PRIM'S AND KRUSKAL'S ALGORITHM

Senyk Anastasiia

Prytula Matvii

Task

The task is to compare the execution time of two algorithms for different numbers of vertices and different filling graphs. Determine in which cases which of the algorithms works better.

Experiment

Experiments are needed to find the execution time of the algorithms, on the completeness of the graph from 0.01 to 1 with a step of 0.01 and for vertices from 5 to 500 with a step of 5.

Number of experiments = (500/5) * (1/0.01) = 10,000

Each experiment should be performed 2-3 times for the results to be effective.

Number of experiments = 10,000*(n)> 28,000 (for each of the algorithms)

RUNNING DEVICE CHARACTERISTICS

- Apple MacBook Air M1
- 8-Core
- CPU clock-rate: 3.2 GHz
- RAM: 16 GB
- macOS Monterey version 12.1

ALGORITHMS

KRUSKAL'S

```python
"""Kruskal's algorithm"""


def kruskal(nodes: list, edges: list) -> tuple[list[list], int]:
    """
    finds a minimum spanning forest of an undirected edge-weighted connected graph
    using Kruskal's algorithm

    :param nodes: list of graph nodes
    :param edges: list of graph edges
    :return: tuple of the minimum spanning forest and the weight of its frame
    """
    nodes, edges = list(map(lambda x: [x], nodes)), sorted(edges, key=lambda x: x[2])
    weight, edges_list = 0, []
    while edges:
        vert1, vert2, vert_weight = edges.pop(0)
        for item in nodes:
            if vert1 in item:
                index_vert1 = nodes.index(item)
            if vert2 in item:
                index_vert2 = nodes.index(item)
        if index_vert1 != index_vert2:
            nodes[index_vert1].extend(nodes.pop(index_vert2))
            weight += vert_weight
            edges_list.append([vert1, vert2])
    return edges_list, weight

```

PRIM'S

```python
"""
Prim's algorithm
"""
import heapq


def find_adj(node, edges):
    """
    find nodes adjacent to the given
    :param node:
    :param edges:
    :return: list[tuples] -> adjacent nodes
    """
    children = []
    for i in edges:
        if i[0] == node:
            children.append((i[2], i[1], i[0]))
        elif i[1] == node:
            children.append((i[2], i[0], i[1]))
    return children


```

```python
def prim_mst(nodes, edges):
    """
    perform Prim's algorithm
    :param nodes:
    :param edges:
    :return: spanning tree and cost of it
    """
    span, mst_cost, visited_set = [], 0, set()
    start_vertex = nodes[0]
    frontier_minheap = find_adj(start_vertex, edges)
    heapq.heapify(frontier_minheap)
    visited_set.add(start_vertex)
    while len(visited_set) < len(nodes):
        curr_vertex = heapq.heappop(frontier_minheap)
        cost = curr_vertex[0]
        if curr_vertex[1] not in visited_set:
            mst_cost += cost
            visited_set.add(curr_vertex[1])
            span.append(curr_vertex)
            children = find_adj(curr_vertex[1], edges)
            for child in children:
                if child[1] not in visited_set:
                    heapq.heappush(frontier_minheap, child)
    return span, mst_cost

