Report 3 - Graph representations and topological search

Karol Cyganik, index number: 148250

Sebastian Chwilczyński, index number: 148248

Theoretical analysis of different graph representations

In this experiment, we compared times needed to add all elements to one graph, and average times needed to search for one element in the graph in four different representations:

- Edge list
- Neighborhood matrix
- Incidence matrix
- List of incidence (Adjacency list)

The edge list is simply a list, where all edges are in form of pairs *{node1, node2}*. It simply has *O(e)* memory consumption, where *e* is a number of edges. *'e'* in the worst case can be equal to $\frac{n*(n-1)}{2}$ nodes in undirected and $n * (n-1)$ nodes in an directed graph. When it comes to searching an existence of one node, the time complexity is similarly *O(e)*. Adding one edge to this structure has constant time *O(1).*

Neighborhood matrix is a $n_x n$ matrix, it has a number of columns equal to the number of rows, both are equal to *n*, the number of nodes in a graph. The whole matrix is filled with ones and zeros. Each field stands for the existence of an edge between two nodes – that one from the column, and row. Zero means there is no such edge where one means the edge exists. It requires *O(n*n)* memory, as it has *n* rows and *n* columns. Searching in this structure is very fast, it happens in *O(1)* time. Same with adding one element, it also requires *O(1)* time.

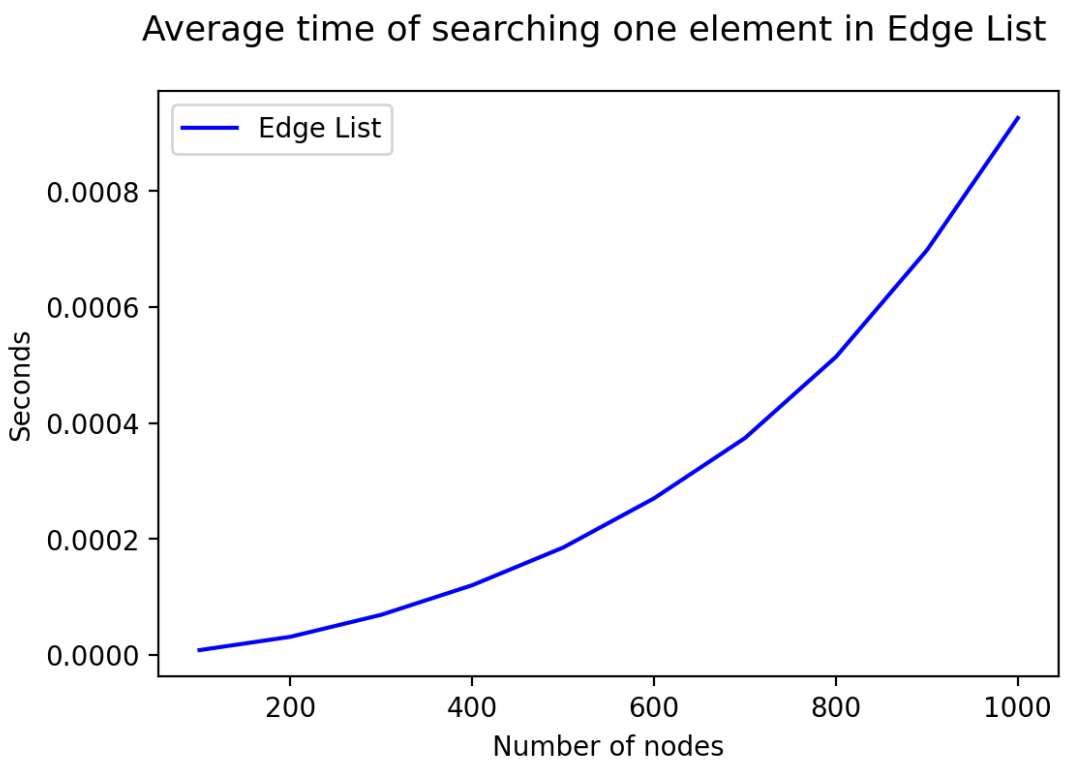
An incidence matrix is a matrix where a number of rows is equal to a number of nodes and number of columns equals a number of edges. This implies *O(n*e)* memory complexity. This matrix is also filled with zeros and ones. Each column represents an edge, 1 in particular cell means this node from the row is one of the vertices creating this edge. So in each column we have only two one's and rest are zeroes so this matrix is rather sparse. Searching for the existence of one edge here is in *O(e)* time, as we need to iterate through all edges and there in *O(1)* checking, is this particular edge that one what we are looking for. Adding theoretically needs *O(1)* time.

