Supplementary material for AAAI 2026 — Improving Human-Robot Teamwork in Urban Search and Rescue Through Knowledge Graph-Based Episodic Memory Learning

All the experiments were carried out on a single machine (AMD Ryzen 9 7950X 16-Core Processor with 128GB memory, Ubuntu 22.04, and RTX 4060 Ti with 16GB)

Start AnonymousPackage

AnonymousPackage is a python package for JanusGraph. The original package name is anonymized for blind submission

```
from anonymous_package.utils import disable_logger

disable_logger()

from gremlin_python.structure.graph import Graph, Vertex, Edge
from gremlin_python.driver.driver_remote_connection import
DriverRemoteConnection
from gremlin_python.driver.serializer import GraphSONSerializersV3d0
from gremlin_python.process.graph_traversal import __
from gremlin_python.process.traversal import P, T, Direction

import json
from anonymous_package.janusgraph import AnonymousPackage
from tqdm.auto import tqdm

anonymous_package = AnonymousPackage()
anonymous_package.connect()


from anonymous_package.utils import disable_logger

disable_logger()
```

Write the co-learning data to AnonymousPackage

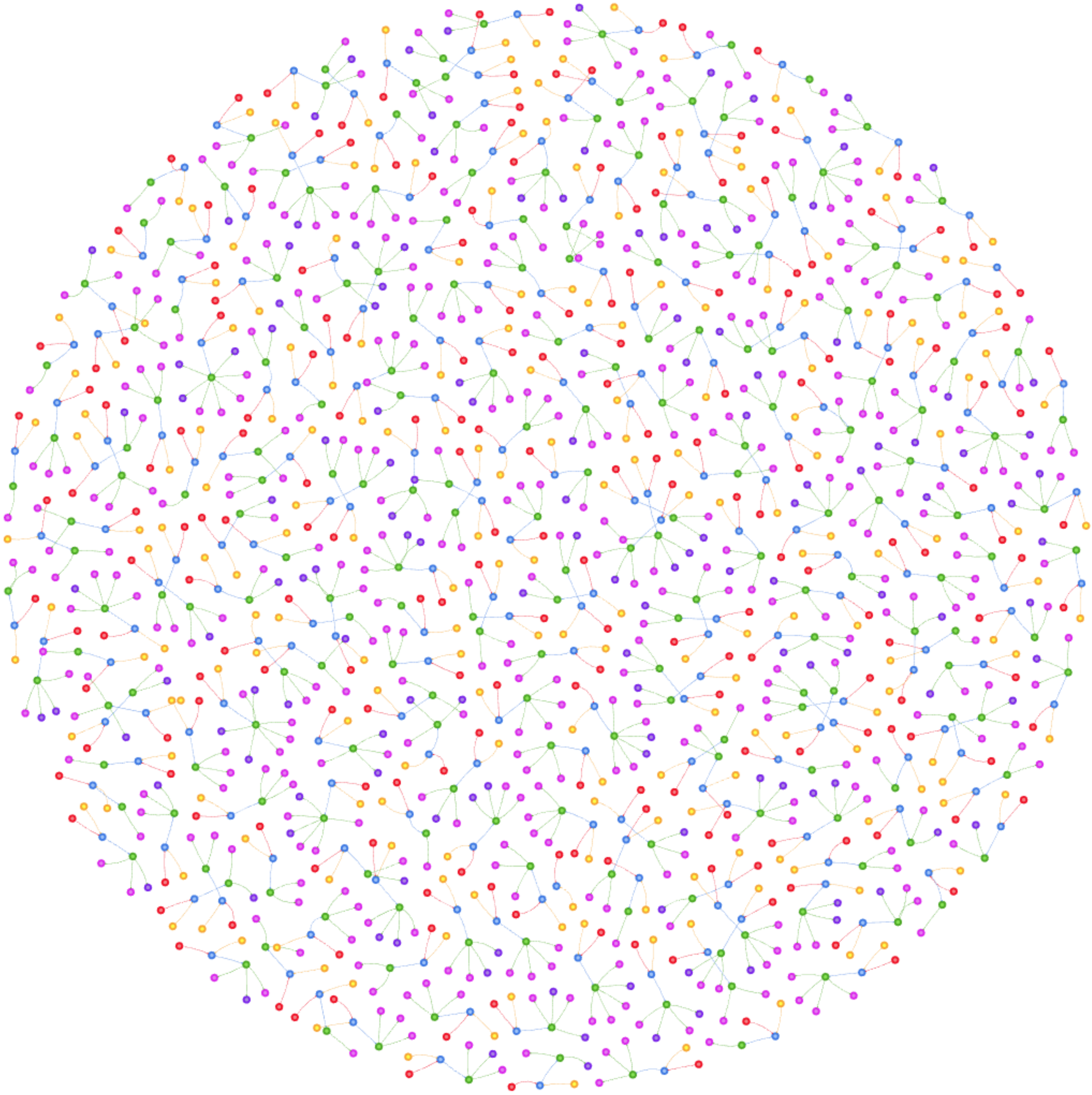
```
with open("./raw-data.json") as f:
    data = json.load(f)


anonymous_package.remove_all_data()

for data_point in tqdm(data):
```

```
    time_added = data_point["timestamp"]

    robot_vertex = anonymous_package.write_long_term_vertex("robot",
{"time_added": time_added})
    cp_properties = {
        "cp_num": data_point["cp_num"],
        "participant_num": data_point["participant"],
        "cp_name": data_point["cp_name"],
        "ticks_lasted": data_point["ticks_lasted"],
        "round_num": data_point["round_num"],
        "time_added": data_point["timestamp"],
        "time_elapsed": data_point["time_elapsed"],
        "remaining_rocks": data_point["remaining_rocks"],
        "victim_harm": data_point["victim_harm"],
        "success": data_point["success"],
    }

    cp_vertex = anonymous_package.write_long_term_vertex("CP",
cp_properties)
    anonymous_package.write_long_term_edge(
        robot_vertex, "has_cp", cp_vertex, {"time_added": time_added}
    )

    participant_vertex = anonymous_package.write_long_term_vertex(
        "participant",
        {"participant_number": data_point["participant"], "time_added":
time_added},
    )
    anonymous_package.write_long_term_edge(
        participant_vertex, "has_cp", cp_vertex, {"time_added": time_added}
    )

    situation = [bar for foo in data_point["situation"] for bar in foo]

    if situation:
        situation_properties = {s["type"]: s["content"] for s in situation}
        situation_properties["time_added"] = time_added
        situation_vertex = anonymous_package.write_long_term_vertex(
            "situation", situation_properties
        )
        anonymous_package.write_long_term_edge(
            cp_vertex, "has_situation", situation_vertex, {"time_added":
time_added}
        )

        for idx, list_ in enumerate(data_point["HumanAction"]):

            if list_:
                properties = {"time_added": time_added}
                for action in list_:
                    properties[action["type"]] = action["content"]
                properties["action_number"] = idx

                human_action_vertex =
```

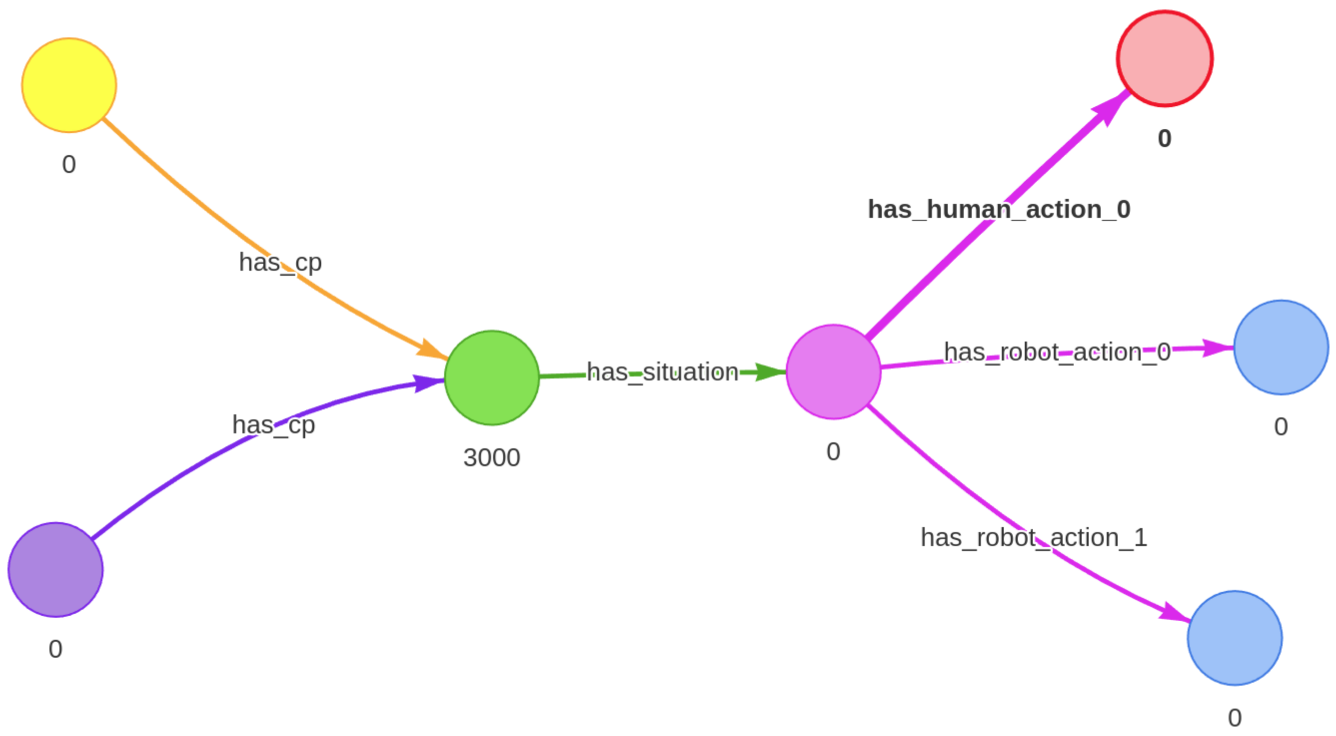
```
anonymous_package.write_long_term_vertex(
                    "human_action", properties
                )
                anonymous_package.write_long_term_edge(
                    situation_vertex,
                    "has_human_action_" + str(idx),
                    human_action_vertex,
                    {"time_added": time_added},
                )

        for idx, list_ in enumerate(data_point["RobotAction"]):

            if list_:
                properties = {"time_added": time_added}
                for action in list_:
                    properties[action["type"]] = action["content"]
                properties["action_number"] = idx

                robot_action_vertex =
anonymous_package.write_long_term_vertex(
                    "robot_action", properties
                )
                anonymous_package.write_long_term_edge(
                    situation_vertex,
                    "has_robot_action_" + str(idx),
                    robot_action_vertex,
                    {"time_added": time_added},
                )
```

```
100%|██████████| 211/211
```