```

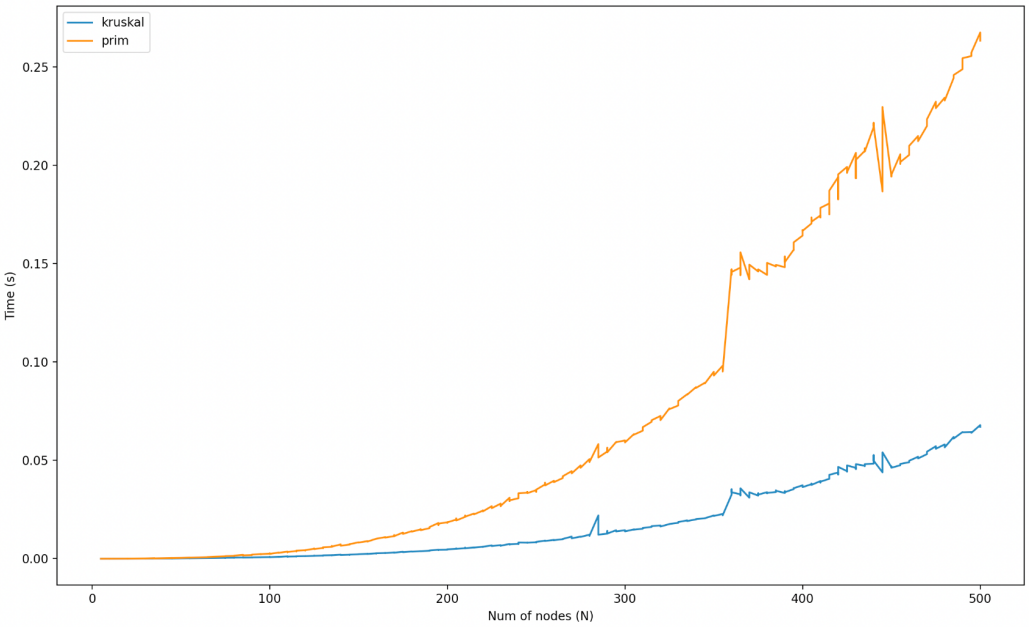
EXPERIMENTS CODE

EXPERIMENTS

For each experiment the running time was measured for both Kruskal's and Prim's algorithms.

