I

In this exercise the objective was to examine the time it takes to check for the existence of an edge between 2 random vertices. Representations undergoing examination are : the adjacency matrix (AM), the incidence matrix (IM), the edge list (EL) and the incidence list (IL). Before thorough line chart analysis, it would be useful to expand on the inner workings of the edge checking algorithms for each of the different graph representations.

AM : In the adjacency matrix, the algorithm consists of a single operation- namely a check whether there is a 1 in the i-th row, j-th column (assuming the investigation aims to obtain information about the edge between -accordingly-the i-th and j-th vertex) – hence the time complexity of the algorithm is O(1) – taking constant time to perform it.

```
bool check_edge(int a, int b) {
    return edges[a][b] == 1;
}
```

Code snippet 1

IL: In our implementation of the incidence list, for each vertex it holds that the vector storing the incident vertices remains sorted – this is not a problem in the particular case of this project, because of the way the graph is generated, however in other cases for the algorithm to work in the same fashion as described below, edges would have to get added in such a way that would not disturb the order. Having included the above disclaimer, one can get down to the brass tacks: Thanks to the afore mentioned nature of the incidence list, assuming one would like to find out if the edge between vertex a and vertex b exists – firstly one finds the vector which contains the neighbors of vertex a – that is the a-th "row", secondly binary search is performed on it - if b cannot be found – it means that the particular edge does not exist. The complexity of binary search is logarithmic and the maximum size of the vector is V – since a vertex can not be adjacent to more than the number of vertices belonging to the graph – therefore the time complexity is O(log(V)).

```
bool check_edge(int a, int b) {
    if (edges[a].size() > 0) {
        int beg = 0, end = edges[a].size() - 1;
        while (beg < end) {
            int mid = (beg + end) / 2;
            int _b = edges[a][mid];
            if (_b >= b)
                end = mid;
            else
                beg = mid + 1;
        }
        if (edges[a][beg] == b)
            return true;
        return false;
    }
    return false;
}
```

Code snippet 2

EL: Regarding the edge list, a similar property exists as in the incidence list *(it holds that if e_{ab} stands for an existing edge between the vertices a and b, if a>c e_{ax} will always appear before e_{cy} on the edge list, analogously e_{xa} will always appear before e_{xc})*, this results in a related way of operating: A binary search is performed, performing checks by the 2 vertices - a and b, if the edge searched for can be found on the edge list – that means the given edge exists. Since the number of edges is equal to (saturation = 0.6)*(max edges = V*(V-1)) the time complexity is $O(\log(V^2)) = O(2*\log(V))=O(\log(V))$.

```
bool check_edge(int a, int b) {
    int beg = 0, end = edges.size()-1;
    while (beg < end) {
        int mid = (beg + end) / 2;
        int _a = edges[mid][0], _b = edges[mid][1];
        if (_a > a || (_a == a && _b >= b))
            end = mid;
        else
            beg = mid + 1;
    }
    if (edges[beg][0] == a && edges[beg][1] == b)
        return true;
    return false;
}
```

Code snippet 3

IM:

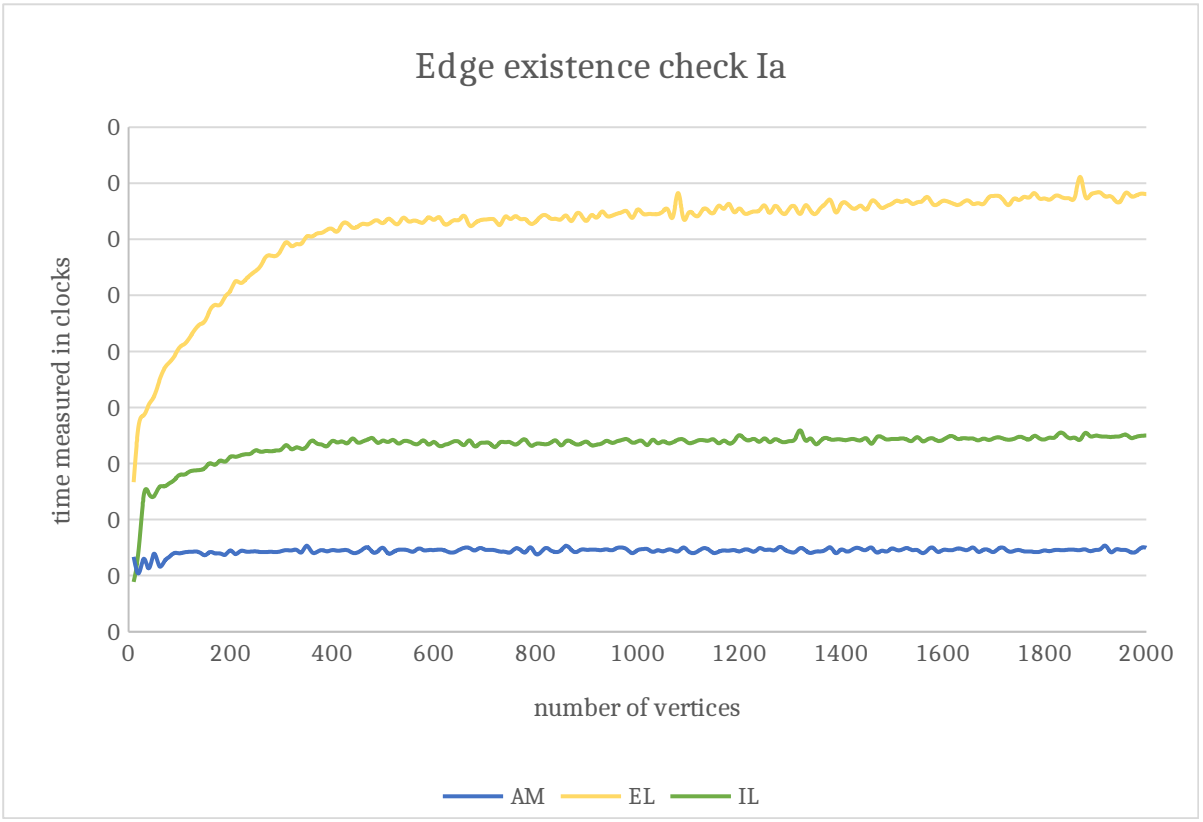
The incidence matrix implementation in this project also has a unique feature, similar to the one in the edge list – that is - *it holds that if* $\mathbf{c_{ab}}$ *stands for an column in the incidence matrix, indicating between which vertices there exists an edge (a and b) , if a<d, a<x and d<y:* c_{ax} *will always appear before* c_{dy}, the second and third condition is due to the fact, that a basic IM can only be used for undirected graphs. Without this condition the algorithm would be $O(E) = O(V^2)$, since every column would have to be checked – for 1s in the a-th and b-th row(a<b), but due to the prior mentioned condition, one can perform a binary search:

Starting from the middle column the algorithm checks if vertex a (c_{ab}) is bigger (by the number assigned to it) than the vertex c (edge e_{cd} is searched for, c<d, so the algorithm checks for the existence of a column with 1s in the c-th and d-th position), if it is equal, the same check for the second component (d) is performed(whether d>b) – that means "checking the middle" is O(V) since there are |V| vertices in a column. If the check is passed the middle becomes the end, if not the middle becomes the beginning. The binary search runs until the stopping condition seizes to be satisfied, and if the particular column corresponds to $e_{cd,}$ that indicates the existence of such an edge. Since the above mentioned "middle check" is O(V), and the binary search is $O(\log(V^2))$, the whole algorithm is O(V*log(V)), which is much worse than the prior mentioned ones, this can be clearly seen on Chart 2, all the other algorithms vastly

