QUESTION 1: What are upper and lower bounds on ρ_{ij}? Provide a justification for using log-normalized return ($r_i(t)$) instead of regular return ($q_i(t)$).

Answer:

For the bounds of the correlation coefficient ρ_{ij}, its values lie within the range:

$$-1 \leq \rho_{ij} \leq 1$$

This bound follows directly from the **Cauchy-Schwarz inequality**. A value of:

- **+1** indicates perfect positive linear correlation,
- **-1** indicates perfect negative linear correlation,
- **0** implies no linear correlation between the two time series.

Why use log-normalized return (r_i(t)) instead of regular return (q_i(t))?

Regular return is defined as:

$$q_i(t) = \frac{p_i(t) - p_i(t-1)}{p_i(t-1)}$$

Log-normalized return is defined as:

$$r_i(t) = \log(1 + q_i(t)) = \log\left(\frac{p_i(t)}{p_i(t-1)}\right)$$

Log-normalized returns are preferred in financial analysis for the following reasons:

- **Better Statistical Properties**: Log returns are more likely to be normally distributed, which is a common assumption in many statistical models.
- **Additivity**: Log returns across multiple periods can be summed directly to obtain the total return, which simplifies analysis over time.
- **Handling Extreme Values**: Logarithmic transformation compresses the effect of large outliers, reducing their impact on statistical measures.
- **Scale Invariance**: Returns expressed in logarithmic form are dimensionless and comparable across assets with different price levels.

Thus, using (r_i(t)) leads to more stable, consistent, and interpretable results, especially in the context of correlation analysis and constructing financial networks.

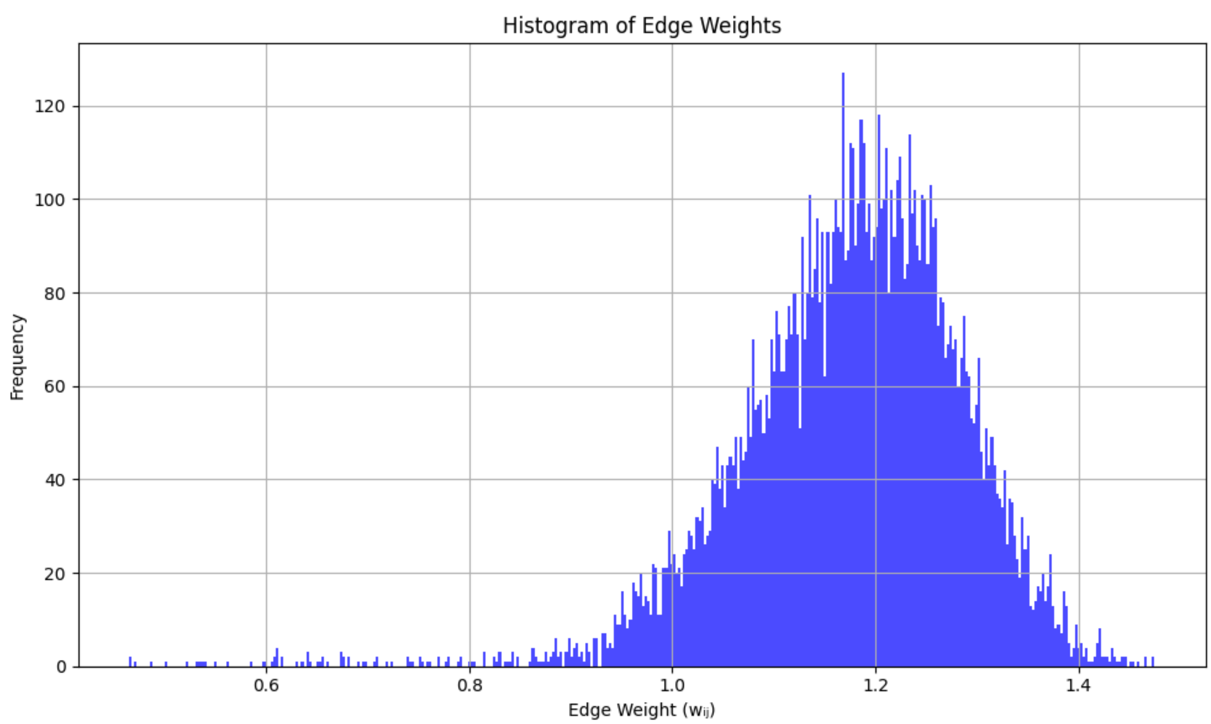
In [1]:
```
from google.colab import drive
drive.mount('/content/drive')

PATH = '/content/drive/MyDrive/Colab Notebooks/ECE232E_Project4/'
```

```
Drive already mounted at /content/drive; to attempt to forcibly remount, cal
l drive.mount("/content/drive", force_remount=True).
```