EXPERIMENT #1

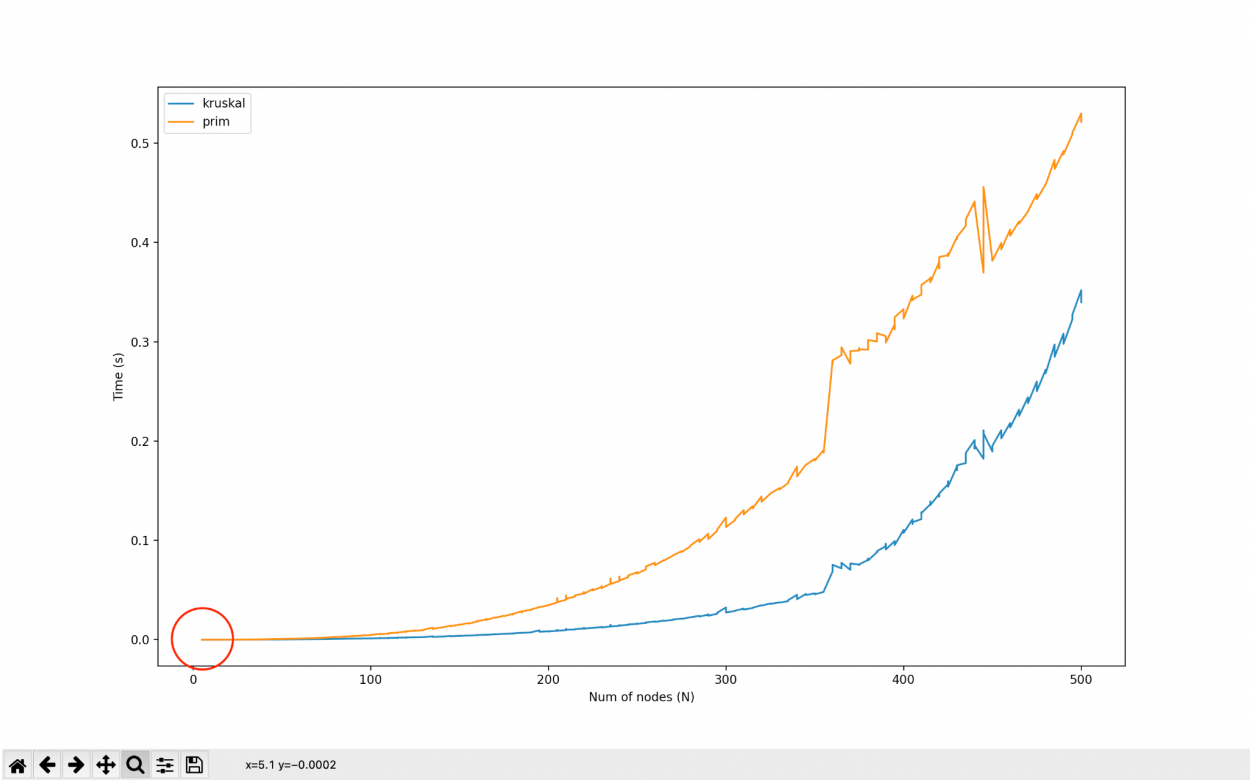
Completeness: 0.1



The fact that both graphs intersect only at the starting point shows that when each vertex is connected to 10% of all other vertices(thus, for 500 vertices there are $[C^2_{500} * 0.1]$ edges), Kruskal's algorithm is way more efficient.

EXPERIMENT #2

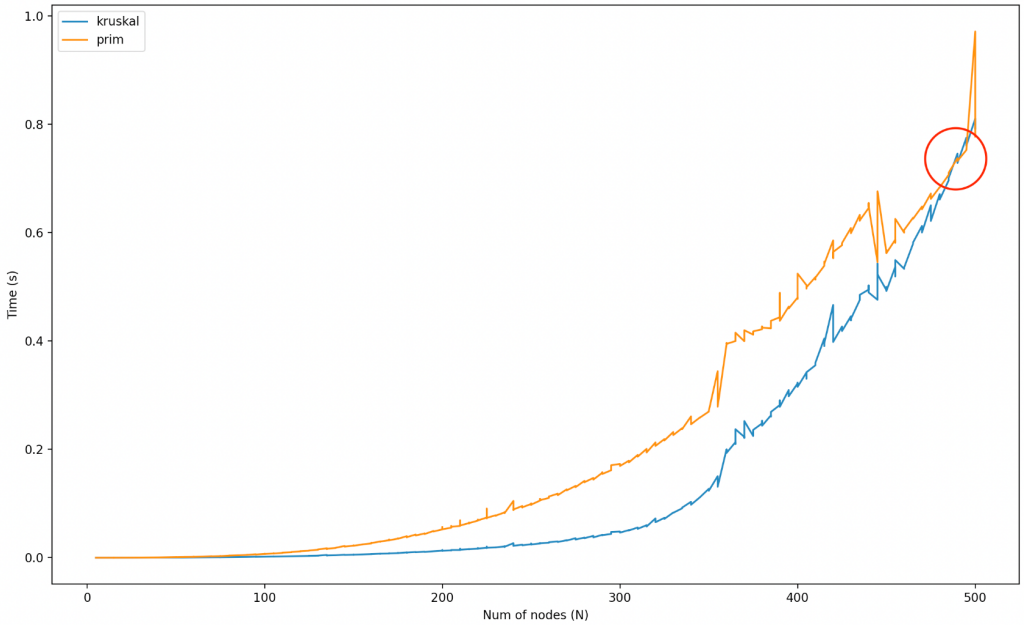
Completeness: 0.2



Still even for when the number of edges is [C^2_{500} * 0.2] Kruskal's algorithm is more efficient.

