Exploration of Projection Spaces

In [1]:
```
# Disable warnings
import warnings
warnings.filterwarnings('ignore', category=FutureWarning)

# Matplotlib inline configuration for Jupyter Notebook
%matplotlib inline

# Core Libraries
import numpy as np
import pandas as pd

# Scipy and Scikit-Learn Libraries
from scipy import interpolate
from sklearn import manifold
from sklearn.decomposition import PCA, FastICA

# Dimensionality Reduction and Clustering Libraries
from openTSNE import TSNE
from umap import UMAP
import hdbscan

# Visualization Libraries
import matplotlib.pyplot as plt
from matplotlib.colors import ListedColormap
from matplotlib.offsetbox import OffsetImage
import altair as alt
from altair import datum
alt.data_transformers.disable_max_rows()

# Import custom module
from utils import CliffWalkingVisualizer
```

```
/opt/anaconda3/envs/xai_proj_space/lib/python3.10/site-packages/tqdm/auto.
py:21: TqdmWarning: IProgress not found. Please update jupyter and ipywidg
ets. See https://ipywidgets.readthedocs.io/en/stable/user_install.html
  from .autonotebook import tqdm as notebook_tqdm
```

Data

In this project, we explore a classic reinforcement learning problem called **Cliff Walking**, where the objective is to navigate a grid world from a starting state to a target destination while avoiding the treacherous "cliff" along the way. This environment, provided by the popular gym library, presents a challenging landscape for learning algorithms. Stepping off the cliff leads in a steep penalty, requiring intelligent decision-making to find optimal paths.

To investigate different strategies in this task, we compare the behaviors of four distinct algorithms: a baseline random policy, SARSA with an epsilon-greedy policy, Q-learning with an epsilon-greedy policy, and Expected SARSA with an epsilon-

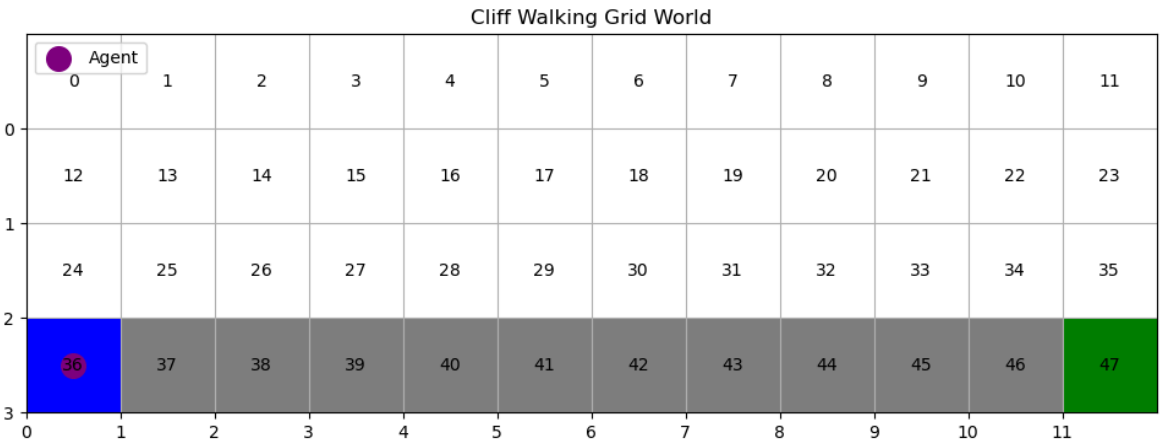
greedy policy. Each algorithm was trained over 5000 episodes, allowing us to analyze convergence patterns and path efficiency.

We thought the Cliff Walking dataset is well-suited for down-projection of learning trajectories for several reasons:

1. **Grid Structure**: The finite, discrete nature of the grid world simplifies state representation, making it easier to visualize and project trajectories.
2. **Convergence Patterns**: As agents learn, their trajectories stabilize, revealing patterns of convergence. Down-projection highlights these trends, showcasing how algorithms adapt their policies to avoid the cliff.
3. **Natural Clusters**: The environment's distinct regions—cliff areas, safe zones, and the goal—create interpretable clusters in state space. Down-projected trajectories make it easier to identify decision points and areas of difficulty.
4. **Algorithm Comparison**: By projecting trajectories into lower dimensions, we can visualize and compare strategies across different algorithms.
5. **High-Revisit States**: States near the cliff are frequently revisited, creating denser regions that reflect the strategic decision-making of the agents.

Overall, the structured environment and dynamic learning behaviors make the Cliff Walking dataset an excellent candidate for visualizing learning trajectories through down-projection techniques.

In [2]:
```
# Initialize the visualizer:
visualizer = CliffWalkingVisualizer()

# Visualize the grid world with the agent in a specified state, e.g., 36:
visualizer.visualize_grid_world(agent_state=36)
```



This plot shows the grid world. The blue state reflects the starting state while the green state corresponds to the goal state. The agent needs to reach this goal state. With every step it takes, it gets a reward of -1. Thus, it is encouraged to take as little steps as possible. However, when the agent falls of the cliff (depicted in grey), it gets punishment of -100. Due to the exploration behaviour of the policies, some random

actions are also taken which might cause the agent to fall off the cliff when it is too close.

Read and Prepare Data

Read in your data from a file or create your own data.

Document any data processing steps.

In [3]:
```
# Load trajectory data for each algorithm
with open('data/cliff_walking/expected_sarsa.npy', 'rb') as f:
    expected_sarsa = np.load(f, allow_pickle=True)
with open('data/cliff_walking/q_learning.npy', 'rb') as f:
    q_learning = np.load(f, allow_pickle=True)
with open('data/cliff_walking/random.npy', 'rb') as f:
    random = np.load(f, allow_pickle=True)
with open('data/cliff_walking/sarsa.npy', 'rb') as f:
    sarsa = np.load(f, allow_pickle=True)

# Set algorithm names and pair with corresponding trajectory data
algorithms = ["Expected SARSA", "Q Learning", "RANDOM", "SARSA"]
trajectories_algos = [expected_sarsa, q_learning, random, sarsa]

# Initialize list to store episode data for DataFrame
data = []

# Process trajectories for each algorithm
for trajectories_algo, algo in zip(trajectories_algos, algorithms):
    for episode_index, trajectory in enumerate(trajectories_algo):
        # Sample every fifth episode
        if episode_index % 5 == 0:
            episode_length = len(trajectory)

            for step_index, step in enumerate(trajectory):
                state, action, reward, next_state, done = step

                # Label step position within the episode
                if step_index == 0:
                    cp = 'start'
                elif step_index == episode_length - 1:
                    cp = 'end'
                else:
                    cp = 'intermediate'

                # Append episode step data to list
                data.append({
                    'line': episode_index,
                    'cp': cp,
                    'algorithm': algo,
                    'state': state
                    #'action': action,
                    #'reward': reward,
                    #'next_state': next_state
                })

# Create DataFrame with all episodes' data
df = pd.DataFrame(data)
```

```
# Separate metadata and projection data
meta_data = df.iloc[:, :3]
proj_data = df.iloc[:, 3:]
```

In [4]:
```
# Convert 'state' column in proj_data to one-hot encoding
one_hot_df = pd.get_dummies(proj_data, columns=['state'], prefix=['state'

one_hot_df.head()
```

Out[4]:

	state_0	state_1	state_2	state_3	state_4	state_5	state_6	state_7	state_8
0	False	False	False	False	False	False	False	False	False
1	False	False	False	False	False	False	False	False	False
2	False	False	False	False	False	False	False	False	False
3	False	False	False	False	False	False	False	False	False
4	False	False	False	False	False	False	False	False	False

5 rows × 37 columns

Comments

- Did you transform, clean, or extend the data? How/Why?

  For each algorithm, the dataset is organized as lists of episodes, where each episode consists of dicts containing (state, action, reward, next_state, done). We filtered the dataset by taking every fifth episode from each algorithm, resulting in a sample of 1,000 episodes per algorithm. This downsampling aims to prevent overcrowding in plots, allowing for clearer visual analysis.

  To simplify and focus our analysis, we used only the state information in the downprojection, as other data (such as actions and rewards) are indirectly represented by the trajectory structure, reducing redundancy while still capturing essential dynamics.

  Additionally, defining the state-space representation is critical for effective analysis and visualization. Here, we applied a one-hot encoding to the states to maintain interpretability and ensure that each unique state has a distinct representation. This encoding allows us to downproject the states in a way that retains their unique properties and relationships, supporting a more nuanced understanding of the agent's movement through the state space and aiding in the clustering and comparison of trajectory patterns across algorithms.

Projection

Project your data into a 2D space. Try multiple (3+) projection methods (e.g., t-SNE, UMAP, MDS, PCA, ICA, other methods) with different settings and compare them.

Make sure that all additional dependencies are included when submitting.

Comments

- Which features did you use? Why?

  We selected only the 48 discrete states of our Cliff Walking environment as features. For each episode, our reinforcement learning agent occupied a current state, chose an action that led to a next state, and received a reward. We used only the current state as our primary feature for trajectory data, as other information can be inferred implicitly by observing these trajectories. The constant reward of -1 (except when the agent reaches a cliff) signifies episode termination, while the action can be deduced from state transitions. This approach simplifies the data without compromising information needed for trajectory analysis.

- Which projection methods did you use? Why?

  To investigate both global and local structures within the trajectory data, we applied three projection techniques with distinct characteristics: **t-SNE**, **PCA**, and **UMAP**. Each method offers unique benefits and limitations, which we explored to understand their suitability and behavior on our dataset.

  t-SNE (t-distributed Stochastic Neighbor Embedding)

  - t-SNE is a nonlinear dimensionality reduction method that is well-suited to data that are not linearly separable. It preserves local structures effectively, making it useful for analyzing clusters of closely related states in trajectories.
  - It is computationally intensive, as it calculates pairwise similarities between points, and often struggles to retain global relationships, focusing instead on preserving the neighborhood structure.
  - Its output can vary between runs due to non-deterministic initialization, and its axes lack interpretability.
  - We chose t-SNE to observe fine-grained local structures within state clusters, despite its limitations in representing broader patterns across clusters.
  - **Hyperparameters**: We experimented with various perplexity values to adjust the balance between local and global structures. We calculated t-sne coordinates for the following perplexities [5, 10, 30, 50, 100, 500, 1000] and choose to plot the one that appears in this particular run to best depict the different tracetories.

  PCA (Principal Component Analysis)

  - PCA is a linear dimensionality reduction method that maintains both global and local relationships in the data and is computationally efficient.
  - It is ideal for identifying principal directions of variance in high-dimensional data, offering interpretable axes that represent maximum variance

directions.
- Though it may miss non-linear structures, PCA effectively retains global structure, making it useful as a baseline for understanding overall patterns in trajectories.
- **Hyperparameters**: We retained enough principal components to capture a substantial proportion of variance, ensuring that the representation was compact but informative.

UMAP (Uniform Manifold Approximation and Projection)

- UMAP is a non-linear technique that provides a balance between preserving local and global structures, offering both interpretable clusters and an approximate overview of overall relationships.
- Its flexibility in retaining both neighborhood structures and larger patterns makes it highly suitable for our data, as it may reveal both fine-grained trajectory clusters and broader state-space patterns.
- **Hyperparameters**: We optimized parameters for minimum distance and number of neighbors to adjust the level of detail in clustering and neighborhood preservation.

ICA (Independent Component Analysis)

- ICA is a downprojection method that finds statistically independent components in the data. These independent components can then reveal underlying patterns.
- It can produce components that are interpretable, especially if the original high-dimensional data were mixtures of independent sources.
- It operates under the assumption that the source signals are non-Gaussian. This makes ICA well-suited for data that do not adhere to Gaussian distributions, however it may struggle to separate data effectively if this assumption is not fullfilled.
- It additionally assumes that the sources are mixed linearly, which may not always be the case. If the sources are mixed nonlinearly, ICA may not be effective.
- **Hyperparameters**: We optimized the number of components to capture the main independent patterns in trajectory data while ensuring meaningful downprojection.

- Why did you choose these hyperparameters?

  Each method's hyperparameters were selected to leverage its strengths: **t-SNE** focused on neighborhood structure, **PCA** maximized variance capture, **UMAP** balanced local/global structures, and **ICA** isolated independent components. These choices were aligned with our goals of uncovering detailed clusters and overall patterns in the state space.

- Are there patterns in the global *and* the local structure?

See the detailed analysis at the end of each method.

TSNE

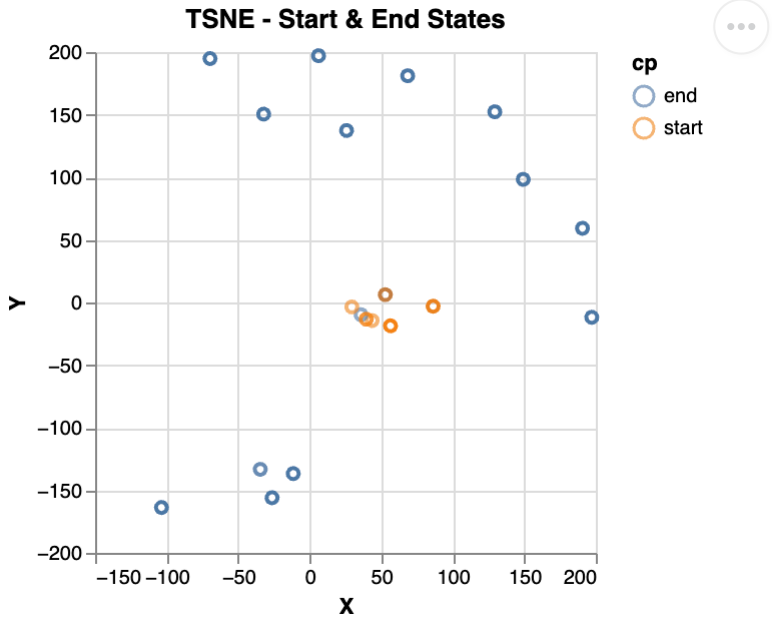
In [5]:
```
# TODO
# tsne_coords = manifold.TSNE(perplexity=p).fit_transform(one_hot_df)
# tsne_coords_perplexities.append(tsne_coords)
t_sne_perplexities = np.load('data/cliff_walking/tsne_coords_perplexities
df_coords = pd.DataFrame(t_sne_perplexities, columns=['X','Y'])
plotting_df = pd.concat([meta_data, df_coords], axis='columns')
plotting_df.head()
```

Out[5]:

	line	cp	algorithm	X	Y
0	0	start	Expected SARSA	43.642590	-14.434265
1	0	intermediate	Expected SARSA	-49.644737	53.423031
2	0	intermediate	Expected SARSA	-49.644737	53.423031
3	0	end	Expected SARSA	35.871559	-9.731361
4	5	start	Expected SARSA	52.834625	6.233274

In [6]:
```
alt.Chart(plotting_df).mark_point(
    opacity=0.6
).encode(
    x='X',
    y='Y',
    color='cp:N'
).transform_filter((datum.cp=='start') | (datum.cp=='end')
).properties(
    width=250,
    height=250,
    title="TSNE - Start & End States"
)
```

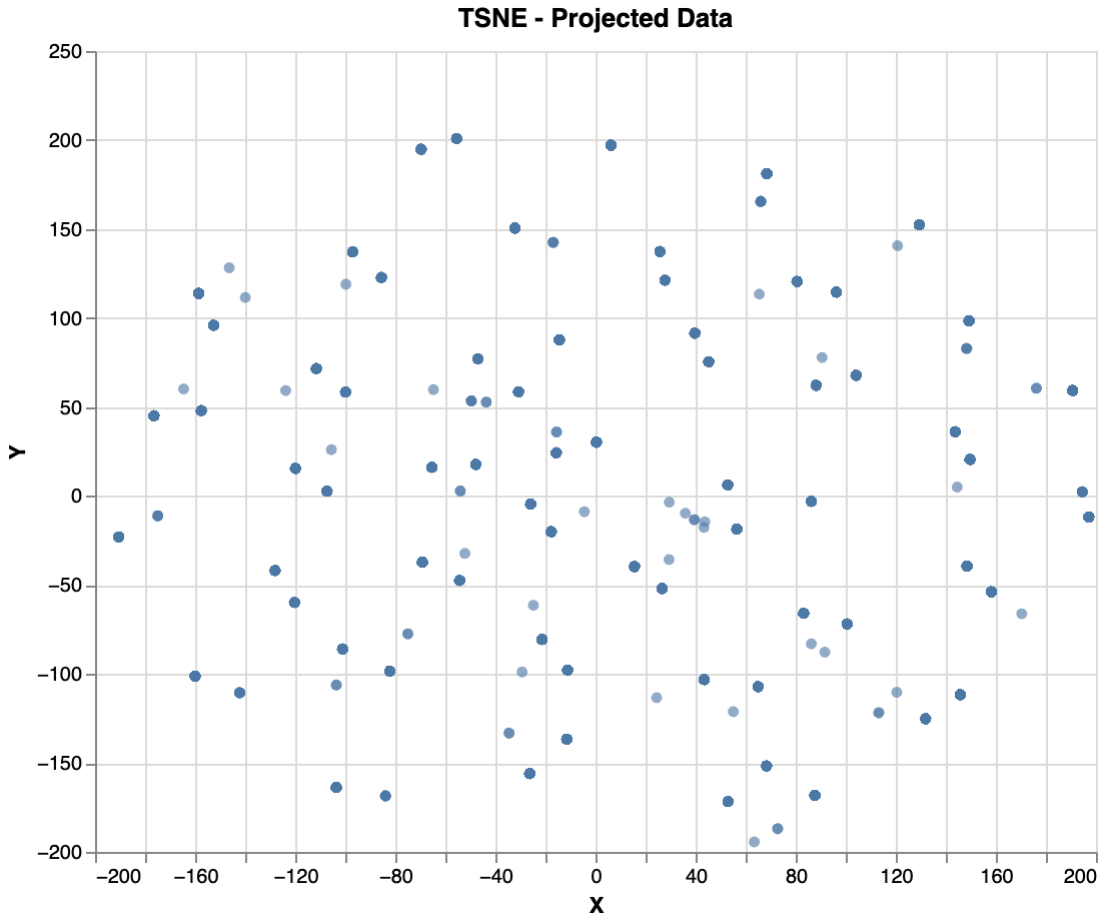
Out[6]:



In [7]:
```
alt.Chart(plotting_df).mark_circle(
    opacity=0.6
```

```
).encode(
    x='X',
    y='Y',
).properties(
    width=500,
    height=400,
    title="TSNE - Projected Data"
)
```

Out[7]:



Observation t-SNE:

In our dataset, we have many duplicate states, such as a single starting state for all trajectories and a limited set of possible end states. The first plot demonstrates an interesting property of t-SNE: because it's an iterative optimization process applied to all points simultaneously, identical states are not guaranteed to be mapped to the exact same coordinates. Instead, while these points may cluster close to one another, they may still have slight variations in position. This effect is particularly noticeable for the starting states (orange points), where multiple (x, y) coordinates represent the exact same state after downprojection.

In the second plot, we observe a greater variety of projected points than the actual 48 unique states, indicating that a single state is represented by several different t-SNE projections. This is likely due to the sensitivity of t-SNE to local variations, causing identical states to be spread slightly in the projection space.

Are there patterns in the global and the local structure?

Larger global structures are not readily discernible from these t-SNE projections. However, we can identify some smaller clusters. The darker points, where opacity

has increased due to overlapping projections, suggest areas with higher revisit frequencies in the original trajectories. These smaller clusters could correspond to specific states among the 48 unique ones, possibly indicating areas of strategic importance or high traffic in the grid (e.g., states near the cliff or around the goal).

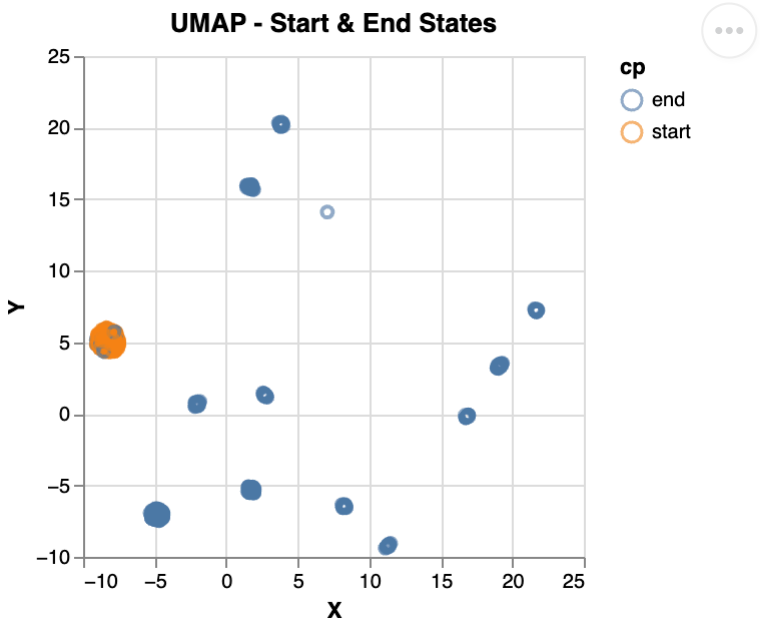
UMAP

In [8]:
```
# Setting the parameters for UMAP
umap_model = UMAP(n_neighbors=15, min_dist=0.1, random_state=42)

# Execute the UMAP on the dataframe with one-hot encoding
umap_coords = umap_model.fit_transform(one_hot_df)
df_umap_coords = pd.DataFrame(umap_coords, columns=['X', 'Y'])

# Combine the metadata with the UMAP coordinates
umap_plotting_df = pd.concat([meta_data, df_umap_coords], axis='columns')
```

```
/opt/anaconda3/envs/xai_proj_space/lib/python3.10/site-packages/umap/umap
_.py:1945: UserWarning: n_jobs value 1 overridden to 1 by setting random_s
tate. Use no seed for parallelism.
  warn(f"n_jobs value {self.n_jobs} overridden to 1 by setting random_stat
e. Use no seed for parallelism.")
```

In [9]:
```
# Graph for UMAP showing the starting and final points
alt.Chart(umap_plotting_df).mark_point(opacity=0.6).encode(
    x='X',
    y='Y',
    color='cp:N'
).transform_filter(
    (datum.cp == 'start') | (datum.cp == 'end')
).properties(
    width=250,
    height=250,
    title="UMAP - Start & End States"
)
```

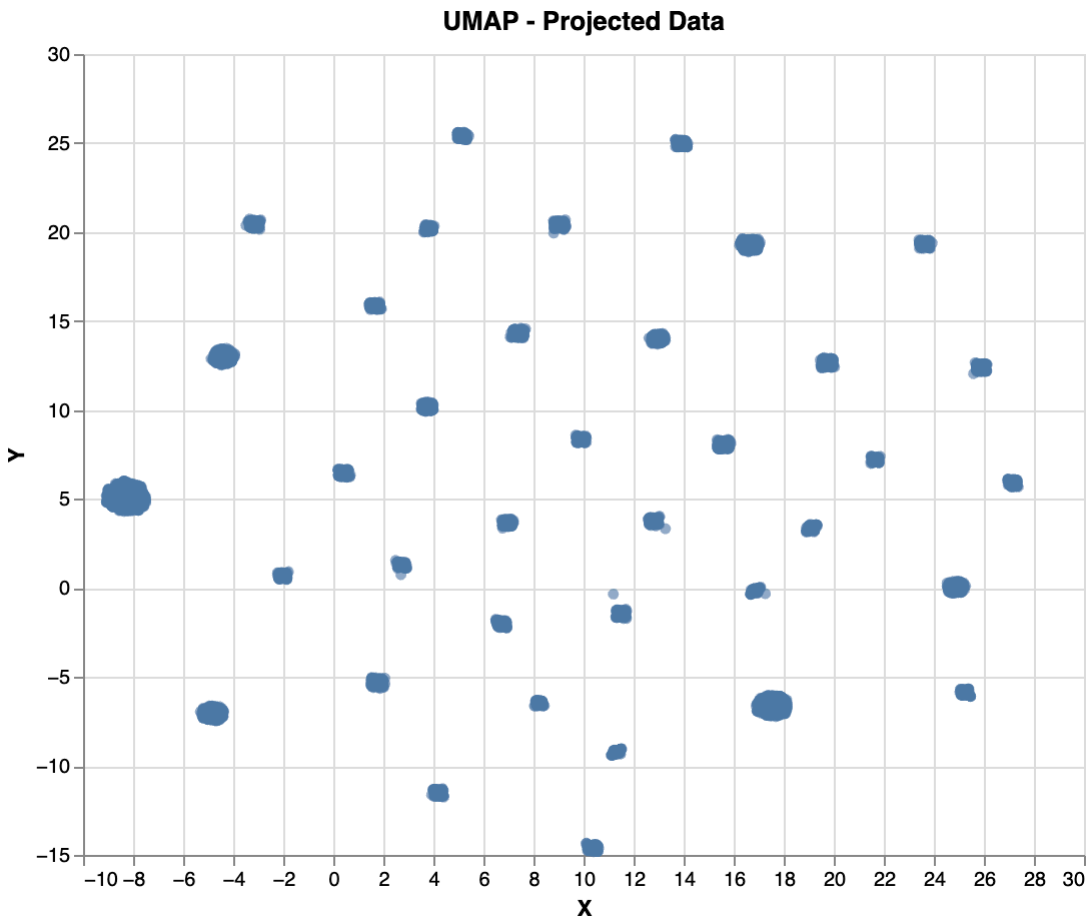
Out[9]:



In [10]:
```
# Graph UMAP of the whole projected dataset
alt.Chart(umap_plotting_df).mark_circle(
```

```
    opacity=0.6
).encode(
    x='X',
    y='Y'
).properties(
    width=500,
    height=400,
    title="UMAP - Projected Data"
)
```

Out[10]:



Observation UMAP:

In the UMAP projection, we also observe that identical states do not always yield identical downprojected coordinates. The starting state, for instance, is represented with slight variations in the downprojected space, as are some of the goal states. Despite this variability, UMAP generally keeps similar states close to each other, though not perfectly identical in position. This can be attributed to the fact that UMAP does not preserve exact distances for every instance, focusing instead on broader neighborhood structures. The total number of points in the plot does not match the expected count of
$4 \text{ algorithms} \times 1000 \text{ episodes} \times \text{number of states per episode}$, indicating that some identical states were mapped very closely or possibly overlapped in the projection.

Are there patterns in the global and the local structure?
UMAP reveals many small clusters, likely corresponding to specific state groups.

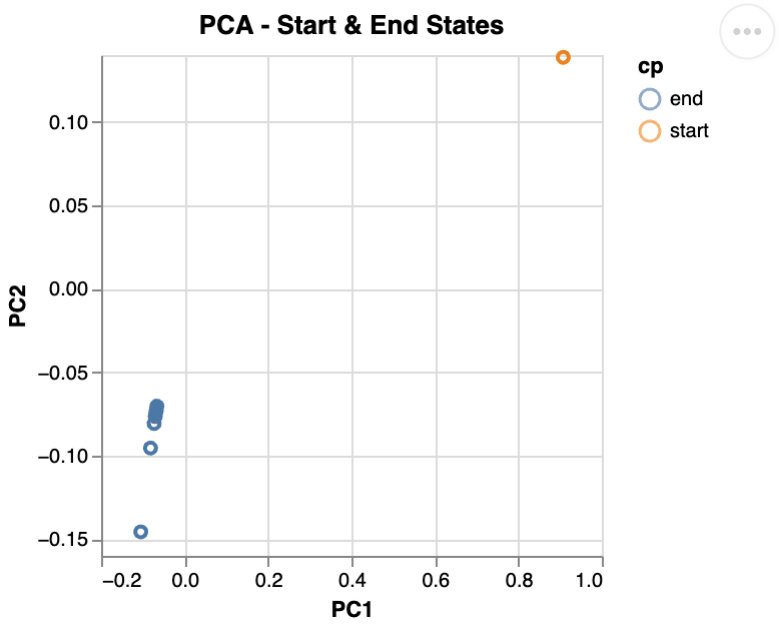
However, no significant global structures are apparent, aside from a prominent cluster for the starting point. The clustering effect suggests local consistency, but UMAP does not display an overarching pattern across the state space, aligning with its emphasis on neighborhood preservation.