Question 2: Plot a histogram showing the un-normalized distribution of edge weights.

In [2]:
```
import pandas as pd
import numpy as np
import os
import itertools
import matplotlib.pyplot as plt

# --- Step 1: Define helper functions ---
def compute_log_normalized_return(df):
    df['Return'] = np.log(1 + (df['Adj Close'] - df['Adj Close'].shift(1)) /
    return df['Return']

def read_stock_data(file_path):
    try:
        df = pd.read_csv(
            file_path,
            parse_dates=['Date'],
            date_format='%Y-%m-%d'  # Preferred way to specify date format
        )
        df.set_index('Date', inplace=True)
        log_returns = compute_log_normalized_return(df)
        return log_returns
    except Exception as e:
        print(f"Error reading {file_path}: {e}")
        return pd.Series()

def calculate_correlation(stock1, stock2):
    return stock1.corr(stock2)

# --- Step 2: Load and filter valid symbols ---
name_sector_df = pd.read_csv( PATH + 'finance_data/Name_sector.csv')
data_folder = PATH + 'finance_data/data'

stock_files = [f for f in os.listdir(data_folder) if f.endswith('.csv')]
available_symbols = [f.rstrip('.csv') for f in stock_files]
stock_symbols = [s for s in name_sector_df['Symbol'].tolist() if s in availa

# --- Step 3: Load return data ---
stock_returns = {}
for symbol in stock_symbols:
    file_path = os.path.join(data_folder, symbol + '.csv')
    stock_returns[symbol] = read_stock_data(file_path)
```

```
# --- Step 4: Compute pairwise correlations ---
stock_combinations = itertools.combinations(stock_symbols, 2)
correlations = {}

for (stock1_symbol, stock2_symbol) in stock_combinations:
    stock1_returns = stock_returns.get(stock1_symbol)
    stock2_returns = stock_returns.get(stock2_symbol)

    if stock1_returns is not None and stock2_returns is not None:
        combined = pd.concat([stock1_returns, stock2_returns], axis=1).dropr
        combined.columns = [stock1_symbol, stock2_symbol]
        if not combined.empty:
            corr = calculate_correlation(combined[stock1_symbol], combined[s
            correlations[(stock1_symbol, stock2_symbol)] = corr

# --- Step 5: Compute edge weights ---
co_weights = {k: np.sqrt(2 * (1 - v)) for k, v in correlations.items()}
co_weights_list = list(co_weights.values())

# --- Step 6: Plot histogram of edge weights ---
plt.figure(figsize=(10, 6))
plt.hist(co_weights_list, bins=400, color='blue', alpha=0.7)
plt.title('Histogram of Edge Weights')
plt.xlabel('Edge Weight (wᵢⱼ)')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```



QUESTION 3:

Extract the MST of the correlation graph. Each stock can be categorized into a sector, which can be found in Name sector.csv file. Plot the MST and color-code the nodes based on sectors. Do you see any pattern in the MST? The structures that you find in MST are called Vine clusters. Provide a detailed explanation about the pattern you observe.

In [3]:
```python
import matplotlib.pyplot as plt
import networkx as nx