List of incidence, also known as **Adjacency List**. It's a list of nodes, where each node has its own list with the nodes, with which it is connected. In the worst case, it may use $O(n^2)$ space to store all connections between nodes. Adding an edge here can be implemented in fast, O(1) time.

Experiment 1 - measuring time needed to check whether a undirected graph contains given edge

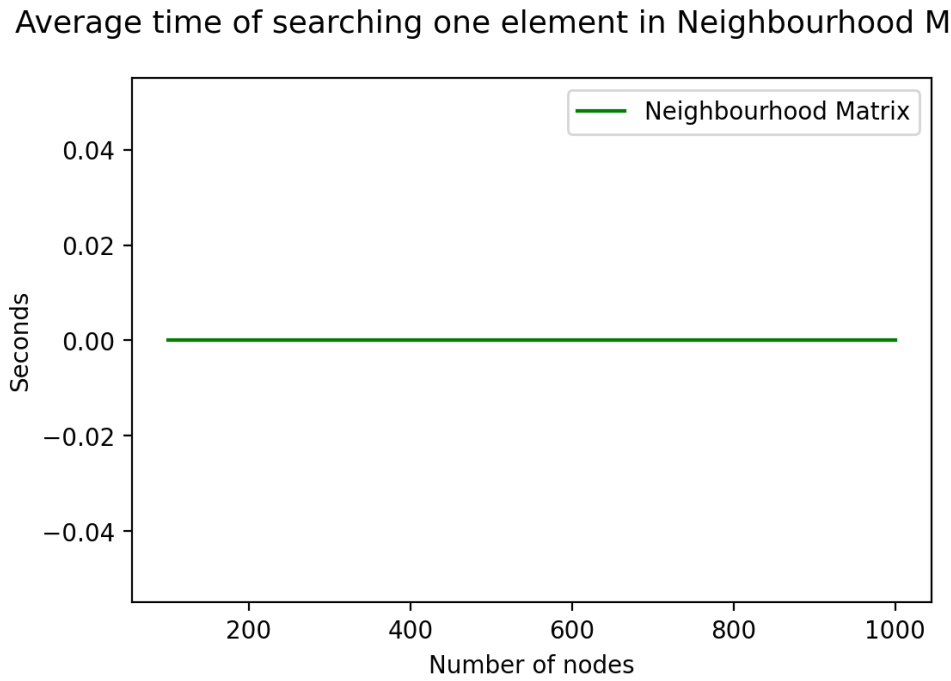
For this experiment, we chosen 10 measuring points using function $n = 100 * x$ where n is the number of vertices in the graph. One could say that this is not enough to observe the behaviour of complexity function, but one has to take into account that for 100 nodes we can possibly create $\frac{n*(n-1)}{2}$ nodes for undirected graph and $n * (n-1)$ for directed graph, so for n = 1000 we can create almost 500 thousand nodes in case of undirected and 1 million in case of a directed graph. Even taking only 60% of them still produces a big number of edges to process. To store graphs in text files we used edge list representation and edge saturation was always 60%, then we created a text file with edges to be found where we put 80% of all possible edges as we wanted to arrange a more life-like situation where not always edge is present in the graph. Next, we searched for them one by one and measure the time needed to look for all, after that we divided that time by the number of all edges to obtain the time needed to find 1 edge.