EXPERIMENT #3

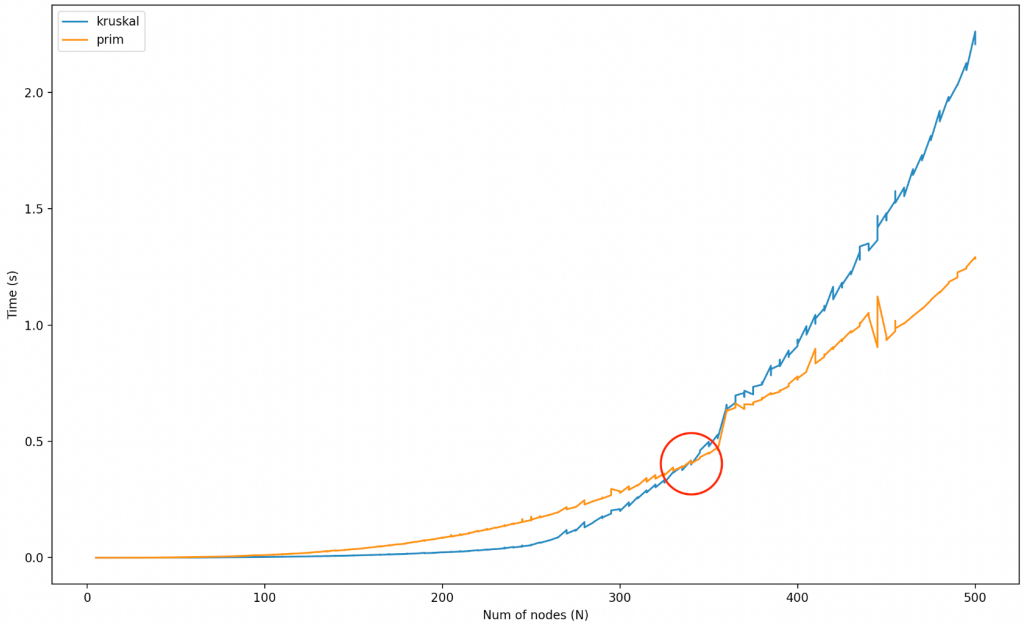
Completeness: 0.3



Here we can see, that when the number of edges is [$C^2_{num\_of\_vertices}$ * 0.3] when the graph has more than 480 vertices the Prim's algorithm requires less execution time.

EXPERIMENT #4

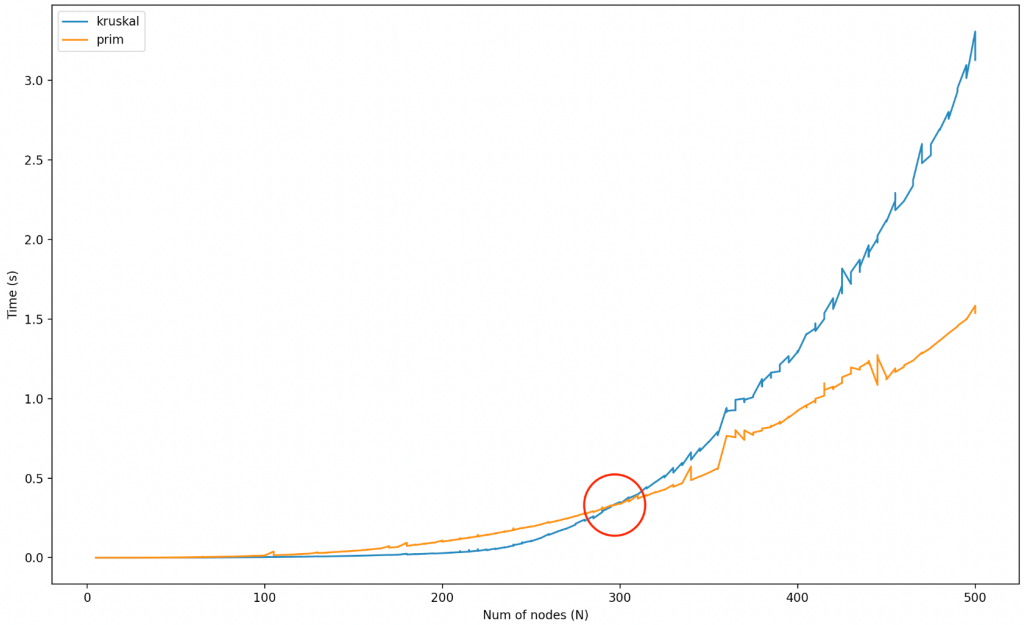
Completeness: 0.4



As the number of edges grows, so does the execution time for Kruskal's algorithm compared to Prim's. When the number of edges is [$C^2_{num\_of\_vertices}$ * 0.4], if the graph has more than 350 vertices it's better to use Prim's algorithm.