Visualize all 211 collaboration patterns (CPs) as knowledge graphs (KGs)



Visualize an example CP (#36)



Get some stats

```
from gremlin_python.process.traversal import TextP

ticks_lasted_min = (

anonymous_package.g.V().hasLabel("CP").values("ticks_lasted").order().limit
(1).next()
)

ticks_lasted_max = (

anonymous_package.g.V().hasLabel("CP").values("ticks_lasted").order().tail(
1).next()
)

time_elapsed_min = (

anonymous_package.g.V().hasLabel("CP").values("time_elapsed").order().limit
(1).next()
)

time_elapsed_max = (

anonymous_package.g.V().hasLabel("CP").values("time_elapsed").order().tail(
1).next()
)

remaining_rocks_min = (

anonymous_package.g.V().hasLabel("CP").values("remaining_rocks").order().li
```

```
mit(1).next()
)

remaining_rocks_max = (

anonymous_package.g.V().hasLabel("CP").values("remaining_rocks").order().ta
il(1).next()
)

victim_harm_min = (

anonymous_package.g.V().hasLabel("CP").values("victim_harm").order().limit(
1).next()
)

victim_harm_max = (

anonymous_package.g.V().hasLabel("CP").values("victim_harm").order().tail(1
).next()
)

success_min =
anonymous_package.g.V().hasLabel("CP").values("success").order().limit(1).n
ext()

success_max =
anonymous_package.g.V().hasLabel("CP").values("success").order().tail(1).ne
xt()

round_num_min =
anonymous_package.g.V().hasLabel("CP").values("round_num").order().limit(1)
.next()

round_num_max =
anonymous_package.g.V().hasLabel("CP").values("round_num").order().tail(1).
next()

print(f"{'Description':<20}{'Min':<10}{'Max':<10}")
print(f"{'-' * 40}")
print(f"{'Ticks lasted:':<20}{ticks_lasted_min:<10}{ticks_lasted_max:<10}")
print(f"{'Time elapsed:':<20}{time_elapsed_min:<10}{time_elapsed_max:<10}")
print(f"{'Remaining rocks:':<20}{remaining_rocks_min:<10}
{remaining_rocks_max:<10}")
print(f"{'Victim harm:':<20}{victim_harm_min:<10}{victim_harm_max:<10}")
print(f"{'Success:':<20}{success_min:<10}{success_max:<10}")
print(f"{'Round number:':<20}{round_num_min:<10}{round_num_max:<10}")
```

```
Description         Min       Max
----------------------------------------
Ticks lasted:       1         3018
Time elapsed:       887       3000
```

```
Remaining rocks:    0          41
Victim harm:        0          1900
Success:            0          1
Round number:       1          8
```

Write the vector representations of the nodes to the database

```python
import numpy as np
import json
from gremlin_python.process.traversal import P, T, Direction, TextP
from sentence_transformers import SentenceTransformer

# Load a pre-trained Sentence-BERT model
model = SentenceTransformer("all-mpnet-base-v2")


def turn_properties_into_string(properties):
    to_return = ""
    if "action" in properties:
        to_return += "Action: " + properties["action"] + ". "
    if "actor" in properties:
        to_return += "Actor: " + properties["actor"] + ". "
    if "location" in properties:
        to_return += "Location: " + properties["location"] + ". "
    if "object" in properties:
        to_return += "Object: " + properties["object"] + ". "

    return to_return


situation_vertices = anonymous_package.g.V().hasLabel("situation").toList()
for situation_vertex in tqdm(situation_vertices):
    # Get situation node details
    situation_label = situation_vertex.label
    situation_properties =
anonymous_package.get_properties(situation_vertex)
    situation_properties_str =
turn_properties_into_string(situation_properties)
    situation_properties_vector = model.encode(situation_properties_str)
    anonymous_package.update_vertex_properties(
        situation_vertex,
        {
            "sentence_representation": situation_properties_str,
            "vector_representation":
json.dumps(situation_properties_vector.tolist()),
        },
    )

    # Get the connected cp vertex
    cp_vertex = (
```

```
anonymous_package.g.V(situation_vertex).inE().hasLabel("has_situation").out
V().next()
    )
    cp_label = cp_vertex.label
    cp_properties = anonymous_package.get_properties(cp_vertex)
    cp_properties_vector = np.array(
        [
            cp_properties["ticks_lasted"] / ticks_lasted_max,
            cp_properties["time_elapsed"] / time_elapsed_max,
            cp_properties["remaining_rocks"] / remaining_rocks_max,
            cp_properties["victim_harm"] / victim_harm_max,
            cp_properties["success"] / success_max,
            cp_properties["round_num"] / round_num_max,
        ]
    )
    anonymous_package.update_vertex_properties(
        cp_vertex,
        {
            "vector_representation":
json.dumps(cp_properties_vector.tolist()),
        },
    )

    # Get human action vertices
    human_actions = (
        anonymous_package.g.V(situation_vertex)
        .outE()
        .hasLabel(TextP.containing("has_human_action"))
        .inV()
        .toList()
    )
    for human_action in human_actions:
        human_action_properties =
anonymous_package.get_properties(human_action)
        human_action_properties_str = turn_properties_into_string(
            human_action_properties
        )
        human_action_properties_vector =
model.encode(human_action_properties_str)
        anonymous_package.update_vertex_properties(
            human_action,
            {
                "sentence_representation": human_action_properties_str,
                "vector_representation": json.dumps(
                    human_action_properties_vector.tolist()
                ),
            },
        )

    # Get robot action vertices
    robot_actions = (
        anonymous_package.g.V(situation_vertex)
        .outE()
        .hasLabel(TextP.containing("has_robot_action"))
```

```
            .inV()
            .toList()
        )
        for robot_action in robot_actions:
            robot_action_properties =
    anonymous_package.get_properties(robot_action)
            robot_action_properties_str = turn_properties_into_string(
                robot_action_properties
            )
            robot_action_properties_vector =
    model.encode(robot_action_properties_str)
            anonymous_package.update_vertex_properties(
                robot_action,
                {
                    "sentence_representation": robot_action_properties_str,
                    "vector_representation": json.dumps(
                        robot_action_properties_vector.tolist()
                    ),
                },
            )
```

```
100%|██████████| 209/209
```

Get the stuff, e.g., adjacency matrix, from the graphs for GNN

Vertex-Type Classes

Node Type	Class ID
situation	0
cp	1
human_action_0	2
human_action_1	3
human_action_2	4
robot_action_0	5
robot_action_1	6
robot_action_2	7
robot_action_3	8
robot_action_4	9

Edge-Type classes

Vertex Class	Class IDs

Vertex Class	Class IDs
has_situation	0
has_human_action_0	1
has_human_action_1	2
has_human_action_2	3
has_robot_action_0	4
has_robot_action_1	5
has_robot_action_2	6
has_robot_action_3	7
has_robot_action_4	8

```
import torch
import torch.nn.functional as F
from torch.nn import Linear
from torch_geometric.nn import RGCNConv
from torch_geometric.data import Data, DataLoader
from torch_geometric.utils import add_self_loops
import random

num_node_types = 10
small_dim = 6
big_dim = 768
num_original_edge_types = 9
relation_mode = ["default", "include_self_loop",
"include_self_loop_and_inv"][2]
if relation_mode == "default":
    num_edge_types = num_original_edge_types
elif relation_mode == "include_self_loop":
    num_edge_types = num_original_edge_types + 1
elif relation_mode == "include_self_loop_and_inv":
    num_edge_types = 2 * num_original_edge_types + 1

situation_vertices = anonymous_package.g.V().hasLabel("situation").toList()

dataset = []
for situation_vertex in tqdm(situation_vertices):

    vertices, edges = anonymous_package.get_within_hops([situation_vertex],
1)

    node_features = torch.zeros(len(vertices), big_dim)
    edge_index = []
    node_classes = []
    edge_type = []
    is_small_dim = []
```

```
    # Extract labels
    vertex_labels = [vertex.label for vertex in vertices]
    vertex_properties = [anonymous_package.get_properties(vertex) for
vertex in vertices]
    inV_labels = [edge.inV.label for edge in edges]
    outV_labels = [edge.outV.label for edge in edges]
    edge_labels = [edge.label for edge in edges]

    # Compute max widths
    max_v_len = max(len(lbl) for lbl in vertex_labels) if vertex_labels
else 0
    max_inV_len = max(len(lbl) for lbl in inV_labels) if inV_labels else 0
    max_edge_len = max(len(lbl) for lbl in edge_labels) if edge_labels else
0
    max_outV_len = max(len(lbl) for lbl in outV_labels) if outV_labels else
0

    # Print vertex labels
    # print("Vertices:")
    for idx, (v, p) in enumerate(zip(vertex_labels, vertex_properties)):
        if v == "situation":
            node_classes.append(0)
            is_small_dim.append(0)
        elif v == "CP":
            node_classes.append(1)
            is_small_dim.append(1)
        elif v == "human_action" and p["action_number"] == 0:
            node_classes.append(2)
            is_small_dim.append(0)
        elif v == "human_action" and p["action_number"] == 1:
            node_classes.append(3)
            is_small_dim.append(0)
        elif v == "human_action" and p["action_number"] == 2:
            node_classes.append(4)
            is_small_dim.append(0)
        elif v == "robot_action" and p["action_number"] == 0:
            node_classes.append(5)
            is_small_dim.append(0)
        elif v == "robot_action" and p["action_number"] == 1:
            node_classes.append(6)
            is_small_dim.append(0)
        elif v == "robot_action" and p["action_number"] == 2:
            node_classes.append(7)
            is_small_dim.append(0)
        elif v == "robot_action" and p["action_number"] == 3:
            node_classes.append(8)
            is_small_dim.append(0)
        elif v == "robot_action" and p["action_number"] == 4:
            node_classes.append(9)
            is_small_dim.append(0)
        else:
            raise ValueError(f"Unknown vertex label: {v}")
```

```
        feats = torch.tensor(
            json.loads(anonymous_package.get_properties(vertices[idx])
["vector_representation"])
        )
        if feats.shape[0] == big_dim:
            node_features[idx] = feats
        elif feats.shape[0] == small_dim:
            node_features[idx, :small_dim] = feats
        else:
            raise ValueError(f"Unknown feature shape: {feats.shape}")

    is_small_dim = torch.tensor(is_small_dim)
    node_classes = torch.tensor(node_classes)

    for edge in edges:
        if "has_situation" in edge.label:
            edge_type.append(0)
        elif "has_human_action_0" in edge.label:
            edge_type.append(1)
        elif "has_human_action_1" in edge.label:
            edge_type.append(2)
        elif "has_human_action_2" in edge.label:
            edge_type.append(3)
        elif "has_robot_action_0" in edge.label:
            edge_type.append(4)
        elif "has_robot_action_1" in edge.label:
            edge_type.append(5)
        elif "has_robot_action_2" in edge.label:
            edge_type.append(6)
        elif "has_robot_action_3" in edge.label:
            edge_type.append(7)
        elif "has_robot_action_4" in edge.label:
            edge_type.append(8)
        else:
            raise ValueError(f"Unknown edge label: {edge.label}")

        edge_index.append(
            [
                vertex_labels.index(edge.outV.label),
                vertex_labels.index(edge.inV.label),
            ]
        )
    edge_index = torch.tensor(edge_index).T

    edge_type = torch.tensor(edge_type)

    if relation_mode == "default":
        pass
    elif relation_mode == "include_self_loop":
        edge_index, edge_type = add_self_loops(
            edge_index, edge_type, fill_value=num_original_edge_types
        )
    elif relation_mode == "include_self_loop_and_inv":
        # Add inverse relations
```

```
        src, dst = edge_index
        inv_edge_index = torch.stack([dst, src], dim=0)
        inv_edge_type = edge_type + num_original_edge_types
        edge_index = torch.cat([edge_index, inv_edge_index], dim=1)
        edge_type = torch.cat([edge_type, inv_edge_type], dim=0)

        # Add self-loops with new relation type = 2N
        edge_index, edge_type = add_self_loops(
            edge_index, edge_type, fill_value=(2 * num_original_edge_types)
        )

    data = Data(
        node_features=node_features,
        edge_index=edge_index,
        node_classes=node_classes,
        edge_type=edge_type,
        is_small_dim=is_small_dim,
    )
    dataset.append(data)
```