PCA

In [11]:
```
# Defining PCA over 2 components on the one-hot encoded data
pca = PCA(n_components=2)
# Project the points on the new coordinate
pca_coords = pca.fit_transform(one_hot_df)

# Create a DataFrame with the new coordinates
df_coords = pd.DataFrame(pca_coords, columns=['PC1', 'PC2'])

# Combine PCA coordinates with metadata
pca_plotting_df = pd.concat([meta_data.reset_index(drop=True), df_coords]
```

In [12]:
```
# Plotting start and end states
alt.Chart(pca_plotting_df).mark_point(opacity=0.6).encode(
    x='PC1',
    y='PC2',
    color='cp:N'
).transform_filter(
    (datum.cp == 'start') | (datum.cp == 'end')
).properties(
    width=250,
    height=250,
    title="PCA - Start & End States"
)
```

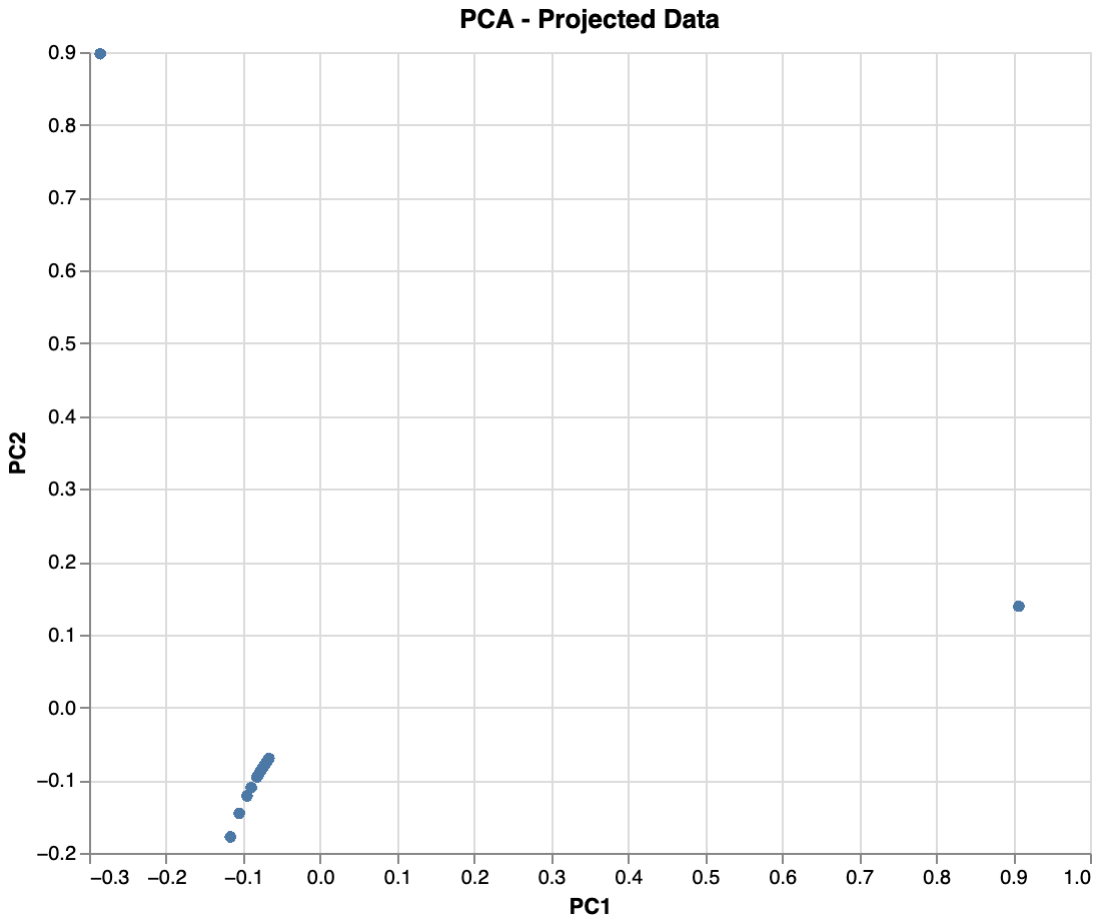
Out[12]:



In [13]:
```
alt.Chart(pca_plotting_df).mark_circle(opacity=0.6).encode(
    x='PC1',
    y='PC2',
).properties(
    width=500,
    height=400,
```

```
    title="PCA - Projected Data"
)
```

Out[13]:



Observation PCA:

In the PCA projection, we observe that the starting state is consistently encoded as a single coordinate, maintaining its position across projections. Since PCA preserves meaningful distances, we can see that most of the end states are closely grouped, with one exception that is slightly farther apart. Overall, the projected space is sparse, with only a few distinct points visible. The intermediate states, located in the upper left corner, are notably separated from the start and end clusters. Interestingly, these intermediate states appear clustered tightly, almost as if represented by a single point, rather than being widely spread out. This compactness suggests that PCA has effectively captured the main variance directions but may have merged some states due to the linear projection.

Are there patterns in the global and the local structure?

The PCA projection reveals a clearer separation of global structure compared to other methods. We see distinct clusters for the starting state, end states, and intermediate states, with meaningful distances between them. While PCA may not capture intricate non-linear relationships, it provides a useful overview of the dataset's main structure, allowing us to identify key clusters with significant separation.

ICA

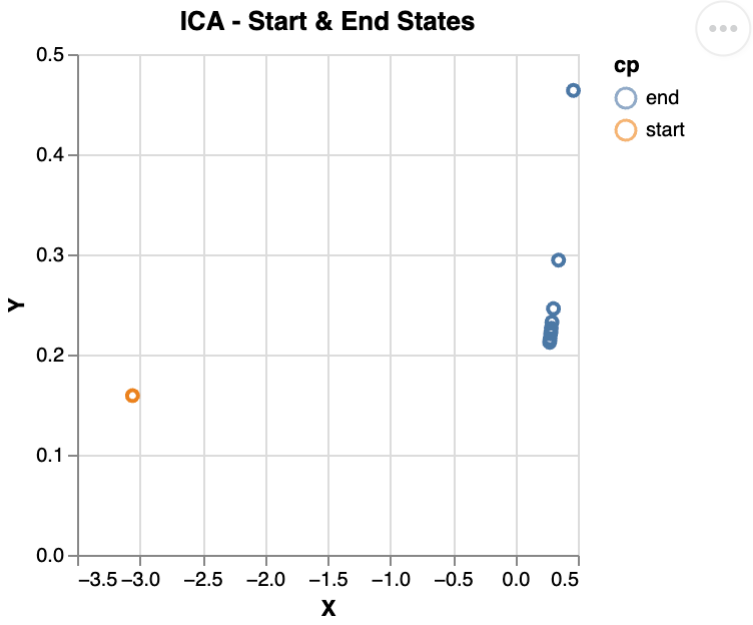
In [14]:
```
# ICA transformation
ica = FastICA(n_components=2, random_state=0)
ica_coords = ica.fit_transform(one_hot_df)
ica_plotting_df = pd.concat([meta_data, pd.DataFrame(ica_coords, columns=
ica_plotting_df.head()
```

Out[14]:

	line	cp	algorithm	X	Y
0	0	start	Expected SARSA	-3.057335	0.159032
1	0	intermediate	Expected SARSA	0.157704	-3.552101
2	0	intermediate	Expected SARSA	0.157704	-3.552101
3	0	end	Expected SARSA	-3.057335	0.159032
4	5	start	Expected SARSA	-3.057335	0.159032

In [15]:
```
alt.Chart(ica_plotting_df).mark_point(
    opacity=0.6
).encode(
    x='X',
    y='Y',
    color='cp:N'
).transform_filter((datum.cp == 'start') | (datum.cp == 'end')
).properties(
    width=250,
    height=250,
    title="ICA - Start & End States"
)
```

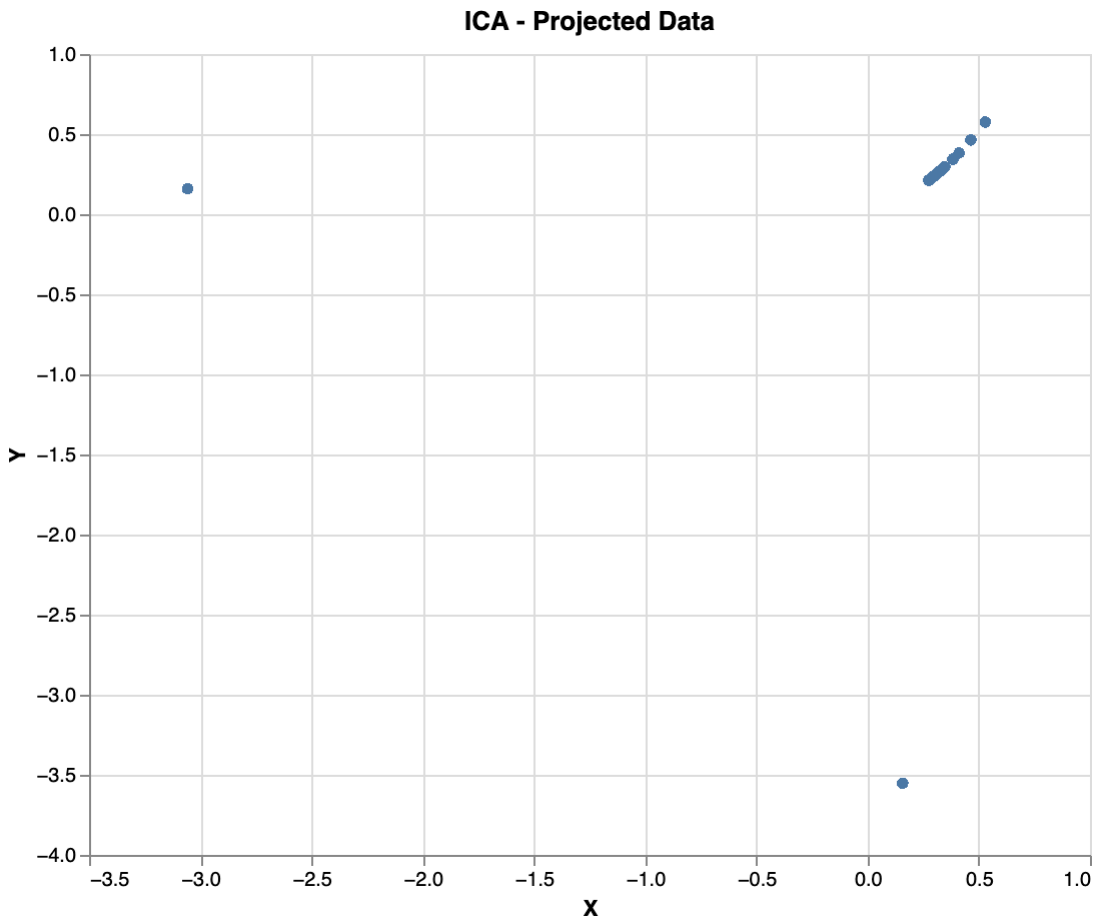
Out[15]:



In [16]:
```
alt.Chart(ica_plotting_df).mark_circle(
    opacity=0.6
).encode(
    x='X',
    y='Y',
).properties(
    width=500,
    height=400,
```

```
    title="ICA - Projected Data"
)
```

Out[16]:



Observation ICA:

In contrast to t-SNE and UMAP, and similar to PCA, ICA consistently maps both the starting state and goal states to specific, stable component values. The data in the ICA projection appears more sparsely distributed than in t-SNE, reflecting ICA's focus on finding independent components rather than preserving local neighborhoods or density.

Are there patterns in the global and the local structure?

Similar to PCA, we observe three main clusters in the ICA projection. Two of these clusters contain only a single (X, Y) component value, while the third cluster consists of multiple distinct component values, suggesting a range of variations within this group. This structure indicates that ICA has identified three primary independent patterns in the data, with two relatively stable clusters and one more variable cluster.

Similarities and Differences of Downprojection Methods

Overall, t-SNE and UMAP show similar behavior, with multiple encodings for identical states, resulting in a denser representation with more scattered points. In contrast, PCA and ICA produce sparser projections, consistently mapping each state to a single coordinate. Both PCA and ICA exhibit similar downprojection patterns: a single

point representing the starting state, a compact cluster of end states, and a single, tightly grouped cluster (or point) for intermediate states. This similarity reflects the linear and independent component focus of PCA and ICA, respectively, in contrast to the neighborhood-preserving nature of t-SNE and UMAP.

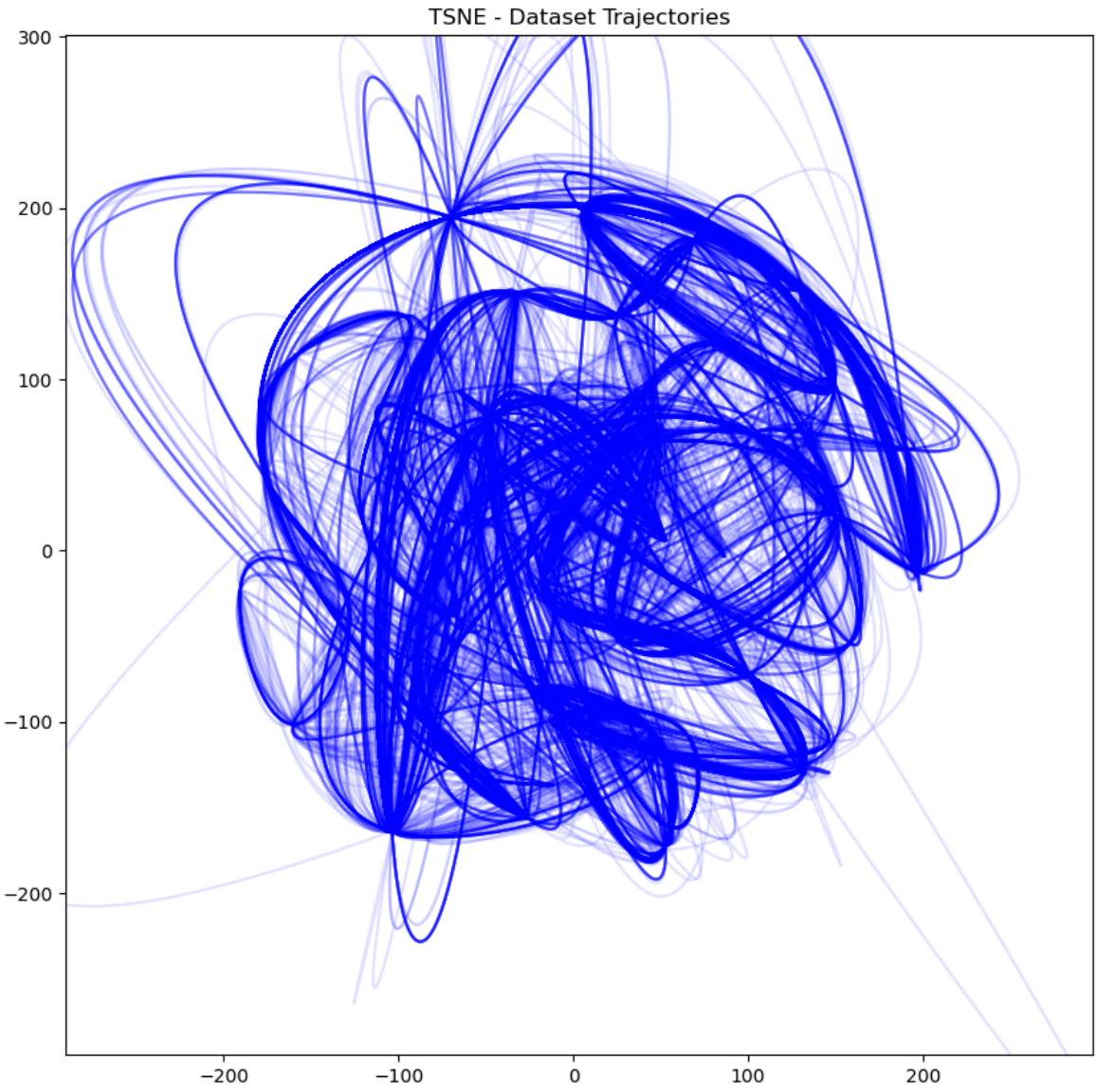
Link States

Connect the states that belong together.

The states of a single solution should be connected to see the path from the start to the end state. How the points are connected is up to you, for example, with straight lines or splines.

TSNE

In [17]:
```
# Function to plot splines
def plot_df_splines(ax, df, x_col='X', y_col='Y', color='blue', alpha=0.3
    x = df[x_col].values
    y = df[y_col].values
    if len(x) >= 4 and len(np.unique(x)) >= 4 and len(np.unique(y)) >= 4:
        try:
            # Prepare the B-spline representation of an N-D curve
            tck, u = interpolate.splprep([x, y], s=smoothing)
            # Evaluate the values of the spline function at specific poin
            x_new, y_new = interpolate.splev(np.linspace(0, 1, n_points),
            ax.plot(x_new, y_new, color=color, alpha=alpha)
        except Exception as e:
            pass
            # Optionally, print the error message for debugging
            # print(f"Error while creating spline: {e}")
    # else:
        # Optionally, print a message if there are not enough points
        # print("Not enough unique points to create a spline.")


# Create a plot
fig, ax = plt.subplots(figsize=(10, 10))

# Get unique algorithms
algorithms = plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = plotting_df[plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax, line_data, alpha=0.1, smoothing=0, n_points=9

ax.set_title("TSNE - Dataset Trajectories")
margin = 100
plt.xlim(plotting_df['X'].min() - margin, plotting_df['X'].max() + margin
plt.ylim(plotting_df['Y'].min() - margin, plotting_df['Y'].max() + margin
plt.show()
```



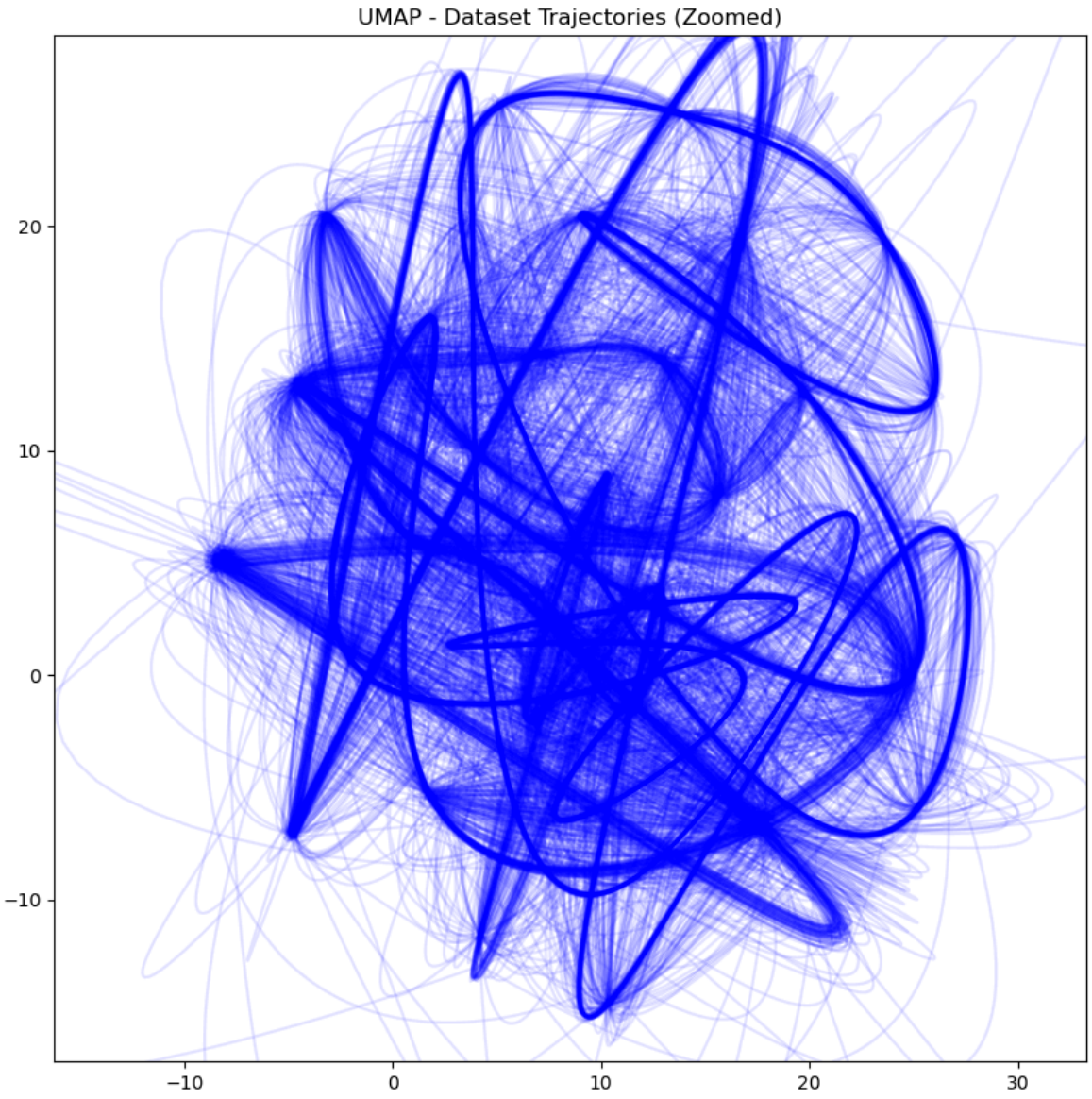
Observation t-SNE:

Without additional color encodings or labels, it's challenging to discern precise patterns in the plot. However, this view highlights some global structures more clearly. Certain paths appear to be more frequently traversed, while others are less common, suggesting preferred routes in the state space. Additionally, there are visible "knots" or clusters where multiple trajectories converge, indicating encoded states that many algorithms pass through repeatedly. These high-traffic areas may represent critical decision points or commonly revisited states near the cliff or goal.

UMAP

In [18]:
```
# Evaluating percentile for X and Y to avoid outlayers

x_min, x_max = umap_plotting_df['X'].quantile([0.05, 0.95])
y_min, y_max = umap_plotting_df['Y'].quantile([0.05, 0.95])

# Creating plot
fig, ax = plt.subplots(figsize=(10, 10))

for i, algo in enumerate(algorithms):
    algo_data = umap_plotting_df[umap_plotting_df['algorithm'] == algo]
```

```
    lines = algo_data['line'].unique()

    # Sampling of lines to enhance the clarity
    sampled_lines = np.random.choice(lines, size=int(len(lines) * 0.1), r

    for line in sampled_lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax, line_data, alpha=0.1, smoothing=0, n_points=1

ax.set_title("UMAP - Dataset Trajectories (Zoomed)")
margin = 8
plt.xlim(x_min - margin, x_max + margin)
plt.ylim(y_min - margin, y_max + margin)
plt.show()
```

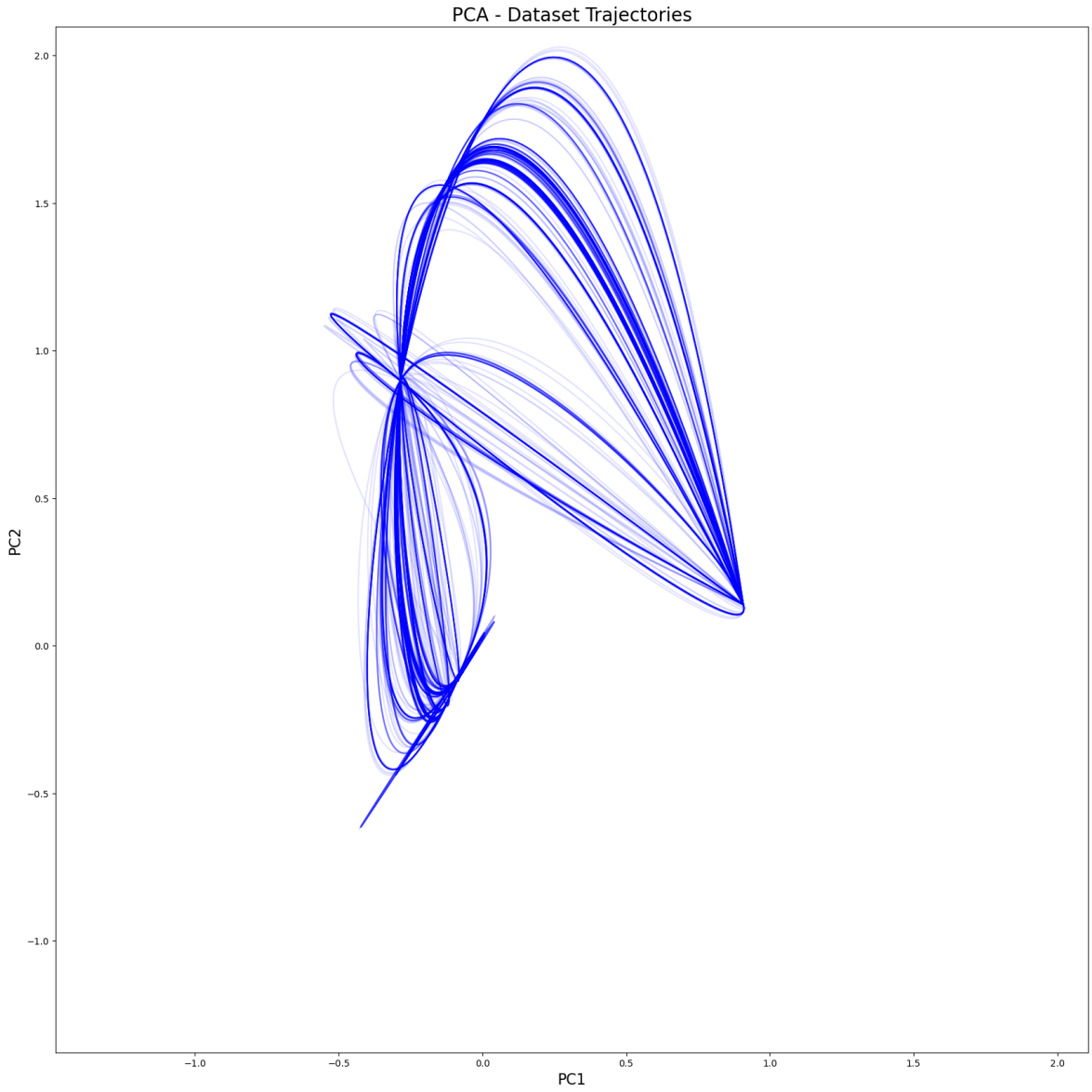


Observation UMAP:

In the UMAP projection, we observe numerous trajectory "bundles," with varying thicknesses. Thicker bundles indicate frequently occurring paths, suggesting that certain trajectories are revisited often. This pattern may imply convergence among the algorithms, as stable policies produce consistent paths through the state space. However, without additional encodings or labels, further interpretation of specific states or transitions is limited.

PCA

In [19]:
```
# Create a plot
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = pca_plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = pca_plotting_df[pca_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax, line_data, x_col='PC1', y_col='PC2', alpha=0.

ax.set_title("PCA - Dataset Trajectories", fontsize=20)
margin = 1.2  # Adjusted margin for PCA data
plt.xlim(pca_plotting_df['PC1'].min() - margin, pca_plotting_df['PC1'].ma
plt.ylim(pca_plotting_df['PC2'].min() - margin, pca_plotting_df['PC2'].ma
plt.xlabel('PC1', fontsize=16)
plt.ylabel('PC2', fontsize=16)
plt.show()
```



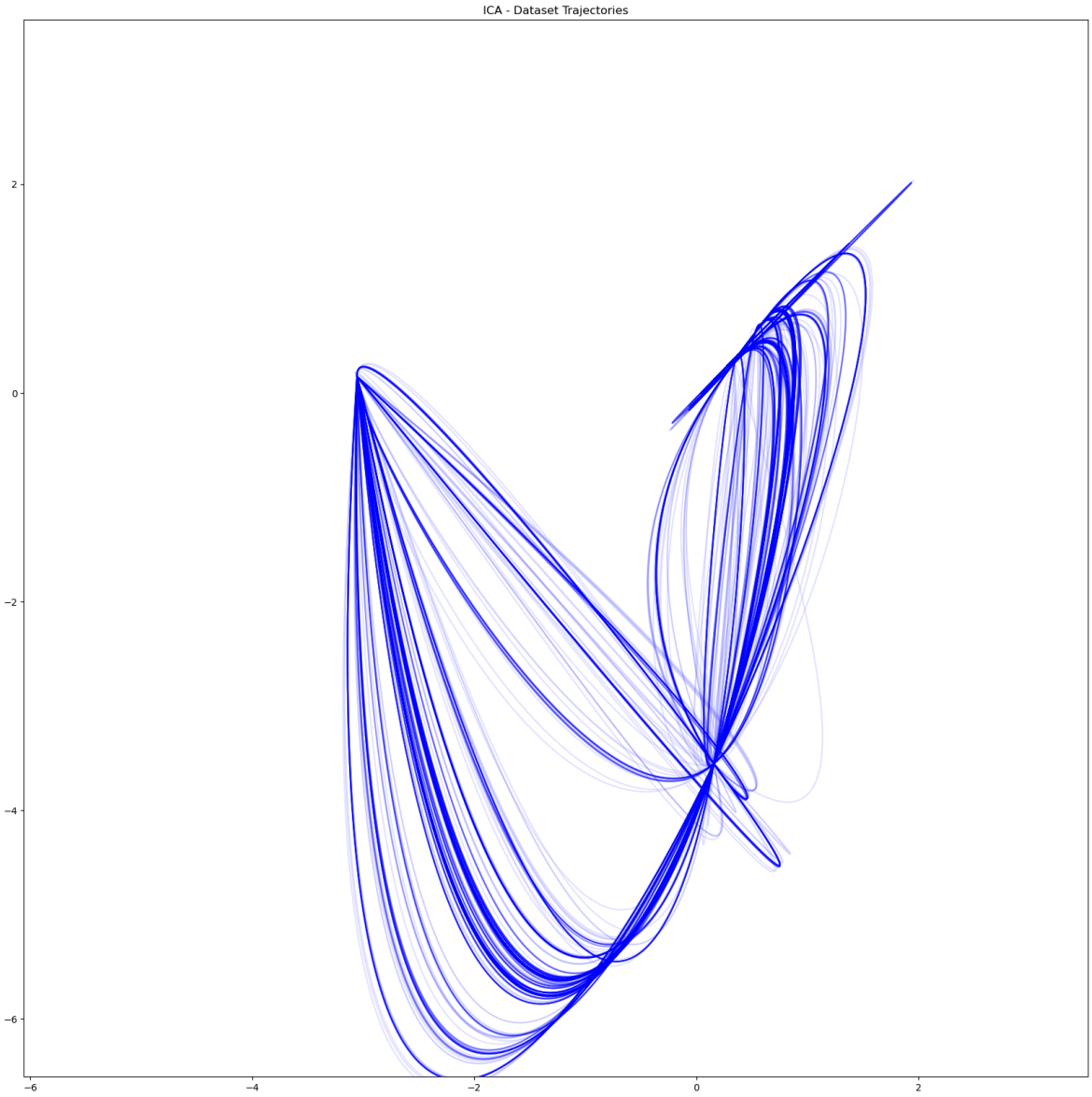
Observation PCA:

The PCA plot appears very different from the t-SNE and UMAP projections. Here, states are not spread out, and identical states are encoded consistently, resulting in more distinct trajectories. The plot reveals three main clusters, which align with our previous observations: the starting state, intermediate states, and end states. All algorithms seem to pass through these key regions, suggesting that PCA has preserved the principal structure of the trajectories while reducing complexity.

ICA

In [20]:
```
fig, ax = plt.subplots(figsize=(20, 20))
algorithms = ica_plotting_df['algorithm'].unique()

for i, algo in enumerate(algorithms):
    algo_data = ica_plotting_df[ica_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax, line_data, alpha=0.1, smoothing=0, n_points=9
```

```
ax.set_title("ICA - Dataset Trajectories")
margin = 3
plt.xlim(ica_plotting_df['X'].min() - margin, ica_plotting_df['X'].max()
plt.ylim(ica_plotting_df['Y'].min() - margin, ica_plotting_df['Y'].max()
plt.show()
```



Observation ICA:

As with PCA, the ICA plot reveals three key regions where all trajectories converge, indicating important areas in the state space. Without additional color encodings or labels, it's challenging to interpret specific patterns, but the resemblance to the PCA plot is noticeable. This similarity suggests that ICA, like PCA, captures the main structural components of the trajectories while maintaining consistent encoding for identical states.