Edge list



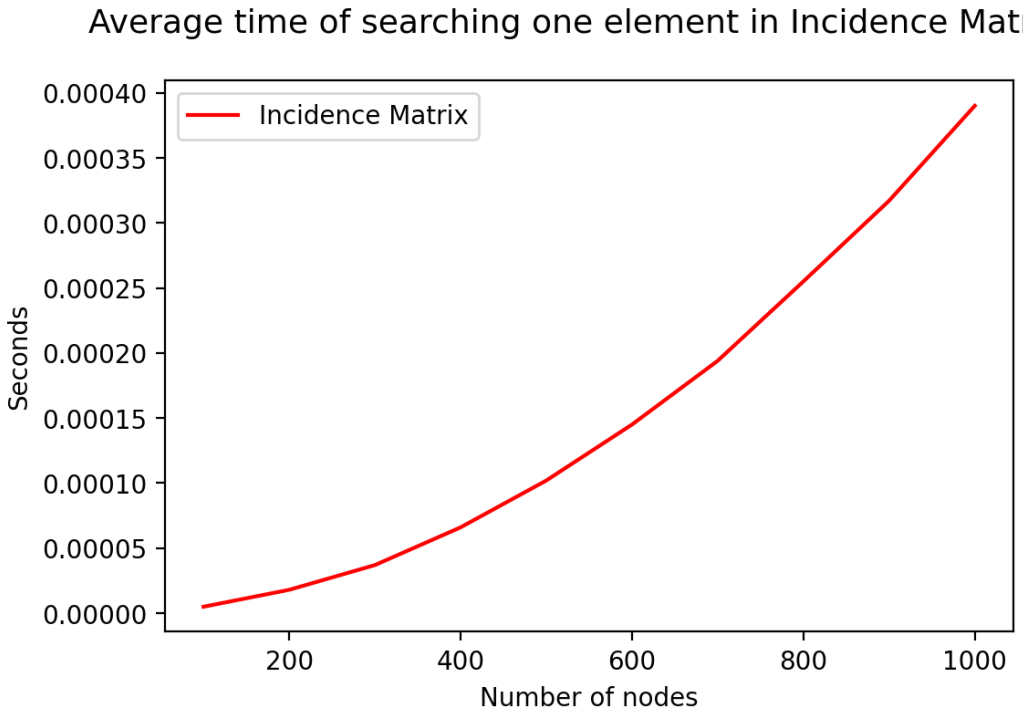
Our assumptions came true - complexity function has a parabolic shape. To be more precise we can say that the complexity function is exactly $saturation * \frac{n*(n-1)}{2}$ since in the worst case we have to go to the end of the list. 8 microseconds when n = 1000 does not look like a big number but we should remember that this is the time needed to search for 1 edge multiply this by the number of edges and we will not get such a good result anymore.