EXPERIMENT #5

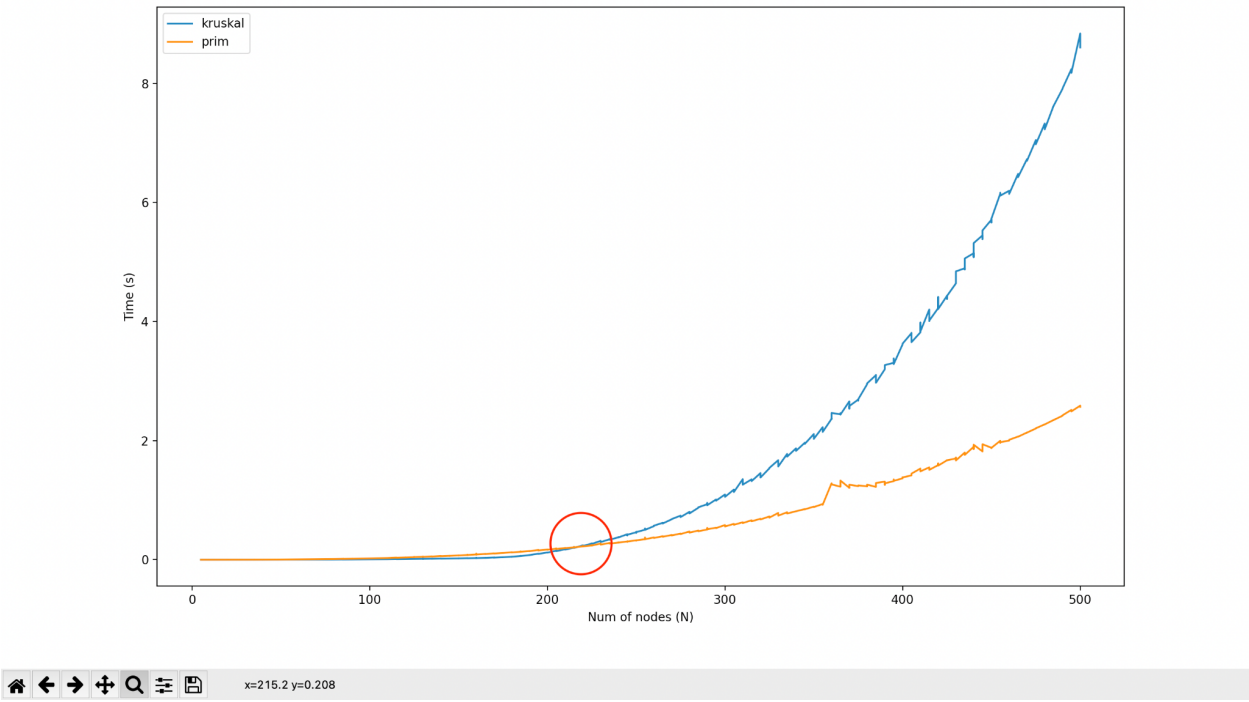
Completeness: 0.5



When the number of edges hits [C^2_{500} * 0.5], Prim's algorithm performs almost two times faster.