Visualize the vectors (before training)

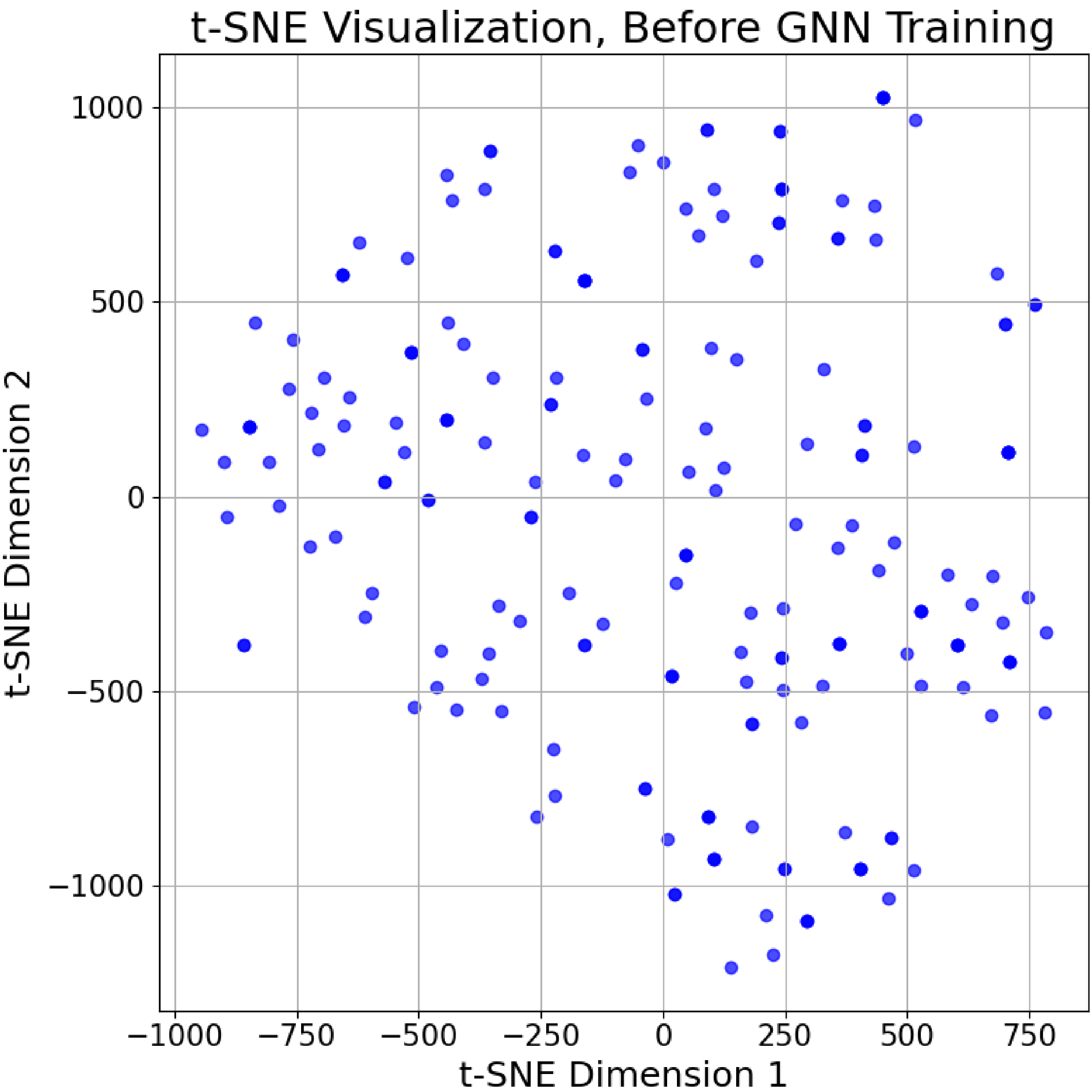
```
import numpy as np
import matplotlib.pyplot as plt
from sklearn.manifold import TSNE


node_features = []
for data_sample in tqdm(dataset):
    node_features_ = np.array(
        [
            node_features_
            for node_features_, is_small_dim_ in zip(
                data_sample.node_features, data_sample.is_small_dim
            )
            if is_small_dim_ == 0
        ]
    ).mean(axis=0)
    node_features.append(node_features_)
node_features = np.array(node_features)

# Apply t-SNE to reduce dimensions to 2D for visualization
tsne = TSNE(
    n_components=2,
    perplexity=30,  # Good balance for most datasets
    n_iter=2000,  # Enough for convergence
    learning_rate=200,  # Default, balanced value
    metric="euclidean",  # Use cosine for high-dimensional data
    init="pca",  # Stable initialization
    random_state=42,  # Reproducible results
)
```