Vertex matrix



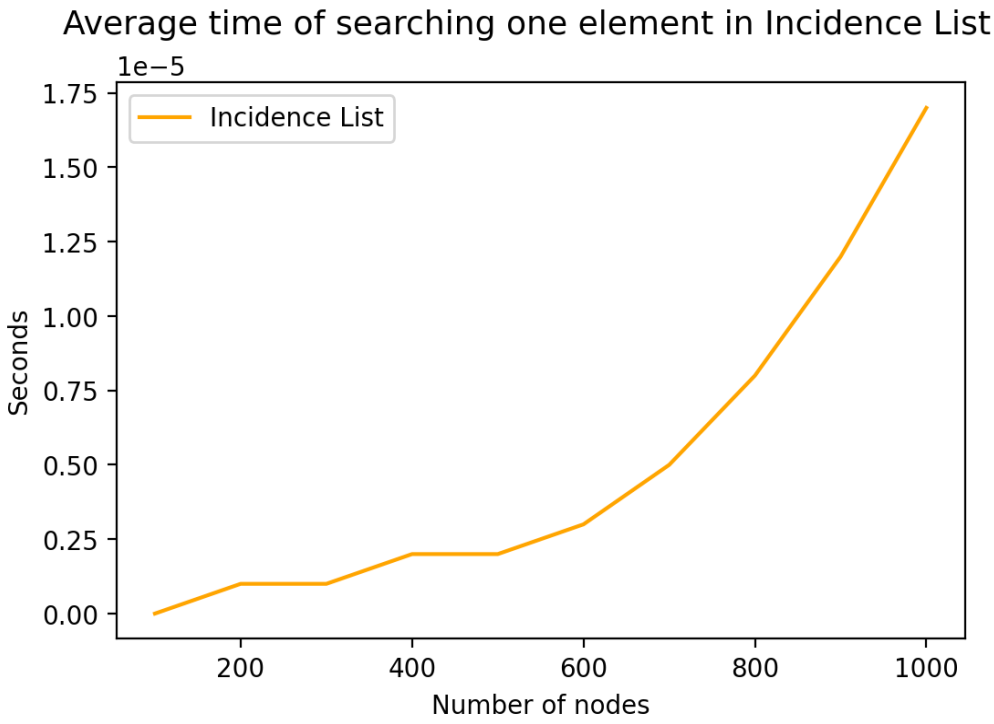
Maybe the straight line is not the most beautiful shape to draw but to see such complexity function is very satisfying. As we expected this representation is able to check if a given edge exists or not in constant time. So when our program has to do a lot of those checks there is no better choice than the vertex matrix to represent our graph.

Incidence matrix



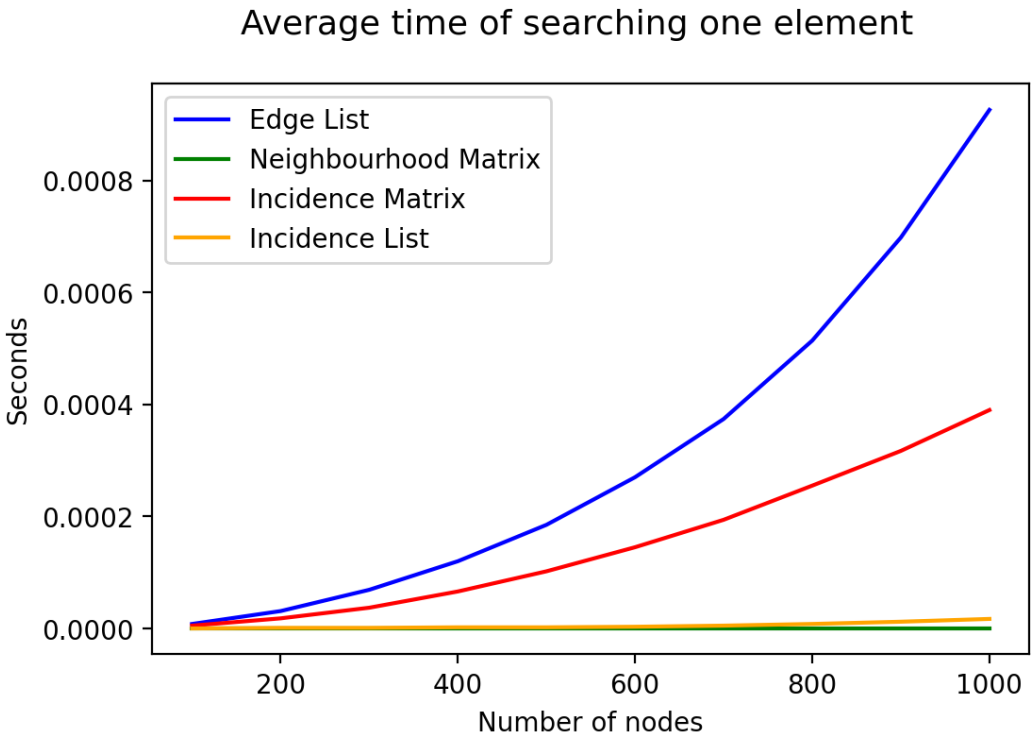
Coming back to parabolic shape. To check whether an edge exists we sometimes need to check all of them and because the amount of edges is approximately equal to n^2 it gives us quadratic complexity. Nevertheless, the incidence matrix is also useful and has neat properties when we look at it in terms of linear algebra.

