FINDING THE MINIMUM SPANNING TREE

Analysis & Design of Algorithms Final Project

By: Jana Walid, Jana Hossam, Sara Ihab, Jana Abotaleb & Mariam Emad

INTRODUCTION

Using five different MST Algorithms, we found the minimum spanning tree of five different datasets, and created videos for those MSTs found. We also created a GitHub documentation and a website to showcase and explain our findings.

ALGORITHM ONE: KRUSKAL

```python
def kruskal_mst_incremental(G):
    n = G.number_of_nodes()
    node_map = {node: i for i, node in enumerate(G.nodes())}
    uf = UnionFind(n)

    # Edge format: (u, v, weight)
    edges = [(u, v, G[u][v]['weight']) for u, v in G.edges()]
    edges.sort(key=lambda x: x[2])

    mst_edges = []

    for u, v, weight in edges:
        u_idx, v_idx = node_map[u], node_map[v]
        if uf.union(u_idx, v_idx):
            mst_edges.append((u, v, weight))
            if len(mst_edges) == n - 1:
                break

    return mst_edges
```

- Node Map orders nodes in case they were not ordered (in other cases it can be used as a way to turn string nodes into integers)
- edges stores each edge in a tuple so that it can then be sorted and iterated over
- uf.union returns true if it is not connected (hence won't cause a cycle)
- if length of mst_edges =n-1, then the mst is complete, we break.
- total cost is O(e log (e)) where e is the number of edges.

ALGORITHM TWO: PRIM'S FIRST IMP.

- Data structure: Uses only a priority queue
- Time Complexity: O(E log V)
- Best for: Dense graphs .
- Drawbacks:
- Poor performance on sparse graphs.
- Requires checking all vertices in each iteration.
- Simple but inefficient for large or sparse graphs.

```python
def prim_mst_incremental(G):
    # Choose any starting node
    start_node = next(iter(G.nodes()))
    mst_edges = []
    visited = {start_node}  # Keep track of visited nodes

    # Priority queue (min-heap) to store edges as (weight, from, to)
    heap = []

    # Add all edges from the start node to the queue
    for neighbor in G.neighbors(start_node):
        weight = G[start_node][neighbor]['weight']
        heapq.heappush(heap, (weight, start_node, neighbor))

    while heap and len(visited) < G.number_of_nodes():
        weight, from_node , to_node  = heapq.heappop(heap)

        if to_node in visited:
            continue

        visited.add(to_node)
        mst_edges.append((from_node, to_node, weight))

        for neighbor in G.neighbors(to_node):
            if neighbor not in visited:
                heapq.heappush(heap, (G[to_node][neighbor]['weight'], to_node, neighbor))

    return mst_edges
```

```python
f prim_mst_incremental_using_adj(adj_list):
    visited = set()
    mst_edges = []

    # Pick any starting node (first key in the adjacency list)
    start_node = list(adj_list.keys())[0]
    visited.add(start_node)

    # Priority queue of edges: (weight, from_node, to_node)
    heap = []

    for neighbor, weight in adj_list[start_node]:
        heapq.heappush(heap, (weight, start_node, neighbor))

    while heap and len(visited) < len(adj_list):
        weight, from_node, to_node = heapq.heappop(heap)

        if to_node in visited:
            continue

        visited.add(to_node)
        mst_edges.append((from_node, to_node, weight))

        for neighbor, w in adj_list[to_node]:
            if neighbor not in visited:
                heapq.heappush(heap, (w, to_node, neighbor))

    return mst_edges
```

ALGORITHM TWO: PRIM'S SECOND IMP.

- Prim's Algorithm with Adjacency List + Min-Heap
- Data structure: Uses a min-heap (priority queue) + adjacency list.
- Time Complexity: O(E log V)
- Best for: Sparse graphs
- Benefits:
- Much faster and scalable than standard Prim.
- Efficient edge lookups and updates using heap.
- Widely used in real-world applications and competitive programming.