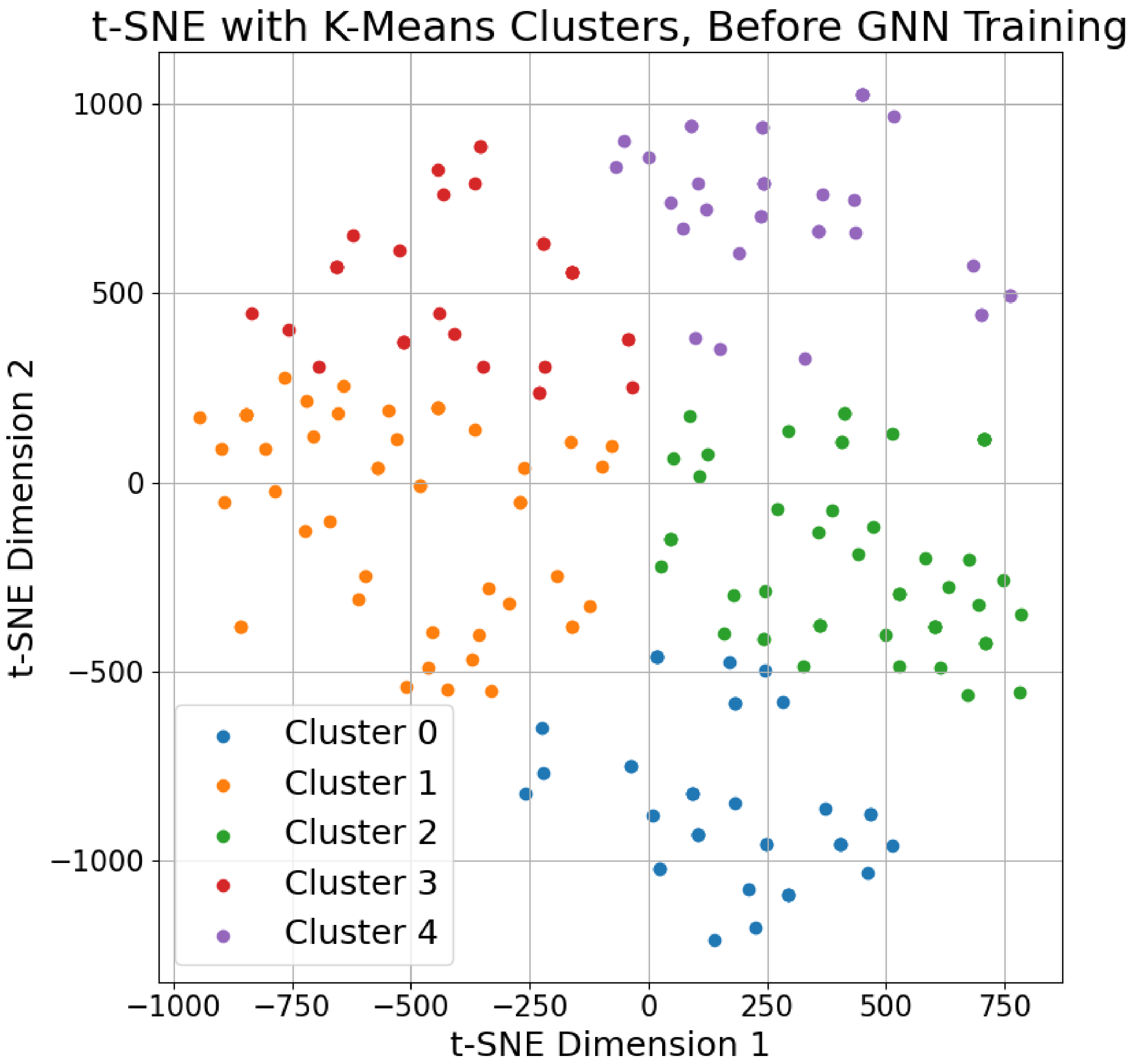
```
vectors_2d = tsne.fit_transform(node_features)

# Plot the 2D visualization
plt.figure(figsize=(8, 8))
plt.scatter(vectors_2d[:, 0], vectors_2d[:, 1], c="blue", alpha=0.7)
plt.title("t-SNE Visualization, Before GNN Training", fontsize=22)
plt.xlabel("t-SNE Dimension 1", fontsize=18)
plt.ylabel("t-SNE Dimension 2", fontsize=18)
plt.grid()

# Change the fontsize for the ticks
plt.tick_params(axis='both', which='major', labelsize=15)  # For major
ticks
plt.tick_params(axis='both', which='minor', labelsize=12)  # For minor
ticks (optional)

# Save and show the plot
plt.tight_layout()
plt.savefig("./figures/t-SNE Visualization, Before GNN Training.png")
plt.savefig("./figures/t-SNE Visualization, Before GNN Training.pdf")
plt.show()

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Assume `tsne_results` is the 2D output of t-SNE (shape: [num_points, 2])
# Number of clusters to form
num_clusters = 5

# Apply K-Means clustering
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(vectors_2d)

# Visualize the clusters
plt.figure(figsize=(8, 8))

for cluster in range(num_clusters):
    plt.scatter(
        vectors_2d[cluster_labels == cluster, 0],
        vectors_2d[cluster_labels == cluster, 1],
        label=f"Cluster {cluster}",
    )
plt.legend(fontsize=18, loc="best")
plt.grid()
plt.title("t-SNE with K-Means Clusters, Before GNN Training", fontsize=22)
plt.xlabel("t-SNE Dimension 1", fontsize=18)
plt.ylabel("t-SNE Dimension 2", fontsize=18)

# Change the fontsize for the ticks
plt.tick_params(axis='both', which='major', labelsize=15)  # For major
ticks
plt.tick_params(axis='both', which='minor', labelsize=12)  # For minor
ticks (optional)
```

```
# Save and show the plot
plt.tight_layout()
plt.savefig("./figures/t-SNE with K-Means Clusters, Before GNN
Training.png")
plt.savefig("./figures/t-SNE with K-Means Clusters, Before GNN
Training.pdf")
plt.show()

from collections import Counter

cluster_dict = dict(Counter(cluster_labels))

ticks_lasted = []
time_elapsed = []
remaining_rocks = []
victim_harm = []
success = []
round_num = []

for vertex in situation_vertices:
    cp_vertex =
anonymous_package.g.V(vertex).inE("has_situation").outV().next()
    properties = anonymous_package.get_properties(cp_vertex)
    ticks_lasted.append(properties["ticks_lasted"])
    time_elapsed.append(properties["time_elapsed"])
    remaining_rocks.append(properties["remaining_rocks"])
    victim_harm.append(properties["victim_harm"])
    success.append(properties["success"])
    round_num.append(properties["round_num"])

ticks_lasted = np.array(ticks_lasted)
time_elapsed = np.array(time_elapsed)
remaining_rocks = np.array(remaining_rocks)
victim_harm = np.array(victim_harm)
success = np.array(success)
round_num = np.array(round_num)

for label, val in cluster_dict.items():
    print(f"Cluster {label}: (count: {val})")
    print(f"Ticks lasted: {int(ticks_lasted[cluster_labels ==
label].mean())}")
    print(f"Time elapsed: {int(time_elapsed[cluster_labels ==
label].mean())}")
    print(f"Remaining rocks: {int(remaining_rocks[cluster_labels ==
label].mean())}")
    print(f"Victim harm: {int(victim_harm[cluster_labels ==
label].mean())}")
    print(f"Success: {success[cluster_labels == label].mean()}")
    print(f"Round num: {int(round_num[cluster_labels == label].mean())}")
    print()

import numpy as np
```

```
# Get the cluster centroids from K-Means
centroids = kmeans.cluster_centers_

# Find the closest data point to each centroid
closest_points = []

for cluster_id in range(num_clusters):
    # Get the indices of points in the current cluster
    cluster_points = vectors_2d[cluster_labels == cluster_id]
    original_indices = np.where(cluster_labels == cluster_id)[0]

    # Compute the distances to the centroid
    distances = np.linalg.norm(cluster_points - centroids[cluster_id], axis=1)

    # Find the index of the closest point
    closest_point_idx = original_indices[np.argmin(distances)]
    closest_points.append(closest_point_idx)
```

```
100%|██████████| 209/209
```


t-SNE Visualization, Before GNN Training
t-SNE Dimension 2
1000
500
0
−500
−1000
t-SNE Dimension 1
−1000
−750
−500
−250
0
250
500
750




```
Cluster 2: (count: 54)
Ticks lasted: 625
Time elapsed: 2309
Remaining rocks: 11
Victim harm: 362
Success: 0.3888888888888889
Round num: 5

Cluster 1: (count: 49)
Ticks lasted: 1174
Time elapsed: 2810
Remaining rocks: 17
Victim harm: 271
Success: 0.2857142857142857
Round num: 4

Cluster 4: (count: 35)
Ticks lasted: 1086
```

```
Time elapsed: 2568
Remaining rocks: 14
Victim harm: 291
Success: 0.2857142857142857
Round num: 5

Cluster 0: (count: 39)
Ticks lasted: 1134
Time elapsed: 2671
Remaining rocks: 13
Victim harm: 305
Success: 0.23076923076923078
Round num: 5

Cluster 3: (count: 32)
Ticks lasted: 737
Time elapsed: 2686
Remaining rocks: 20
Victim harm: 284
Success: 0.09375
Round num: 5
```

Define classes and functions for training RGCN

```
import torch
import torch.nn.functional as F
from torch.nn import Linear
from torch_geometric.nn import RGCNConv
from torch_geometric.data import Data, DataLoader
from torch_geometric.utils import add_self_loops
import random


class RGCN(torch.nn.Module):
    """
    RGCN model for node classification.

    Attributes:
        small_dim (int): The dimension of small-dimensional node features.
        big_dim (int): The dimension of big-dimensional node features.
        num_hidden_channels (int): The number of hidden channels in the
RGCN layers.
        num_edge_types (int): The number of edge types in the RGCN model.
        num_node_types (int): The number of node types in the RGCN model.

    Methods:
        forward(node_features, edge_index, edge_type, is_small_dim):
Forward pass of the model.

    Example:
        model = RGCN(small_dim=5, big_dim=768, num_hidden_channels=64,
```

```
num_edge_types=9, num_node_types=10)
        output = model(node_features, edge_index, edge_type, is_small_dim)

        # node_features is [num_nodes, small_dim or big_dim]
        # is_small_dim is a boolean mask: True where node features were
originally M-dim

        # output is [num_nodes, num_node_types]

    """

    def __init__(
        self, small_dim, big_dim, num_hidden_channels, num_edge_types,
num_node_types
    ):
        """
        Initialize the RGCN model.

        Args:
            small_dim (int): The dimension of small-dimensional node
features.
            big_dim (int): The dimension of big-dimensional node features.
            num_hidden_channels (int): The number of hidden channels in the
RGCN layers.
            num_edge_types (int): The number of edge types in the RGCN
model.
            num_node_types (int): The number of node types in the RGCN
model.

        """

        super().__init__()
        # This MLP will be used to upscale small_dim features to big_dim
        self.small_dim = small_dim
        self.big_dim = big_dim
        self.upscale = Linear(small_dim, big_dim)

        # After upscaling, all features are big_dim
        self.conv1 = RGCNConv(big_dim, num_hidden_channels, num_edge_types)
        self.conv2 = RGCNConv(num_hidden_channels, num_hidden_channels,
num_edge_types)
        self.fc = Linear(num_hidden_channels, num_node_types)

    def forward(self, node_features, edge_index, edge_type, is_small_dim):
        """
        Forward pass of the RGCN model.

        Args:
            node_features (torch.Tensor): The node features tensor.
            edge_index (torch.Tensor): The edge index tensor.
            edge_type (torch.Tensor): The edge type tensor.
            is_small_dim (torch.Tensor): The boolean mask tensor.

        Returns:
```

```
            torch.Tensor: The output tensor.
        """
        # node_features is [num_nodes, small_dim or big_dim]
        # is_m_dim is a boolean mask: True where node features were
originally M-dim

        # Upscale only those nodes that are small_dim
        # Extract small_dim node features (only first small_dim entries are
relevant)

        # shape: [num_small_dim_nodes, small_dim]
        small_dim_nodes = node_features[is_small_dim, : self.small_dim]

        # shape: [num_small_dim_nodes, big_dim]
        small_dim_nodes_upscaled = self.upscale(small_dim_nodes)

        # Replace the small_dim node rows in node_features with the
upscaled features
        node_features_new = node_features.clone()
        node_features_new[is_small_dim] = small_dim_nodes_upscaled
        node_features = node_features_new

        # Now all nodes are effectively big_dim
        self.node_features_1 = self.conv1(node_features, edge_index,
edge_type)
        self.node_features_1_relu = F.relu(self.node_features_1)
        self.node_features_2 = self.conv2(
            self.node_features_1_relu, edge_index, edge_type
        )
        self.output = self.fc(self.node_features_2)

        return self.output


def count_parameters(model):
    """
    Count the number of trainable parameters in a model.

    Args:
        model (torch.nn.Module): The model to count parameters for.

    Returns:
        int: The number of trainable parameters in the model.
    """
    return sum(p.numel() for p in model.parameters() if p.requires_grad)


def train_rgcn(
    model,
    dataset,
    num_edge_types,
    num_hidden_channels,
    batch_size,
    num_epochs,
```

```
    small_dim,
    big_dim,
    device,
    num_node_types,
) -> dict:
    """
    Train an RGCN model on the given dataset.

    Args:
        model (torch.nn.Module): The RGCN model to train.
        dataset (torch_geometric.data.Data): The dataset to use for
training.
        num_edge_types (int): The number of edge types in the dataset.
        num_hidden_channels (int): The number of hidden channels in the
model.
        batch_size (int): The batch size to use for training.
        num_epochs (int): The number of epochs to train the model for.
        small_dim (int): The dimension of small-dimensional node features.
        big_dim (int): The dimension of big-dimensional node features.
        device (torch.device): The device to use for training the model.
        num_node_types (int): The number of node types in the dataset.

    Returns:
        dict: A dictionary containing the training loss and accuracy for
each epoch.
    """
    stats = {"loss": [], "accuracy": []}
    loader = DataLoader(dataset, batch_size=batch_size, shuffle=True)

    optimizer = torch.optim.Adam(model.parameters())
    loss_fn = torch.nn.CrossEntropyLoss()

    for epoch in range(num_epochs):
        model.train()
        total_loss = 0
        correct_predictions = 0
        total_samples = 0

        for batch in loader:
            batch = batch.to(device)
            optimizer.zero_grad()
            out = model(
                batch.node_features,
                batch.edge_index,
                batch.edge_type,
                batch.is_small_dim,
            )

            # Compute loss
            loss = loss_fn(out, batch.node_classes)
            loss.backward()
            optimizer.step()

            total_loss += loss.item()
```

```
            # Compute accuracy
            _, predicted = torch.max(out, dim=1)  # Get predicted class
indices
            correct_predictions += (predicted ==
batch.node_classes).sum().item()
            total_samples += batch.node_classes.size(0)

        # Calculate average loss and accuracy for the epoch
        avg_loss = total_loss / len(loader)
        accuracy = correct_predictions / total_samples

        if epoch % 99 == 0:
            print(
                f"Mode {relation_mode}, Epoch {epoch+1}, Loss:
{avg_loss:.4f}, Accuracy: {accuracy:.4f}"
            )

        stats["loss"].append(avg_loss)
        stats["accuracy"].append(accuracy)

    return stats
```