List of incidence



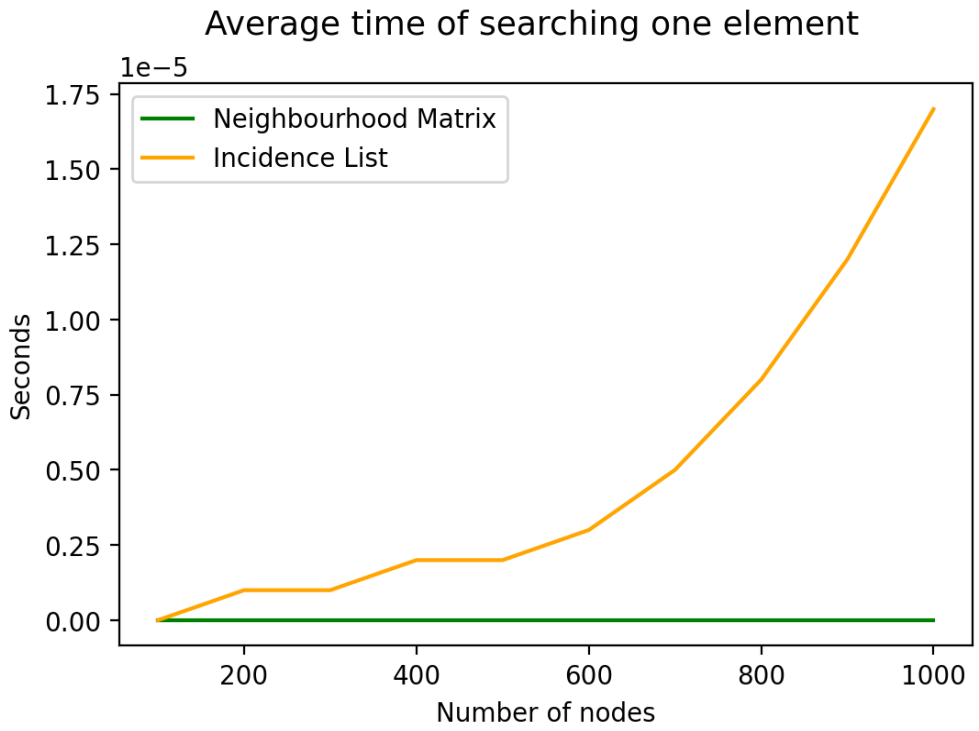
Function still looks like a parabola but we can observe that searching is 10 times faster compared to Edge list and Incidence matrix. Why it is so? We traverse only a list of one node and in the worst case, we go at the very end of it. An edge can have at most n neighbour so this is definitely less to consider than in previous representations.

Comparison of all



Number of nodes	Edge List	Neighbourhood Matrix	Incidence Matrix	Incidence List
100.0	0.000008	0.000000	0.000005	0.000000
200.0	0.000031	0.000000	0.000018	0.000001
300.0	0.000069	0.000000	0.000037	0.000001
400.0	0.000120	0.000000	0.000066	0.000002
500.0	0.000185	0.000000	0.000102	0.000002
600.0	0.000270	0.000000	0.000145	0.000003
700.0	0.000374	0.000000	0.000194	0.000005
800.0	0.000514	0.000000	0.000255	0.000008
900.0	0.000698	0.000000	0.000317	0.000012
1000.0	0.000926	0.000000	0.000390	0.000017

We can easily see which representation is the slowest one but in this plot, one may say that difference between Incidence List and Neighbourhood matrix is no substantial, so we present plot comparing only those two below.