ALGORITHM THREE: BORUVKA

- Borůvka’s algorithm
- Goal: Finds the Minimum Spanning Tree (MST) of a weighted, undirected graph.
- Method: Each component picks its lightest outgoing edge in every round.
- Merge: Selected edges are added, and components are merged.
- Parallelizable: Components operate independently → ideal for distributed computing.
- Efficiency: Runs in O(E log V) time with at most log V rounds.

```
boruvka_mst_incremental(G):
n = G.number_of_nodes()#retrieves the num of nodes
node_map = {node: i for i, node in enumerate(G.nodes())}#maps each node for to its unique integer
uf = UnionFind(n)#used to manage the components

mst_edges = []#to store the edges to build the MSTs

while len(mst_edges) < n - 1:#keeps looping until it has n-1 edges the exact num of edges required
    cheapest = [-1] * n#stores the min edge coming out of each component and it treats each node initailly as its

    for u, v in G.edges():#used to initally collect the cheapest edge between components
        weight = G[u][v]['weight']
        u_idx, v_idx = node_map[u], node_map[v]#converts each node to its label
        comp_u, comp_v = uf.find(u_idx), uf.find(v_idx)#retreives the root of the components of these two nodes

        if comp_u != comp_v:#only wants to consider edges which belongs to different components
            if cheapest[comp_u] == -1 or weight < cheapest[comp_u][0]:#checks if the current edge is smaller than
                cheapest[comp_u] = (weight, u, v)#if true it updates the cheapest to the new edge
            if cheapest[comp_v] == -1 or weight < cheapest[comp_v][0]:#repeats the same steps but for the other c
                cheapest[comp_v] = (weight, u, v)

    for edge_info in cheapest:#iterates over the cheapest edges found
        if edge_info != -1:#skips if no cheapest edge was yet recorded for that component
            weight, u, v = edge_info
            u_idx, v_idx = node_map[u], node_map[v]
            if uf.union(u_idx, v_idx):#adds only edges to the MST's edges if they don't form a cycle
                mst_edges.append((u, v, weight))

return mst_edges
```

ALGORITHM FOUR: REVERSE-DELETE

- Sorts all edges descendingly and starts removing the heaviest edges as long as it doesn't disconnect the graph
- Time Complexity: O(E^^2)
- Space Complexity: O(V+E)

```python
def reverse_delete_mst_incremental(G):
    edges = [(G[u][v]['weight'], u, v) for u, v in G.edges()]
    edges.sort(reverse=True)  # Start with heaviest edges

    current_graph = G.copy()
    removed_edges = []

    for weight, u, v in edges:
        current_graph.remove_edge(u, v)#removes the edge
        if not nx.is_connected(current_graph):#checks if the graph becomess disconnected
            current_graph.add_edge(u, v, weight=weight)#if true then it can't remove this edge and adds it back to the graph
        else:
            removed_edges.append((u, v, weight))#else the edge it removed and added to the list of removed edges

    mst_edges = [(u, v, current_graph[u][v]['weight']) for u, v in current_graph.edges()]#the remaining edges are the ones which form the mst
    return mst_edges #to return the mst of the graph
```