Meta Data Encoding

Encode addtional features in the visualization.

Use features of the source data and include them in the projection, e.g., by using color, opacity, different shapes, or line styles, etc.

Comments

- Which features did you use? Why?
- How are the features encoded?

TODO
We tried out various features for our downprojection methods to illustrate a variety of approaches.
We worked with different color encodings to mark the 4 different algorithms. In addition, we encoded start, end, and intermediate steps b varying the shapes of the scattered points. Intermediate points are smaller and filled (to avoid overplotting) and start and end points get bigger shapes (circles and squares).

In the trajectory density cluster plots, each trajectory line represents the sequence of states visited by an agent under one of the four algorithms in the Cliff Walking environment. The following elements enhance the interpretability of recurring paths and high-density clusters:

1. **Density-Based Cluster Highlights**:
    - States that are frequently visited across different trajectories are identified as clusters using a density-based approach, specifically the DBSCAN algorithm.
    - These clusters are marked by enlarged, semi-transparent circles to represent the state density—larger and more opaque markers indicate a higher frequency of visits.
    - Text labels annotate each cluster with the respective state number and scale in size according to cluster density, allowing for quick recognition of high-traffic areas in the grid.
2. **Line Encoding for Trajectories**:
    - Each trajectory is represented by a line that connects the sequential states visited by the agent. The lines are semi-transparent, making paths shared by multiple algorithms more prominent as they become visually reinforced through overlap.
    - This approach highlights common routes among algorithms while still preserving individual paths. The low opacity of each line reduces visual clutter, especially in areas where multiple trajectories converge.

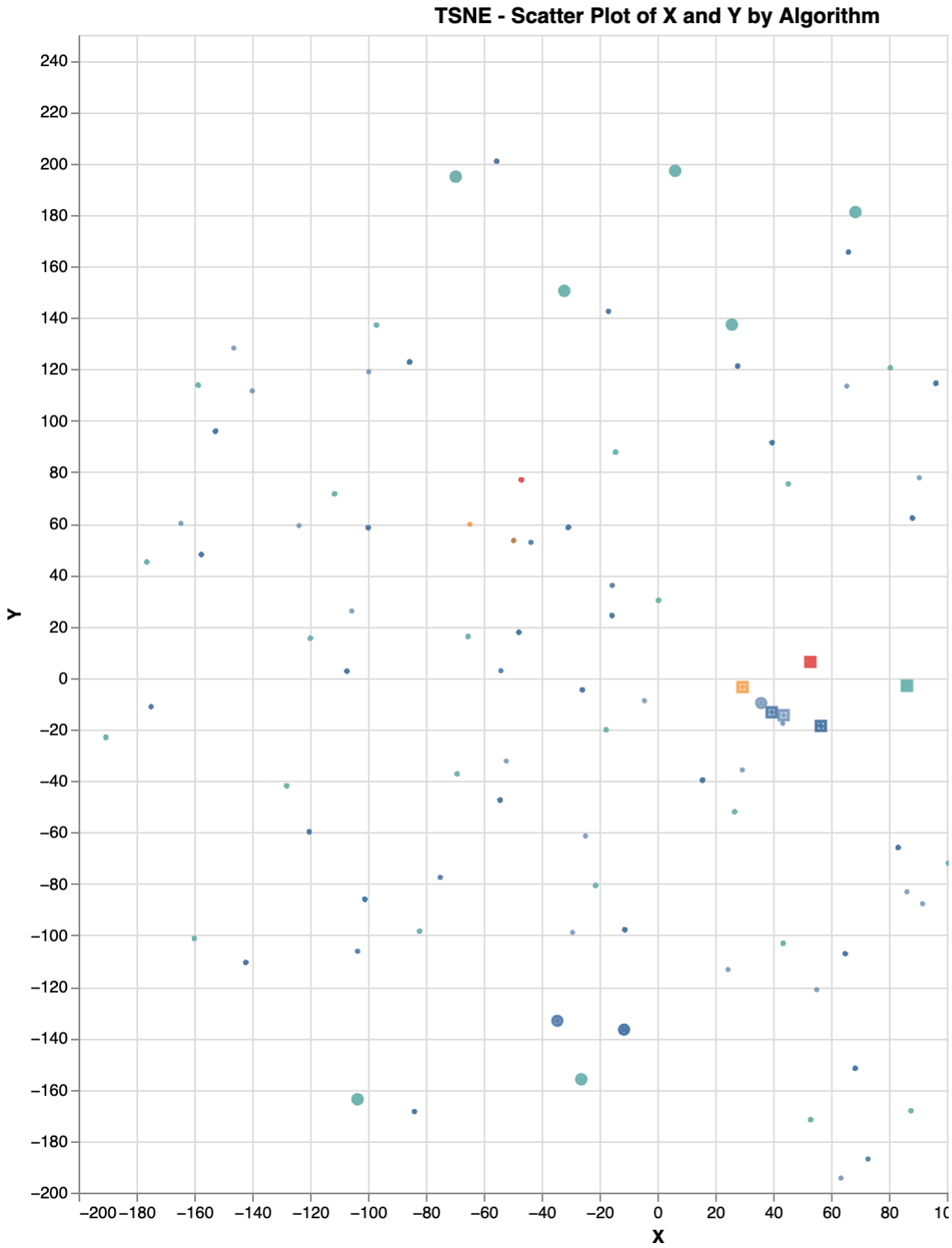
Through these density-based highlights and trajectory lines, the plot reveals both common pathways and high-visit states, making it easier to see how each algorithm behaves in the grid, particularly in terms of exploration versus exploitation.

TSNE

In [21]:
```
alt.Chart(plotting_df).mark_point(
    opacity=0.7,
    filled=True,  # Ensure points are filled
```

```
    size=10
).encode(
    x='X',
    y='Y',
    # draw one line per attempt, but ...
    color='algorithm:N', # .. color the lines per solving strategy
).properties(
    width=700,
    height=700,
    title="TSNE - Scatter Plot of X and Y by Algorithm"
) + alt.Chart(plotting_df).transform_filter(
    (datum.cp == 'end') | (datum.cp == 'start') # no intermediate states
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape='cp:N',
    color='algorithm:N', # .. color the lines per solving strategy
).properties(
    width=700,
    height=700
)
```

Out[21]:

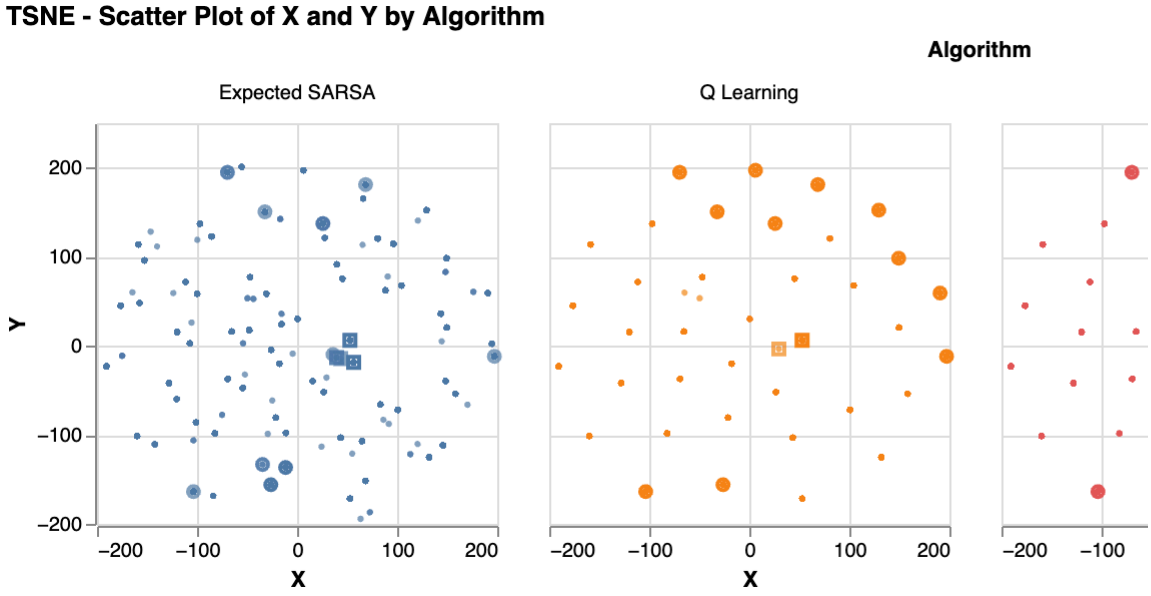


In [22]:

```
# Base chart for intermediate states
base = alt.Chart(plotting_df).mark_point(
    opacity=0.7,
    filled=True,  # Ensure points are filled
    size=10
).encode(
    x='X',
    y='Y',
    color='algorithm:N'
).properties(
    width=200,
    height=200
)
```

```
# Chart for start and end states only
start_end = alt.Chart(plotting_df).transform_filter(
    (alt.datum.cp == 'end') | (alt.datum.cp == 'start')
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape=alt.Shape('cp:N', legend=alt.Legend(title="State")),
    color='algorithm:N'
).properties(
    width=200,
    height=200
)

# Combine both charts and apply faceting to arrange by algorithm in a row
chart = (base + start_end).facet(
    facet=alt.Facet('algorithm:N', title="Algorithm"),
    columns=len(plotting_df['algorithm'].unique())
).properties(
    title="TSNE - Scatter Plot of X and Y by Algorithm"
)

chart
```

Out[22]:

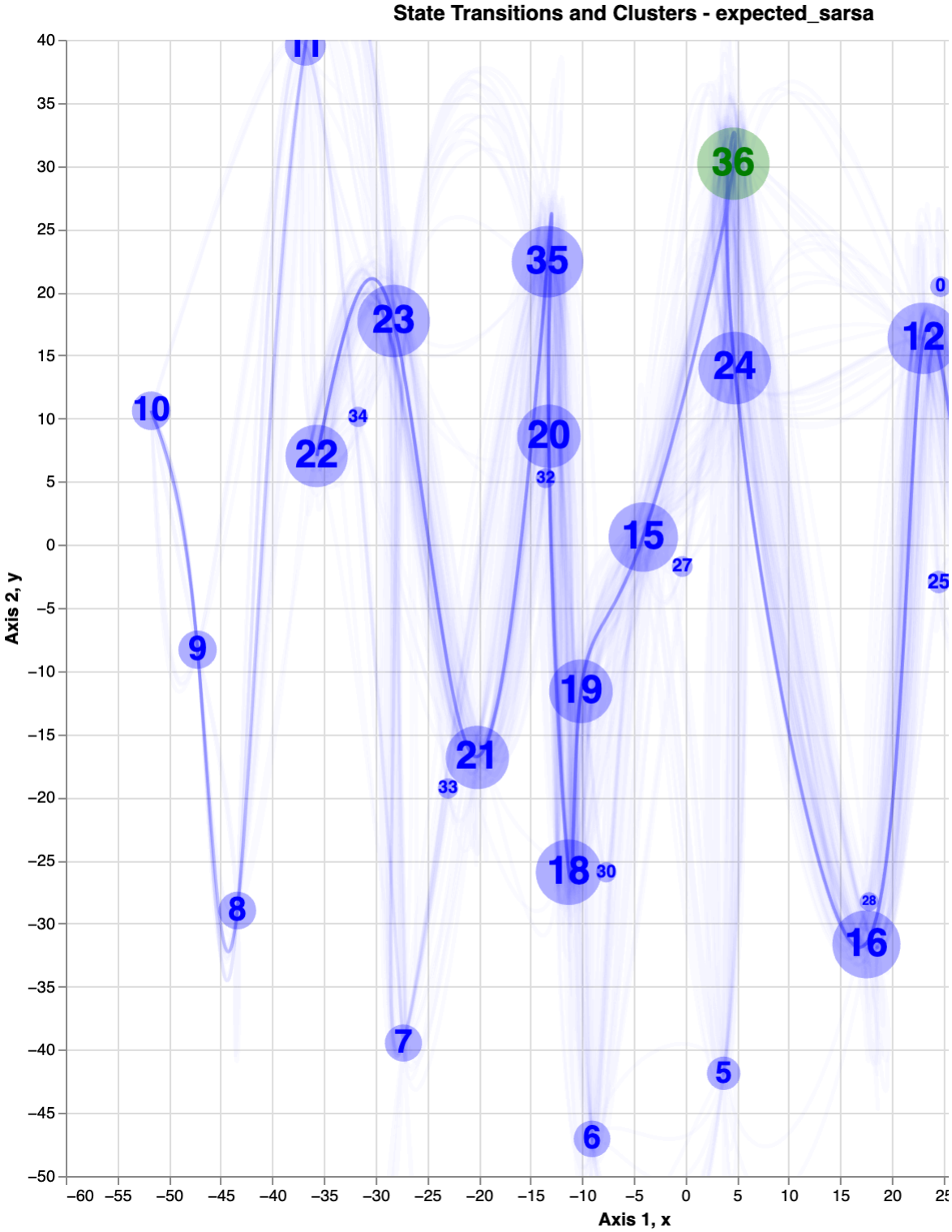


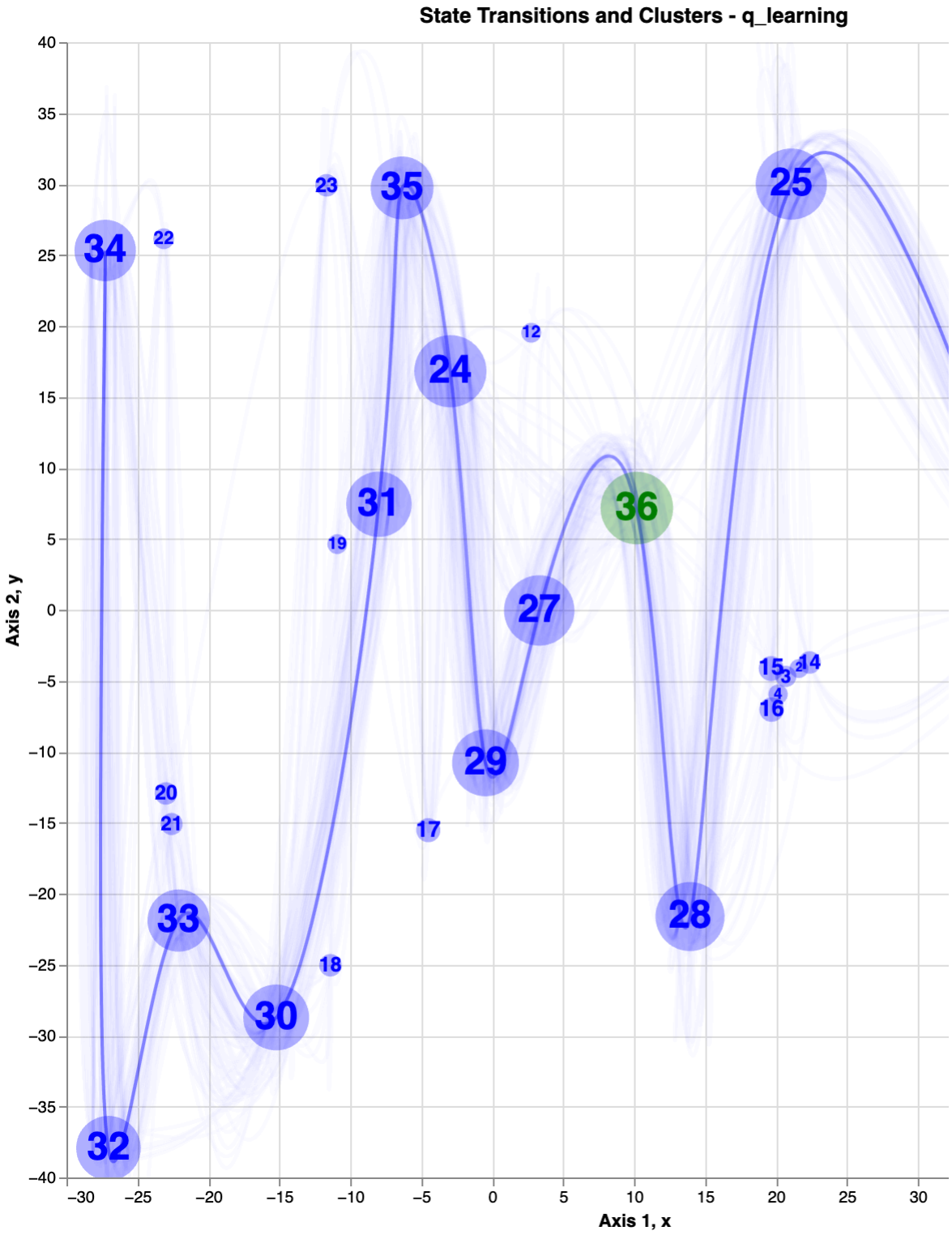
In [23]:

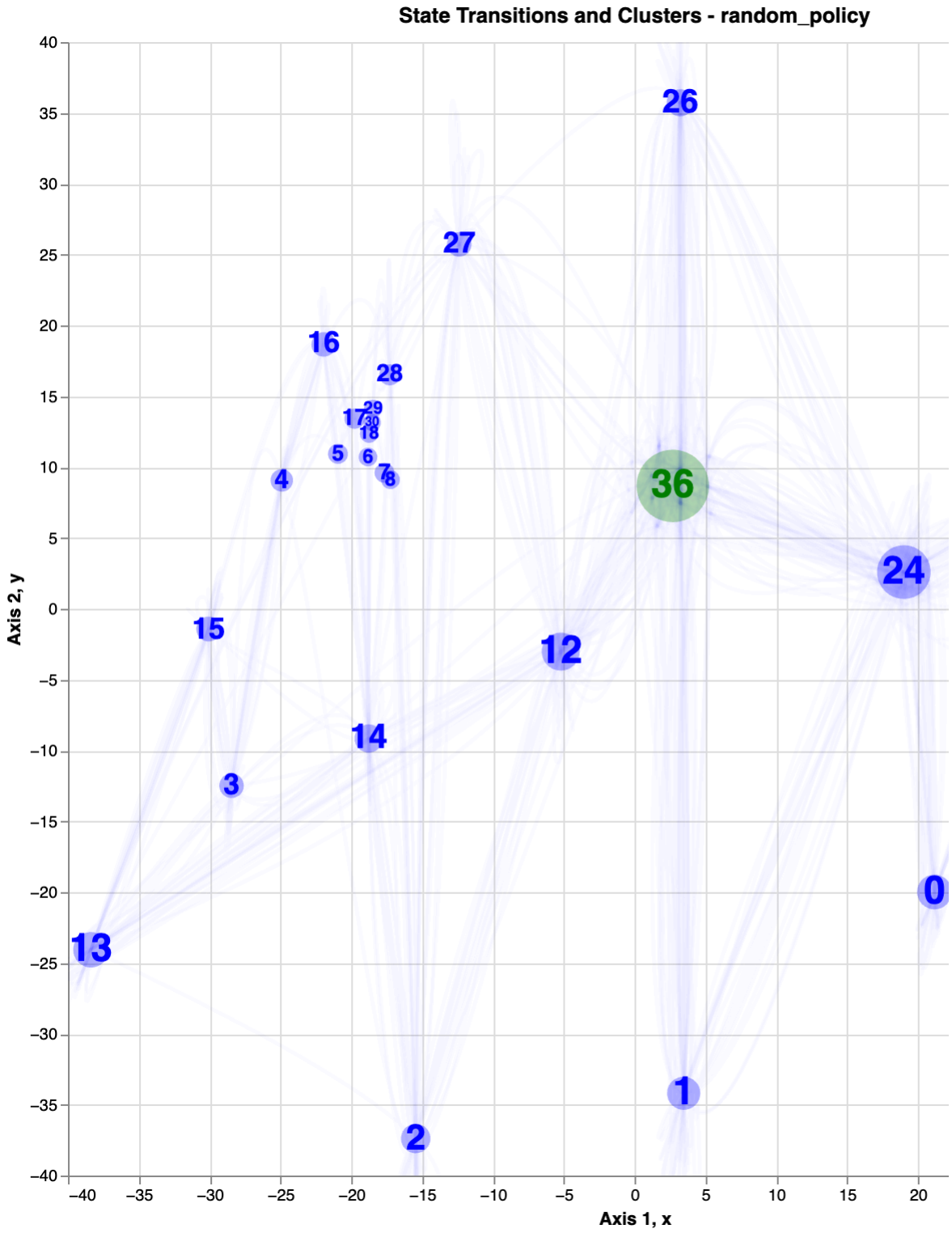
```
expected_sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='expected_sarsa', encoding_type='one-hot', down_projec
)

q_learning_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='q_learning', encoding_type='one-hot', down_project_me
)

random_policy_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='random_policy', encoding_type='one-hot', down_project
)

sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='sarsa', encoding_type='one-hot', down_project_method=
)

expected_sarsa_connected_density_plot.display()
q_learning_connected_density_plot.display()
```

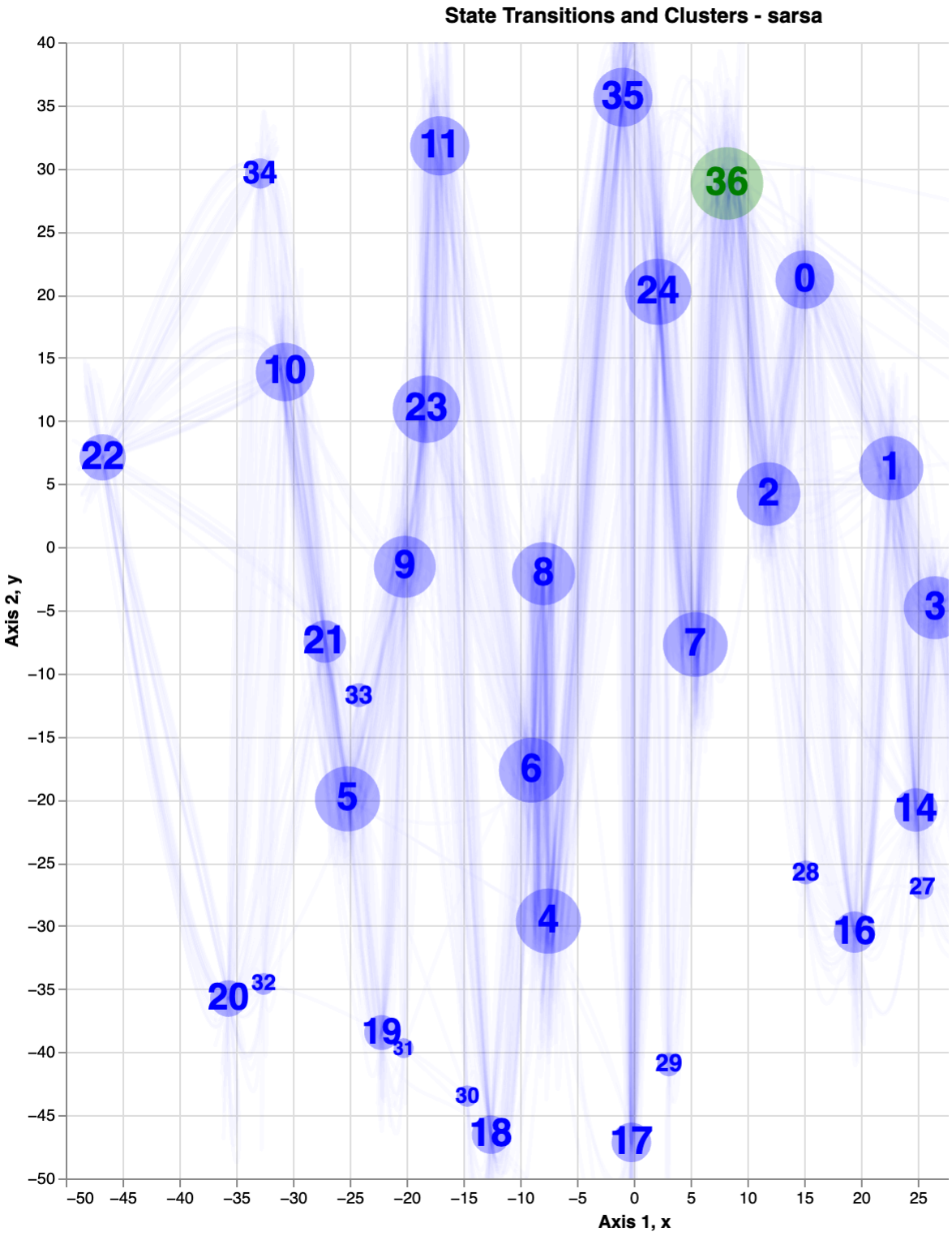
```
random_policy_connected_density_plot.display()
sarsa_connected_density_plot.display()
```

```
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
```




State Transitions and Clusters - q_learning
Axis 1, x
Axis 2, y
40
35
30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−35
−40
−30
−25
−20
−15
−10
−5
0
5
10
15
20
25
30
34
22
23
35
25
12
24
31
36
19
27
15
3
2
14
4
16
29
20
21
17
33
28
18
30
32


State Transitions and Clusters - random_policy
Axis 2, y
Axis 1, x
40
35
30
25
20
15
10
5
0
−5
−10
−15
−20
−25
−30
−35
−40
−40
−35
−30
−25
−20
−15
−10
−5
0
5
10
15
20
26
27
16
28
29
17
30
18
5
6
7
8
4
36
24
15
12
14
3
0
13
1
2



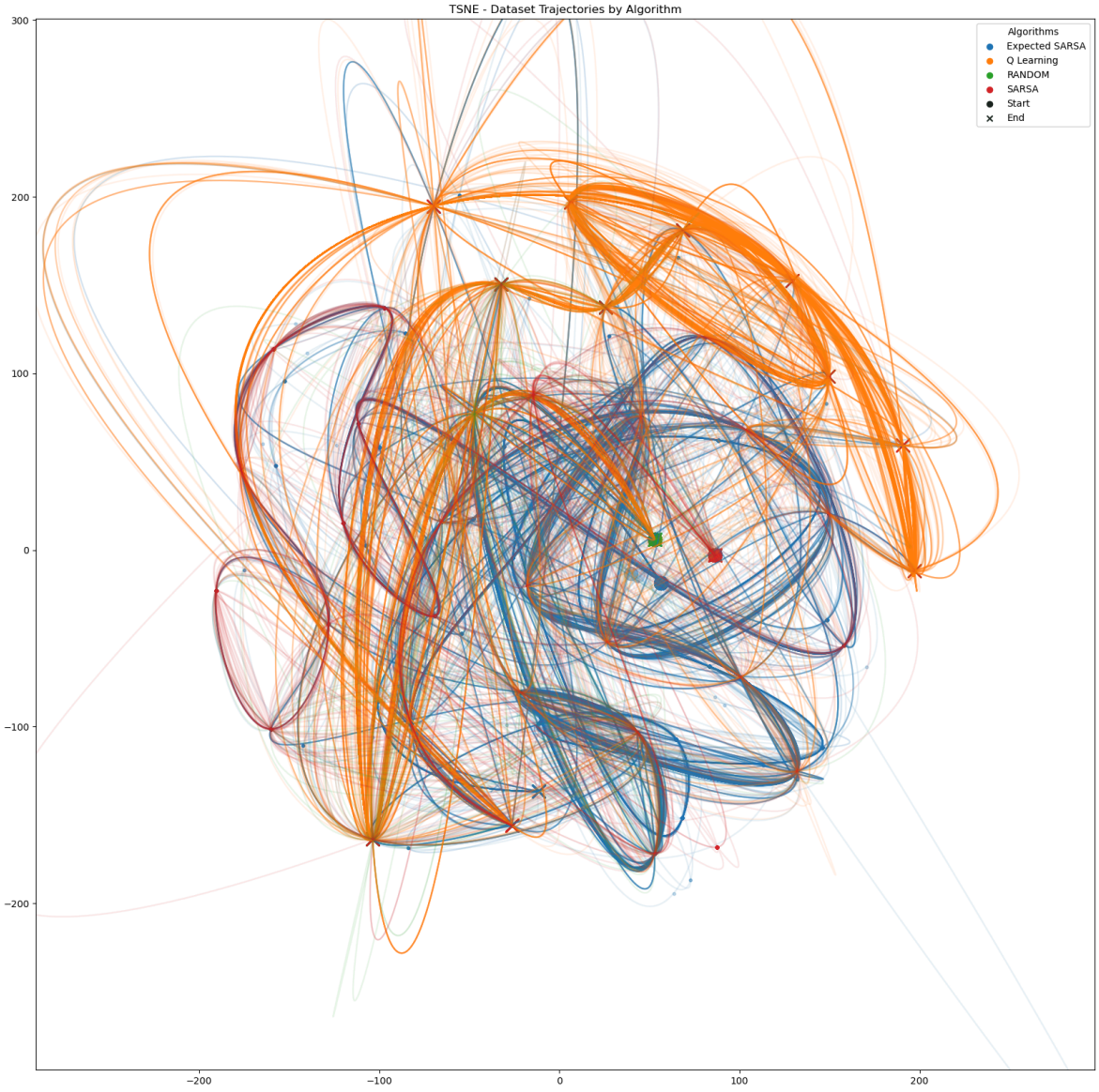
Observation t-SNE:

In the first scatter plot, we see fewer points for Q-learning and random policies, likely due to the limited number of states and slight variations in encoding, resulting in many values occupying the same space. The second plot reveals insights into the agent's behavior: Q-learning, Random, and SARSA policies exhibit numerous end points, indicating frequent mistakes. Expected SARSA, however, performs better at avoiding the cliff, as fewer end points are prominent, suggesting a more cautious approach.

In the four plots displaying state transitions and clusters, we gain further understanding of state prominence and distribution. The behavior of each policy is

distinct. The starting state is consistently the most prominent. For the random policy, there are few frequently revisited states, which aligns with its lack of learning. Expected SARSA frequently visits states toward the middle, maintaining a balance between safety and efficiency. Q-learning shows a tendency to revisit states near the cliff, reflecting a riskier strategy. SARSA, in contrast, takes a path that is farthest from the cliff, favoring a longer but safer trajectory. This behavior is explained in more detail below.