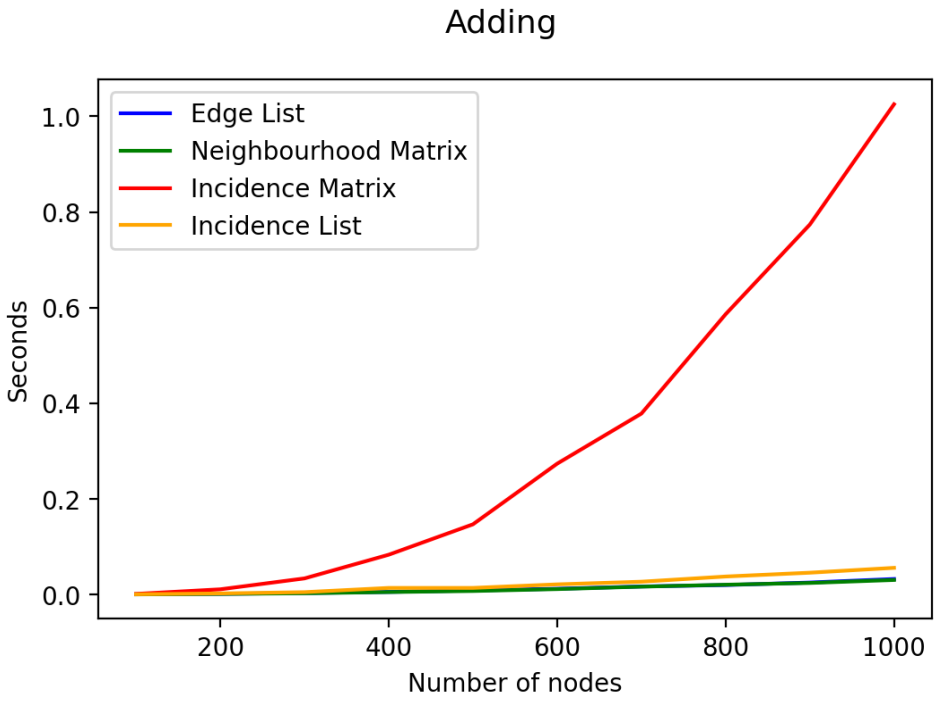


And now clearly we can say that vertex matrix is the winner!

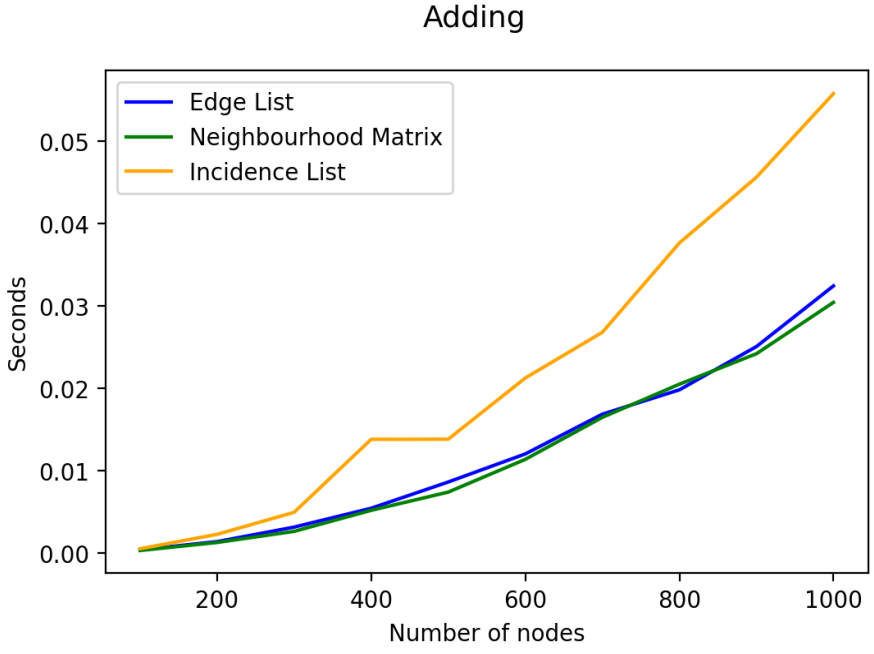
After doing this experiment we were curious if maybe searching the vertex matrix is the fastest but before the search, we have to create a representation so we also measured the time needed to build a representation from the txt file.

