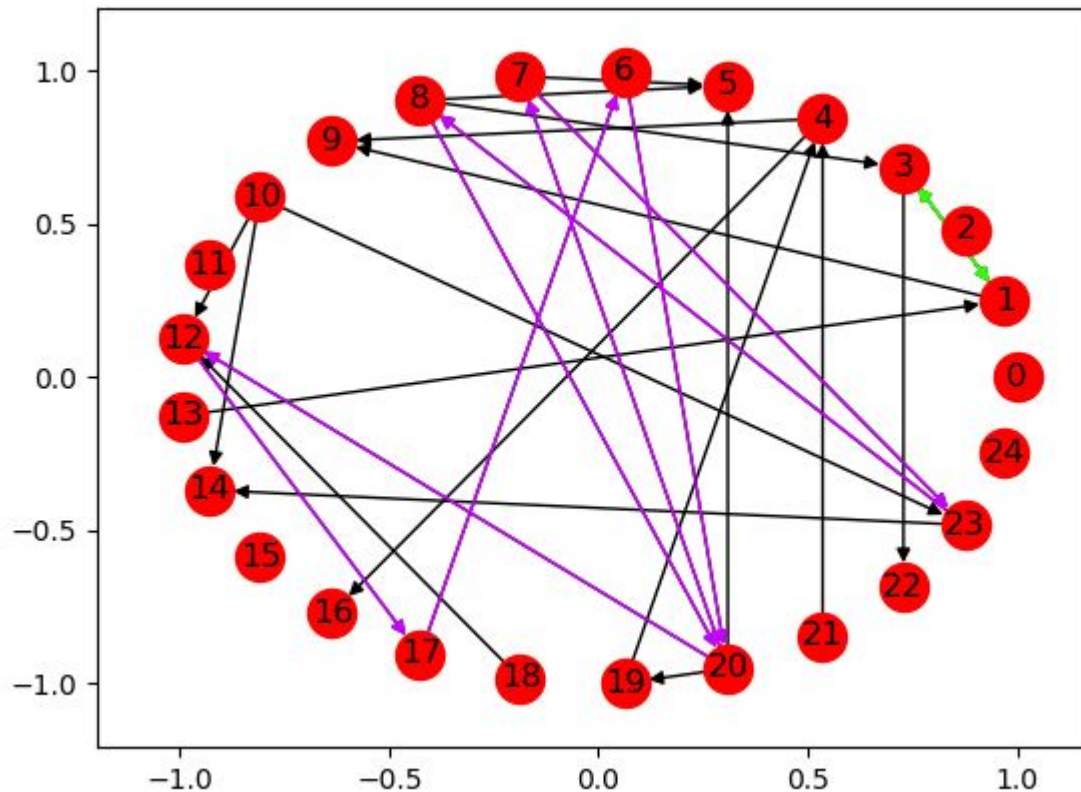
1. Generate a graph
2. Apply all algorithms on that graph
 - a. Save SSCs references of all nodes for each algorithm
3. For each combination of nodes
 - a. If a path between them exist in both direction
 - i. They belong to the same SSC
 - b. Check if they belong to the same SSC reference in all algorithms result
4. Plot graph with coloured SSCs
 - a. If graph is under 40 nodes for the sake of legibility and my computer



Two SCCs:

1. Green one
2. Violet one

Black edges are just edges
among single SCC nodes.





Missing part of project

- + Memory profiling of algorithms performance



Profile memory usage - memory.py

- Through MEMORY_TEST global variable from main.py
- Hard-coded experiments in memory.py
 - GRAPH = True
 - Generate a personalised graph and save it in a file
 - GRAPH = False
 - Implies test of an algorithm according to ALG_TEST
 - ALG_TEST = 1 → Tarjan
 - ALG_TEST = 2 → Nuutila
 - ALG_TEST = 3 → Pearce
 - Results manually inserted in an Excel table
 - Charts generated through Excel



Profile memory usage - memory.py /2

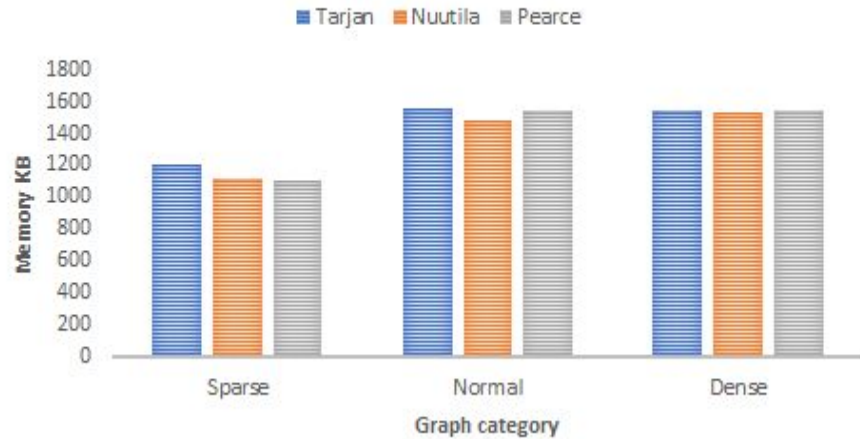
Hands on

- A tara is fixed AFTER graph loading
- Function is called
- Memory is checked
- Results are collected and reasoned

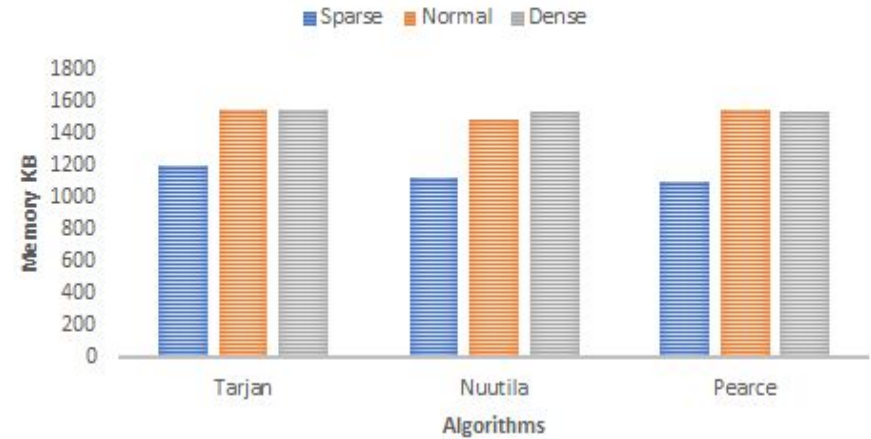
Due to small variations of memory usage, only so called “big graph” have been considered (600 nodes). Smaller graph didn’t improve memory usage than 0.05% and very few differences among them.

Profile memory usage - Charts

PROFILE MEMORY USAGE



PROFILE MEMORY USAGE





Comments

1. Graphs and evaluation have been iteratively tried
 - No variation of % increment of algorithms
2. I would have expected a slightly better memory usage by the Pearce algorithm
 - Slightly better for *sparse* graphs
 - Quite comparable for *normal* and *dense* graphs
3. Possibly Python handle better data structures used in Nuutila way