In [24]:
```
# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = plotting_df[plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Mark start and end points
for i, algo in enumerate(algorithms):
    start_data = plotting_df[(plotting_df['cp'] == 'start') & (plotting_d
    #checkpoint_data = tsne_algo1[(tsne_algo1['cp'] == ' checkpoint') & (
    intermediate_data = plotting_df[(plotting_df['cp'] == 'intermediate')
    end_data = plotting_df[(plotting_df['cp'] == 'end') & (plotting_df['a

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    #ax.scatter(checkpoint_data['X'], checkpoint_data['Y'], color=colors[
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)  # Empty scatter for

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("TSNE - Dataset Trajectories by Algorithm")
ax.legend(title="Algorithms", loc="best")

margin = 100
plt.xlim(plotting_df['X'].min() - margin, plotting_df['X'].max() + margin
plt.ylim(plotting_df['Y'].min() - margin, plotting_df['Y'].max() + margin
plt.show()
```



Observation t-SNE:

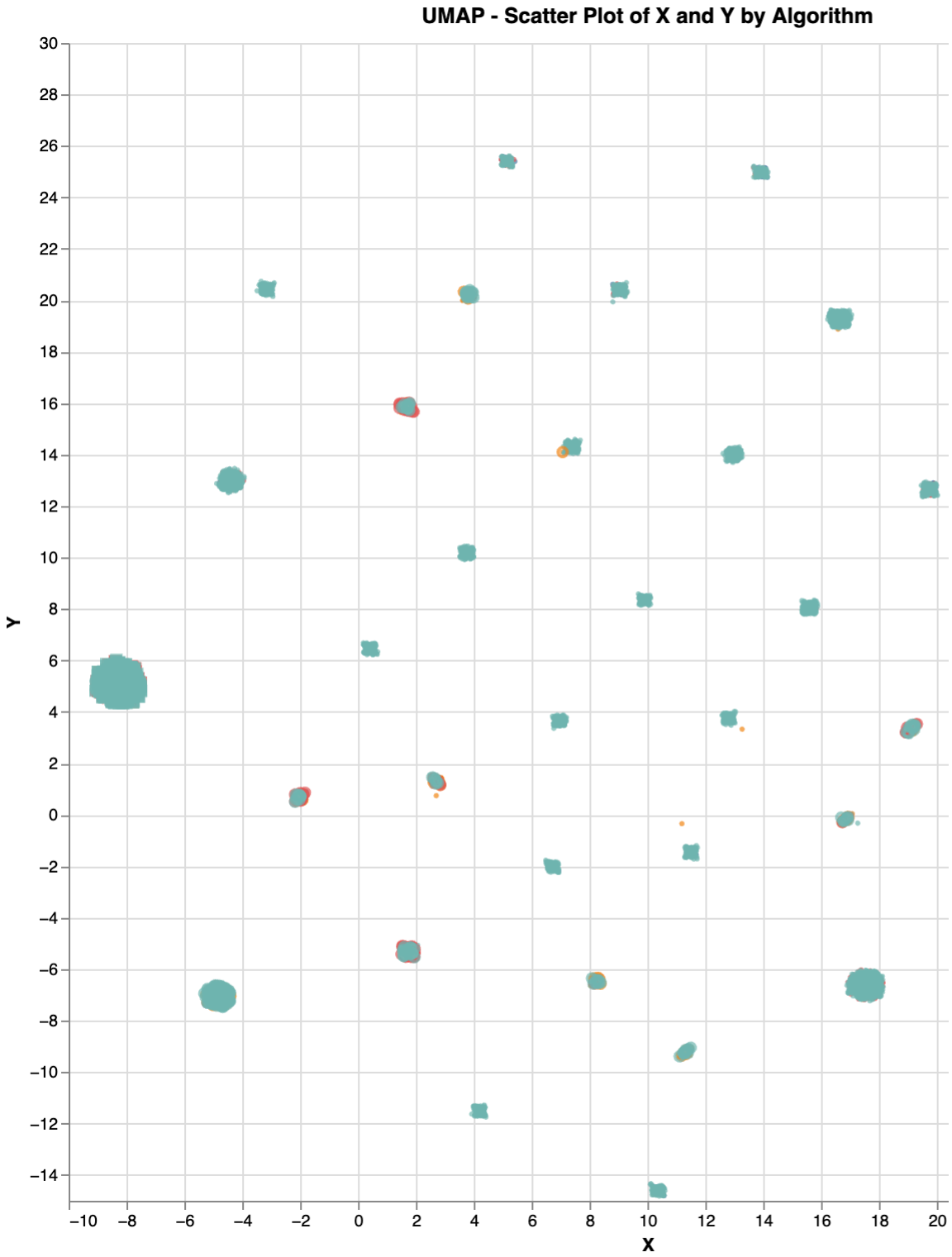
With color encodings by algorithm and special markers (start='o' and end='X'), the differences in behavior among the algorithms become more evident. The trajectories for Q-learning and Expected SARSA are particularly prominent. Although SARSA also displays visible paths, the algorithms traverse different trajectories, as expected given their distinct behaviors. Even without clustering the data or knowing the specific states, we observe that Q-learning trajectories frequently pass through end points and are notably concentrated near the cliff, reflecting a higher risk approach. The random policy, on the other hand, produces less frequent, non-repetitive paths, resulting in faint green lines in the background due to the lack of intentional structure or repeated trajectories.

UMAP

In [25]:
```
alt.Chart(umap_plotting_df).mark_point(
    opacity=0.7,
    filled=True,  #
    size=10
).encode(
    x='X',
```

```
    y='Y',
    color='algorithm:N',
).properties(
    width=700,
    height=700,
    title="UMAP - Scatter Plot of X and Y by Algorithm"
) + alt.Chart(umap_plotting_df).transform_filter(
    (datum.cp == 'end') | (datum.cp == 'start')
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape='cp:N',
    color='algorithm:N'
).properties(
    width=700,
    height=700
)
```

Out[25]:

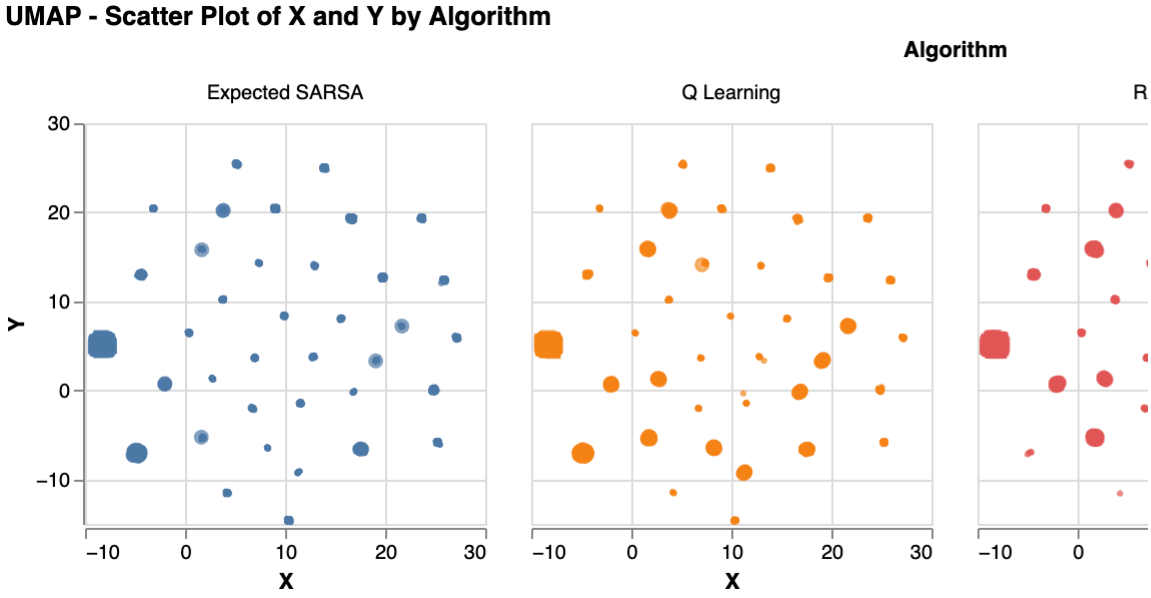


In [26]:

```
# Graph for intermediate state
base = alt.Chart(umap_plotting_df).mark_point(
    opacity=0.7,
    filled=True,
    size=10
).encode(
    x='X',
    y='Y',
    color='algorithm:N'
).properties(
    width=200,
    height=200
)
```

```
# Graph for starting/ending point
start_end = alt.Chart(umap_plotting_df).transform_filter(
    (alt.datum.cp == 'end') | (alt.datum.cp == 'start')
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape=alt.Shape('cp:N', legend=alt.Legend(title="State")),
    color='algorithm:N'
).properties(
    width=200,
    height=200
)

chart = (base + start_end).facet(
    facet=alt.Facet('algorithm:N', title="Algorithm"),
    columns=len(umap_plotting_df['algorithm'].unique())
).properties(
    title="UMAP - Scatter Plot of X and Y by Algorithm"
)

chart
```

Out[26]:

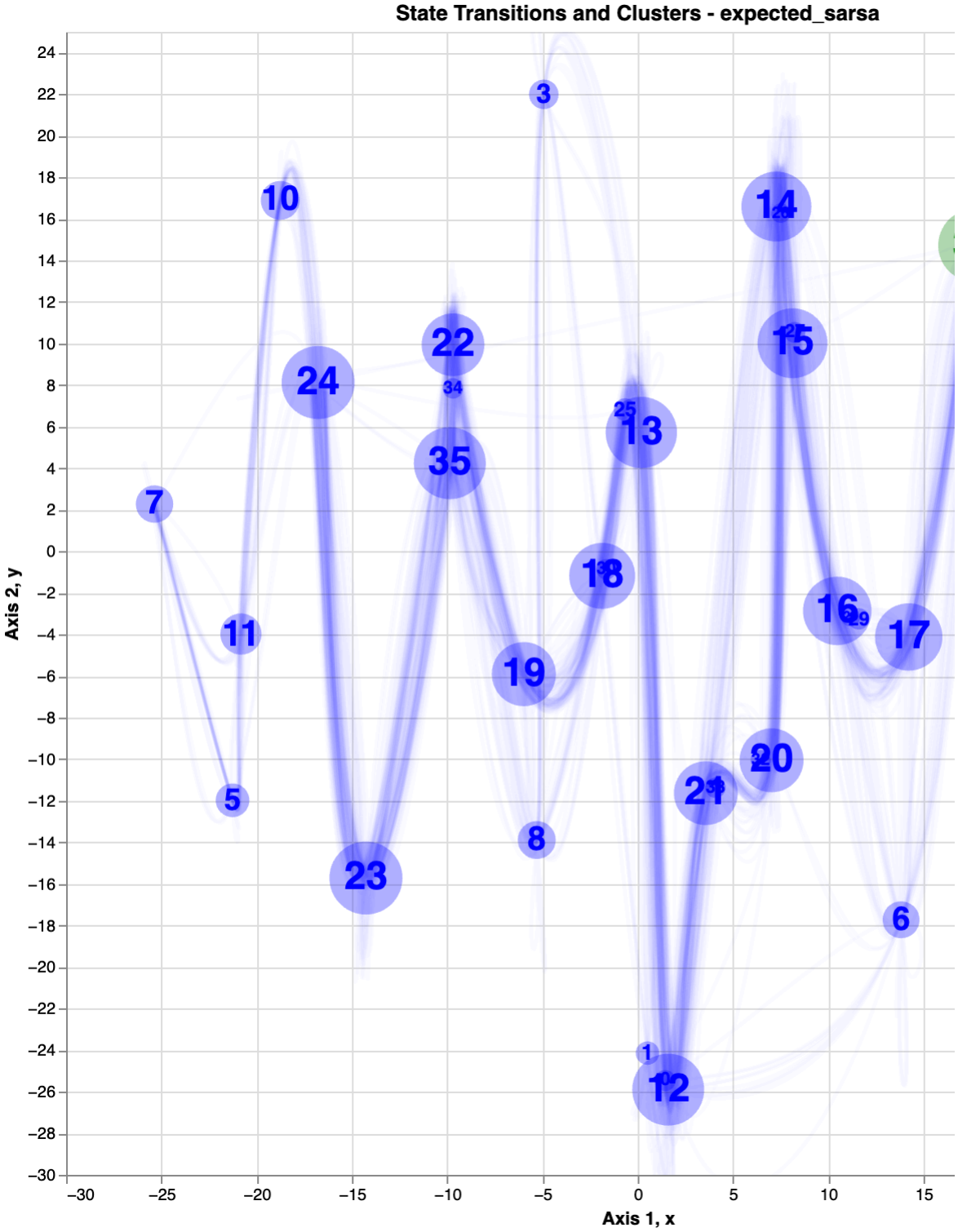


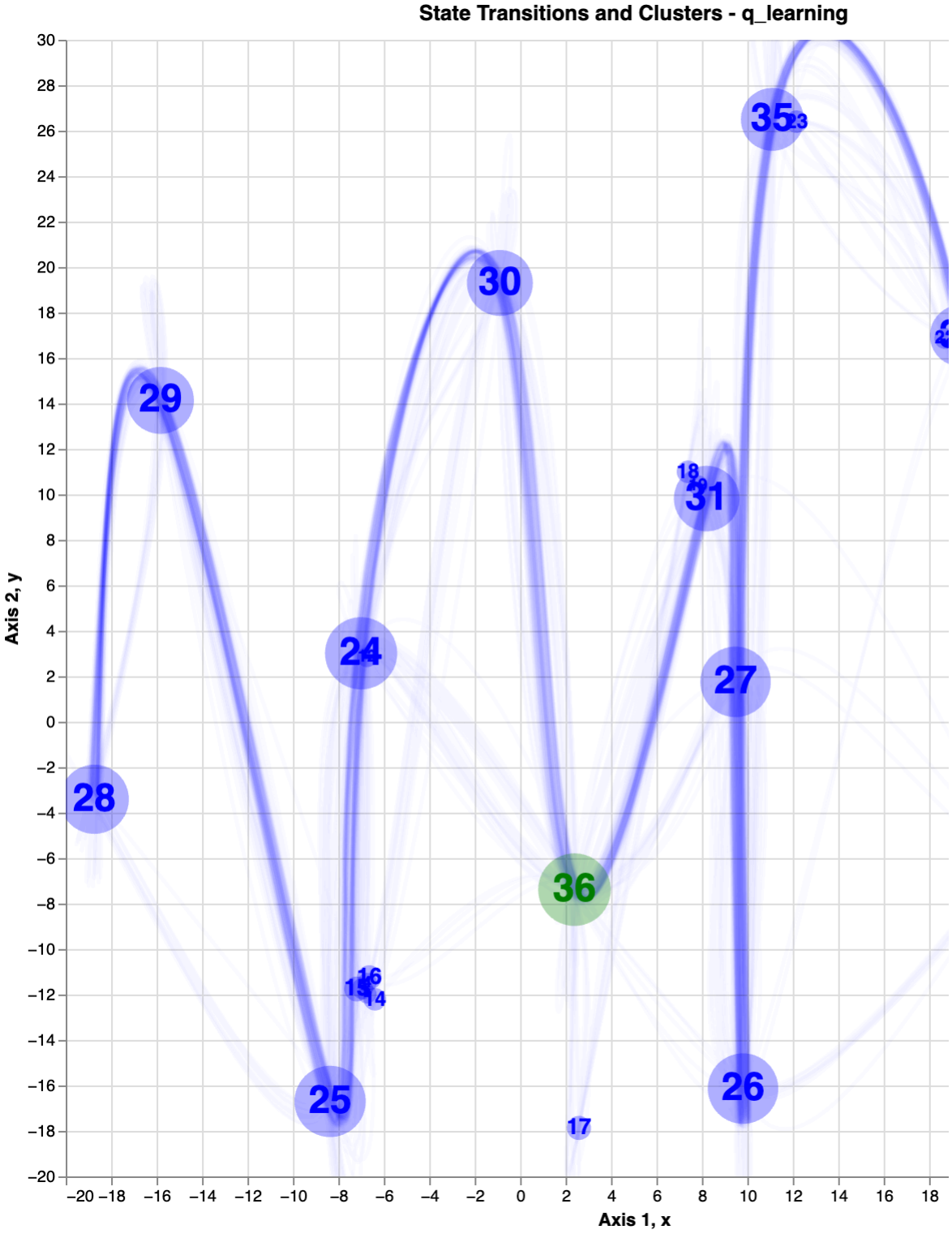
In [27]:

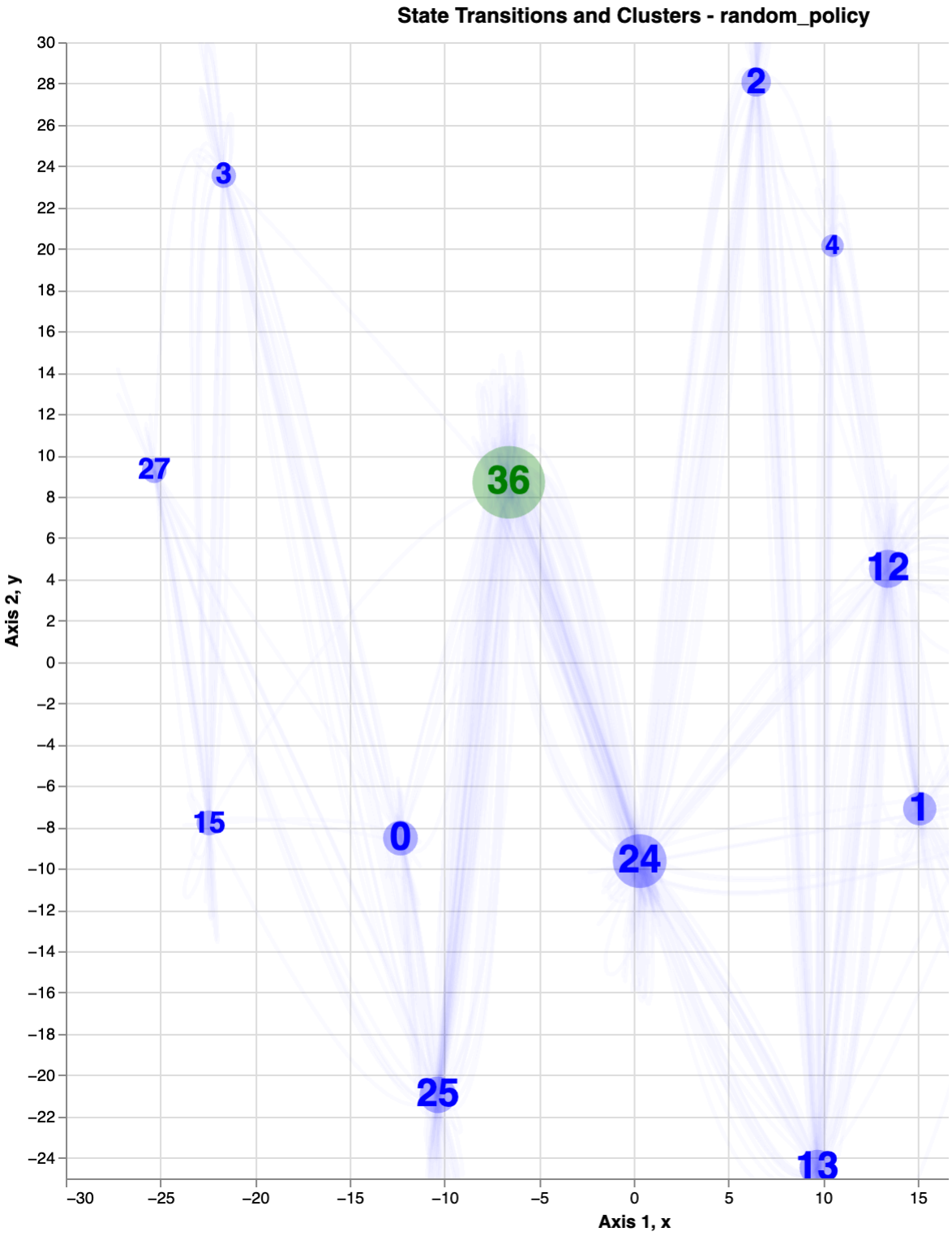
```
expected_sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='expected_sarsa', encoding_type='one-hot', down_projec
)

q_learning_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='q_learning', encoding_type='one-hot', down_project_me
)

random_policy_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='random_policy', encoding_type='one-hot', down_project
)

sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='sarsa', encoding_type='one-hot', down_project_method=
)

expected_sarsa_connected_density_plot.display()
q_learning_connected_density_plot.display()
```

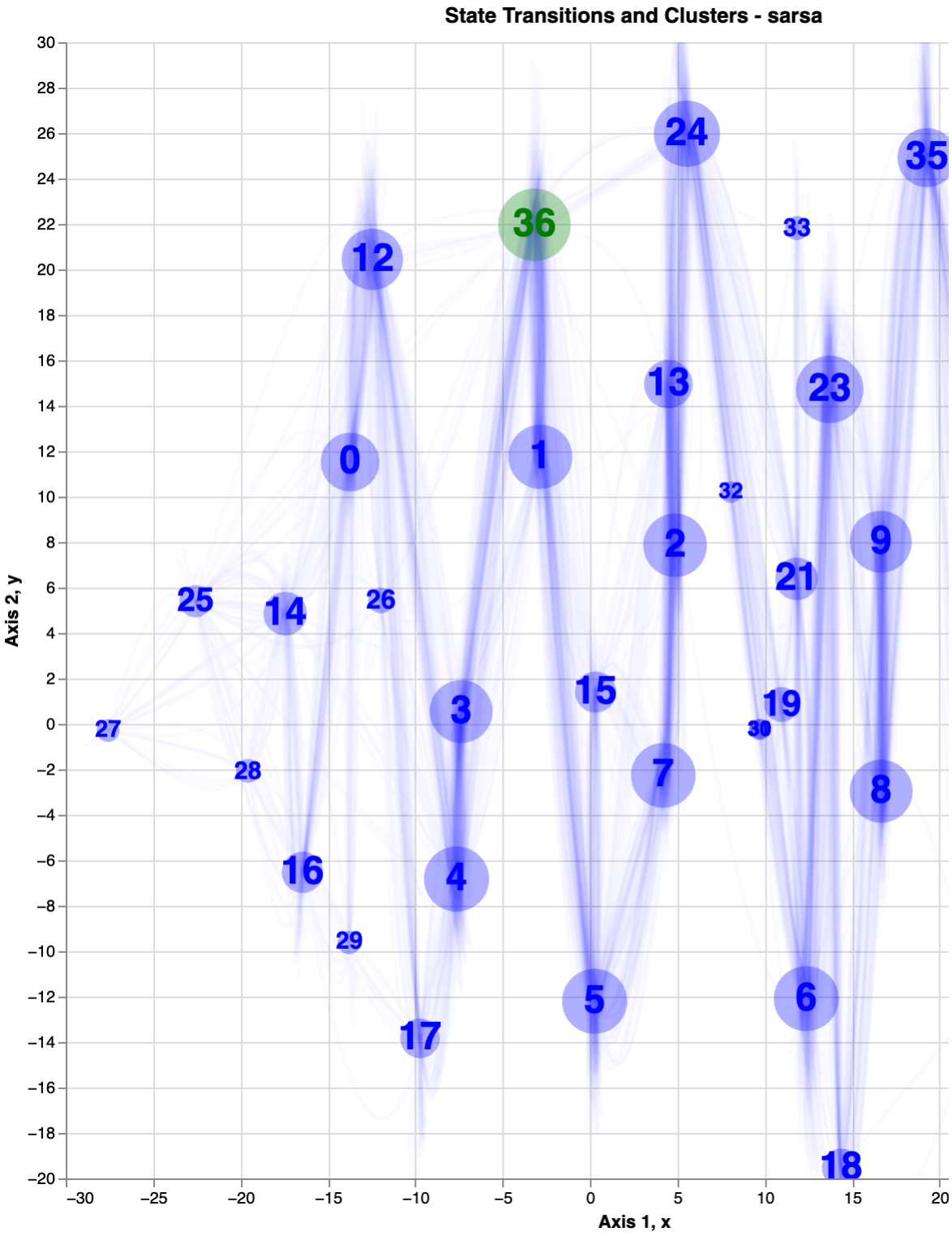
```
random_policy_connected_density_plot.display()
sarsa_connected_density_plot.display()
```

```
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
```




State Transitions and Clusters - q_learning
Axis 2, y
Axis 1, x
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
−2
−4
−6
−8
−10
−12
−14
−16
−18
−20
−20 −18 −16 −14 −12 −10 −8 −6 −4 −2 0 2 4 6 8 10 12 14 16 18
35
23
30
29
18
31
24
27
28
36
16
14
25
17
26

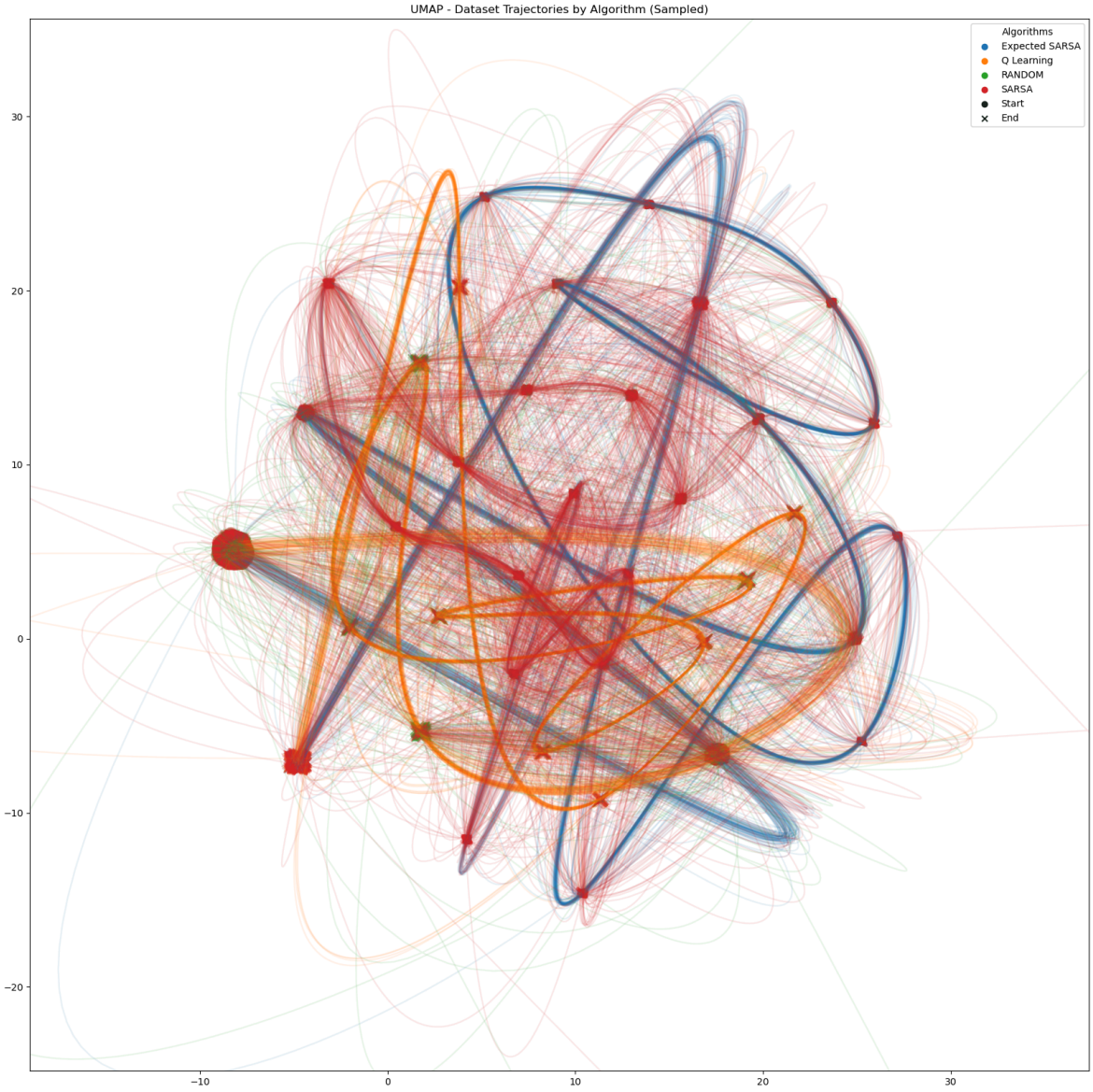

State Transitions and Clusters - random_policy
Axis 2, y
Axis 1, x
30
28
26
24
22
20
18
16
14
12
10
8
6
4
2
0
−2
−4
−6
−8
−10
−12
−14
−16
−18
−20
−22
−24
−30
−25
−20
−15
−10
−5
0
5
10
15
2
3
4
27
36
12
15
0
24
1
25
13



Observation UMAP:

Similar to t-SNE, UMAP does not distinctly separate unique states by algorithm based solely on color coding. However, by examining the scatter plots of individual algorithms, we can observe that the distribution of clusters varies. The SARSA, Q-learning, and Random policies display similar cluster patterns, with points spread across various states. Expected SARSA, in contrast, has fewer clusters with points densely packed in specific areas, suggesting a more consistent set of preferred states and potentially more stable trajectories. This clustering behavior highlights Expected SARSA's tendency to avoid the cliff, as it appears to focus on a narrower range of safer states.