Measuring time needed to create all representations

Number of nodes	Edge List	Neighbourhood Matrix	Incidence Matrix	Incidence List
100.0	0.000378	0.000301	0.001333	0.000511
200.0	0.001380	0.001276	0.010904	0.002264
300.0	0.003141	0.002625	0.033719	0.004940
400.0	0.005434	0.005179	0.083257	0.013809
500.0	0.008621	0.007393	0.146818	0.013821
600.0	0.012031	0.011372	0.273657	0.021254
700.0	0.016861	0.016476	0.378294	0.026807
800.0	0.019818	0.020505	0.586043	0.037659
900.0	0.025074	0.024203	0.773577	0.045645
1000.0	0.032447	0.030444	1.025189	0.055805



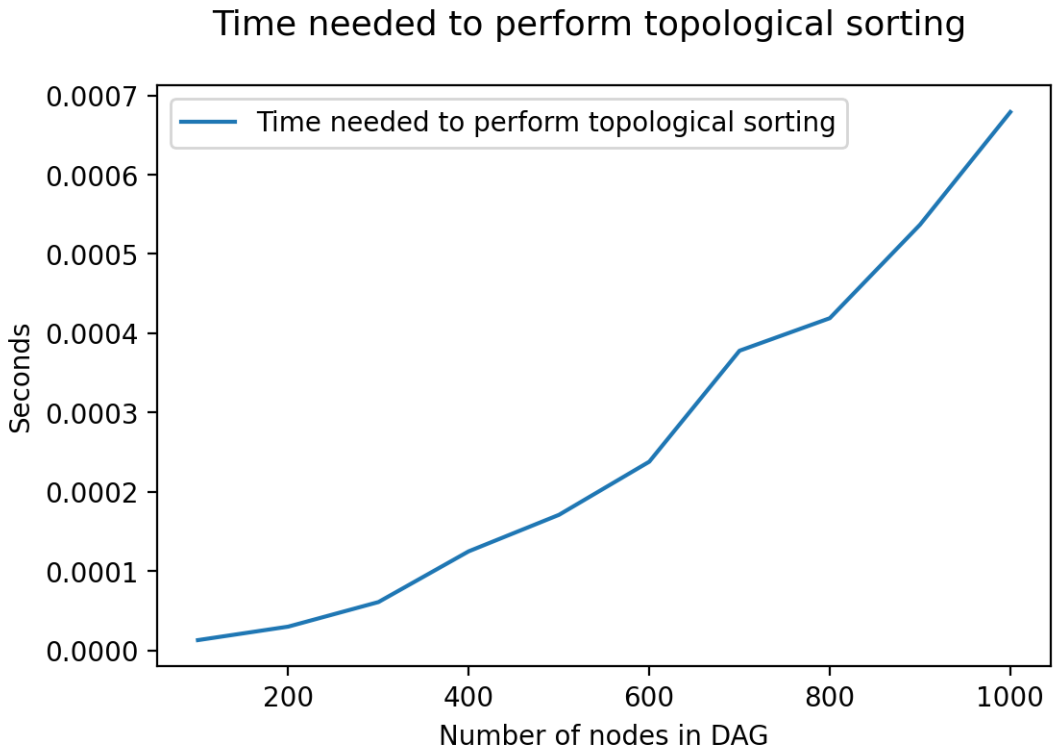
Without any doubt, the incidence matrix is the slowest one to create as we have matrix n by m which in fact takes n^3 space, and allocating as much memory may be time-consuming. We used calloc function here and it should be quite efficient in allocating memory already filled with 0, but maybe for very big sizes of array it is not as fast as we expected . But how lists can be so fast? As we can remember from the previous report it takes O(n) to add one element in the worst case, but it was an ordered list. In this example order of edges doesn't really matter so we can either put each new edge at the end of the list or either at the beginning of it and it takes O(1) so adding all elements takes O(n). In the case of the Vertex matrix time also should be linear as we just change the value of the given cell from 0 to 1 n times. Now let's zoom the plot to see better.