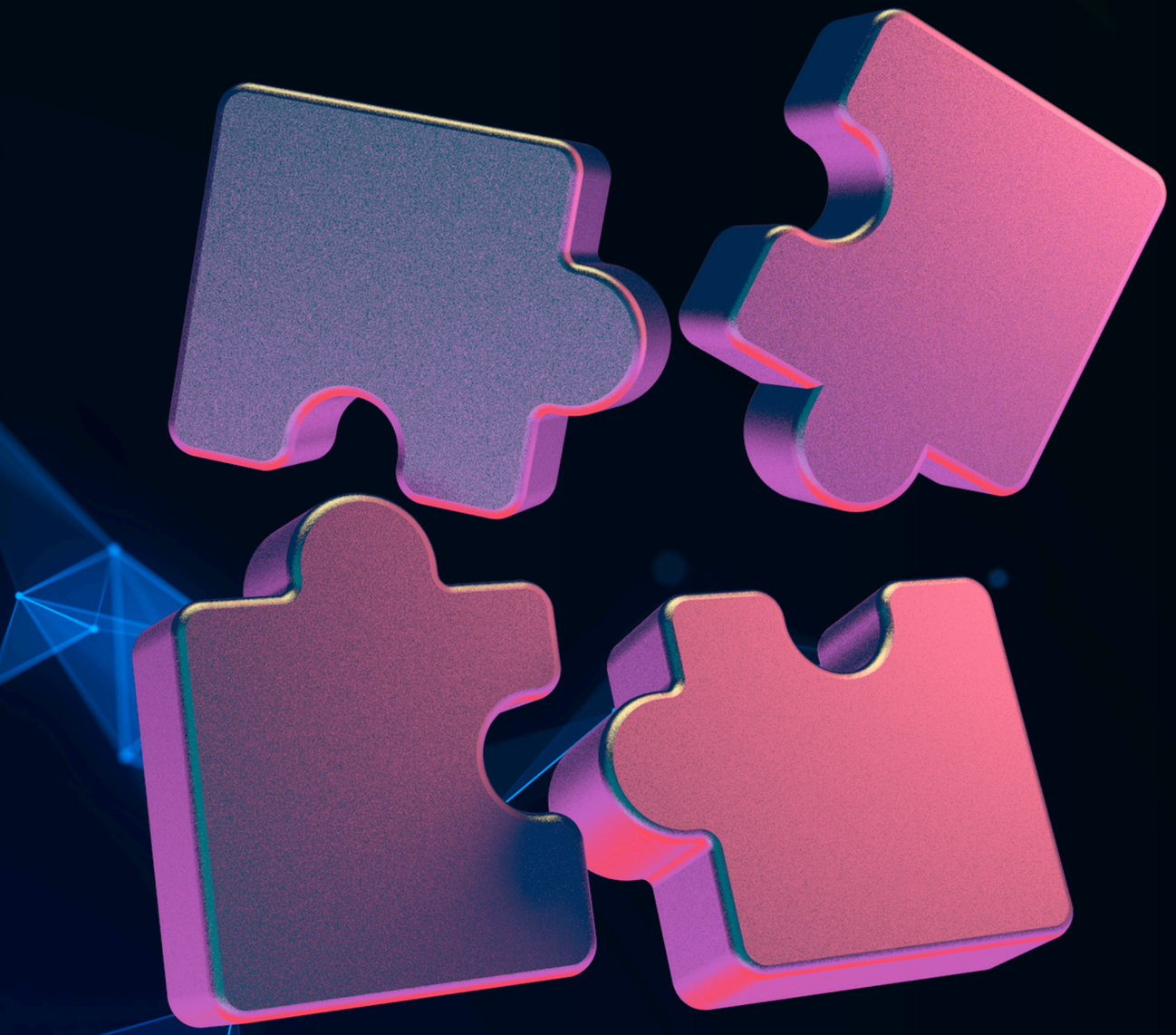
ALGORITHM FIVE: KARGER

- Karger's Algorithm
- Goal: Finds a minimum cut in an undirected graph.
- Method: Randomly contracts edges, merging two nodes into one.
- Process: Repeats edge contraction until only two nodes remain.
- Cut Selection: The edges between the two remaining nodes form the minimum cut.
- Randomized: Needs to be run multiple times to increase the chance of finding the actual minimum cut.

```python
import random

def karger_modified_mst(G, iterations=100):
    best_mst = []
    best_cost = float('inf')
    nodes = list(G.nodes())

    for _ in range(iterations):
        edges = list(G.edges(data=True))
        random.shuffle(edges)

        uf = UnionFind(len(nodes))
        node_index = {node: i for i, node in enumerate(nodes)}
        mst = []
        cost = 0

        for u, v, data in edges:
            idx_u, idx_v = node_index[u], node_index[v]
            if uf.union(idx_u, idx_v):
                weight = data['weight']
                mst.append((u, v, weight))
                cost += weight
                if len(mst) == len(nodes) - 1:
                    break

        if len(mst) == len(nodes) - 1 and cost < best_cost:
            best_cost = cost
            best_mst = mst

    return best_mst
```

DATASET ONE: ECO-FLORIDA

- eco-florida dataset is a foodweb nodes where each node represents a species or a group of specices
- the edges represent an interaction relationship between a predator and a prey
- the weights represents the magnitude of the carbon flow passed from the prey to the predator.