In [28]:
```
# Defining colors
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Creating the plot
fig, ax = plt.subplots(figsize=(20, 20))

# Sampling the 10%
sampling_fraction = 0.1

# Loop overall the algorithms
for i, algo in enumerate(algorithms):
    algo_data = umap_plotting_df[umap_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    # Sampling
    sampled_lines = np.random.choice(lines, size=int(len(lines) * samplin

    for line in sampled_lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Start/End points
for i, algo in enumerate(algorithms):
    start_data = umap_plotting_df[(umap_plotting_df['cp'] == 'start') & (
    intermediate_data = umap_plotting_df[(umap_plotting_df['cp'] == 'inte
    end_data = umap_plotting_df[(umap_plotting_df['cp'] == 'end') & (umap

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("UMAP - Dataset Trajectories by Algorithm (Sampled)")
ax.legend(title="Algorithms", loc="best")

margin = 10
plt.xlim(umap_plotting_df['X'].min() - margin, umap_plotting_df['X'].max(
plt.ylim(umap_plotting_df['Y'].min() - margin, umap_plotting_df['Y'].max(
plt.show()
```



Observation UMAP:

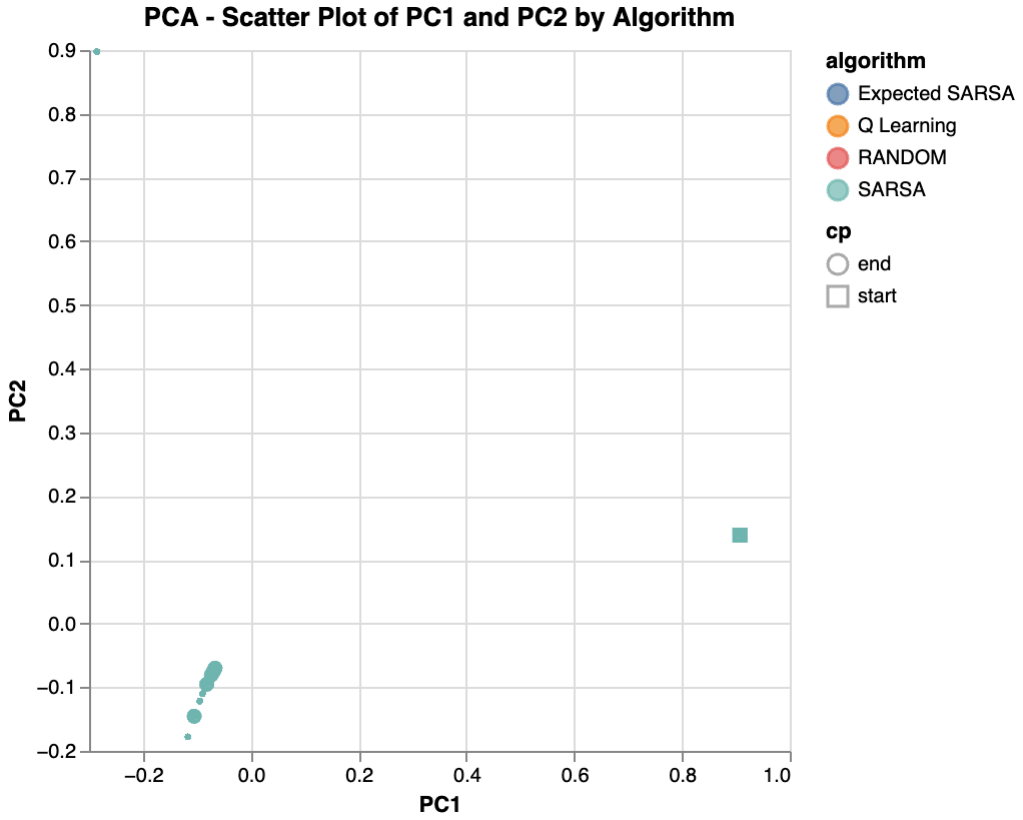
Also here in the spline plot, we see very similar patterns to t-SNE. The detected bundles correspond the the trajectories of the three policies (the random policy has naturally no intentional bundles as same trajectories are not frequent due to the random action selection). Again, Q-learning passes through lots of different endpoints near the cliff, indicating the preference for selecting this shortest path with more risk. Moreover, we can also see some intersections at encoded starting states where all the trajectories start/pass through.

PCA

In [29]:
```
import altair as alt
from altair import datum

# Scatter plot of PC1 and PC2 by Algorithm
pca_chart = alt.Chart(pca_plotting_df).mark_point(
    opacity=0.7,
    filled=True,  #
    size=10
).encode(
    x='PC1',
```

```
    y='PC2',
    color='algorithm:N',
).properties(
    width=350,
    height=350,
    title="PCA - Scatter Plot of PC1 and PC2 by Algorithm"
) + alt.Chart(pca_plotting_df).transform_filter(
    (datum.cp == 'end') | (datum.cp == 'start')
).mark_point(filled=False).encode(
    x='PC1',
    y='PC2',
    shape='cp:N',
    color='algorithm:N',
).properties(
    width=350,
    height=350
)

pca_chart
```

Out[29]:

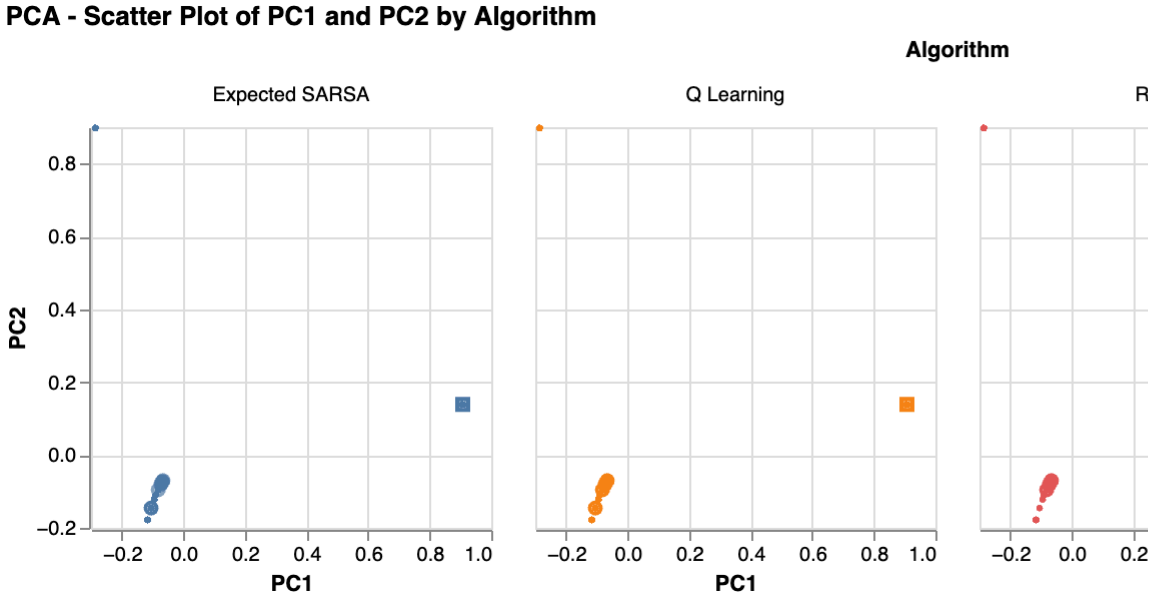


In [30]:

```
# Base chart for intermediate states
base_pca = alt.Chart(pca_plotting_df).mark_point(
    opacity=0.7,
    filled=True,
    size=10
).encode(
    x='PC1',
    y='PC2',
    color='algorithm:N'
).properties(
    width=200,
    height=200
)

# Chart for start and end states only
```

```
start_end_pca = alt.Chart(pca_plotting_df).transform_filter(
    (alt.datum.cp == 'end') | (alt.datum.cp == 'start')
).mark_point(filled=False).encode(
    x='PC1',
    y='PC2',
    shape=alt.Shape('cp:N', legend=alt.Legend(title="State")),
    color='algorithm:N'
).properties(
    width=200,
    height=200
)

# Combine both charts and apply faceting
pca_facet_chart = (base_pca + start_end_pca).facet(
    facet=alt.Facet('algorithm:N', title="Algorithm"),
    columns=len(pca_plotting_df['algorithm'].unique())
).properties(
    title="PCA - Scatter Plot of PC1 and PC2 by Algorithm"
)

pca_facet_chart
```

Out[30]:

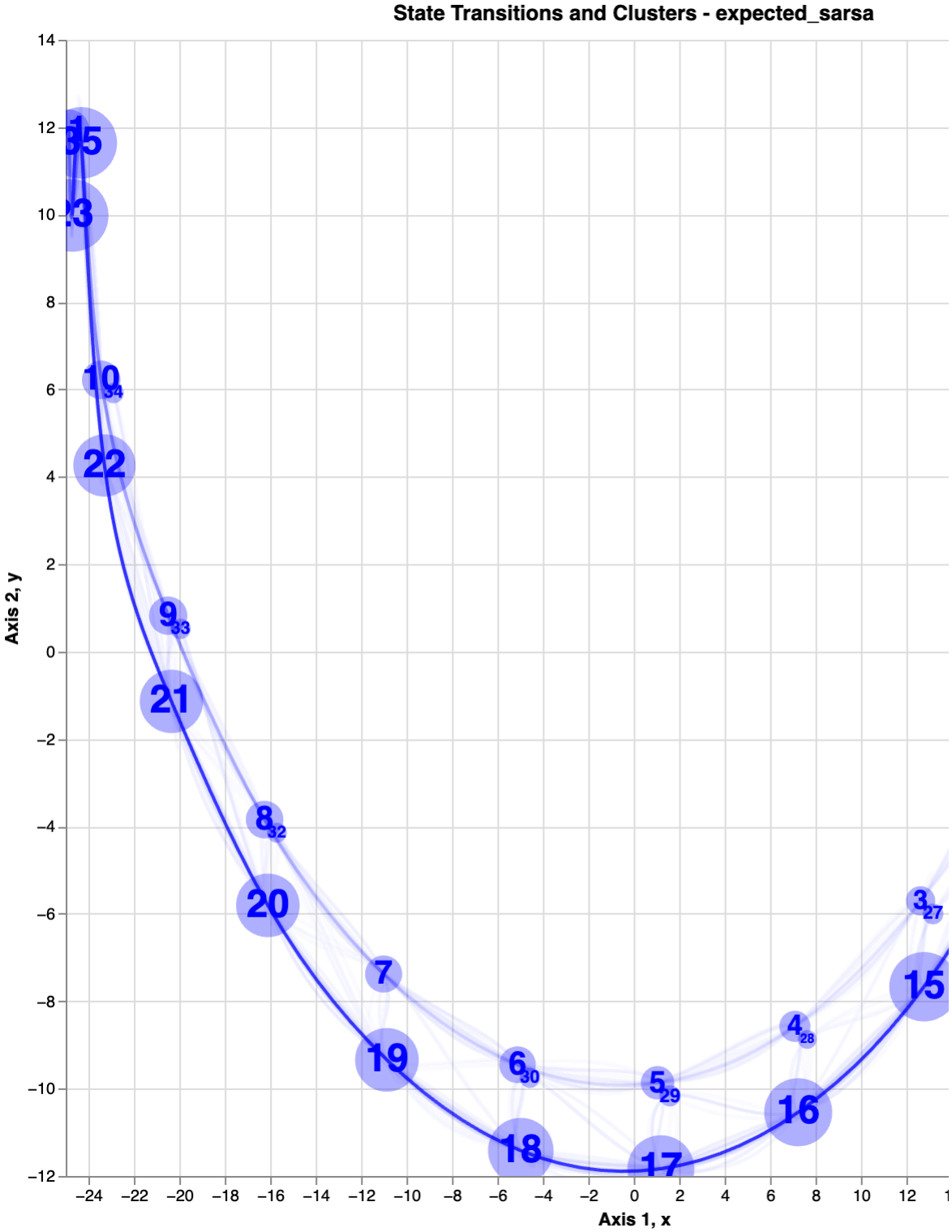


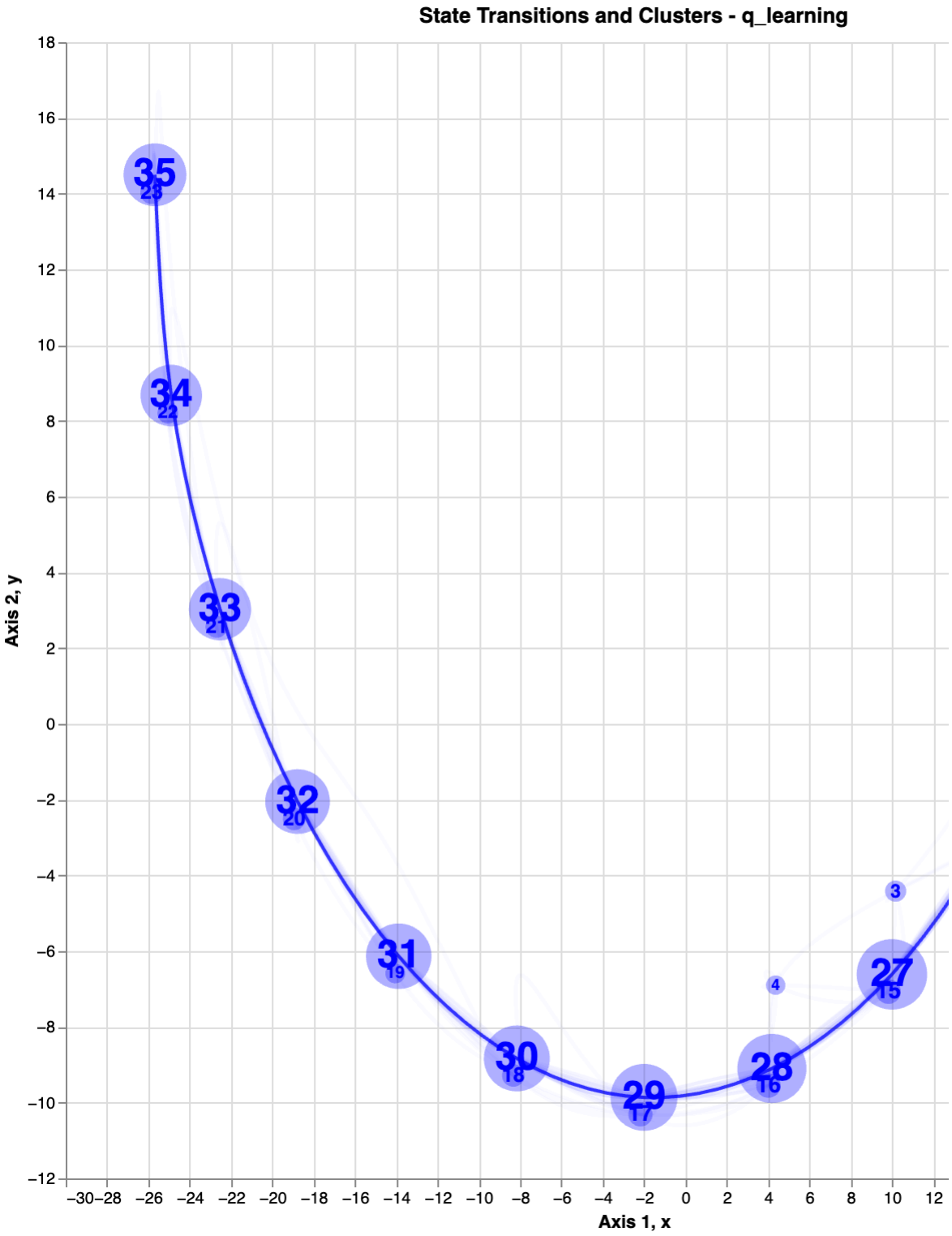
In [31]:

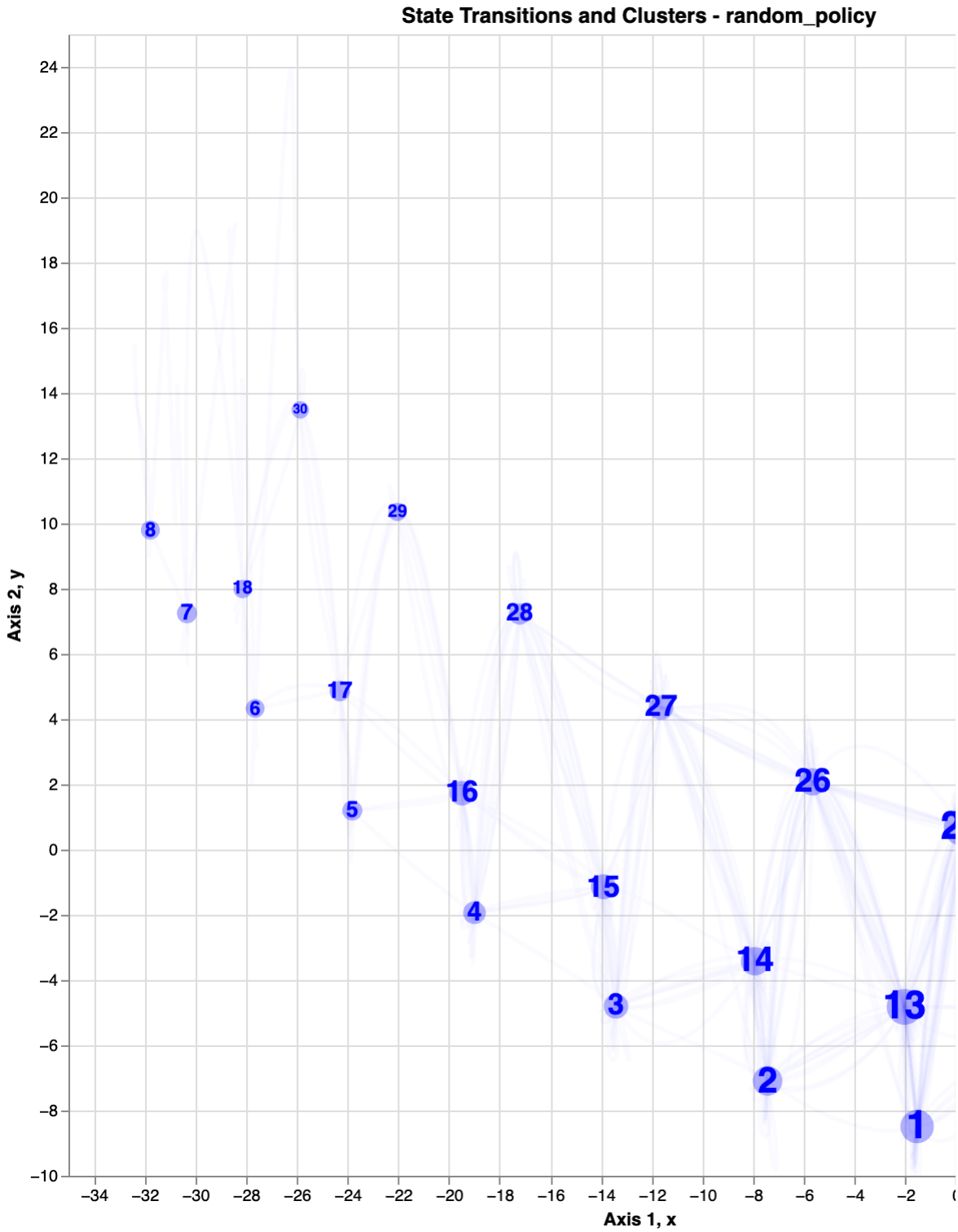
```
expected_sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='expected_sarsa', encoding_type='one-hot', down_projec
)

q_learning_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='q_learning', encoding_type='one-hot', down_project_me
)

random_policy_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='random_policy', encoding_type='one-hot', down_project
)

sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='sarsa', encoding_type='one-hot', down_project_method=
)

expected_sarsa_connected_density_plot.display()
q_learning_connected_density_plot.display()
```

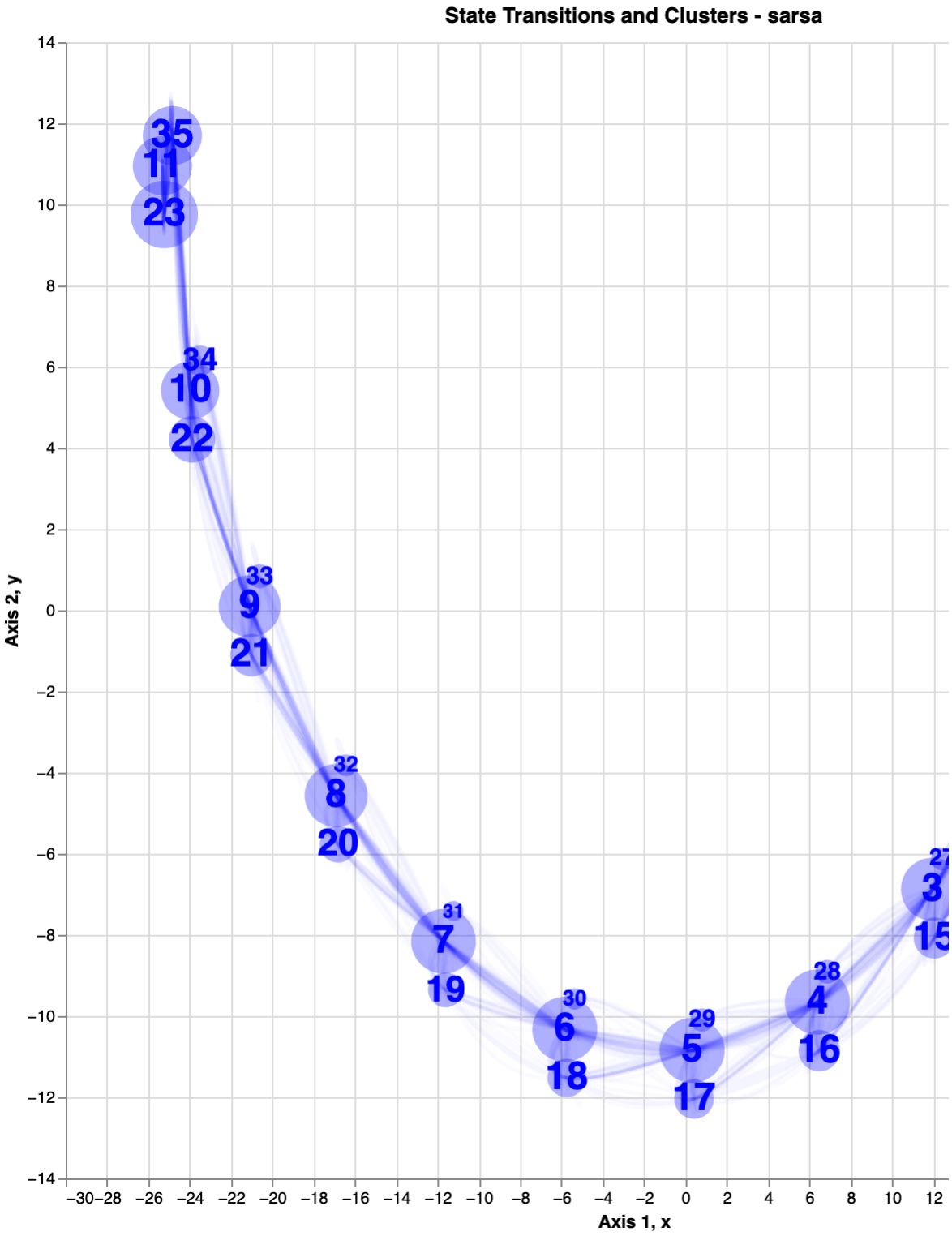
```
random_policy_connected_density_plot.display()
sarsa_connected_density_plot.display()
```

```
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
```




State Transitions and Clusters - q_learning
Axis 2, y
Axis 1, x
35
23
34
22
33
21
32
20
31
19
30
18
29
17
28
16
27
15
4
3


State Transitions and Clusters - random_policy
Axis 2, y
Axis 1, x
30
29
8
18
7
28
17
6
27
26
16
5
15
4
14
3
13
2
1



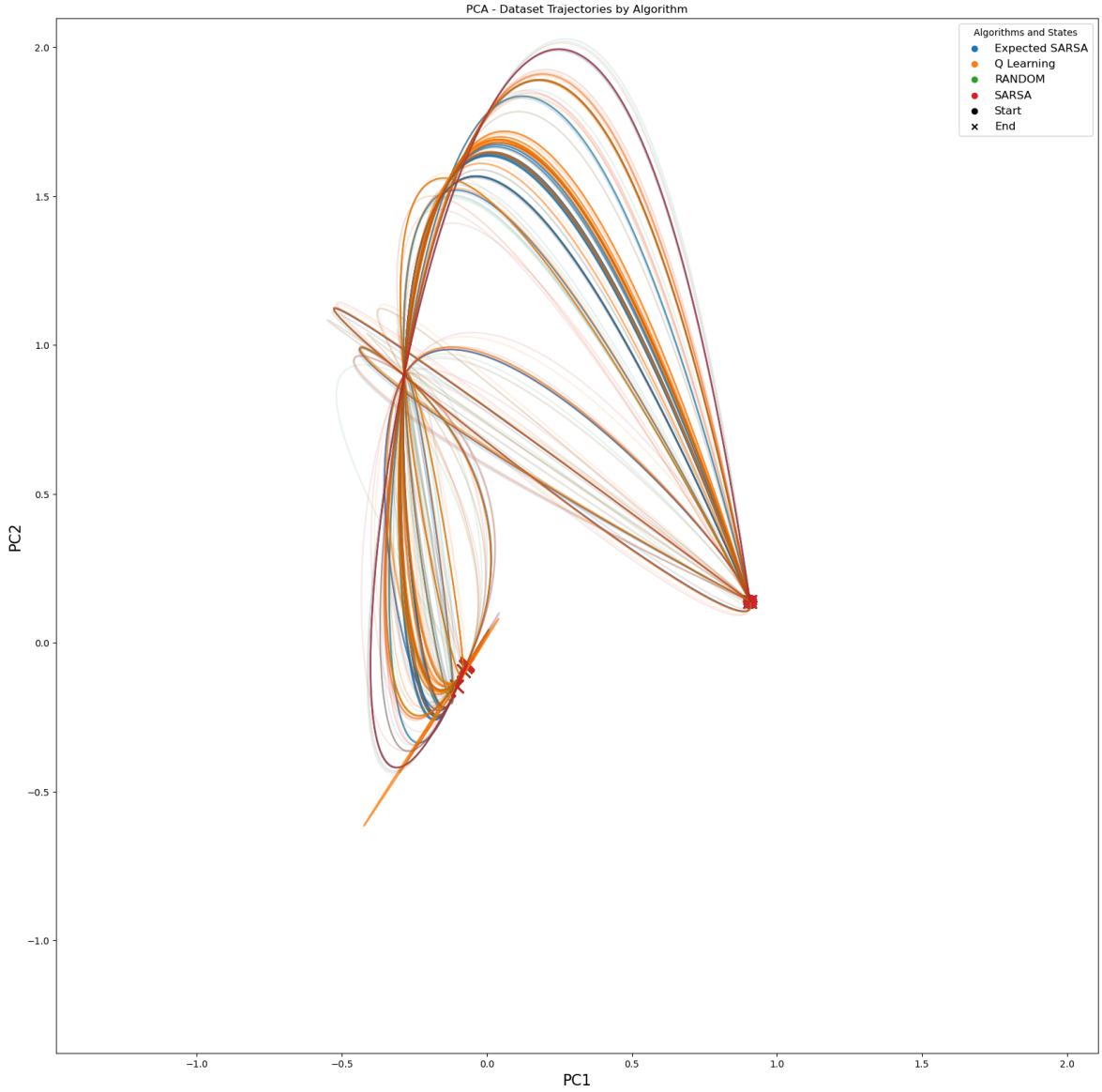
Observation PCA:

As previously stated, adding color encoding does not reveal significant insights into the frequently visited downprojected states across algorithms. In the scatter plots of individual algorithms, minor differences are present but are not meaningful enough for deeper interpretation. These four PCA plots largely confirm the interpretations made for t-SNE: they illustrate how different downprojection methods represent state distributions in a 2D grid, but without distinct algorithm-specific clustering patterns.

The PCA projections further emphasize the consistency of the algorithms in visiting similar states, though PCA's linear nature limits its ability to separate subtle nuances

between them. This visualization reinforces that PCA is best for capturing global variance rather than distinct algorithm trajectories or state prominence.

In [32]:
```
import matplotlib.pyplot as plt
import matplotlib.patches as mpatches

# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = pca_plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = pca_plotting_df[pca_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, x_col='PC1', y_col='PC2', co

# Mark start, intermediate, and end points
for i, algo in enumerate(algorithms):
    start_data = pca_plotting_df[(pca_plotting_df['cp'] == 'start') & (pc
    intermediate_data = pca_plotting_df[(pca_plotting_df['cp'] == 'interm
    end_data = pca_plotting_df[(pca_plotting_df['cp'] == 'end') & (pca_pl

    ax.scatter(start_data['PC1'], start_data['PC2'], color=colors[i], mar
    ax.scatter(intermediate_data['PC1'], intermediate_data['PC2'], color=
    ax.scatter(end_data['PC1'], end_data['PC2'], color=colors[i], marker=

# Create legend entries for algorithms
for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)  # Empty scatter for

# Create legend entries for states
state_markers = {'Start': 'o', 'End': 'x'}
for state_name, state_marker in state_markers.items():
    ax.scatter([], [], color='black', marker=state_marker, label=state_na

ax.set_title("PCA - Dataset Trajectories by Algorithm")
ax.legend(title="Algorithms and States", loc="best", fontsize=12)

margin = 1.2  # Adjusted margin for PCA data
plt.xlim(pca_plotting_df['PC1'].min() - margin, pca_plotting_df['PC1'].ma
plt.ylim(pca_plotting_df['PC2'].min() - margin, pca_plotting_df['PC2'].ma
plt.xlabel('PC1', fontsize=16)
plt.ylabel('PC2', fontsize=16)
plt.show()
```



Observation PCA:

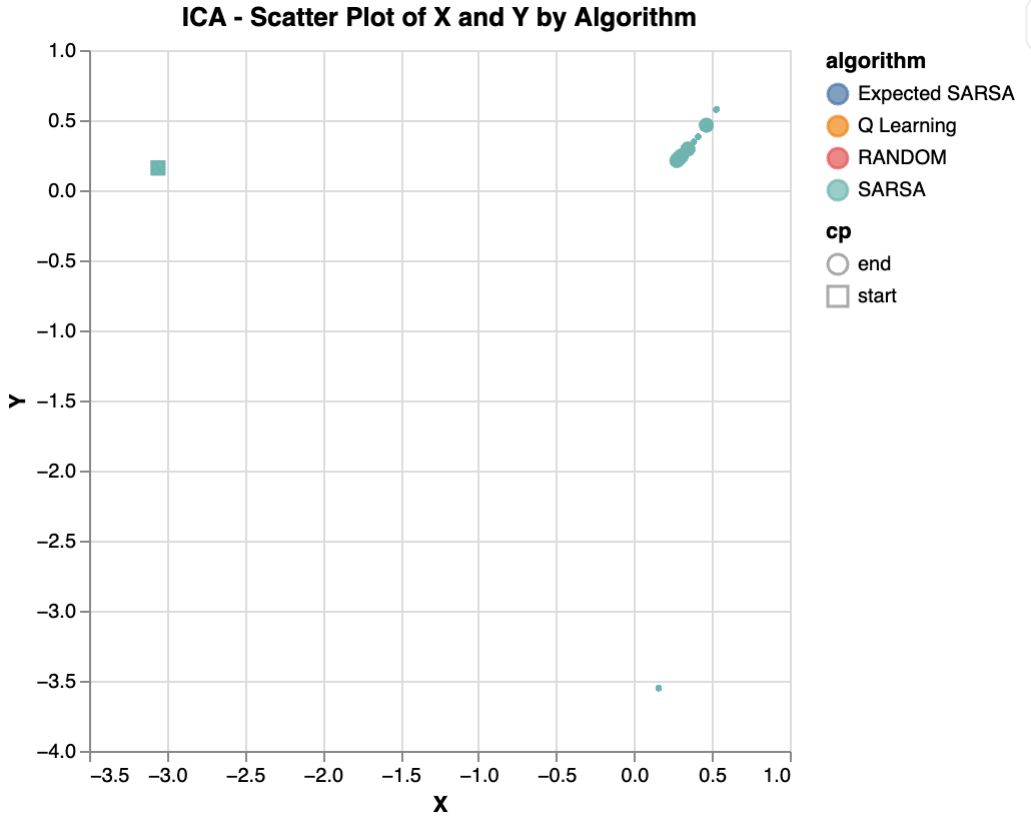
In contrast to t-SNE and UMAP, the we do not see lots of bundles that highlight differences in the behaviour of the algorithms. All trajectories pass through the same starting state, traverse to the next state/cluster and then, closeer to the lower left corner, we see that many states are encoded in this region (e.g. all final states). Thus it can be suspected that there are nuances in the tracetories visible in this one cluster.