So incidence List is not as fast as Edge List and vertex matrix but we can see the difference is not so substantial as in the case of the incidence matrix.

Experiment 2 - measuring time needed to perform topological sorting on Directed Acyclic Graph

For this experiment, we chosen a topological sorting algorithm that uses BFS to traverse through the graph. The algorithm visits one node, then it needs to search all neighbours of this node where it is right now. This is the reason why we chose a list of incidence as our graph representation. In the edge list, we would need to traverse through all lists every time we visit one node, which would give us time complexity *O(n*e).* The same situation happens while using the incidence matrix, for each node we would need to iterate over all edges, complexity - *O(n*e).* So two representations last – vertex matrix and adjacency list. Both have the same worst-case complexity in this algorithm – *O(n*n),* as we traverse all nodes and in every edge, we would need to search for the existence of the connection between this and one another node. In the neighbourhood matrix for searching an existence of edge in all cases, we would need to iterate over all n nodes while visiting one node. In the list of incidence this situation happens very rarely, only, if every node is connected to each other, so it will only occur in a complete graph. This implies, that the adjacency list will be a little faster than the vertex matrix. We should also take into account that DAG is rather sparse type of graph so using either vertex or incidence matrix may be memory consuming. The total time complexity using the adjacency list and BFS algorithm will be *O(n + e).*



A maximum number of edges in DAG is $n * (n - 1)$, so we can simplify it to n2. In our random graphs, we had an edge saturation equal to 30% of all possible edges. As the time complexity of the topological search *is O(n + e),* we could assume that it will be $O(n + n^2)$, so

$O(n^2)$. The thing is, we have a 30% edge saturation, so the coefficient in this quadratic function is very low. This is exactly, what the plot shows. We can see here a parabola, which is a little flattened

Conclusions

Taking into account those experiments and our thoughts, without any doubts an edge list is representation that is the least useful. It performs slow in searching, doesn't have any special attributes and it is hard to write any efficient algorithm using this representation. One advantage is that is it very easy to implement and it may be beneficial to use it when we want to store our graph in txt file and easily import it into our program.

Incidence matrix is very slow when we use it for searching an edge and it is the most memory consuming representation of all. On the other hand it has some properties that are very useful in linear algebra so for sure this representation has some pros and cons and in some particular cases it will be the best one to use.

Neighbourhood matrix is definitely the fastest one when we want to know whether some edge exists or not it is also quite easy to implement. So when this operation is crucial one should use vertex matrix for sure. The only drawback of this representation we found is that it may be not the best choice when graph is sparse, as we will store a lot of zeroes and consume memory unnecessarily.

Edge list is very good in searching for an edge, of course not as good as vertex matrix but still performs very well. Big advantage of adjacency list is that it is created using dynamic data structure, namely linked list so we always consume only as much memory as we need. It is also readable and is good for representing graph in txt file. What's more it suits to topological search and variety of another algorithms very well.

Summing this part, for us, two most useful representations are Neighbourhood matrix and Edge list.

Considering topological sort it is very useful algorithm and it has many real life applications such as: instruction scheduling, ordering of formula cell evaluation when recomputing formula values in spreadsheets, logic synthesis, determining the order of compilation tasks to perform in makefiles, data serialization, and resolving symbol dependencies in linkers. It is also used to decide in which order to load tables with foreign keys in databases. Algorithm can be implemented using either BFS and DFS approaches, here we selected BFS as it does not require to use recurrence.