# --- Fix 1: Safer sector mapping with fallback color ---
sector_colors = {
    'Telecommunication Services': 'skyblue',
    'Consumer Staples': 'olive',
    'Utilities': 'gold',
    'Information Technology': 'rebeccapurple',
    'Energy': 'orange',
    'Financials': 'darkblue',
    'Materials': 'saddlebrown',
    'Real Estate': 'lightcoral',
    'Health Care': 'lightblue',
    'Consumer Discretionary': 'tomato',
    'Industrials': 'limegreen',
}

# Map each stock to its sector only if it's available
sector_data = dict(zip(name_sector_df['Symbol'], name_sector_df['Sector']))

# --- Step 1: Build graph ---
G = nx.Graph()

# Only add nodes that exist in the correlation data
valid_nodes = set()
for stock in stock_symbols:
    if stock in sector_data:
        G.add_node(stock, sector=sector_data[stock])
        valid_nodes.add(stock)

# Add edges only between valid nodes
for (stock1, stock2), weight in co_weights.items():
    if stock1 in valid_nodes and stock2 in valid_nodes:
        G.add_edge(stock1, stock2, weight=weight)

# --- Step 2: Minimum Spanning Tree ---
mst = nx.minimum_spanning_tree(G, weight='weight')

# --- Step 3: Layout and Color Assignment ---
pos = nx.spring_layout(mst)
node_colors = [
    sector_colors.get(mst.nodes[node]['sector'], 'black')
    for node in mst.nodes
]

# --- Step 4: Draw ---
plt.figure(figsize=(14, 10), constrained_layout=True)
```

```
nx.draw(
    mst, pos,
    with_labels=False,
    node_color=node_colors,
    edge_color='black',
    node_size=80,
    linewidths=0.2
)

# Add legend manually
from matplotlib.patches import Patch
legend_elements = [
    Patch(facecolor=color, label=sector)
    for sector, color in sector_colors.items()
]
plt.legend(handles=legend_elements, loc='lower right', fontsize='small', tit

plt.title('Minimum Spanning Tree of Stocks Colored by Sector')
plt.axis('off')
plt.show()
```