ICA

In [33]:
```
alt.Chart(ica_plotting_df).mark_point(
    opacity=0.7,
    filled=True,
    size=10
).encode(
    x='X',
    y='Y',
    color='algorithm:N'
).properties(
    width=350,
    height=350,
```

```
    title="ICA - Scatter Plot of X and Y by Algorithm"
) + alt.Chart(ica_plotting_df).transform_filter(
    (datum.cp == 'end') | (datum.cp == 'start')
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape='cp:N',
    color='algorithm:N'
).properties(
    width=350,
    height=350
)
```

Out[33]:

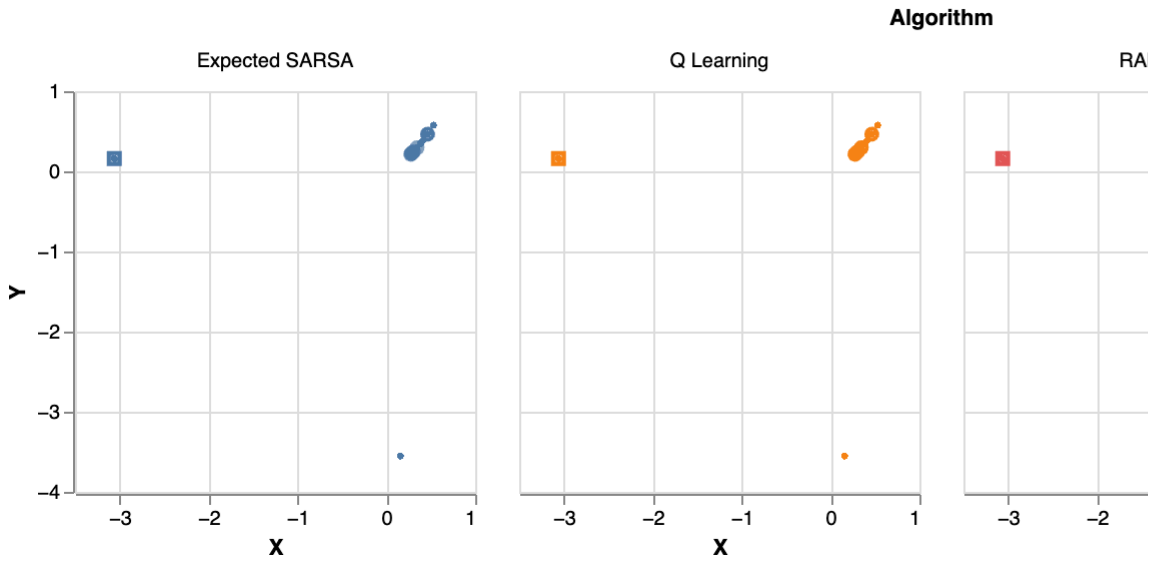


In [34]:

```
# Base chart for intermediate states
base = alt.Chart(ica_plotting_df).mark_point(
    opacity=0.7,
    filled=True,  # Ensure points are filled
    size=10
).encode(
    x='X',
    y='Y',
    color='algorithm:N'
).properties(
    width=200,
    height=200
)

# Chart for start and end states only
start_end = alt.Chart(ica_plotting_df).transform_filter(
    (alt.datum.cp == 'end') | (alt.datum.cp == 'start')
).mark_point(filled=False).encode(
    x='X',
    y='Y',
    shape=alt.Shape('cp:N', legend=alt.Legend(title="State")),
    color='algorithm:N'
```

```
).properties(
    width=200,
    height=200
)

# Combine both charts and apply faceting to arrange by algorithm in a row
chart = (base + start_end).facet(
    facet=alt.Facet('algorithm:N', title="Algorithm"),
    columns=len(ica_plotting_df['algorithm'].unique())
).properties(
    title="ICA - Scatter Plot of X and Y by Algorithm"
)

chart
```

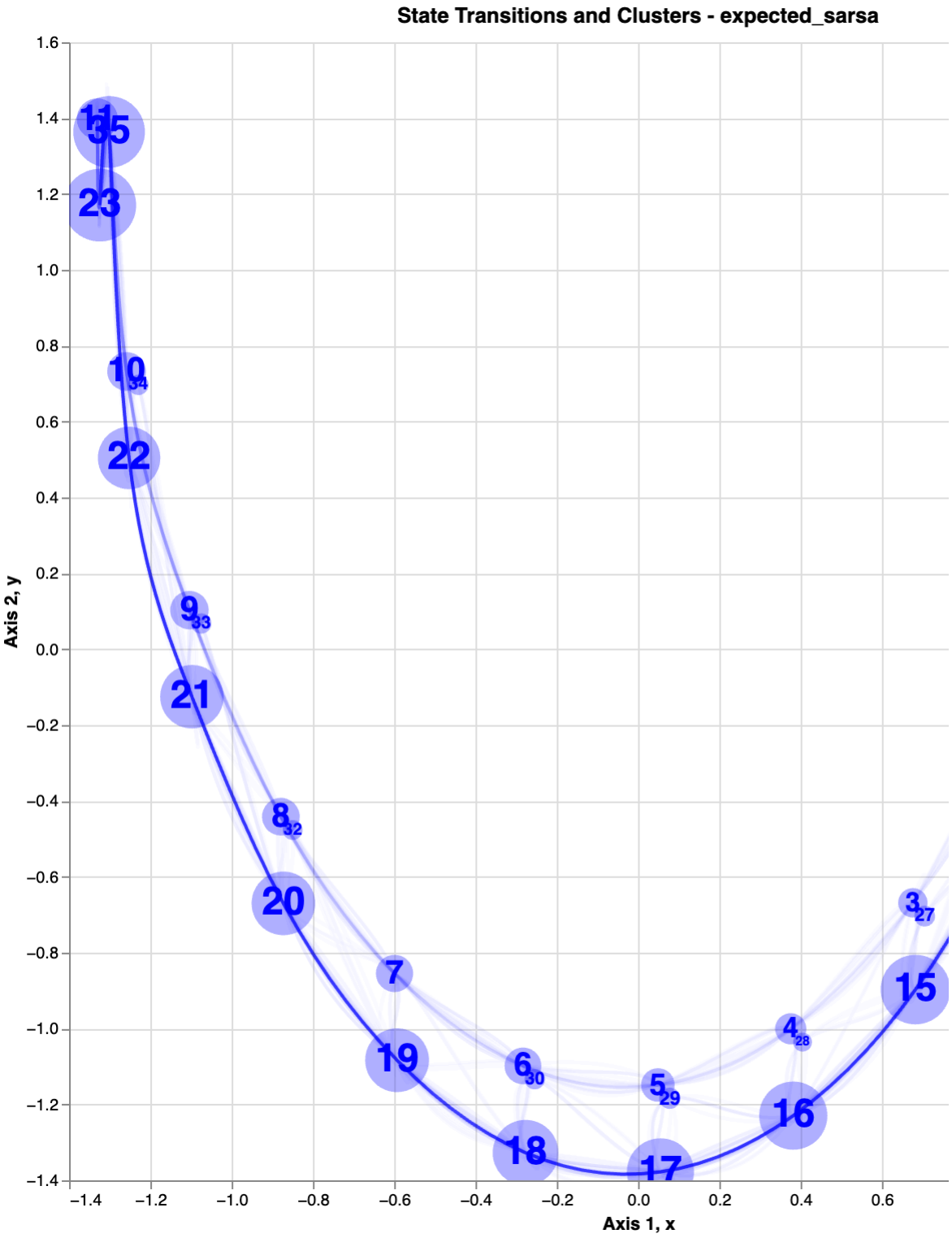
Out[34]:

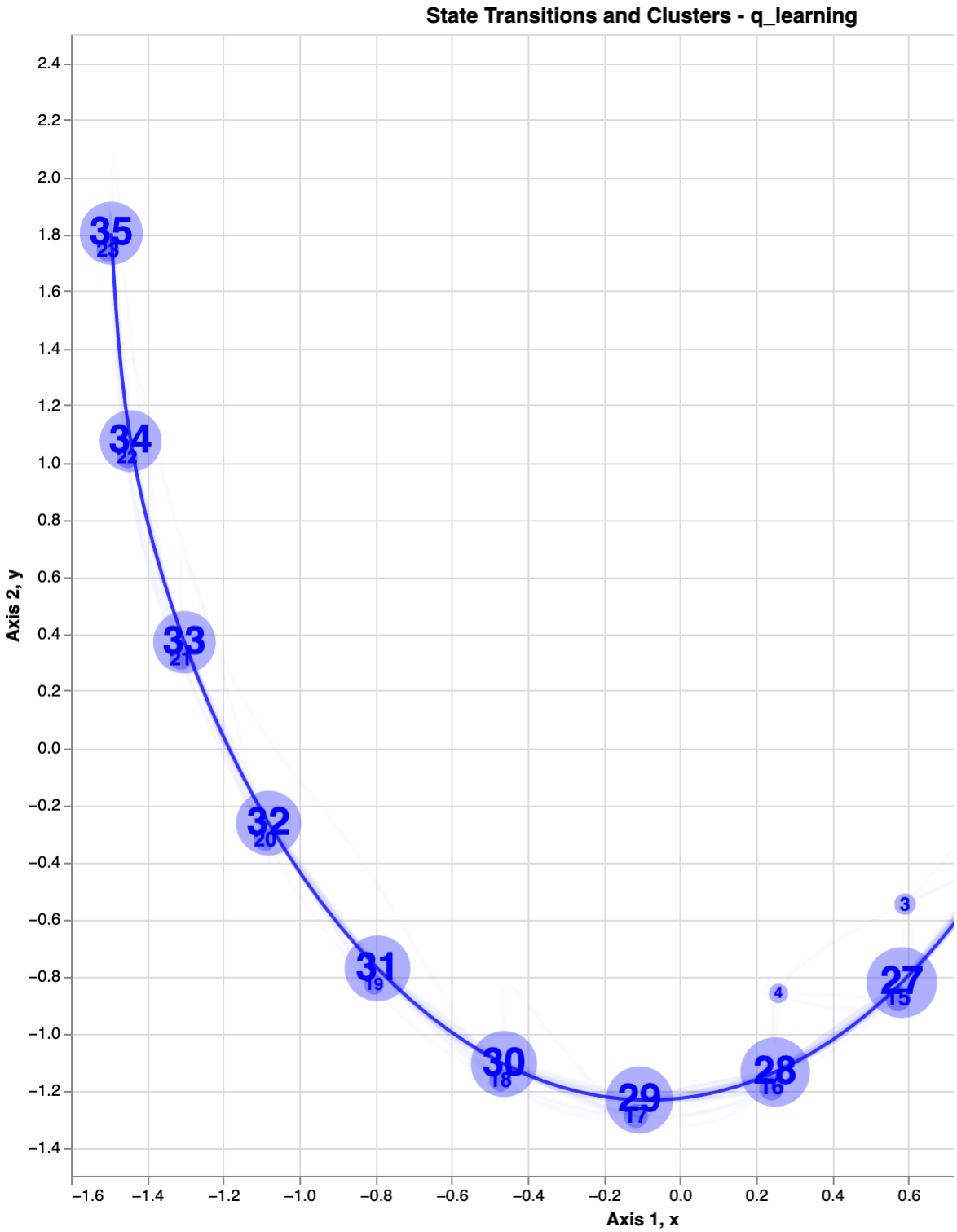


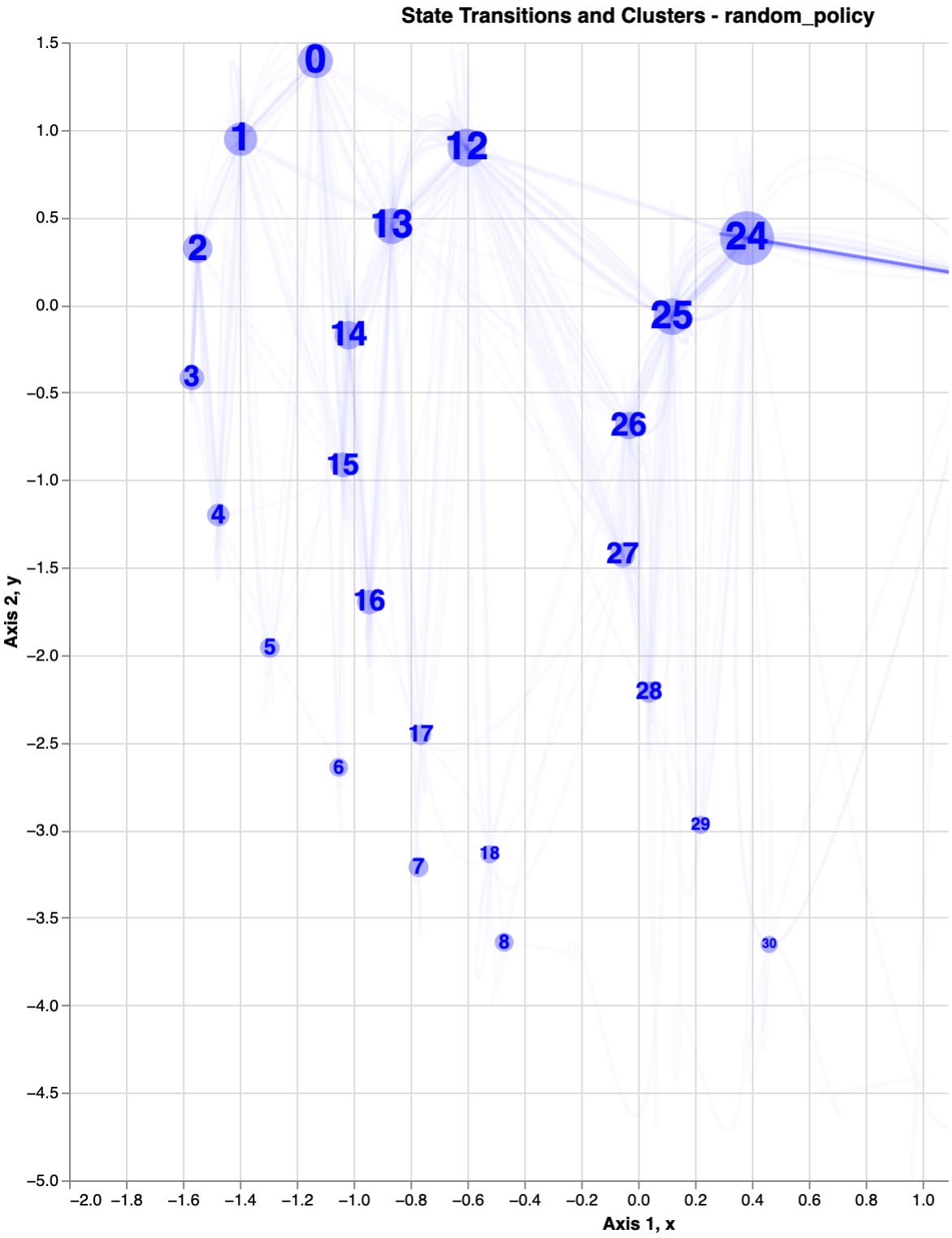
In [35]:

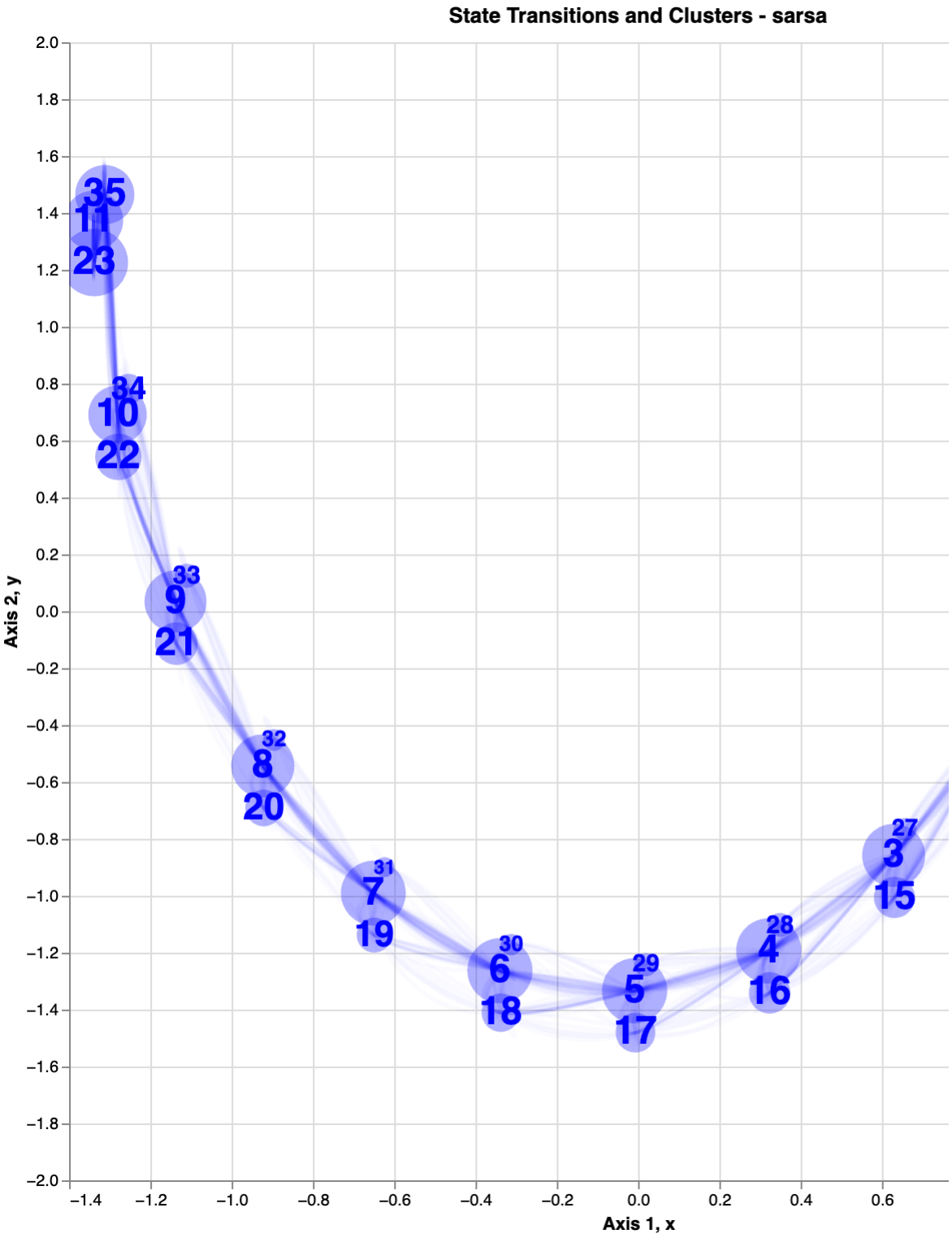
```
expected_sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='expected_sarsa', encoding_type='one-hot', down_projec
)

q_learning_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='q_learning', encoding_type='one-hot', down_project_me
)

random_policy_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='random_policy', encoding_type='one-hot', down_project
)

sarsa_connected_density_plot = visualizer.process_and_visualize(
    algorithm_name='sarsa', encoding_type='one-hot', down_project_method=
)

expected_sarsa_connected_density_plot.display()
q_learning_connected_density_plot.display()
random_policy_connected_density_plot.display()
sarsa_connected_density_plot.display()
```

```
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
Loading down-projected data from cache.
```


State Transitions and Clusters - expected_sarsa
Axis 2, y
Axis 1, x
1.6
1.4
1.2
1.0
0.8
0.6
0.4
0.2
0.0
−0.2
−0.4
−0.6
−0.8
−1.0
−1.2
−1.4
−1.4
−1.2
−1.0
−0.8
−0.6
−0.4
−0.2
0.0
0.2
0.4
0.6
11
35
23
10
34
22
9
33
21
8
32
20
7
19
6
30
18
5
29
17
4
28
16
3
27
15


State Transitions and Clusters - q_learning
Axis 2, y
Axis 1, x
35
23
34
22
33
21
32
20
31
19
30
18
29
17
28
16
27
15
4
3


State Transitions and Clusters - random_policy
Axis 2, y
Axis 1, x
0
1
12
13
24
2
25
14
3
26
15
4
27
16
5
28
17
6
29
18
7
8
30

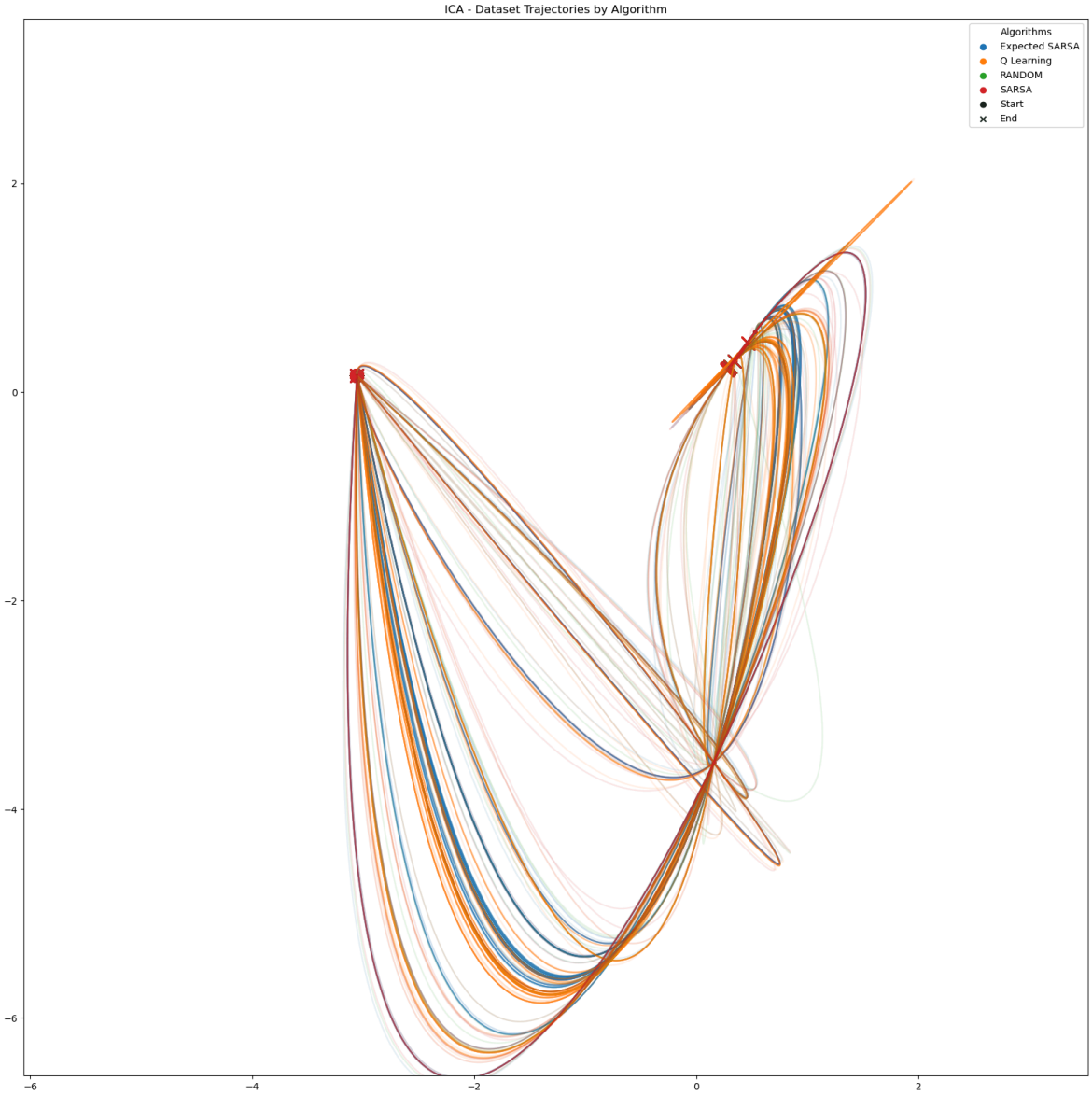


Observation ICA:

Very similar to PCA no downprojected clusters for the individual algrithms are visible in these plots and differences between the scattered points are only nuanced.

In [36]:
```
# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot for ICA trajectories
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = ica_plotting_df['algorithm'].unique()
```

```
# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = ica_plotting_df[ica_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Mark start and end points
for i, algo in enumerate(algorithms):
    start_data = ica_plotting_df[(ica_plotting_df['cp'] == 'start') & (ic
    intermediate_data = ica_plotting_df[(ica_plotting_df['cp'] == 'interm
    end_data = ica_plotting_df[(ica_plotting_df['cp'] == 'end') & (ica_pl

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

# Legend for algorithms and state markers
for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)  # Empty scatter for

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("ICA - Dataset Trajectories by Algorithm")
ax.legend(title="Algorithms", loc="best")

margin = 3
plt.xlim(ica_plotting_df['X'].min() - margin, ica_plotting_df['X'].max()
plt.ylim(ica_plotting_df['Y'].min() - margin, ica_plotting_df['Y'].max()
plt.show()
```



Observation ICA:

Like PCA, our starting state is encoded where all trajectories pass through and go on to a next state. All other states are encoded in one cluster in the upper right corner.

Optional

▶ Projection Space Explorer (click to reveal)

Results

You may add additional screenshots of the Projection Space Explorer.

Interpretion

- What can be seen in the projection(s)?
- Was it what you expected? If not what did you expect?
- Can you confirm prior hypotheses from the projection?
- Did you get any unexpected insights?