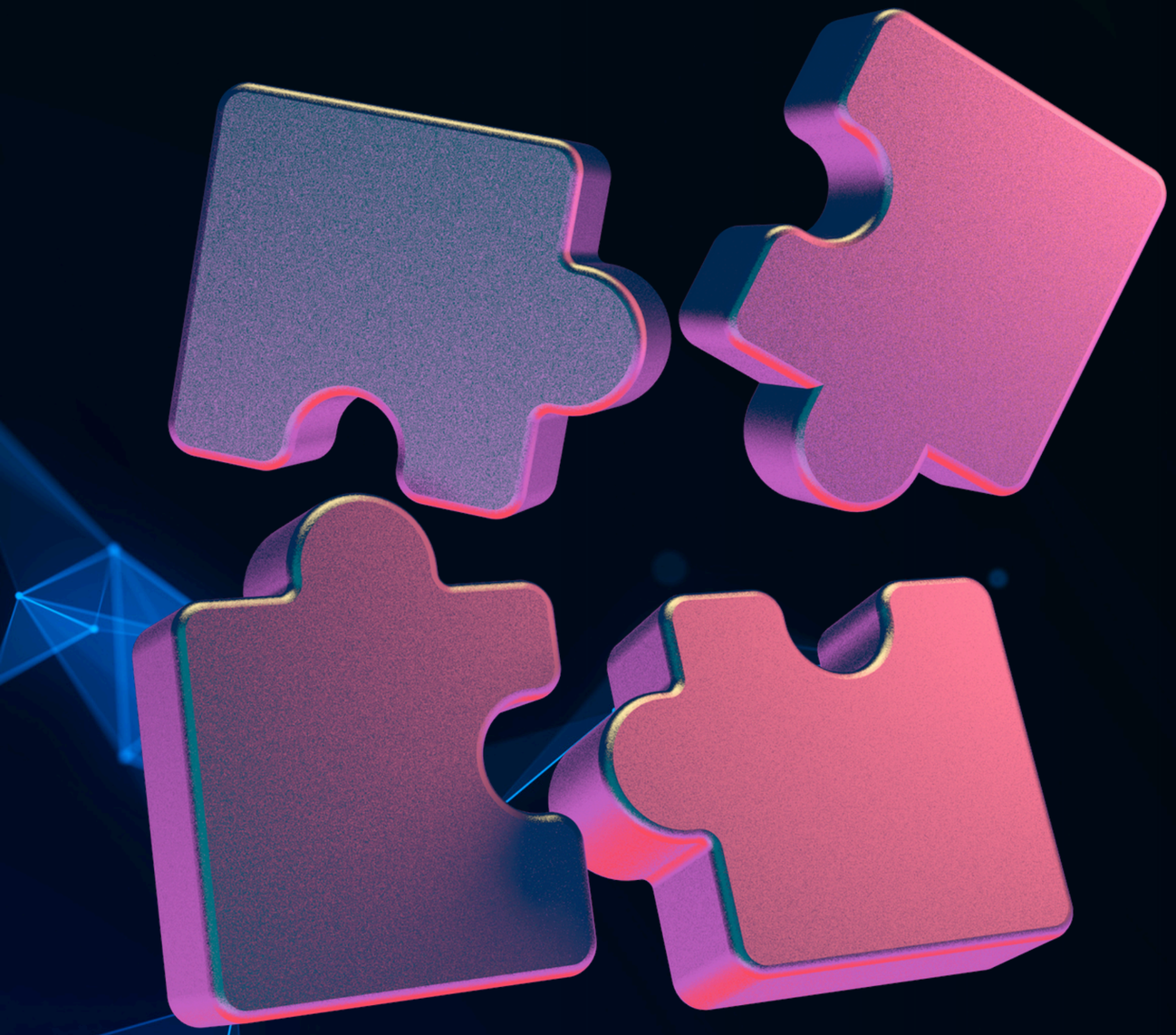


DATASET TWO: G12

- This is a graph with interconnected 800 nodes that doesn’t represent anything real.
- Nodes here act as a sort of grid where each node has connections to its immediate neighbours above, below and to either side of it.
- weights are assigned randomly (1 & -1).
- The graph was created purely for science and doesn’t have any real world meaning

DATASET THREE: DUBCOVA1

-comes from a 2D/3D scientific/engineering simulations (simulating liquid flow, solving partial differential equations, etc)
-nodes represent points/ variables in a 2D/3D space
-edges show whether or not they interact or affect each other
-weights show how much they interact or affect each other
-Dubcova1 specifically comes from a fluid flow/heat transfer problem simulation