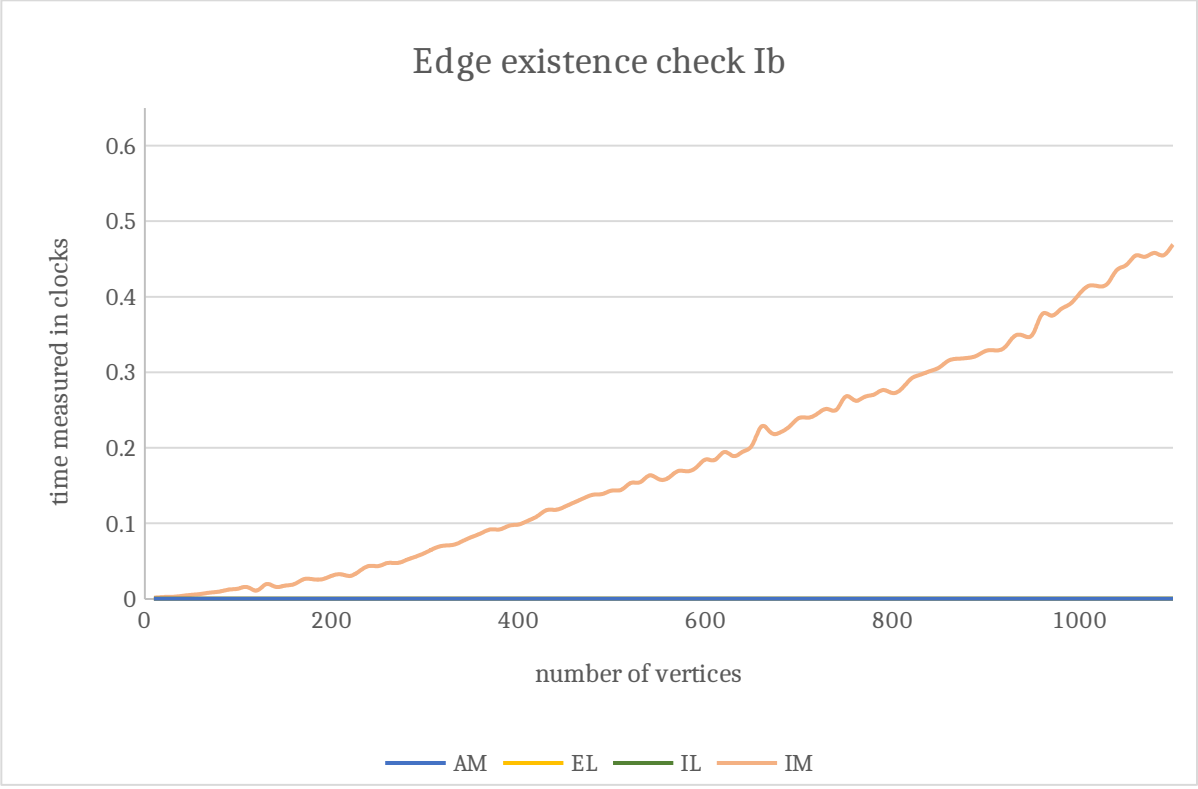
```
bool check_edge(int a, int b) {
    int beg = 0, end = E - 1;
    int _a = -1, _b = -1;

    /// start binary search
    while (beg < end) {
        int mid = (beg + end) / 2;
        /// _a, _b is edge stored in mid, but we have to find this values in O(V)
        _a = -1, _b = -1;
        for (int i = 0; i < V; i++) {
            if (edges[i][mid] == 1 && _a == -1)
                _a = i;
            else if(edges[i][mid] == 1)
                _b = i;
        }
        if (_a > a || (_a == a && _b >= b))
            end = mid;
        else
            beg = mid + 1;
    }

    _a = -1, _b = -1;
    for (int i = 0; i < V; i++) {
        if (edges[i][beg] == 1 && _a == -1)
            _a = i;
        else if(edges[i][beg] == 1)
            _b = i;
    }
    if (_a == a && _b == b)
        return true;
    return false;
```

Code snippet 4

outperform it.



Line chart 1



Line chart 2

Botch of the charts on the previous page illustrate the time needed to retrieve information concerning a single edge. The results math the previous descriptions:

(Line chart 1)

- In AM, the operation is performed in constant time(a single operation), -unrelated to the number of vertices
- The line corresponding to the IL, indeed is logarithmic, values are therefore greater than for AM.
- For EL, which is $O(\log(V^2)) = O(2*\log(V)) = O(\log(V))$, the time needed to perform the operation indeed is about 2 times as much as for IL – this would not have to be necessarily true, due to the nature of the big O notation, but it is – which further validates the description above.

(Line chart 2)

- The edge check algorithm for IM performs way worse than all the others – it's $O(n\log(n))$, so much so, that it is impossible to obtain any information about the other lines, since they are all squished together