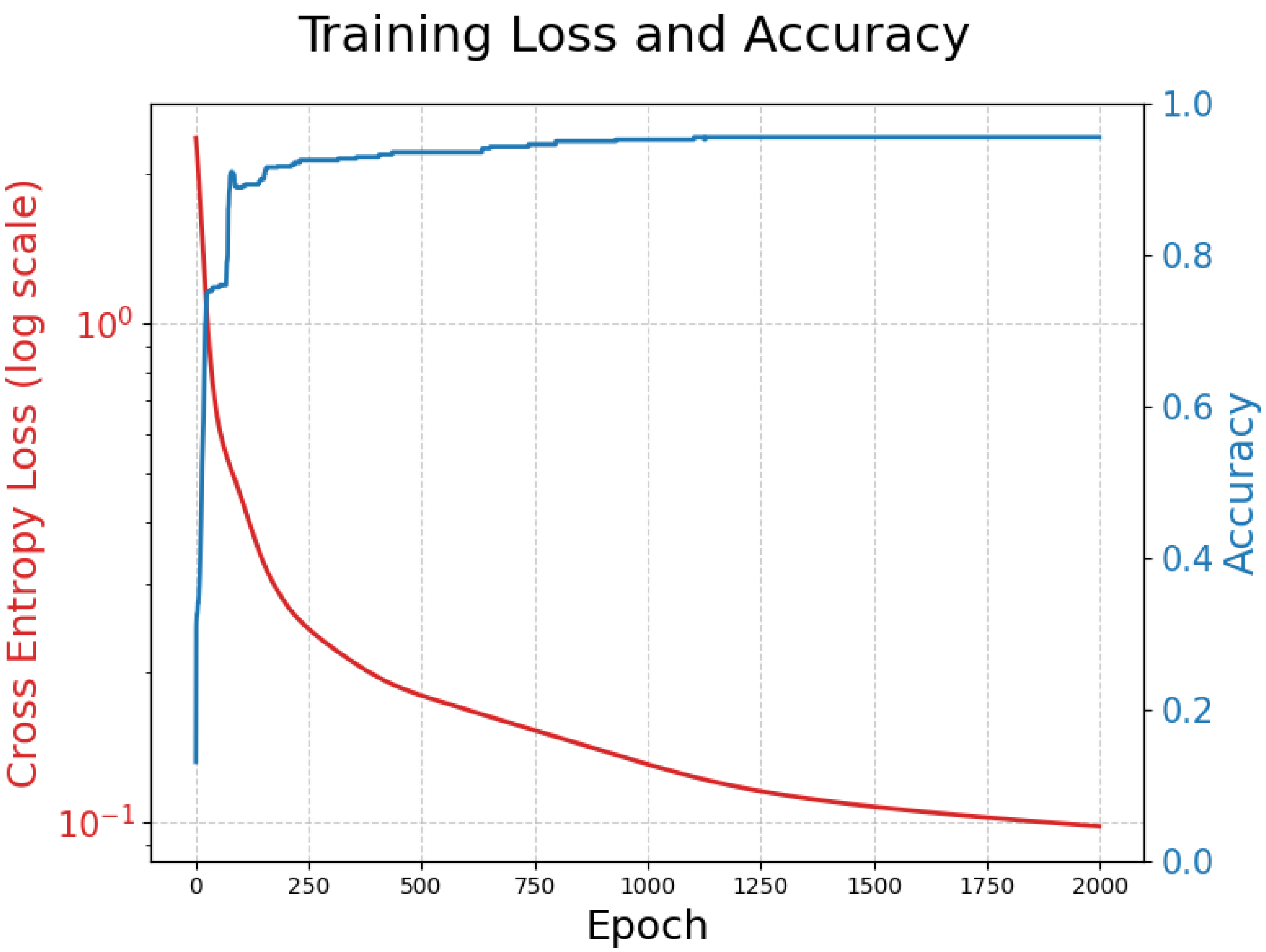
Train!

```
import matplotlib.pyplot as plt

num_epochs = 2000
batch_size = len(dataset)
num_hidden_channels = 8
device = "cuda"
print(f"Training with {relation_mode} relations:")

model = RGCN(
    small_dim=small_dim,
    big_dim=big_dim,
    num_hidden_channels=num_hidden_channels,
    num_edge_types=num_edge_types,
    num_node_types=num_node_types,
).to(device)
print(f"Number of parameters: {count_parameters(model)}")

stats = train_rgcn(
    model=model,
    dataset=dataset,
    num_edge_types=num_edge_types,
    num_hidden_channels=num_hidden_channels,
    batch_size=batch_size,
    num_epochs=num_epochs,
    small_dim=small_dim,
    big_dim=big_dim,
```

```
    device=device,
    num_node_types=num_node_types,
)

# Create a single figure with two y-axes
fig, ax1 = plt.subplots(figsize=(8, 6))

# Plot Loss
ax1.plot(stats["loss"], label="Loss", color="tab:red", linewidth=2)
ax1.set_xlabel("Epoch", fontsize=18)
ax1.set_ylabel("Cross Entropy Loss (log scale)", fontsize=18,
color="tab:red")
ax1.set_yscale("log")
ax1.tick_params(axis="y", which="major", labelsize=15,
labelcolor="tab:red")
ax1.tick_params(axis="y", which="minor", labelsize=12,
labelcolor="tab:red")
ax1.grid(True, linestyle="--", alpha=0.6)

# Create a secondary y-axis for Accuracy
ax2 = ax1.twinx()
ax2.plot(stats["accuracy"], label="Accuracy", color="tab:blue",
linewidth=2)
ax2.set_ylabel("Accuracy", fontsize=18, color="tab:blue")
ax2.set_ylim(0, 1)
ax2.tick_params(axis="y", which="major", labelsize=15,
labelcolor="tab:blue")
ax2.tick_params(axis="y", which="minor", labelsize=12,
labelcolor="tab:blue")

# Title and Legends
fig.suptitle("Training Loss and Accuracy", fontsize=22)
fig.tight_layout()

# Save the figure
plt.savefig("./figures/Training_Loss_and_Accuracy.png", dpi=300)
plt.savefig("./figures/Training_Loss_and_Accuracy.pdf")

# Show the plot
plt.show()
```

```
Training with include_self_loop_and_inv relations:
Number of parameters: 129642
Mode include_self_loop_and_inv, Epoch 1, Loss: 2.3562, Accuracy: 0.1307


Mode include_self_loop_and_inv, Epoch 100, Loss: 0.4520, Accuracy: 0.8890
Mode include_self_loop_and_inv, Epoch 199, Loss: 0.2752, Accuracy: 0.9170
Mode include_self_loop_and_inv, Epoch 298, Loss: 0.2256, Accuracy: 0.9253
Mode include_self_loop_and_inv, Epoch 397, Loss: 0.1968, Accuracy: 0.9295
Mode include_self_loop_and_inv, Epoch 496, Loss: 0.1800, Accuracy: 0.9357
```

```
Mode include_self_loop_and_inv, Epoch 595, Loss: 0.1688, Accuracy: 0.9357
Mode include_self_loop_and_inv, Epoch 694, Loss: 0.1584, Accuracy: 0.9429
Mode include_self_loop_and_inv, Epoch 793, Loss: 0.1487, Accuracy: 0.9461
Mode include_self_loop_and_inv, Epoch 892, Loss: 0.1398, Accuracy: 0.9502
Mode include_self_loop_and_inv, Epoch 991, Loss: 0.1314, Accuracy: 0.9523
Mode include_self_loop_and_inv, Epoch 1090, Loss: 0.1240, Accuracy: 0.9523
Mode include_self_loop_and_inv, Epoch 1189, Loss: 0.1183, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1288, Loss: 0.1139, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1387, Loss: 0.1105, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1486, Loss: 0.1077, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1585, Loss: 0.1054, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1684, Loss: 0.1034, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1783, Loss: 0.1016, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1882, Loss: 0.0999, Accuracy: 0.9554
Mode include_self_loop_and_inv, Epoch 1981, Loss: 0.0984, Accuracy: 0.9554
```