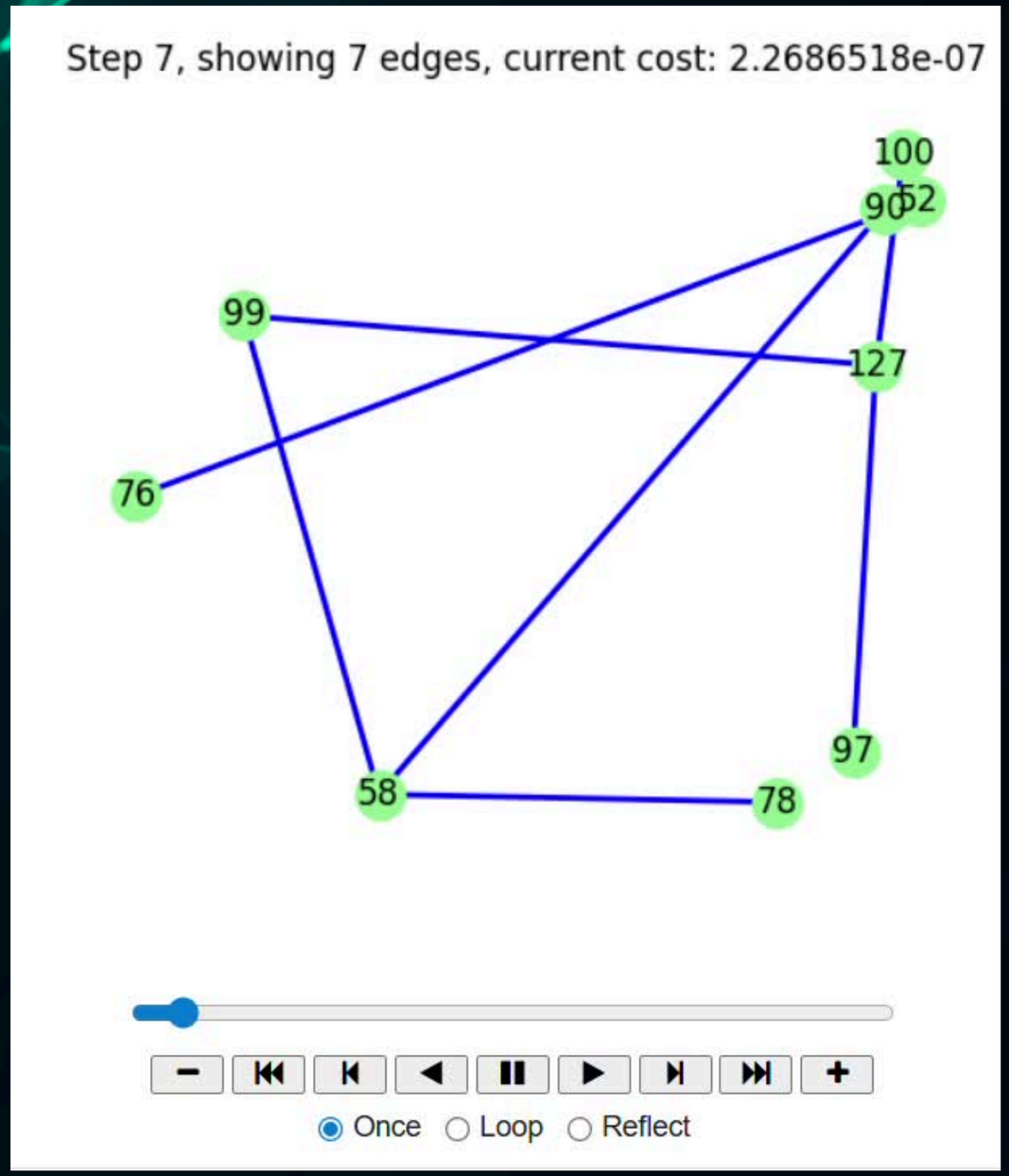


DATASET FOUR: GAASH6

-The $GaAsH_6$ dataset models quantum interactions between atomic orbitals in a Ga-As-H molecule using nodes, edges, and weighted connections.
-It helps scientists design advanced materials like high-efficiency solar cells by simulating electron behavior.

DATASET FIVE: FULL CHIP

-The FullChip dataset represents microchip layouts, with nodes as components (e.g., transistors) and edges as their connections.
-It’s used to optimize chip design—improving speed and efficiency in real devices like smartphones.