TSNE

In [37]:
```
# Detect Clusters in the dataset
clusterer = hdbscan.HDBSCAN(min_cluster_size=400)
cluster_labels = clusterer.fit_predict(plotting_df[['X', 'Y']].values)
plotting_df['cluster'] = cluster_labels
display(plotting_df.head())
print()

# create a dictionary with the proportion of samples having one state bel
df['cluster'] = cluster_labels
df.head()
cluster_state_dict = {}

for cluster_id, cluster_data in df.groupby('cluster'):
    #if cluster_id != -1:
    cluster_state_dict[cluster_id] = {}
    unique_states = cluster_data['state'].unique()  # Extract unique stat
    print(f"Cluster {cluster_id} has the following unique states:")
    for state in unique_states:
        cluster_state_dict[cluster_id][state.item()] = len(cluster_data[c
        print(f"  - {state}")
    print(f"samples in cluster: {len(cluster_data)}")
    print()  # Blank line for readability
```

	line	cp	algorithm	X	Y	cluster
0	0	start	Expected SARSA	43.642590	-14.434265	35
1	0	intermediate	Expected SARSA	-49.644737	53.423031	31
2	0	intermediate	Expected SARSA	-49.644737	53.423031	31
3	0	end	Expected SARSA	35.871559	-9.731361	35
4	5	start	Expected SARSA	52.834625	6.233274	35

```
Cluster -1 has the following unique states:
  - 36
  - 12
samples in cluster: 284

Cluster 0 has the following unique states:
  - 34
samples in cluster: 1085

Cluster 1 has the following unique states:
  - 30
samples in cluster: 1012

Cluster 2 has the following unique states:
  - 5
samples in cluster: 1594

Cluster 3 has the following unique states:
  - 25
samples in cluster: 1912

Cluster 4 has the following unique states:
  - 0
samples in cluster: 1801

Cluster 5 has the following unique states:
  - 1
samples in cluster: 1894

Cluster 6 has the following unique states:
  - 32
samples in cluster: 956

Cluster 7 has the following unique states:
  - 15
samples in cluster: 1736

Cluster 8 has the following unique states:
  - 16
samples in cluster: 1606

Cluster 9 has the following unique states:
  - 21
samples in cluster: 1367

Cluster 10 has the following unique states:
  - 28
samples in cluster: 1167

Cluster 11 has the following unique states:
  - 4
samples in cluster: 1681

Cluster 12 has the following unique states:
  - 9
samples in cluster: 1395

Cluster 13 has the following unique states:
  - 26
samples in cluster: 1423
```

```
Cluster 14 has the following unique states:
  - 20
samples in cluster: 1287

Cluster 15 has the following unique states:
  - 31
samples in cluster: 989

Cluster 16 has the following unique states:
  - 23
samples in cluster: 2491

Cluster 17 has the following unique states:
  - 2
samples in cluster: 1856

Cluster 18 has the following unique states:
  - 3
samples in cluster: 1732

Cluster 19 has the following unique states:
  - 33
samples in cluster: 975

Cluster 20 has the following unique states:
  - 29
samples in cluster: 1113

Cluster 21 has the following unique states:
  - 10
samples in cluster: 1341

Cluster 22 has the following unique states:
  - 18
samples in cluster: 1376

Cluster 23 has the following unique states:
  - 27
samples in cluster: 1253

Cluster 24 has the following unique states:
  - 35
samples in cluster: 2835

Cluster 25 has the following unique states:
  - 14
samples in cluster: 1925

Cluster 26 has the following unique states:
  - 7
samples in cluster: 1569

Cluster 27 has the following unique states:
  - 8
samples in cluster: 1470

Cluster 28 has the following unique states:
  - 6
samples in cluster: 1543
```

```
Cluster 29 has the following unique states:
  - 36
samples in cluster: 3428

Cluster 30 has the following unique states:
  - 24
samples in cluster: 1871

Cluster 31 has the following unique states:
  - 24
samples in cluster: 3002

Cluster 32 has the following unique states:
  - 11
samples in cluster: 1407

Cluster 33 has the following unique states:
  - 17
samples in cluster: 1510

Cluster 34 has the following unique states:
  - 22
samples in cluster: 1501

Cluster 35 has the following unique states:
  - 36
samples in cluster: 2870

Cluster 36 has the following unique states:
  - 19
samples in cluster: 1342

Cluster 37 has the following unique states:
  - 12
samples in cluster: 3002

Cluster 38 has the following unique states:
  - 13
samples in cluster: 2272
```

In [38]:
```python
# Plot the Gridworld with iproportional state size for each cluster
# Define the grid dimensions (4 rows and 12 columns for Cliff Walking)
grid_height = 4
grid_width = 12
total_states = grid_height * grid_width

# Function to convert state index to (x, y) grid position
def state_to_grid_position(state, grid_width):
    return divmod(state, grid_width)  # Returns (row, column)

def visualize_grid_world(cluster_id, cluster_state_dict, filname=None):
    fig, ax = plt.subplots(figsize=(6, 2))
    # Create a grid using a matrix
    grid = np.zeros((grid_height, grid_width))

    # Define special states: start, goal, and cliff
    start_state = 36
    goal_state = 47
```

```
    cliff_states = list(range(37, 47))

    # Mark the cliff, start, and goal states in the grid
    # Using different values for different states
    for state in cliff_states:
        x, y = state_to_grid_position(state, grid_width)
        grid[x, y] = 1  # Cliff (will be gray)

    start_x, start_y = state_to_grid_position(start_state, grid_width)
    goal_x, goal_y = state_to_grid_position(goal_state, grid_width)
    grid[start_x, start_y] = 2  # Start (will be blue)
    grid[goal_x, goal_y] = 3    # Goal (will be green)

    # Create custom colormap with specified colors
    colors = ['white',    # Empty cells (0)
              'gray',     # Cliff (1)
              'blue',     # Start (2)
              'green']    # Goal (3)
    custom_cmap = ListedColormap(colors)

    # Plot the grid with the custom color map
    ax.imshow(grid, cmap=custom_cmap, extent=[0, grid_width, 0, grid_heig

    # Plot the agent's current position
    base_agent_marker_size = 200
    cluster_states = cluster_state_dict[cluster_id]
    for agent_state, cluster_proportion in cluster_states.items():
        agent_x, agent_y = state_to_grid_position(agent_state, grid_width
        agent_marker_size = base_agent_marker_size * cluster_proportion
        ax.scatter(agent_y + 0.5, grid_height - agent_x - 0.5, color='pur
                   s=agent_marker_size,
                   label=f'Agent {cluster_proportion:.2f}')

    # Annotate the grid with state numbers
    #for state in range(total_states):
    #    x, y = state_to_grid_position(state, grid_width)
    #    ax.text(y + 0.5, grid_height - x - 0.5, str(state), ha='center',

    # Labels and titles
    ax.set_xticks(np.arange(grid_width))
    ax.set_yticks(np.arange(grid_height))
    ax.set_xticklabels(np.arange(grid_width))
    ax.set_yticklabels(np.arange(grid_height - 1, -1, -1))
    ax.grid(True)
    #ax.set_title("Cliff Walking Grid World")
    #ax.legend(loc='upper left')

    # Display the plot

    if filename is not None:
        fig.savefig(filename)
        plt.close(fig)
    else:
        plt.show()
```

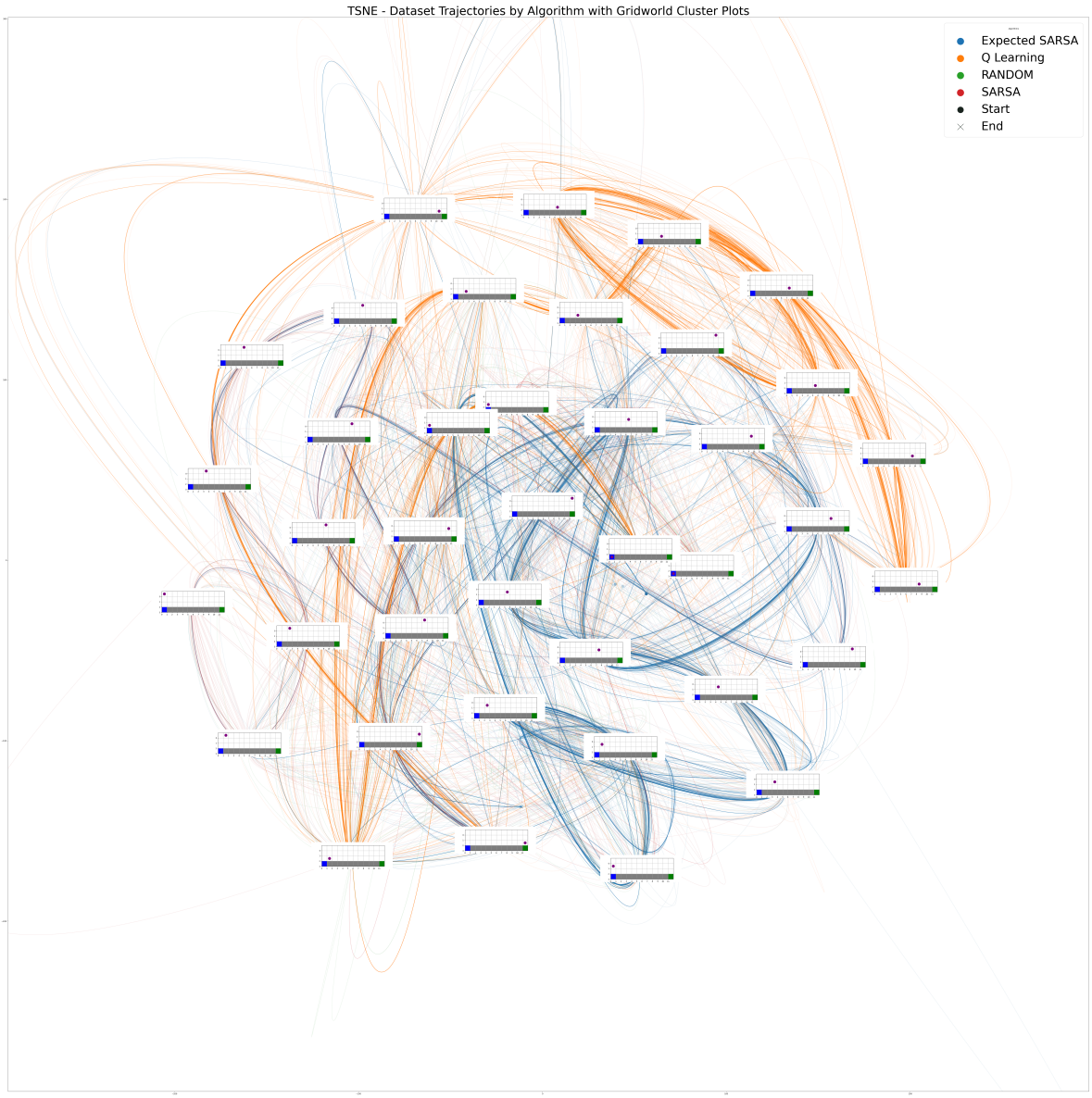
In [39]:
```
# storing gridworld plots for each cluster
# Group by cluster and calculate the center point for each cluster
cluster_centers = plotting_df.groupby('cluster')[['X', 'Y']].mean()

for cluster_id in plotting_df['cluster'].unique():
```

```
    if cluster_id != -1:
        #representative_state = plotting_df[plotting_df['cluster'] == clu
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        visualize_grid_world(cluster_id, cluster_state_dict, filename)
```

In [40]:

```
# Embedding Cluster Plots in Plots
from mpl_toolkits.axes_grid1.inset_locator import InsetPosition, mark_ins
from matplotlib.offsetbox import OffsetImage, AnnotationBbox
### SPLINES ###

# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot
fig, ax = plt.subplots(figsize=(100, 100))

# Get unique algorithms
algorithms = plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = plotting_df[plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Mark start and end points
for i, algo in enumerate(algorithms):
    start_data = plotting_df[(plotting_df['cp'] == 'start') & (plotting_d
    intermediate_data = plotting_df[(plotting_df['cp'] == ' intermediate'
    end_data = plotting_df[(plotting_df['cp'] == 'end') & (plotting_df['a

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo, marker='.', s=5000)

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("TSNE - Dataset Trajectories by Algorithm with Gridworld Clu
ax.legend(title="Algorithms", loc="best", fontsize = 60)



### GRIDWORLD CLUSTERS ###
# Plot each point and embed each cluster's gridworld plot at its center
cluster_centers = plotting_df.groupby('cluster')[['X', 'Y']].mean()
for i, (cluster_id, center) in enumerate(cluster_centers.iterrows()):
    if cluster_id != -1:
        #cluster_data = plotting_df[plotting_df['cluster'] == cluster_id]
        #ax.scatter(cluster_data['X'], cluster_data['Y'], label=f'Cluster

        # Load and add the saved gridworld image at the cluster center
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
```

```
        img = plt.imread(filename)
        imagebox = OffsetImage(img, zoom=0.7)
        ab = AnnotationBbox(imagebox, (center['X'], center['Y']), frameon
        ax.add_artist(ab)

margin = 100
plt.xlim(plotting_df['X'].min() - margin, plotting_df['X'].max() + margin
plt.ylim(plotting_df['Y'].min() - margin, plotting_df['Y'].max() + margin
plt.show()
fig.savefig('data/cliff_walking/TSNE_1000_plot.png')
```



In [41]:
```
#from IPython.display import Image, display
#display(Image(filename="data/cliff_walking/TSNE_1000_plot.png"))
```

Observation t-SNE:

This final plot, was produced for the t-SNE downprojections by first clustering the points with HDBscan and then plotting the gridworld with the agent states of the cluster into its center.

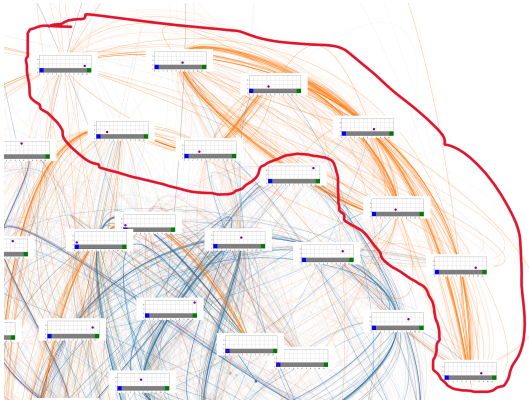
From the overall structure, we can derive somewhat dense starting and end points (taking different encodings for the same state into consideration). We also have some bundles, very prominent for Q-learning and expected Sarsa. Besides, also some sparse trajectories/points can be detected.

We see the different charcteristics of the 3 different policies (algorithms), excluding the random policy: In our observations, we expected to see the following:

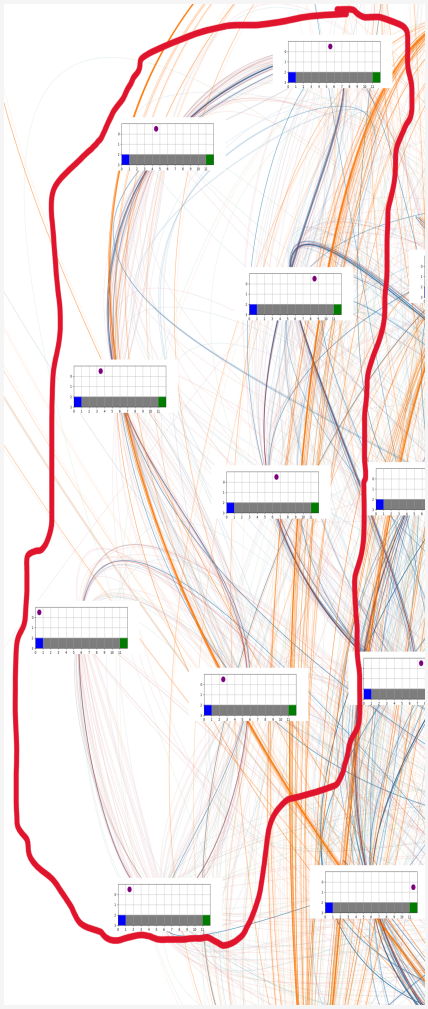
- Sarsa learns from the actions that the policy slects, also suboptimal actions. Thus, we get a more conservative policy that results in a safer behaviour
- Q-learning looks at the maximum Q-value over all actions for the next state. Thus it is more greed and always aims for the best possible outcome.
- Expected Sarsa is a trade of between Sarsa and Q-learning. It accounts for all possible actions and doesn't focus on a single action.

We can easily confirm this by looking into the plots:

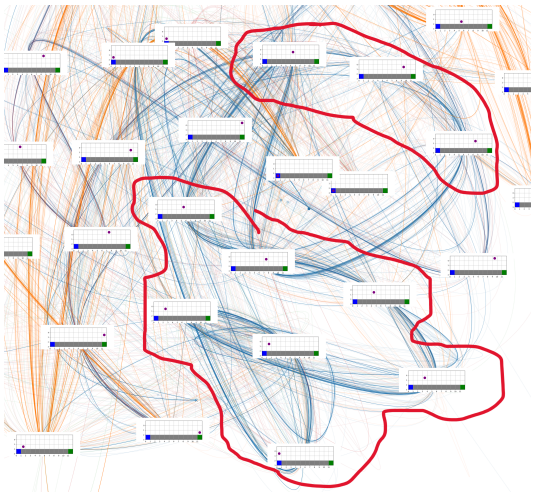
In this part of the projection, we see that an agent passes this area under a Q-learning policy a lot. This policy takes always the greediest action to get the best possible outcome. In our gridworl, we get a reward of -1 for every step. Thus, under this greedy policy, we try to make as little steps as possible to reach the goal. This is only possible when walking very close to the cliff (grey area). Consequently, our trajectories are very prominent at these state near/at the cliff.



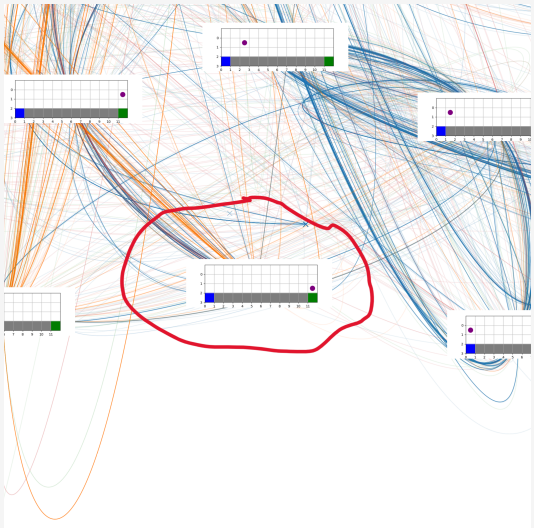
Although the red Sarsa trajectories are not that visible in the plot, we provided here a snippet where the cluster states are visible: Sarsa is the most conservative algorithm and in the snippet, it has not many overlaps with the other algorithms. The paths are the farthest away from the cliff to avoid the penalities of falling of it.



Here, we can see that under the expected Sarsa algorithm, the agent takes different steps: They can be interpreted as this middle ground between Sarsa and Q-learning as it is more conservative than Q-learning but not as conservative as Sarsa. Thus, we see that the agent took steps that are a litte distant from the cliff but not that far away.



In the snippet of the goal state, we see that some trajectories of agents that managed to get past the cliff meet again. In addition, we can see another property of t-SNE: spacial properties are not preserved, thus close states to this goal state are not necessarily in this states neighborhood.



UMAP

In [42]:
```
# Detect Clusters in the dataset
clusterer = hdbscan.HDBSCAN(min_cluster_size=400)
cluster_labels = clusterer.fit_predict(umap_plotting_df[['X', 'Y']].value
umap_plotting_df['cluster'] = cluster_labels
display(umap_plotting_df.head())
print()

# create a dictionary with the proportion of samples having one state bel
df['cluster'] = cluster_labels
df.head()
cluster_state_dict = {}

for cluster_id, cluster_data in df.groupby('cluster'):
    #if cluster_id != -1:
    cluster_state_dict[cluster_id] = {}
    unique_states = cluster_data['state'].unique()  # Extract unique stat
    print(f"Cluster {cluster_id} has the following unique states:")
    for state in unique_states:
        cluster_state_dict[cluster_id][state.item()] = len(cluster_data[c
        print(f"  - {state}")
    print(f"samples in cluster: {len(cluster_data)}")
    print()  # Blank line for readability
```

	line	cp	algorithm	X	Y	cluster
0	0	start	Expected SARSA	-7.783310	4.864756	0
1	0	intermediate	Expected SARSA	17.651447	-6.973587	8
2	0	intermediate	Expected SARSA	17.666525	-6.818447	8
3	0	end	Expected SARSA	-8.490652	4.694337	0
4	5	start	Expected SARSA	-7.902337	5.341905	0

```
Cluster 0 has the following unique states:
  - 36
samples in cluster: 6430

Cluster 1 has the following unique states:
  - 17
samples in cluster: 1510

Cluster 2 has the following unique states:
  - 35
samples in cluster: 2835

Cluster 3 has the following unique states:
  - 4
samples in cluster: 1681

Cluster 4 has the following unique states:
  - 13
samples in cluster: 2272

Cluster 5 has the following unique states:
  - 20
samples in cluster: 1286

Cluster 6 has the following unique states:
  - 12
  - 33
samples in cluster: 3155

Cluster 7 has the following unique states:
  - 11
samples in cluster: 1407

Cluster 8 has the following unique states:
  - 24
samples in cluster: 4873

Cluster 9 has the following unique states:
  - 16
  - 34
samples in cluster: 1607

Cluster 10 has the following unique states:
  - 18
samples in cluster: 1376

Cluster 11 has the following unique states:
  - 23
  - 32
samples in cluster: 2492

Cluster 12 has the following unique states:
  - 25
samples in cluster: 1912

Cluster 13 has the following unique states:
  - 19
samples in cluster: 1342

Cluster 14 has the following unique states:
```

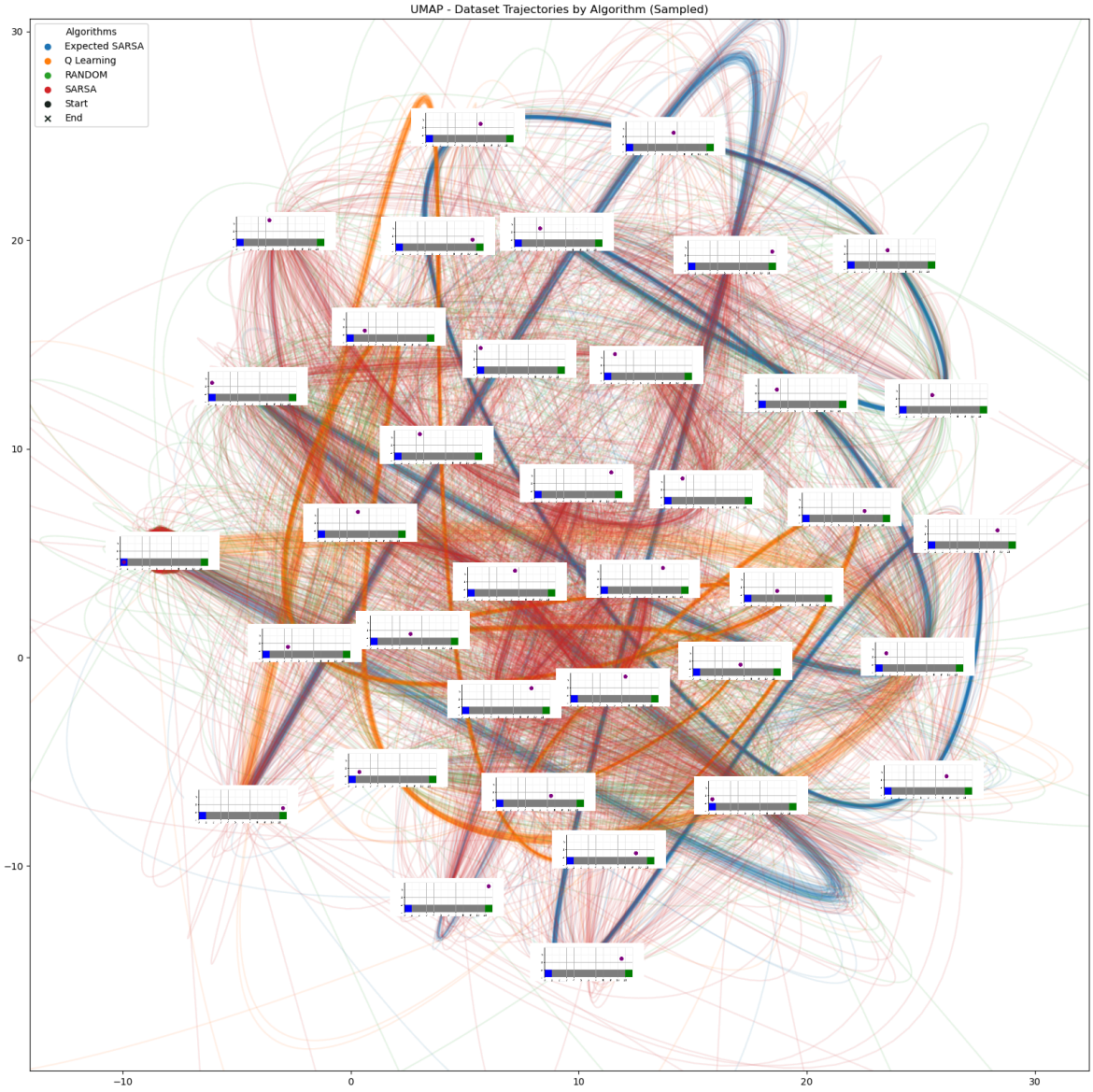
```
  - 1
samples in cluster: 1894

Cluster 15 has the following unique states:
  - 21
samples in cluster: 1367

Cluster 16 has the following unique states:
  - 15
  - 20
samples in cluster: 1736

Cluster 17 has the following unique states:
  - 0
  - 33
samples in cluster: 1802

Cluster 18 has the following unique states:
  - 14
samples in cluster: 1925

Cluster 19 has the following unique states:
  - 34
samples in cluster: 1082

Cluster 20 has the following unique states:
  - 26
samples in cluster: 1423

Cluster 21 has the following unique states:
  - 3
samples in cluster: 1732

Cluster 22 has the following unique states:
  - 5
samples in cluster: 1594

Cluster 23 has the following unique states:
  - 22
samples in cluster: 1501

Cluster 24 has the following unique states:
  - 10
samples in cluster: 1341

Cluster 25 has the following unique states:
  - 32
  - 33
  - 15
samples in cluster: 957

Cluster 26 has the following unique states:
  - 30
  - 33
samples in cluster: 1013

Cluster 27 has the following unique states:
  - 28
samples in cluster: 1167
```

```
Cluster 28 has the following unique states:
  - 6
samples in cluster: 1543

Cluster 29 has the following unique states:
  - 27
samples in cluster: 1253

Cluster 30 has the following unique states:
  - 29
  - 34
samples in cluster: 1113

Cluster 31 has the following unique states:
  - 2
samples in cluster: 1856

Cluster 32 has the following unique states:
  - 9
samples in cluster: 1395

Cluster 33 has the following unique states:
  - 7
  - 29
samples in cluster: 1570

Cluster 34 has the following unique states:
  - 8
  - 34
samples in cluster: 1471

Cluster 35 has the following unique states:
  - 31
samples in cluster: 989

Cluster 36 has the following unique states:
  - 33
samples in cluster: 971
```

In [43]:
```
# storing gridworld plots for each cluster
# Group by cluster and calculate the center point for each cluster
cluster_centers = umap_plotting_df.groupby('cluster')[['X', 'Y']].mean()

for cluster_id in umap_plotting_df['cluster'].unique():
    if cluster_id != -1:
        #representative_state = umap_plotting_df[umap_plotting_df['cluste
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        visualize_grid_world(cluster_id, cluster_state_dict, filename)
```

In [44]:
```
# Defining colors
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Creating the plot
fig, ax = plt.subplots(figsize=(20, 20))

# Sampling the 10%
sampling_fraction = 0.1
```

```
# Loop overall the algorithms
for i, algo in enumerate(algorithms):
    algo_data = umap_plotting_df[umap_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    # Sampling
    sampled_lines = np.random.choice(lines, size=int(len(lines) * samplin

    for line in sampled_lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Start/End points
for i, algo in enumerate(algorithms):
    start_data = umap_plotting_df[(umap_plotting_df['cp'] == 'start') & (
    intermediate_data = umap_plotting_df[(umap_plotting_df['cp'] == 'inte
    end_data = umap_plotting_df[(umap_plotting_df['cp'] == 'end') & (umap

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("UMAP - Dataset Trajectories by Algorithm (Sampled)")
ax.legend(title="Algorithms", loc="best")

### GRIDWORLD CLUSTERS ###
# Plot each point and embed each cluster's gridworld plot at its center
cluster_centers = umap_plotting_df.groupby('cluster')[['X', 'Y']].mean()
for i, (cluster_id, center) in enumerate(cluster_centers.iterrows()):
    if cluster_id != -1:
        #cluster_data = umap_plotting_df[umap_plotting_df['cluster'] == c
        #ax.scatter(cluster_data['X'], cluster_data['Y'], label=f'Cluster

        # Load and add the saved gridworld image at the cluster center
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        img = plt.imread(filename)
        imagebox = OffsetImage(img, zoom=0.2)
        ab = AnnotationBbox(imagebox, (center['X'], center['Y']), frameon
        ax.add_artist(ab)

margin = 5
plt.xlim(umap_plotting_df['X'].min() - margin, umap_plotting_df['X'].max(
plt.ylim(umap_plotting_df['Y'].min() - margin, umap_plotting_df['Y'].max(
plt.show()
fig.savefig('data/cliff_walking/UMAP_plot.png')
```

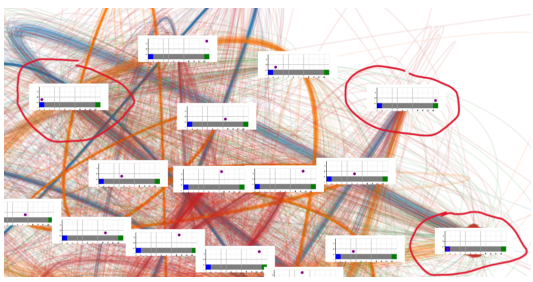


In [45]:
```
#from IPython.display import Image, display
#display(Image(filename="data/cliff_walking/UMAP_plot.png"))
```

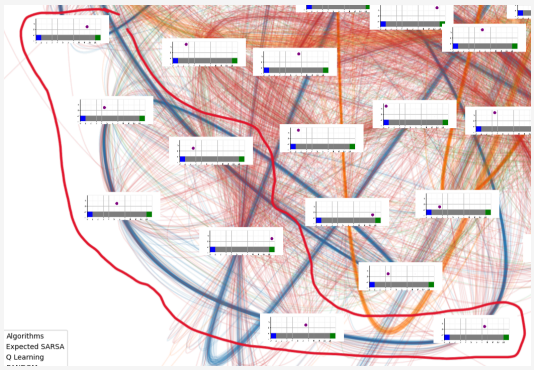
Observation UMAP:

From the overall structure, we can derive bundles, for the three learning algorithms. When we following the trajecotires and looking at the clustered points along these trajectories, we can see as mentioned above the three different characteristics/behaviours of Q-learning, Sarsa, and expected Sarsa. We provide some snippets of interesting regions below:

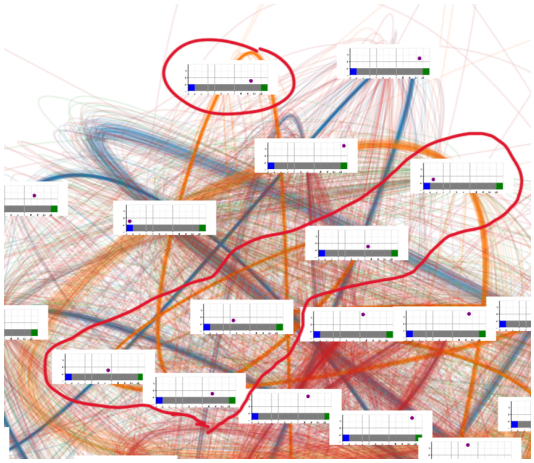
In the first snipped, shared clusters by different algorithms are highlighted. We see that these clusters represent the starting state, the first action (which is always the same for all algorithms) and for the three learning algorithms the goal state.



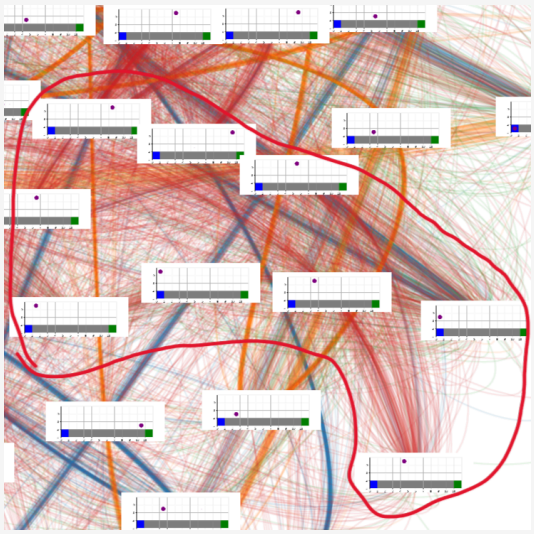
In the second snipped, we see that the cluster components all have one state in common, belognging to the trajectory in the middle which is the expected behaviour of this policy.



Similarly, we plotted the states for the clusters and see also here that a single cluster encodes exactly one state and here, the bundles for Q-learning near the cliff are highlighted.