Visualize again (after training)

Represent every graph as one vector by the mean of the node_features

```
node_features = []
for data_sample in tqdm(dataset):
    with torch.no_grad():
```

```
        prediction = model(
            data_sample.node_features.to(device),
            data_sample.edge_index.to(device),
            data_sample.edge_type.to(device),
            data_sample.is_small_dim.to(device),
        )
    node_features.append(model.node_features_2.mean(dim=0).clone())

node_features = torch.stack(node_features).detach().cpu().numpy()

# Apply t-SNE to reduce dimensions to 2D for visualization
tsne = TSNE(
    n_components=2,
    perplexity=30,  # Good balance for most datasets
    n_iter=2000,  # Enough for convergence
    learning_rate=200,  # Default, balanced value
    metric="euclidean",  # Use cosine for high-dimensional data
    init="pca",  # Stable initialization
    random_state=42,  # Reproducible results
)

vectors_2d = tsne.fit_transform(node_features)

# Plot the 2D visualization
plt.figure(figsize=(8, 8))

plt.scatter(vectors_2d[:, 0], vectors_2d[:, 1], c="blue", alpha=0.7)
plt.title("t-SNE Visualization of Vectors, After GNN Training",
fontsize=22)
plt.xlabel("t-SNE Dimension 1", fontsize=18)
plt.ylabel("t-SNE Dimension 2", fontsize=18)
plt.grid()

# Change the fontsize for the ticks
plt.tick_params(axis="both", which="major", labelsize=15)  # For major
ticks
plt.tick_params(axis="both", which="minor", labelsize=12)  # For minor
ticks (optional)

plt.tight_layout()
plt.savefig("./figures/t-SNE Visualization of Vectors, After GNN
Training.png")
plt.savefig("./figures/t-SNE Visualization of Vectors, After GNN
Training.pdf")
plt.show()

from sklearn.cluster import KMeans
import matplotlib.pyplot as plt

# Assume `tsne_results` is the 2D output of t-SNE (shape: [num_points, 2])
# Number of clusters to form
num_clusters = 5

# Apply K-Means clustering
```

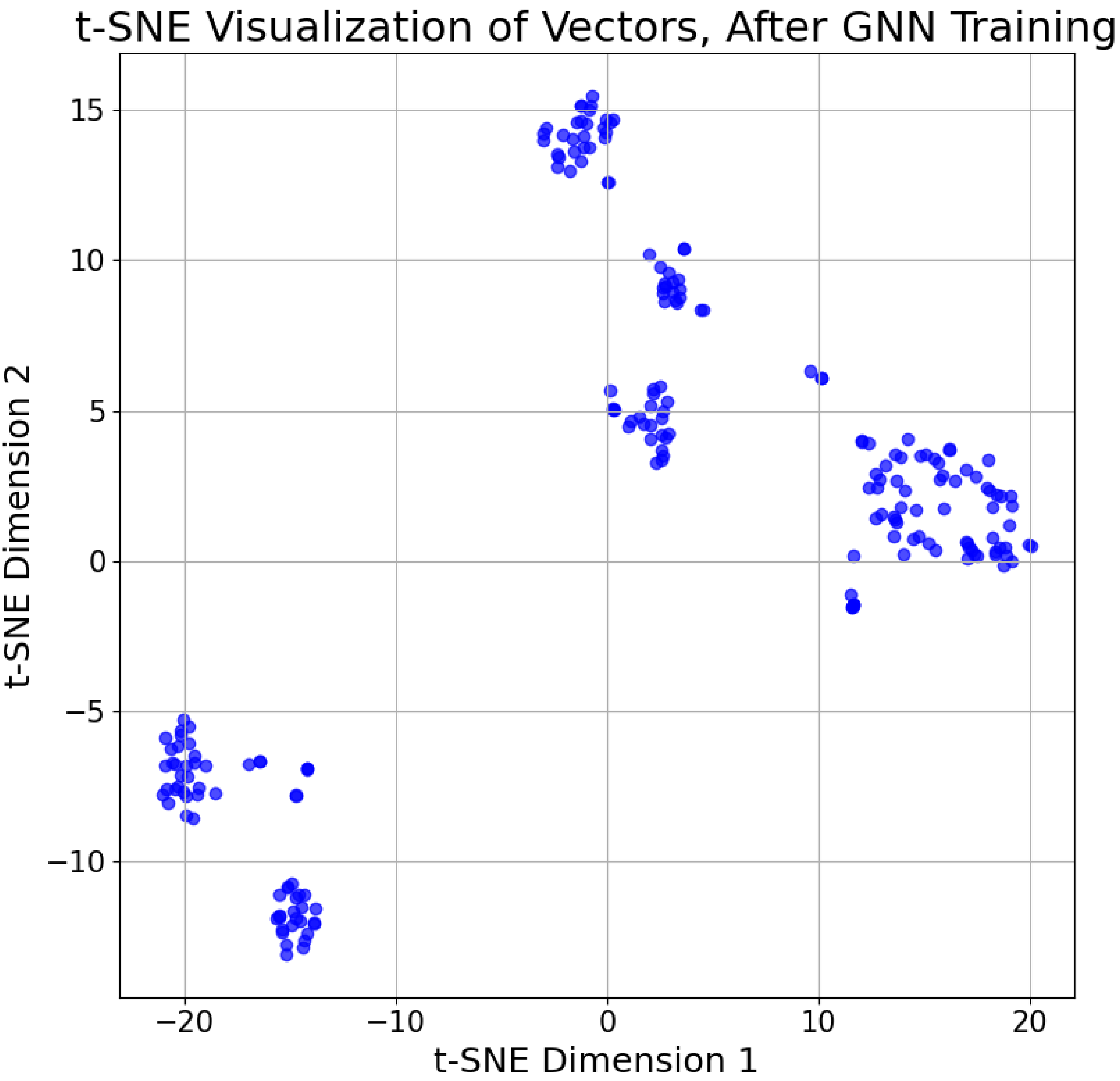
```
kmeans = KMeans(n_clusters=num_clusters, random_state=42)
cluster_labels = kmeans.fit_predict(vectors_2d)

# Visualize the clusters
plt.figure(figsize=(8, 8))
for cluster in range(num_clusters):
    plt.scatter(
        vectors_2d[cluster_labels == cluster, 0],
        vectors_2d[cluster_labels == cluster, 1],
        label=f"Cluster {cluster}",
    )
plt.title("t-SNE with K-Means Clusters, After GNN Training", fontsize=22)
plt.xlabel("t-SNE Dimension 1", fontsize=18)
plt.ylabel("t-SNE Dimension 2", fontsize=18)

# Change the fontsize for the ticks
plt.tick_params(axis="both", which="major", labelsize=15)  # For major
ticks
plt.tick_params(axis="both", which="minor", labelsize=12)  # For minor
ticks (optional)

# change the legend fontsize
plt.legend(fontsize=18, loc="best")
plt.grid()

plt.tight_layout()
plt.savefig("./figures/t-SNE with K-Means Clusters, After GNN
Training.png")
plt.savefig("./figures/t-SNE with K-Means Clusters, After GNN
Training.pdf")
plt.show()

from collections import Counter

cluster_dict = dict(Counter(cluster_labels))

ticks_lasted = []
time_elapsed = []
remaining_rocks = []
victim_harm = []
success = []
round_num = []

for vertex in situation_vertices:
    cp_vertex =
anonymous_package.g.V(vertex).inE("has_situation").outV().next()
    properties = anonymous_package.get_properties(cp_vertex)
    ticks_lasted.append(properties["ticks_lasted"])
    time_elapsed.append(properties["time_elapsed"])
    remaining_rocks.append(properties["remaining_rocks"])
    victim_harm.append(properties["victim_harm"])
    success.append(properties["success"])
    round_num.append(properties["round_num"])
```