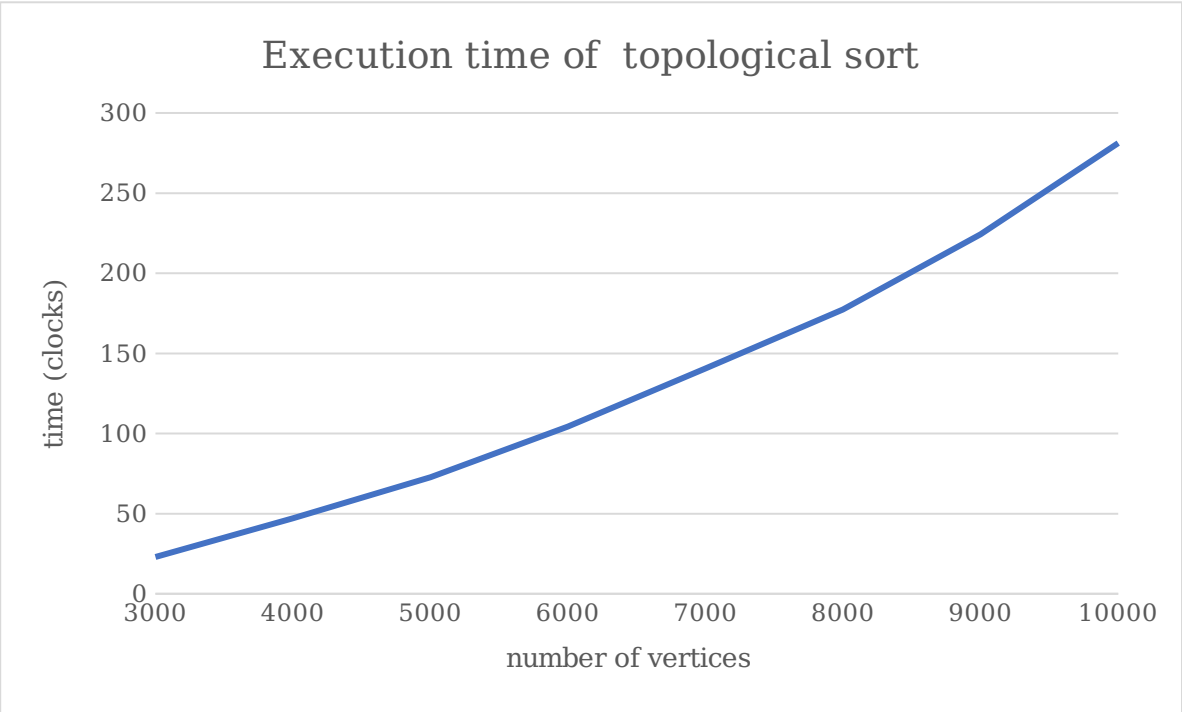
II

For the topological sort procedure, the edge list was used as the representation of choice. The IL seems to be particularly suited for this task due to the easy access (and low complexity of this operation in comparison to the other representations) to the vector of neighbors of a given node. Using different representations, this task would require more time to complete: In the EL one would first have to conduct a search – of time complexity $O(\log(V^2))$ – and then iterate through the "block" (assuming the list is sorted) of edges with the neighbors of the node of interest. Regarding the IM –since a DAG is an directed graph - zeros and ones in a single matrix would not be enough to represent it, one would have to (for example) use -1/0/+1 instead. Moving on to the AM, one would need to iterate through a row corresponding to the given vertex – $O(V)$. As mentioned at the beginning since checking the neighbors of a node is the primary operation in the topological sort algorithm, the incidence list is best suited for it.

The adjacency matrices and incidence lists are the two representations that arguably are best suited for directed graph representation, but for graphs in which the number of edges is significantly smaller than the number of vertices squared, the incidence lists consumes less memory. However they also have their downsides – removing edges is not $O(1)$ as in the AM, but in the top sort algorithm such operations were not performed.

Memory complexity of the representations:

	IM	AM	EL	IL
Memory complexity	$O(E*V) = O(V^3)$	$O(V^2)$	$O(E) = O(V^2)$	$O(E) = O(V^2)$