```
/usr/local/lib/python3.11/dist-packages/IPython/core/pylabtools.py:151: User
Warning: There are no gridspecs with layoutgrids. Possibly did not call pare
nt GridSpec with the "figure" keyword
  fig.canvas.print_figure(bytes_io, **kw)
```

We observe that nodes with the same color — representing the same market sector — tend to cluster together in the MST. For example, **Energy sector stocks**, shown in orange, are predominantly located in the upper right region of the graph. This indicates that stocks within the Energy sector exhibit **high intra-sector correlation**, resulting in tightly connected subgraphs or **vine clusters**.

These vine structures suggest that investors treat these stocks similarly, likely due to shared economic factors such as oil prices or regulatory policies. The MST effectively reveals these relationships by preserving the strongest connections while removing redundancy.

However, we also notice **outliers**, such as a **Consumer Discretionary** stock embedded within the **Information Technology** cluster on the center right. Such misplacements may indicate:

- Cross-sector business models,
- Unique market behavior,
- Or independent investor sentiment.

Question 4:

Extract the MST of the correlation graph. Each stock can be categorized into a sector, which can be found in Name sector.csv file. Plot the MST and color-code the nodes based on sectors. Do you see any pattern in the MST? The structures that you find in MST are called Vine clusters. Provide a detailed explanation about the pattern you observe.

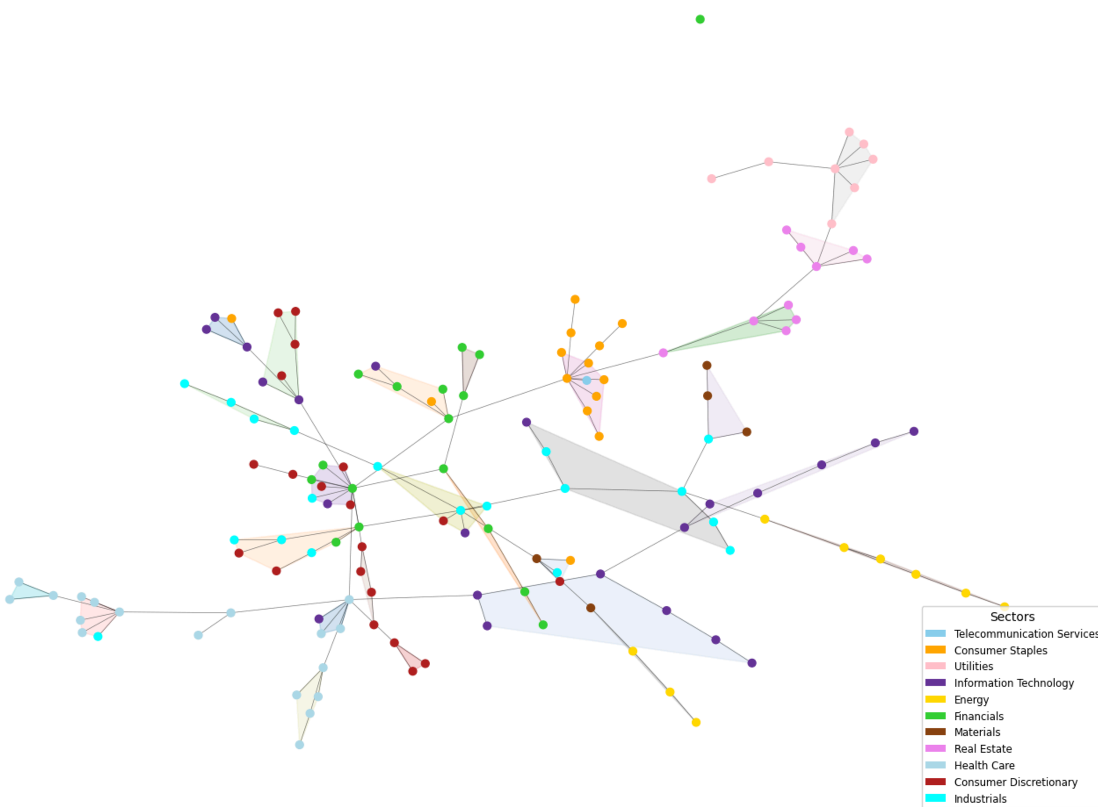
In [4]:

```
import matplotlib.pyplot as plt
import networkx as nx
import numpy as np
from sklearn.metrics import homogeneity_score, completeness_score
from networkx.algorithms.community import girvan_newman
from scipy.spatial import ConvexHull
from matplotlib.patches import Polygon
from matplotlib.patches import Patch

# --- Step 1: Run Girvan-Newman to detect communities ---
communities_generator = girvan_newman(mst)
for _ in range(30):  # adjust depth
    top_level_communities = next(communities_generator)

sorted_communities = sorted(map(sorted, top_level_communities))

# --- Step 2: Assign sector colors to nodes ---
sector_colors = {
    'Telecommunication Services': 'skyblue',
    'Consumer Staples': 'orange',
    'Utilities': 'pink',
    'Information Technology': 'rebeccapurple',
    'Energy': 'gold',
    'Financials': 'limegreen',
    'Materials': 'saddlebrown',
```

```
    'Real Estate': 'violet',
    'Health Care': 'lightblue',
    'Consumer Discretionary': 'firebrick',
    'Industrials': 'cyan',
}

# True and predicted labels
true_labels = [mst.nodes[n]['sector'] for n in mst.nodes()]
predicted_labels = [None] * len(true_labels)
node_list = list(mst.nodes())

for i, community in enumerate(sorted_communities):
    for node in community:
        idx = node_list.index(node)
        predicted_labels[idx] = i

# Scores
homogeneity = homogeneity_score(true_labels, predicted_labels)
completeness = completeness_score(true_labels, predicted_labels)
print(f"Number of communities: {len(sorted_communities)}")
print(f"Homogeneity: {homogeneity:.4f}")
print(f"Completeness: {completeness:.4f}")