EXPERIMENT #6

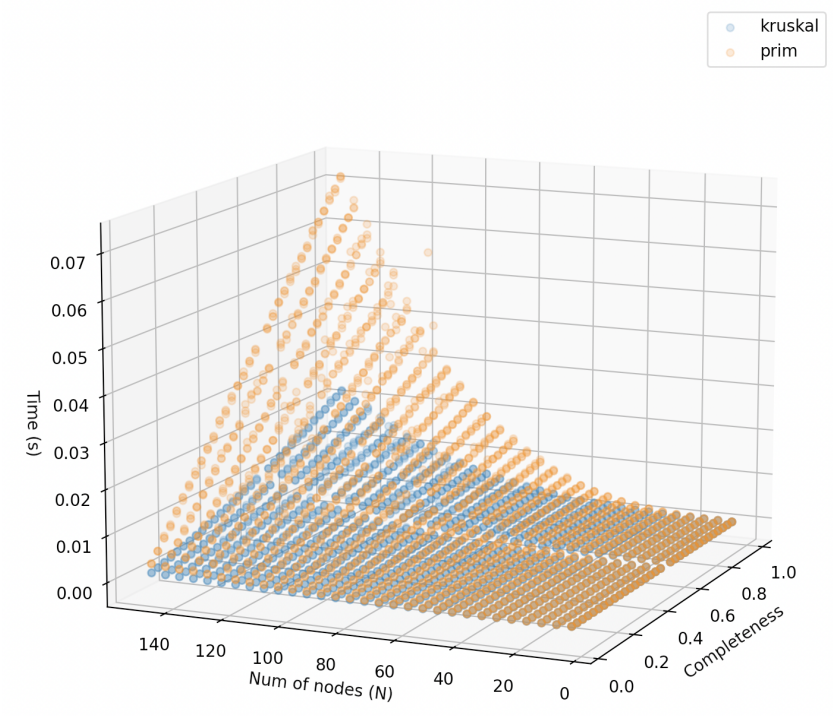
Completeness: 1



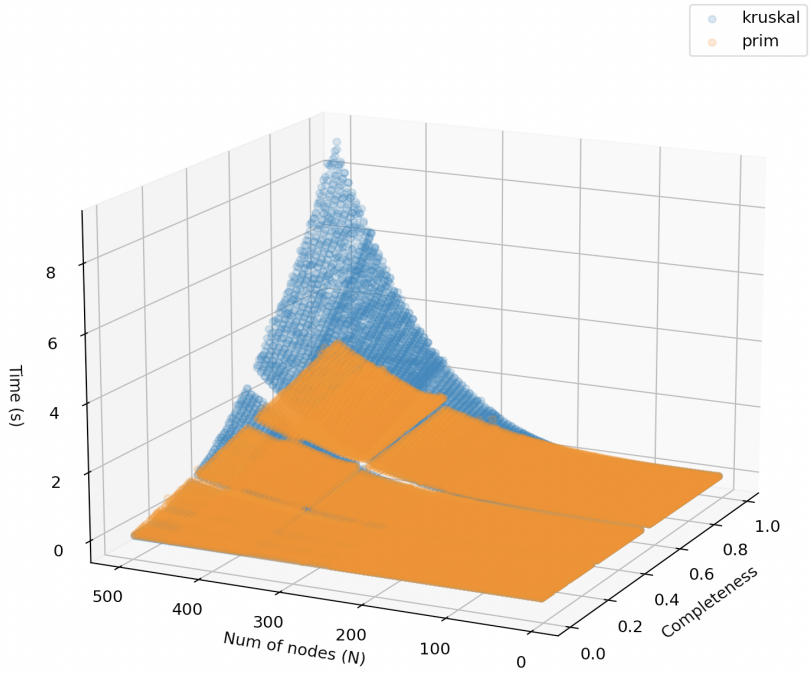
Here, it is shown that as soon as a complete graph has 215 vertices, Prim's algorithm becomes more efficient for it.

Kruskal's time complexity: O(E*V)
Prim's time complexity: O(E*log(V))

Conclusion



The three-dimensional graph with 150 vertices shows that the Kruskal's algorithm works much more efficiently than the Prim's algorithm with different completeness of the graph.



While the second graph with 500 vertices shows the opposite results.

Since the time complexity of Kruskal is O (E * V) and the time complexity of Prima is O (E * log (V)), it is better to use the first algorithm with a small number of vertices to find the path of the graph, while Prim's algorithm becomes more efficient as they increase(as the closer the number of vertices is to zero, the more equally fast grows the function V and log(V)). Just the same goes for the completeness of the graph. The closer it is to zero, the less time is required to perform both algorithms(as completeness determines the number of edges in the graph, whereas time complexities of the algorithms depend on this variable).

This is shown in the graphs above