Line Chart 3

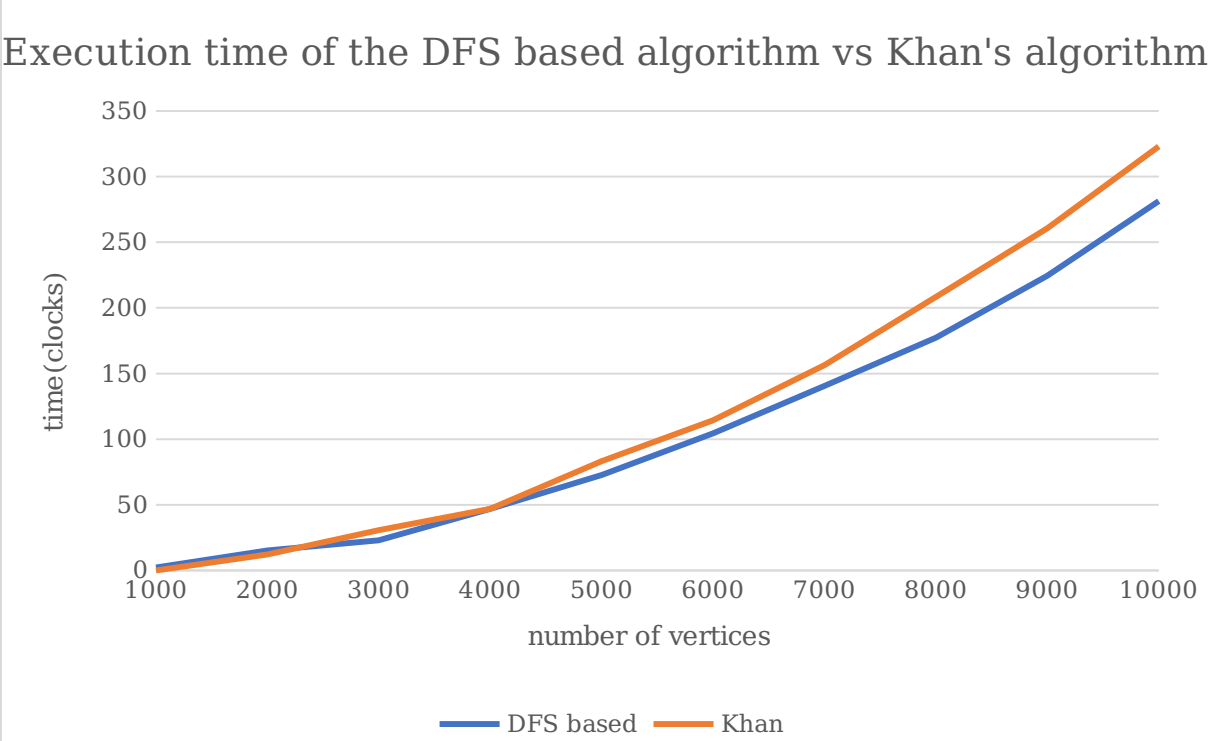
The topological sort invokes DFS for every vertex, if it hasn't been visited before – each vertex then, gets to be pushed to the front of the list (visited vertices are pushed to the front after their successors already have been placed on the list) – which ultimately serves as the outcome of the top sort algorithm. The objective is for all of the nodes to be "aligned" on the list in such a way that the edges "point" right (every outcoming edge of a node in an incoming edge to another, to the right of the prior mentioned one). Since the DFS algorithm has a complexity of O(e + v), every edge gets to be "used" and no vertex gets to be visited more than once) – the top sort algorithm is of time complexity O(E + V) = $O(V^2)$, the memory complexity of the top sort algorithm being O(V), this conflicts with the results displayed by the means of the above line chart, does not to be notably quadratic – this may be due to the sparseness of the graphs (0.3 saturation).

```
class Digraph : public Graph_IL {
private:

    /*** recursive method used by topological_sort method ***/
    void __topological_sort_recursive(int v, bool *visited, std::list<int> *result) {
        visited[v] = true;
        for (auto i : edges[v])
            if (!visited[i])
                __topological_sort_recursive(i, visited, result);
        result->push_front(v);
    }

public:

    Digraph(int V) : Graph_IL(V) {}

    ~Digraph() {
        edges.clear();
    }

    /*** O(1) ***/
    void add_edge(int a, int b) {
        edges[a].push_back(b);
        E++;
    }

    /*** uses DFS to sort nodes in topological order
    ******** O(V + E) ~ O(V + V^2) ~ O(V^2) *****/
    std::list<int>* topological_sort() {
        std::list<int> *result = new std::list<int>();
        bool *visited = new bool[V];
        for (int i = 0; i < V; i++)
            visited[i] = 0;
        for (int i = 0; i < V; i++)
            if (!visited[i])
                __topological_sort_recursive(i, visited, result);
        delete []visited;
        return result;
    }
};
```

Code snippet 5

Remarks:

Interestingly – the two prevalent algorithms employed to conduct topological sorting – namely Khan's algorithm and the DFS-based algorithm (the one previously described) differ in the amount of time needed for the sorting to be executed. Although having the same complexity O(E + V) one can see that for larger graphs the latter performs better.



Line chart 4

This can be explained by the fact that a list of in-degrees has to be maintained (after a vertex is added to the result (dequeued) O(V), the in-degree of its successors has to be decreased, checked for any 0s (O(E)) added to the queue O(V), which introduced additional operations (associated with the queue itself) in comparison to the recursive approach in the DFS based algorithm, but does not change the overall complexity.