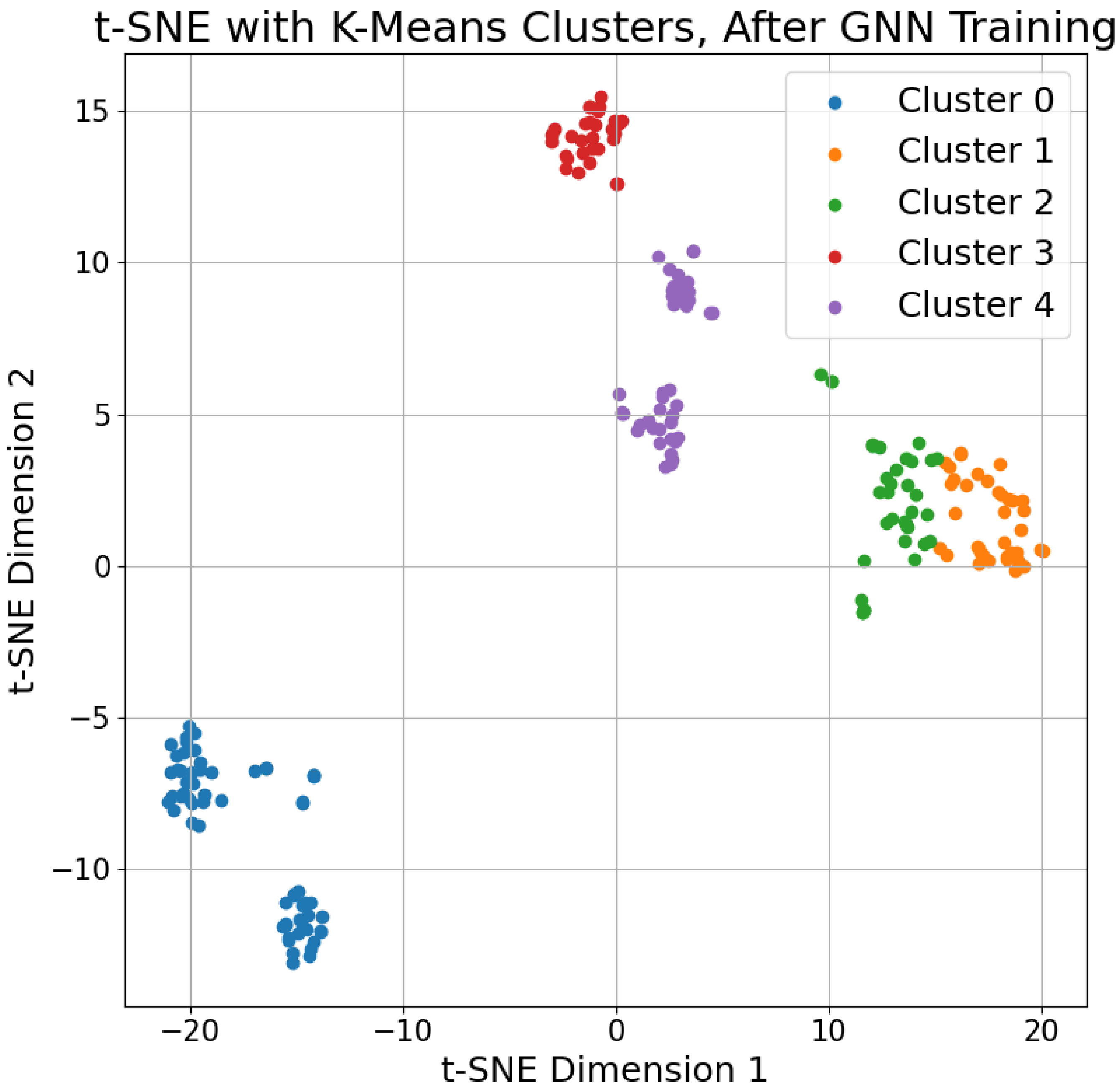
```
ticks_lasted = np.array(ticks_lasted)
time_elapsed = np.array(time_elapsed)
remaining_rocks = np.array(remaining_rocks)
victim_harm = np.array(victim_harm)
success = np.array(success)
round_num = np.array(round_num)

for label, val in cluster_dict.items():
    print(f"Cluster {label}: (count: {val})")
    print(f"Ticks lasted: {int(ticks_lasted[cluster_labels ==
label].mean())}")
    print(f"Time elapsed: {int(time_elapsed[cluster_labels ==
label].mean())}")
    print(f"Remaining rocks: {int(remaining_rocks[cluster_labels ==
label].mean())}")
    print(f"Victim harm: {int(victim_harm[cluster_labels ==
label].mean())}")
    print(f"Success: {success[cluster_labels == label].mean()}")
    print(f"Round num: {int(round_num[cluster_labels == label].mean())}")
    print()

import numpy as np

# Get the cluster centroids from K-Means
centroids = kmeans.cluster_centers_

# Find the closest data point to each centroid
closest_points = []

for cluster_id in range(num_clusters):
    # Get the indices of points in the current cluster
    cluster_points = vectors_2d[cluster_labels == cluster_id]
    original_indices = np.where(cluster_labels == cluster_id)[0]

    # Compute the distances to the centroid
    distances = np.linalg.norm(cluster_points - centroids[cluster_id],
axis=1)

    # Find the index of the closest point
    closest_point_idx = original_indices[np.argmin(distances)]
    closest_points.append(closest_point_idx)


# Output the closest data points for each cluster
for cluster_id, point_idx in enumerate(closest_points):
    cp_vertex = (

anonymous_package.g.V(situation_vertices[point_idx]).inE("has_situation").o
utV().next()
    )
    cp_num = anonymous_package.get_properties(cp_vertex)["cp_num"]
    print(f"Closest point to centroid of Cluster {cluster_id}: CP
{cp_num}")
```

```
import pandas as pd
import numpy as np

# Compute mean values per cluster
cluster_means = {
    "time_elapsed": [],
    "remaining_rocks": [],
    "victim_harm": [],
    "success": [],
}

for label in range(num_clusters):
    cluster_indices = cluster_labels == label

cluster_means["time_elapsed"].append(np.mean(time_elapsed[cluster_indices])
)

cluster_means["remaining_rocks"].append(np.mean(remaining_rocks[cluster_ind
ices]))

cluster_means["victim_harm"].append(np.mean(victim_harm[cluster_indices]))
    cluster_means["success"].append(np.mean(success[cluster_indices]))

# Compute overall mean for difference calculations
overall_mean = {
    "time_elapsed": np.mean(time_elapsed),
    "remaining_rocks": np.mean(remaining_rocks),
    "victim_harm": np.mean(victim_harm),
    "success": np.mean(success),
}

# Create a DataFrame
df_clusters = pd.DataFrame(
    {
        "Time elapsed": [int(val) for val in
cluster_means["time_elapsed"]],
        "Remaining rocks": [int(val) for val in
cluster_means["remaining_rocks"]],
        "Victim harm": [int(val) for val in cluster_means["victim_harm"]],
        "Success": [round(val, 2) for val in cluster_means["success"]],
        "Time elapsed Diff from Mean": [
            round(val - overall_mean["time_elapsed"], 1)
            for val in cluster_means["time_elapsed"]
        ],
        "Remaining rocks Diff from Mean": [
            round(val - overall_mean["remaining_rocks"], 1)
            for val in cluster_means["remaining_rocks"]
        ],
        "Victim harm Diff from Mean": [
            round(val - overall_mean["victim_harm"], 1)
            for val in cluster_means["victim_harm"]
        ],
        "Success Diff from Mean": [
            round(val - overall_mean["success"], 2) for val in
```

```
cluster_means["success"]
        ],
    },
    index=[f"Cluster {i}" for i in range(num_clusters)],
)
df_clusters
```

```
100%|██████████| 209/209
```






```
Cluster 4: (count: 43)
Ticks lasted: 896
Time elapsed: 2629
Remaining rocks: 16
Victim harm: 355
Success: 0.18604651162790697
Round num: 5

Cluster 0: (count: 62)
Ticks lasted: 1344
Time elapsed: 2716
Remaining rocks: 16
Victim harm: 291
Success: 0.24193548387096775
Round num: 5

Cluster 3: (count: 30)
```

```
Ticks lasted: 608
Time elapsed: 2869
Remaining rocks: 18
Victim harm: 246
Success: 0.16666666666666666
Round num: 4

Cluster 1: (count: 38)
Ticks lasted: 581
Time elapsed: 2133
Remaining rocks: 11
Victim harm: 318
Success: 0.42105263157894735
Round num: 5

Cluster 2: (count: 36)
Ticks lasted: 970
Time elapsed: 2607
Remaining rocks: 11
Victim harm: 311
Success: 0.3611111111111111
Round num: 5

Closest point to centroid of Cluster 0: CP 194
Closest point to centroid of Cluster 1: CP 90
Closest point to centroid of Cluster 2: CP 124
Closest point to centroid of Cluster 3: CP 166
Closest point to centroid of Cluster 4: CP 153
```

```
.dataframe tbody tr th {
    vertical-align: top;
}

.dataframe thead th {
    text-align: right;
}
```

	Time elapsed	Remaining rocks	Victim harm	Success	Time elapsed Diff from Mean	Remaining rocks Diff from Mean	Victim harm Diff from Mean	Success Diff from Mean
Cluster 0	2716	16	291	0.24	120.6	1.0	-14.8	-0.03
Cluster 1	2133	11	318	0.42	-462.4	-3.9	11.7	0.15

	Time elapsed	Remaining rocks	Victim harm	Success	Time elapsed Diff from Mean	Remaining rocks Diff from Mean	Victim harm Diff from Mean	Success Diff from Mean
Cluster 2	2607	11	311	0.36	12.1	-3.2	4.4	0.09
Cluster 3	2869	18	246	0.17	273.4	3.9	-60.0	-0.11
Cluster 4	2629	16	355	0.19	33.9	1.9	49.1	-0.09

```
import numpy as np

# Create a dictionary to hold all CP numbers for each cluster
cluster_cp_dict = {}

for cluster_id in range(num_clusters):
    # Get indices of points belonging to the current cluster
    cluster_indices = np.where(cluster_labels == cluster_id)[0]

    # Collect CP numbers for all points in the cluster
    cp_nums = []
    for idx in cluster_indices:
        cp_vertex =
anonymous_package.g.V(situation_vertices[idx]).inE("has_situation").outV().
next()
        cp_num = anonymous_package.get_properties(cp_vertex)["cp_num"]
        cp_nums.append(cp_num)

    # Store them in the dictionary (optional)
    cluster_cp_dict[cluster_id] = cp_nums

    # Print them if desired
    print(f"All CP numbers in Cluster {cluster_id}: {cp_nums}")

# Now cluster_cp_dict contains all CP numbers keyed by cluster ID

from collections import Counter

for key, val in cluster_cp_dict.items():
    print(f"Cluster {key}:")
    vertex_labels = []
    edge_labels = []

    # Collect labels
    for cp_num in val:
        vertices, edges = anonymous_package.get_within_hops(
            [anonymous_package.g.V().has("cp_num",
```

```
cp_num).outE().inV().next()],
            1
        )
        for vertex in vertices:
            vertex_labels.append(vertex.label)
        for edge in edges:
            edge_labels.append(edge.label)

    # Filter out labels you do NOT want to see
    vertex_labels = [v for v in vertex_labels if v not in ("situation",
"CP")]
    edge_labels   = [e for e in edge_labels if e != "has_situation"]

    # Count and sort in descending order
    vertex_labels_counter = Counter(vertex_labels)
    edge_labels_counter   = Counter(edge_labels)
    vertex_labels_sorted  = vertex_labels_counter.most_common()  # returns
list sorted by count desc
    edge_labels_sorted    = edge_labels_counter.most_common()

    # Totals for percentage calculation
    total_vertex_labels = sum(vertex_labels_counter.values())
    total_edge_labels   = sum(edge_labels_counter.values())

    print("Vertex labels (descending order with percentages):")
    for label, count in vertex_labels_sorted:
        percentage = (count / total_vertex_labels) * 100
        print(f"  {label}: {count} ({percentage:.2f}%)")

    print("\nEdge labels (descending order with percentages):")
    for label, count in edge_labels_sorted:
        percentage = (count / total_edge_labels) * 100
        print(f"  {label}: {count} ({percentage:.2f}%)")

    print("-" * 50, "\n")
```

```
All CP numbers in Cluster 0: [44, 206, 51, 115, 154, 38, 118, 188, 42, 182,
185, 200, 196, 194, 117, 161, 162, 133, 29, 195, 207, 156, 22, 73, 173,
152, 150, 113, 47, 160, 76, 24, 210, 157, 179, 53, 203, 116, 25, 91, 56,
199, 52, 189, 114, 77, 165, 149, 61, 159, 209, 158, 191, 39, 50, 155, 208,
54, 205, 20, 143, 145]
All CP numbers in Cluster 1: [100, 111, 71, 66, 112, 104, 86, 72, 89, 88,
109, 65, 105, 62, 102, 35, 184, 34, 67, 186, 63, 90, 32, 33, 40, 120, 92,
31, 108, 93, 110, 106, 94, 68, 101, 69, 87, 107]
All CP numbers in Cluster 2: [96, 131, 181, 202, 127, 136, 139, 128, 132,
192, 164, 174, 171, 177, 97, 201, 124, 138, 95, 123, 126, 204, 129, 163,
122, 103, 137, 135, 130, 172, 198, 125, 176, 197, 167, 134]
All CP numbers in Cluster 3: [43, 16, 78, 45, 48, 49, 4, 98, 141, 17, 9,
13, 166, 21, 18, 3, 28, 142, 7, 99, 46, 79, 11, 2, 55, 14, 5, 140, 1, 41]
```