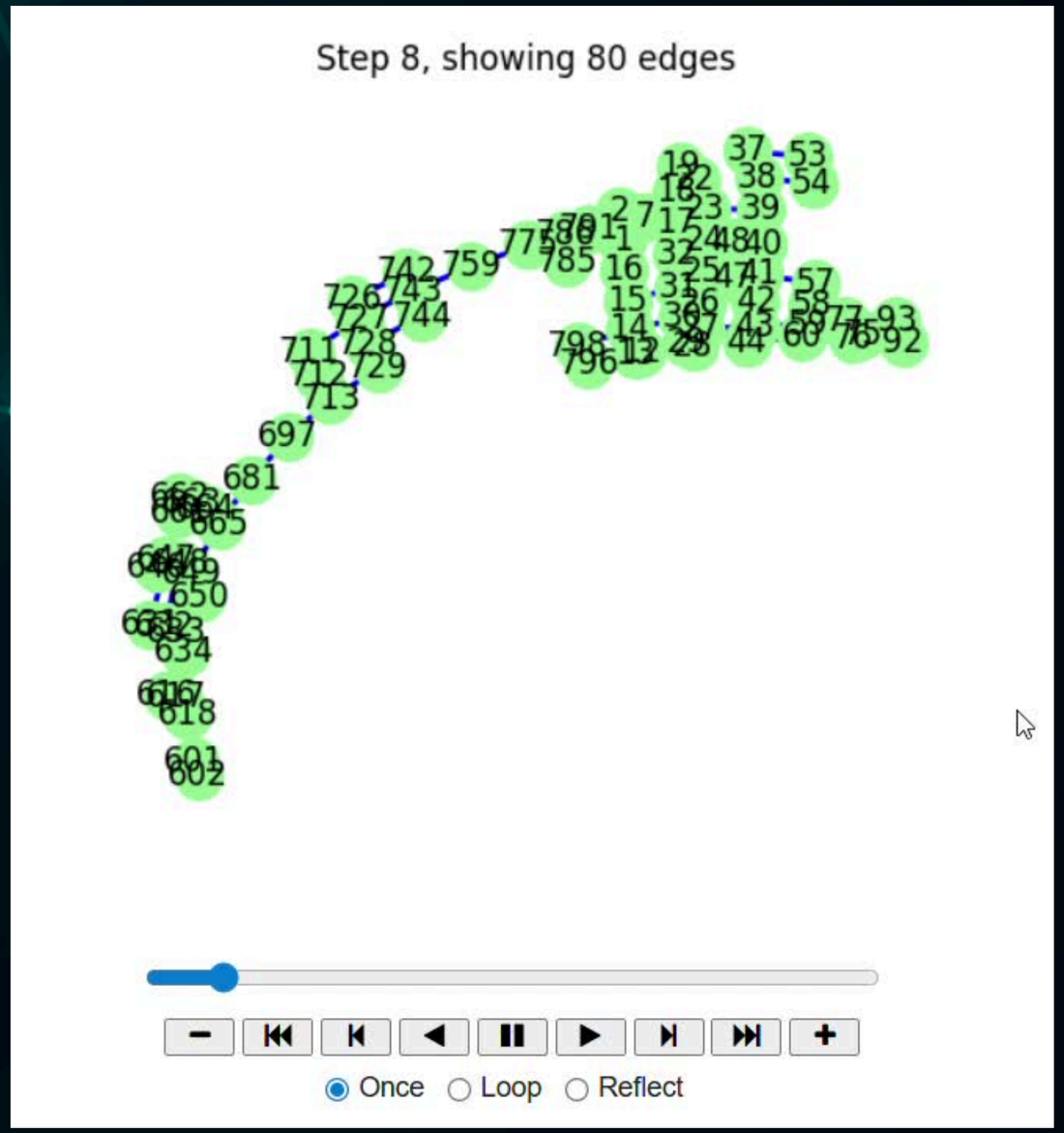


SAMPLE VIDEO

ECO-FLORIDA USING KRUSKAL

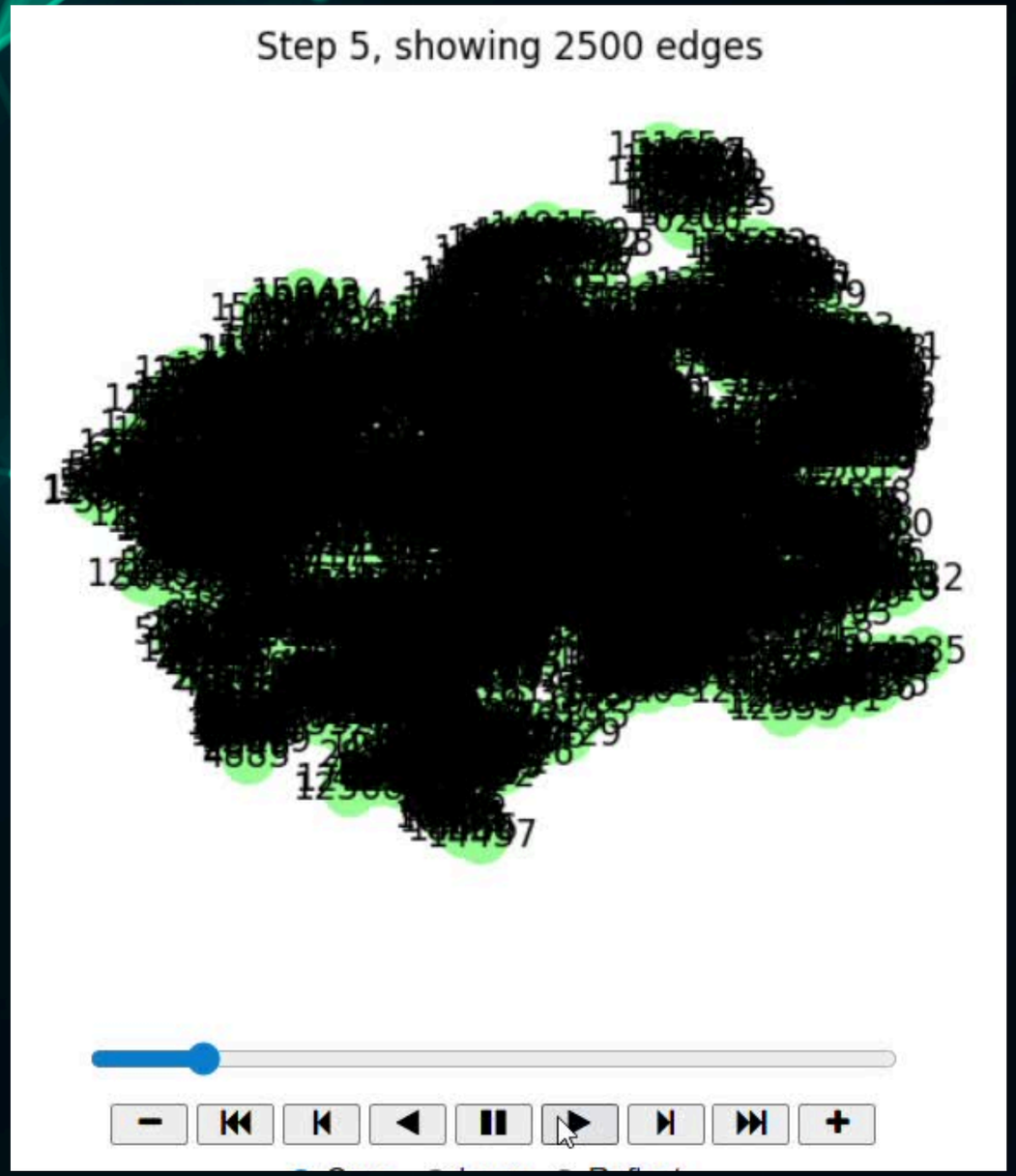
1 edge at a time

all videos in GitHub



SAMPLE VIDEO

G12 USING PRIM'S
10 edges at a time
all videos in GitHub

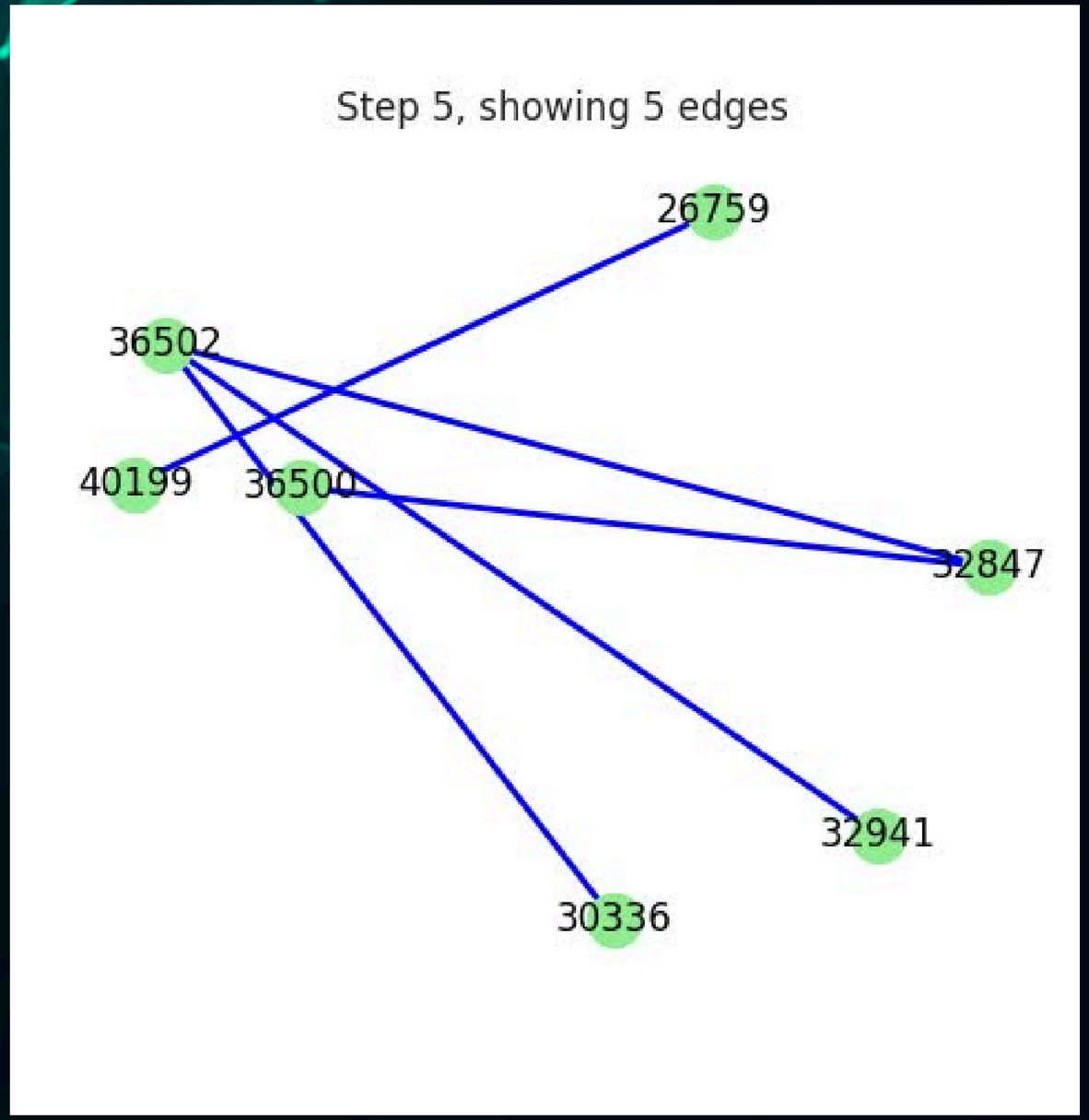


SAMPLE VIDEO

DUBCOVA1 USING BORUVKA

500 edges at a time

all videos in GitHub



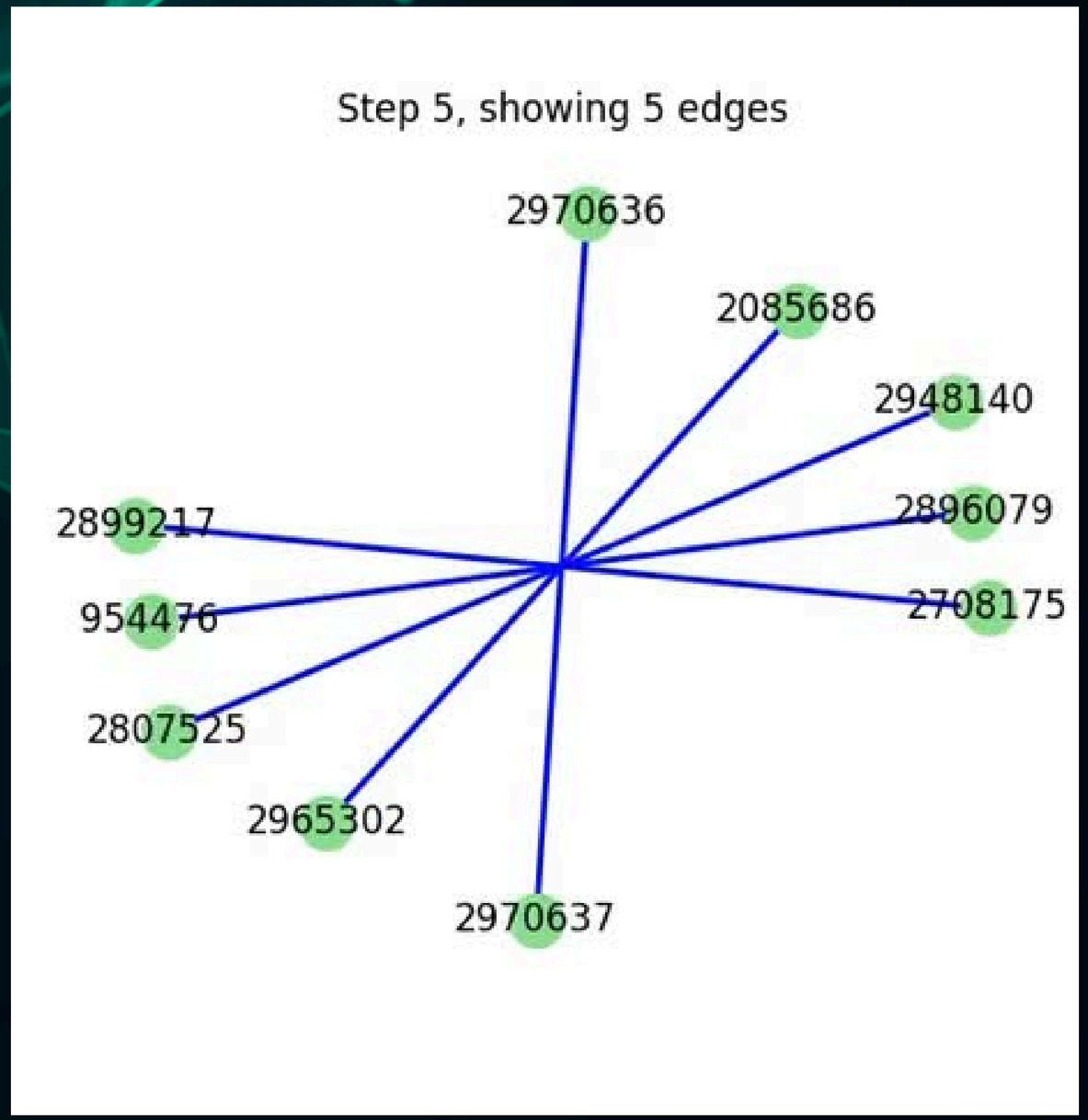
SAMPLE VIDEO

Sample of GaAsH6 USING REVERSE-DELETE

1 edge at a time

all videos in GitHub

MST algorithm applied on whole dataset



SAMPLE VIDEO

Sample of FullChip USING KRUSKAL
1 edge at a time
all videos in GitHub
MST algorithm applied on whole dataset

GITHUB

TIME COMPLEXITY COMPARISONS

✅ Updated Time Complexity Comparison Table

Algorithm	Theoretical Time Complexity	Time Usage Description
Kruskal	O(E log E)	Sorting edges + Union-Find operations
Prim (Standard)	O(E log V)	Uses a priority queue (min-heap); assumes adjacency matrix or basic structure
Prim (Adjacency List)	O(E log V)	Uses adjacency list with min-heap (priority queue); much faster for sparse graphs
Borůvka	O(E log V)	Iteratively merges components using the lightest outgoing edges
Reverse-Delete	$O(E^2)$	Tries removing edges (starting from heaviest) while checking connectivity after each
Karger (Modified)	$O(I \times E \times \alpha(V)) \approx O(I \times E)$	Randomized; I = iterations, α(V) is inverse Ackermann function (nearly constant in practice)

HTML

MST Visualization Tool

Dataset: G12 **Algorithm:** Kruskal **Visualization:** Network Run Algorithm

Total MST Cost: --

Execution Time: --

THANK YOU!

FOR YOUR ATTENTION