For Sarsa, we see similar patters, and the clusters reflect states far away from the cliff.



PCA

In [46]:
```
# Detect Clusters in the dataset
clusterer = hdbscan.HDBSCAN(min_cluster_size=400)
cluster_labels = clusterer.fit_predict(pca_plotting_df[['PC1', 'PC2']].va
pca_plotting_df['cluster'] = cluster_labels
display(pca_plotting_df.head())
print()

# create a dictionary with the proportion of samples having one state bel
df['cluster'] = cluster_labels
df.head()
cluster_state_dict = {}

for cluster_id, cluster_data in df.groupby('cluster'):
```

```
    #if cluster_id != -1:
    cluster_state_dict[cluster_id] = {}
    unique_states = cluster_data['state'].unique()  # Extract unique stat
    print(f"Cluster {cluster_id} has the following unique states:")
    for state in unique_states:
        cluster_state_dict[cluster_id][state.item()] = len(cluster_data[c
        print(f"  - {state}")
    print(f"samples in cluster: {len(cluster_data)}")
    print()  # Blank line for readability
```

	line	cp	algorithm	PC1	PC2	cluster
0	0	start	Expected SARSA	0.907516	0.138661	0
1	0	intermediate	Expected SARSA	-0.285555	0.897602	1
2	0	intermediate	Expected SARSA	-0.285555	0.897602	1
3	0	end	Expected SARSA	0.907516	0.138661	0
4	5	start	Expected SARSA	0.907516	0.138661	0

```
Cluster 0 has the following unique states:
  - 36
samples in cluster: 6430

Cluster 1 has the following unique states:
  - 24
samples in cluster: 4873

Cluster 2 has the following unique states:
  - 12
samples in cluster: 3154

Cluster 3 has the following unique states:
  - 35
samples in cluster: 2835

Cluster 4 has the following unique states:
  - 23
samples in cluster: 2491

Cluster 5 has the following unique states:
  - 13
samples in cluster: 2272

Cluster 6 has the following unique states:
  - 0
samples in cluster: 1801

Cluster 7 has the following unique states:
  - 4
samples in cluster: 1681

Cluster 8 has the following unique states:
  - 2
samples in cluster: 1856

Cluster 9 has the following unique states:
  - 15
samples in cluster: 1736

Cluster 10 has the following unique states:
  - 3
samples in cluster: 1732

Cluster 11 has the following unique states:
  - 20
samples in cluster: 1287

Cluster 12 has the following unique states:
  - 27
samples in cluster: 1253

Cluster 13 has the following unique states:
  - 30
samples in cluster: 1012

Cluster 14 has the following unique states:
  - 28
samples in cluster: 1167
```

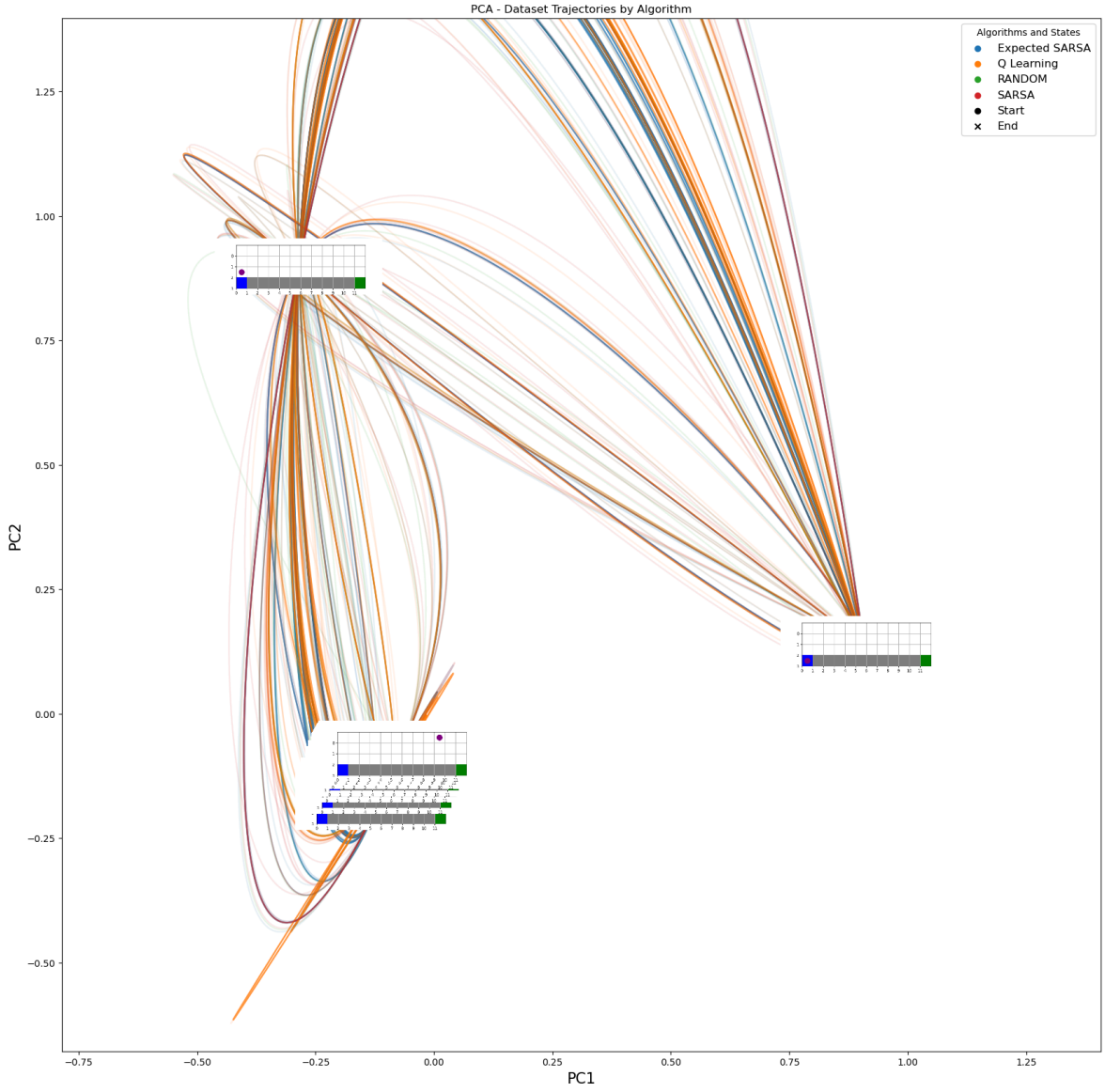
```
Cluster 15 has the following unique states:
  - 1
samples in cluster: 1894

Cluster 16 has the following unique states:
  - 32
samples in cluster: 956

Cluster 17 has the following unique states:
  - 34
samples in cluster: 1085

Cluster 18 has the following unique states:
  - 29
samples in cluster: 1113

Cluster 19 has the following unique states:
  - 14
samples in cluster: 1925

Cluster 20 has the following unique states:
  - 25
samples in cluster: 1912

Cluster 21 has the following unique states:
  - 8
samples in cluster: 1470

Cluster 22 has the following unique states:
  - 6
samples in cluster: 1543

Cluster 23 has the following unique states:
  - 19
samples in cluster: 1342

Cluster 24 has the following unique states:
  - 10
samples in cluster: 1341

Cluster 25 has the following unique states:
  - 31
samples in cluster: 989

Cluster 26 has the following unique states:
  - 33
samples in cluster: 975

Cluster 27 has the following unique states:
  - 22
samples in cluster: 1501

Cluster 28 has the following unique states:
  - 17
samples in cluster: 1510

Cluster 29 has the following unique states:
  - 7
samples in cluster: 1569
```

```
Cluster 30 has the following unique states:
  - 18
samples in cluster: 1376

Cluster 31 has the following unique states:
  - 21
samples in cluster: 1367

Cluster 32 has the following unique states:
  - 26
samples in cluster: 1423

Cluster 33 has the following unique states:
  - 5
samples in cluster: 1594

Cluster 34 has the following unique states:
  - 16
samples in cluster: 1606

Cluster 35 has the following unique states:
  - 11
samples in cluster: 1407

Cluster 36 has the following unique states:
  - 9
samples in cluster: 1395
```

In [47]:
```
# storing gridworld plots for each cluster
# Group by cluster and calculate the center point for each cluster
cluster_centers = pca_plotting_df.groupby('cluster')[['PC1', 'PC2']].mean

for cluster_id in pca_plotting_df['cluster'].unique():
    if cluster_id != -1:
        #representative_state = pca_plotting_df[pca_plotting_df['cluster'
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        visualize_grid_world(cluster_id, cluster_state_dict, filename)
```

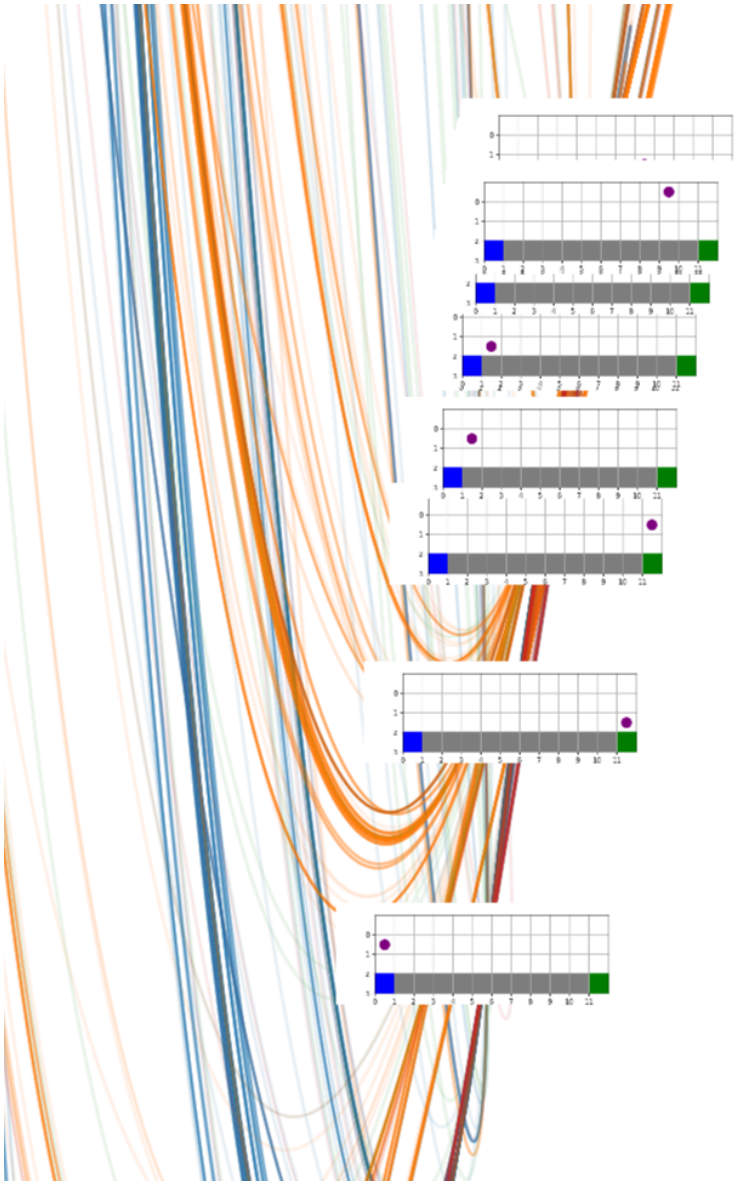
In [48]:
```
# Embedding Cluster Plots in Plots
# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = pca_plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = pca_plotting_df[pca_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, x_col='PC1', y_col='PC2', co

# Mark start, intermediate, and end points
```

```
for i, algo in enumerate(algorithms):
    start_data = pca_plotting_df[(pca_plotting_df['cp'] == 'start') & (pc
    intermediate_data = pca_plotting_df[(pca_plotting_df['cp'] == 'interm
    end_data = pca_plotting_df[(pca_plotting_df['cp'] == 'end') & (pca_pl

    ax.scatter(start_data['PC1'], start_data['PC2'], color=colors[i], mar
    ax.scatter(intermediate_data['PC1'], intermediate_data['PC2'], color=
    ax.scatter(end_data['PC1'], end_data['PC2'], color=colors[i], marker=

# Create legend entries for algorithms
for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)  # Empty scatter for

# Create legend entries for states
state_markers = {'Start': 'o', 'End': 'x'}
for state_name, state_marker in state_markers.items():
    ax.scatter([], [], color='black', marker=state_marker, label=state_na

ax.set_title("PCA - Dataset Trajectories by Algorithm")
ax.legend(title="Algorithms and States", loc="best", fontsize=12)

### GRIDWORLD CLUSTERS ###
# Plot each point and embed each cluster's gridworld plot at its center
cluster_centers = pca_plotting_df.groupby('cluster')[['PC1', 'PC2']].mean
for i, (cluster_id, center) in enumerate(cluster_centers.iterrows()):
    if cluster_id != -1:
        #cluster_data = pca_plotting_df[pca_plotting_df['cluster'] == clu
        #ax.scatter(cluster_data['X'], cluster_data['Y'], label=f'Cluster

        # Load and add the saved gridworld image at the cluster center
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        img = plt.imread(filename)
        imagebox = OffsetImage(img, zoom=0.3)
        ab = AnnotationBbox(imagebox, (center['PC1'], center['PC2']), fra
        ax.add_artist(ab)

margin = 0.5  # Adjusted margin for PCA data
plt.xlim(pca_plotting_df['PC1'].min() - margin, pca_plotting_df['PC1'].ma
plt.ylim(pca_plotting_df['PC2'].min() - margin, pca_plotting_df['PC2'].ma
plt.xlabel('PC1', fontsize=16)
plt.ylabel('PC2', fontsize=16)
plt.show()
fig.savefig('data/cliff_walking/PCA_plot.png')
```



In [49]:
```
#from IPython.display import Image, display
#display(Image(filename="data/cliff_walking/PCA_plot.png"))
```

In [50]:
```
from IPython.display import Image, display
display(Image(filename="data/cliff_walking/PCA_plot_zoomed_v2.png"))
```



Observation PCA:

Even with the additional added cluster states, we do not get any insights into the behvaiour of the different algorithms. Interestingly, starting state and the immediate next state can be well separated but for all the other states we have one big cluster. Even when zooming into this cluster, we cannot get any meanigful information.

ICA

In [51]:
```
# Detect Clusters in the dataset
clusterer = hdbscan.HDBSCAN(min_cluster_size=400)
cluster_labels = clusterer.fit_predict(ica_plotting_df[['X', 'Y']].values
ica_plotting_df['cluster'] = cluster_labels
display(ica_plotting_df.head())
print()

# create a dictionary with the proportion of samples having one state bel
df['cluster'] = cluster_labels
df.head()
cluster_state_dict = {}

for cluster_id, cluster_data in df.groupby('cluster'):
```

```
    #if cluster_id != -1:
    cluster_state_dict[cluster_id] = {}
    unique_states = cluster_data['state'].unique()  # Extract unique stat
    print(f"Cluster {cluster_id} has the following unique states:")
    for state in unique_states:
        cluster_state_dict[cluster_id][state.item()] = len(cluster_data[c
        print(f"  - {state}")
    print(f"samples in cluster: {len(cluster_data)}")
    print()  # Blank line for readability
```

	line	cp	algorithm	X	Y	cluster
0	0	start	Expected SARSA	-3.057335	0.159032	1
1	0	intermediate	Expected SARSA	0.157704	-3.552101	0
2	0	intermediate	Expected SARSA	0.157704	-3.552101	0
3	0	end	Expected SARSA	-3.057335	0.159032	1
4	5	start	Expected SARSA	-3.057335	0.159032	1

```
Cluster 0 has the following unique states:
  - 24
samples in cluster: 4873

Cluster 1 has the following unique states:
  - 36
samples in cluster: 6430

Cluster 2 has the following unique states:
  - 12
samples in cluster: 3154

Cluster 3 has the following unique states:
  - 35
samples in cluster: 2835

Cluster 4 has the following unique states:
  - 13
samples in cluster: 2272

Cluster 5 has the following unique states:
  - 23
samples in cluster: 2491

Cluster 6 has the following unique states:
  - 4
samples in cluster: 1681

Cluster 7 has the following unique states:
  - 0
samples in cluster: 1801

Cluster 8 has the following unique states:
  - 30
samples in cluster: 1012

Cluster 9 has the following unique states:
  - 28
samples in cluster: 1167

Cluster 10 has the following unique states:
  - 20
samples in cluster: 1287

Cluster 11 has the following unique states:
  - 27
samples in cluster: 1253

Cluster 12 has the following unique states:
  - 3
samples in cluster: 1732

Cluster 13 has the following unique states:
  - 15
samples in cluster: 1736

Cluster 14 has the following unique states:
  - 2
samples in cluster: 1856
```

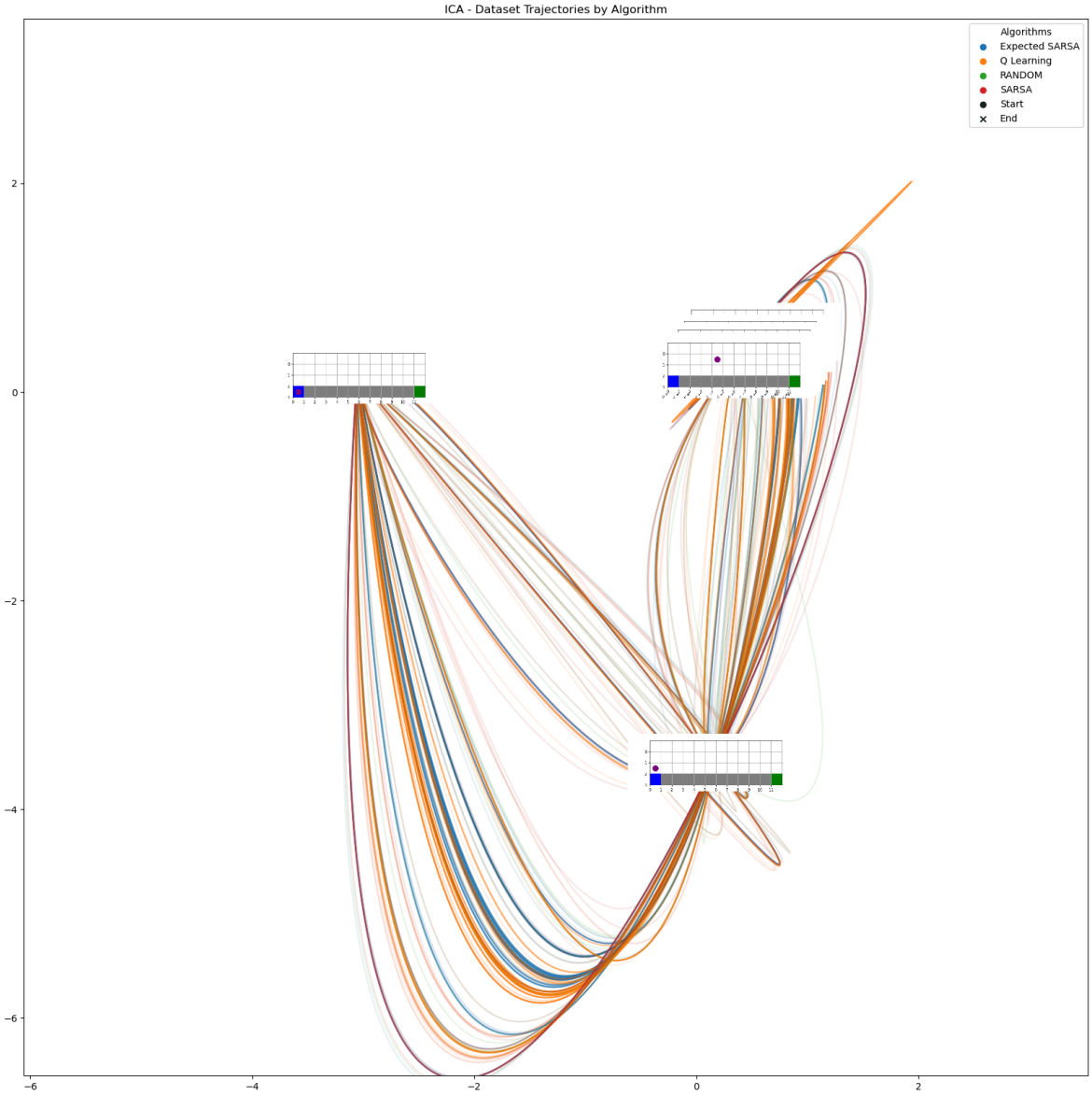
```
Cluster 15 has the following unique states:
  - 32
samples in cluster: 956

Cluster 16 has the following unique states:
  - 29
samples in cluster: 1113

Cluster 17 has the following unique states:
  - 34
samples in cluster: 1085

Cluster 18 has the following unique states:
  - 1
samples in cluster: 1894

Cluster 19 has the following unique states:
  - 31
samples in cluster: 989

Cluster 20 has the following unique states:
  - 33
samples in cluster: 975

Cluster 21 has the following unique states:
  - 19
samples in cluster: 1342

Cluster 22 has the following unique states:
  - 10
samples in cluster: 1341

Cluster 23 has the following unique states:
  - 8
samples in cluster: 1470

Cluster 24 has the following unique states:
  - 6
samples in cluster: 1543

Cluster 25 has the following unique states:
  - 25
samples in cluster: 1912

Cluster 26 has the following unique states:
  - 14
samples in cluster: 1925

Cluster 27 has the following unique states:
  - 18
samples in cluster: 1376

Cluster 28 has the following unique states:
  - 21
samples in cluster: 1367

Cluster 29 has the following unique states:
  - 26
samples in cluster: 1423
```

```
Cluster 30 has the following unique states:
  - 17
samples in cluster: 1510

Cluster 31 has the following unique states:
  - 22
samples in cluster: 1501

Cluster 32 has the following unique states:
  - 7
samples in cluster: 1569

Cluster 33 has the following unique states:
  - 9
samples in cluster: 1395

Cluster 34 has the following unique states:
  - 11
samples in cluster: 1407

Cluster 35 has the following unique states:
  - 5
samples in cluster: 1594

Cluster 36 has the following unique states:
  - 16
samples in cluster: 1606
```

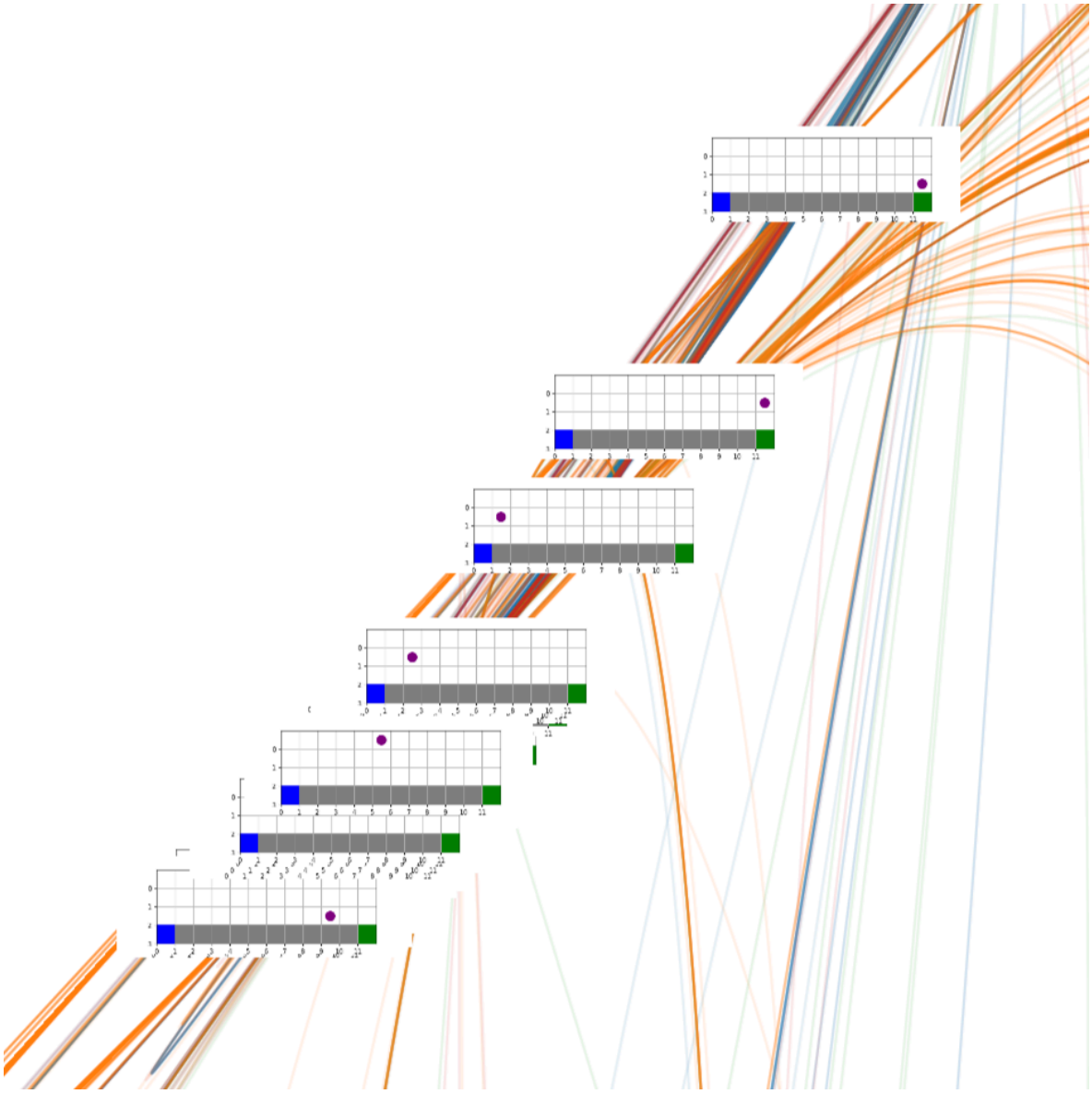
In [52]:
```
# storing gridworld plots for each cluster
# Group by cluster and calculate the center point for each cluster
cluster_centers = ica_plotting_df.groupby('cluster')[['X', 'Y']].mean()

for cluster_id in ica_plotting_df['cluster'].unique():
    if cluster_id != -1:
        #representative_state = ica_plotting_df[ica_plotting_df['cluster'
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        visualize_grid_world(cluster_id, cluster_state_dict, filename)
```

In [53]:
```
# Define colors for each unique algorithm
colors = ['#1f77b4', '#ff7f0e', '#2ca02c', '#d62728']
colors_cp = ['#1A2421', '#1A2421']

# Create a plot for ICA trajectories
fig, ax = plt.subplots(figsize=(20, 20))

# Get unique algorithms
algorithms = ica_plotting_df['algorithm'].unique()

# Loop through each unique algorithm
for i, algo in enumerate(algorithms):
    algo_data = ica_plotting_df[ica_plotting_df['algorithm'] == algo]
    lines = algo_data['line'].unique()

    for line in lines:
        line_data = algo_data[algo_data['line'] == line]
        plot_df_splines(ax=ax, df=line_data, color=colors[i], alpha=0.1,

# Mark start and end points
for i, algo in enumerate(algorithms):
```

```
    start_data = ica_plotting_df[(ica_plotting_df['cp'] == 'start') & (ic
    intermediate_data = ica_plotting_df[(ica_plotting_df['cp'] == 'interm
    end_data = ica_plotting_df[(ica_plotting_df['cp'] == 'end') & (ica_pl

    ax.scatter(start_data['X'], start_data['Y'], color=colors[i], marker=
    ax.scatter(intermediate_data['X'], intermediate_data['Y'], color=colo
    ax.scatter(end_data['X'], end_data['Y'], color=colors[i], marker='x',

# Legend for algorithms and state markers
for i, algo in enumerate(algorithms):
    ax.scatter([], [], color=colors[i], label=algo)  # Empty scatter for

for state_name, state_marker in zip(['Start', 'End'], ['o', 'x']):
    ax.scatter([], [], color=colors_cp[0], label=state_name, marker=state

ax.set_title("ICA - Dataset Trajectories by Algorithm")
ax.legend(title="Algorithms", loc="best")

### GRIDWORLD CLUSTERS ###
# Plot each point and embed each cluster's gridworld plot at its center
cluster_centers = ica_plotting_df.groupby('cluster')[['X', 'Y']].mean()
for i, (cluster_id, center) in enumerate(cluster_centers.iterrows()):
    if cluster_id != -1:
        #cluster_data = ica_plotting_df[ica_plotting_df['cluster'] == clu
        #ax.scatter(cluster_data['X'], cluster_data['Y'], label=f'Cluster

        # Load and add the saved gridworld image at the cluster center
        filename = f'gridworld/gridworld_cluster_{cluster_id}.png'
        img = plt.imread(filename)
        imagebox = OffsetImage(img, zoom=0.3)
        ab = AnnotationBbox(imagebox, (center['X'], center['Y']), frameon
        ax.add_artist(ab)


margin = 3
plt.xlim(ica_plotting_df['X'].min() - margin, ica_plotting_df['X'].max()
plt.ylim(ica_plotting_df['Y'].min() - margin, ica_plotting_df['Y'].max()
plt.show()
fig.savefig('data/cliff_walking/ICA_plot.png')
```



In [54]:
```
from IPython.display import Image, display
display(Image(filename="data/cliff_walking/ICA_plot_zoomed_v2.png"))
```



Observation ICA:

Exaclty the same as PCA, we have well separated starting state and successor state but one big cluster that is not meaningful indicating that ICA is not able to separate the data.

Similarities and Differneces of Downprojection Methods

From the results we can derive the following:

- t-SNE and UMAP are both nonlinear dimensionality reduction techniques that are better suited for this dataset. However, UMAP appears to be best suited for this visualization task as it balances local and global structures very well and provides meaningful trajectory bundels for all three learning algorithms.
- PCA and ICA result in very similar downprojections. Their encodings are not able to separate the data well and no meaningful trajectories can be derived. A possible explanation could be the nature of the data: PCA and ICA capture linear and independent components, respectively, but they don't capture the nonlinear

relationships well or the non-Gaussian assumption is violated which might have been a problem for this dataset.

Submission

When you've finished working on this assignment please download this notebook as HTML and add it to your repository in addition to the notebook file.

In [55]:
```
print('I am at the end of the Notebook')
```

```
I am at the end of the Notebook
```