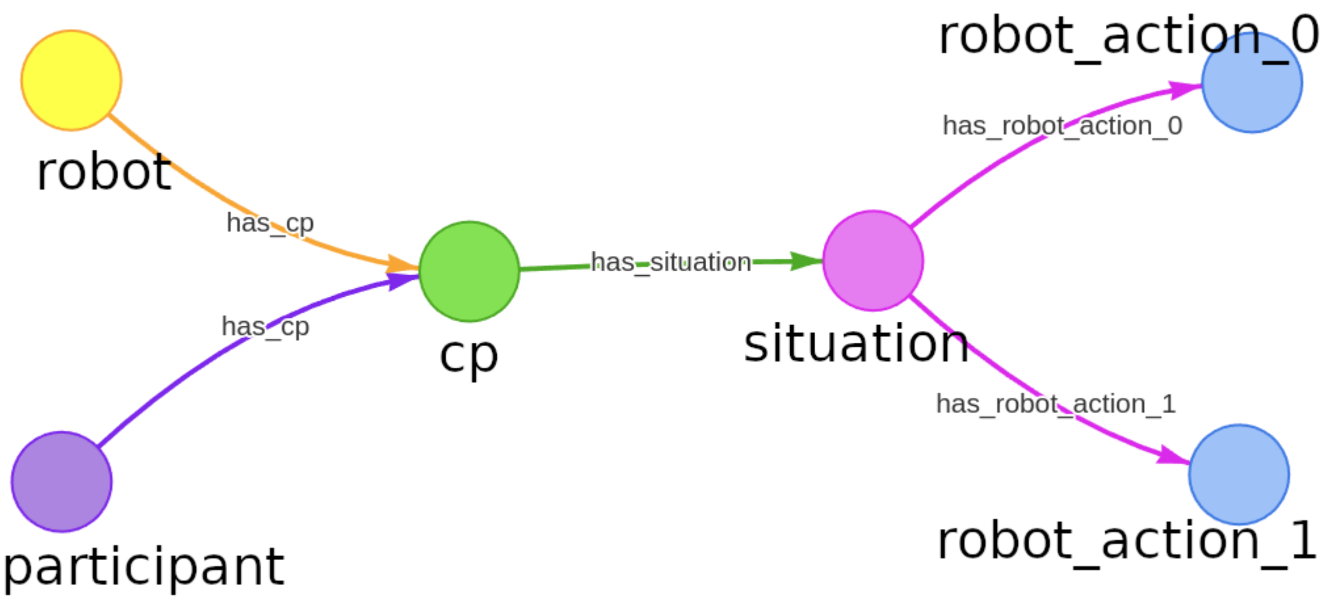
```
All CP numbers in Cluster 4: [19, 37, 119, 187, 57, 26, 169, 30, 183, 193,
178, 6, 74, 27, 59, 12, 168, 8, 10, 58, 15, 82, 83, 144, 36, 84, 85, 148,
175, 75, 64, 80, 190, 151, 60, 153, 23, 70, 121, 81, 147, 146, 180]
Cluster 0:
Vertex labels (descending order with percentages):
  robot_action: 71 (63.96%)
  human_action: 40 (36.04%)

Edge labels (descending order with percentages):
  has_robot_action_0: 62 (55.86%)
  has_human_action_0: 30 (27.03%)
  has_human_action_1: 8 (7.21%)
  has_robot_action_2: 3 (2.70%)
  has_robot_action_3: 3 (2.70%)
  has_robot_action_4: 3 (2.70%)
  has_human_action_2: 2 (1.80%)
--------------------------------------------------

Cluster 1:
Vertex labels (descending order with percentages):
  robot_action: 78 (100.00%)

Edge labels (descending order with percentages):
  has_robot_action_1: 38 (48.72%)
  has_robot_action_0: 38 (48.72%)
  has_robot_action_2: 2 (2.56%)
--------------------------------------------------

Cluster 2:
Vertex labels (descending order with percentages):
  robot_action: 110 (95.65%)
  human_action: 5 (4.35%)

Edge labels (descending order with percentages):
  has_robot_action_1: 36 (31.30%)
  has_robot_action_0: 36 (31.30%)
  has_robot_action_2: 31 (26.96%)
  has_robot_action_3: 7 (6.09%)
  has_human_action_1: 4 (3.48%)
  has_human_action_2: 1 (0.87%)
--------------------------------------------------

Cluster 3:
Vertex labels (descending order with percentages):
  robot_action: 48 (61.54%)
  human_action: 30 (38.46%)

Edge labels (descending order with percentages):
  has_human_action_0: 30 (38.46%)
  has_robot_action_1: 30 (38.46%)
  has_robot_action_2: 16 (20.51%)
  has_robot_action_3: 2 (2.56%)
--------------------------------------------------
```

```
Cluster 4:
Vertex labels (descending order with percentages):
  robot_action: 99 (60.37%)
  human_action: 65 (39.63%)

Edge labels (descending order with percentages):
  has_human_action_0: 43 (26.22%)
  has_robot_action_1: 43 (26.22%)
  has_robot_action_0: 42 (25.61%)
  has_human_action_1: 20 (12.20%)
  has_robot_action_2: 7 (4.27%)
  has_robot_action_3: 7 (4.27%)
  has_human_action_2: 2 (1.22%)
--------------------------------------------------
```

```
vertices, edges = anonymous_package.get_within_hops(
    [anonymous_package.g.V().has("cp_num", 90).outE().inV().next()], 1
)

for vertex in vertices:
    try:
        print(
            vertex.label + "_" +
str(anonymous_package.get_properties(vertex)["action_number"]),
            anonymous_package.get_properties(vertex)
["sentence_representation"],
        )
    except:
        foo = {
            key: val
            for key, val in
anonymous_package.get_properties(vertex).items()
            if key != "vector_representation"
        }

        print(vertex.label, foo)
```

```
situation {'num_recalled': 0, 'sentence_representation': 'Location: Top of
rock pile. Object: Large rock. ', 'location': 'Top of rock pile',
'time_added': '2024-07-03T10:43:10', 'object': 'Large rock'}
CP {'num_recalled': 0, 'cp_num': 90, 'ticks_lasted': 183,
'participant_num': 4087, 'remaining_rocks': 10, 'success': False,
'time_elapsed': 2553, 'cp_name': 'Move', 'victim_harm': 300, 'round_num':
6, 'time_added': '2024-07-03T10:43:10'}
robot_action_1 Action: Drop <object> in <location>. Location: <Right> side
of field. Object: Large rock.
robot_action_0 Action: Pick up <object> in <location>. Location: Top of
rock pile. Object: Large rock.
```

Final CP chosen




```
import scipy.stats as stats
import numpy as np

# Your data
n = 8 * 20  # 160 observations per group
mean_1 = 0.257
std_1 = 0.439
mean_2 = 0.413
std_2 = 0.494

# Calculate pooled standard error
pooled_std = np.sqrt(((n-1)*std_1**2 + (n-1)*std_2**2) / (2*n - 2))
standard_error = pooled_std * np.sqrt(2/n)

# Calculate t-statistic
t_stat = (mean_2 - mean_1) / standard_error

# Degrees of freedom
df = 2*n - 2

# One-tailed p-value (since you want to show mean_2 > mean_1)
p_value = 1 - stats.t.cdf(t_stat, df)

print(f"t-statistic: {t_stat:.4f}")
print(f"p-value (one-tailed): {p_value:.6f}")
print(f"Effect size (Cohen's d): {(mean_2 - mean_1) / pooled_std:.4f}")

# Add confidence interval for the difference
import scipy.stats as stats
import numpy as np

# Calculate 95% confidence interval for the difference
diff = mean_2 - mean_1
```

```
margin_error = stats.t.ppf(0.975, df) * standard_error
ci_lower = diff - margin_error
ci_upper = diff + margin_error

print(f"\nDifference in means: {diff:.3f}")
print(f"95% CI for difference: [{ci_lower:.3f}, {ci_upper:.3f}]")
print(f"Percentage improvement: {((mean_2 - mean_1) / mean_1 *
100):.1f}%\n")


# Summary for reporting
print("=== STATISTICAL RESULTS SUMMARY ===")
print(f"Group 1 (baseline): {mean_1:.1%} success rate (SD = {std_1:.3f})")
print(f"Group 2 (treatment): {mean_2:.1%} success rate (SD = {std_2:.3f})")
print(f"Difference: {diff:.3f} ({((mean_2 - mean_1) / mean_1 * 100):.1f}%
improvement)")
print(f"95% CI: [{ci_lower:.3f}, {ci_upper:.3f}]")
print(f"t({df}) = {t_stat:.3f}, p = {p_value:.4f} (one-tailed)")
print(f"Effect size (Cohen's d) = {(mean_2 - mean_1) / pooled_std:.3f}")
```

```
t-statistic: 2.9858
p-value (one-tailed): 0.001524
Effect size (Cohen's d): 0.3338

Difference in means: 0.156
95% CI for difference: [0.053, 0.259]
Percentage improvement: 60.7%

=== STATISTICAL RESULTS SUMMARY ===
Group 1 (baseline): 25.7% success rate (SD = 0.439)
Group 2 (treatment): 41.3% success rate (SD = 0.494)
Difference: 0.156 (60.7% improvement)
95% CI: [0.053, 0.259]
t(318) = 2.986, p = 0.0015 (one-tailed)
Effect size (Cohen's d) = 0.334
```