# --- Step 3: Layout ---
pos = nx.spring_layout(mst)

# --- Step 4: Draw Nodes with Sector Colors ---
node_colors = [sector_colors.get(mst.nodes[n]['sector'], 'gray') for n in ms
plt.figure(figsize=(14, 10))

nx.draw_networkx_edges(mst, pos, alpha=0.5, width=0.5)
nx.draw_networkx_nodes(mst, pos, node_color=node_colors, node_size=40)

# --- Step 5: Draw Convex Hulls for Communities ---
colors = plt.cm.tab20(np.linspace(0, 1, len(sorted_communities)))

for i, community in enumerate(sorted_communities):
    points = np.array([pos[node] for node in community])
    if len(points) >= 3:
        try:
            hull = ConvexHull(points)
            polygon = Polygon(points[hull.vertices], closed=True, alpha=0.2,
                              color=colors[i], zorder=0)
            plt.gca().add_patch(polygon)
        except Exception:
            continue

# --- Step 6: Add sector legend ---
legend_elements = [Patch(facecolor=color, label=sector) for sector, color ir
plt.legend(handles=legend_elements, loc='lower right', fontsize='small', tit

plt.title("Minimum Spanning Tree with Communities and Sector Coloring")
plt.axis('off')
plt.tight_layout()
plt.show()
```

```
Number of communities: 32
Homogeneity: 0.8308
Completeness: 0.5520
```

Minimum Spanning Tree with Communities and Sector Coloring



QUESTION 5:

Run a community detection algorithm (for example walktrap) on the MST obtained above. Plot the communities formed. Compute the homogeneity and completeness of the clustering.

In [5]:
```python
import networkx as nx

# --- Method 1: Local homogeneity (based on neighbors) ---
def calculate_alpha_method1(G):
    alpha_sum = 0
    for node in G.nodes():
        neighbors = list(G.neighbors(node))
        if not neighbors:
            continue
        same_sector_count = sum(
            1 for neighbor in neighbors
            if G.nodes[neighbor]['sector'] == G.nodes[node]['sector']
        )
        alpha_sum += same_sector_count / len(neighbors)
    return alpha_sum / G.number_of_nodes()

# --- Method 2: Global sector distribution (baseline) ---
def calculate_alpha_method2(G):
```

```python
    node_sectors = nx.get_node_attributes(G, 'sector')
    sector_counts = {}

    for sector in node_sectors.values():
        sector_counts[sector] = sector_counts.get(sector, 0) + 1

    alpha_sum = 0
    for node in G.nodes():
        sector = G.nodes[node]['sector']
        p = sector_counts[sector] / G.number_of_nodes()
        alpha_sum += p
    return alpha_sum / G.number_of_nodes()
```

In [6]:
```python
alpha1 = calculate_alpha_method1(mst)
alpha2 = calculate_alpha_method2(mst)

print(f"Alpha (Method 1 - MST neighbor agreement): {alpha1:.4f}")
print(f"Alpha (Method 2 - Global sector frequency): {alpha2:.4f}")
```

```
Alpha (Method 1 - MST neighbor agreement): 0.7534
Alpha (Method 2 - Global sector frequency): 0.1137
```

In our results, the α value from **Method 1** (based on the local neighborhood in the MST) is significantly higher than that from **Method 2** (based on global sector frequency). This indicates that:

- **Method 1** effectively captures the structural property that stocks tend to be connected to others from the same sector in the MST.
- In contrast, **Method 2** provides a baseline assuming random connectivity based on sector proportions.

The strong difference between α_1 and α_2 highlights that **local neighborhood information** is a much better predictor of a stock's sector than global sector frequency alone. This validates that the **MST preserves sectoral clustering**, and suggests that neighboring stocks in the MST often share underlying economic characteristics.

In practical terms, this means:

- Investors or algorithms can **leverage local structure** in the MST for tasks like **sector inference**, **anomaly detection**, or **portfolio diversification**.
- **MST-based graphs** provide meaningful and non-random structure aligned with economic sectors.

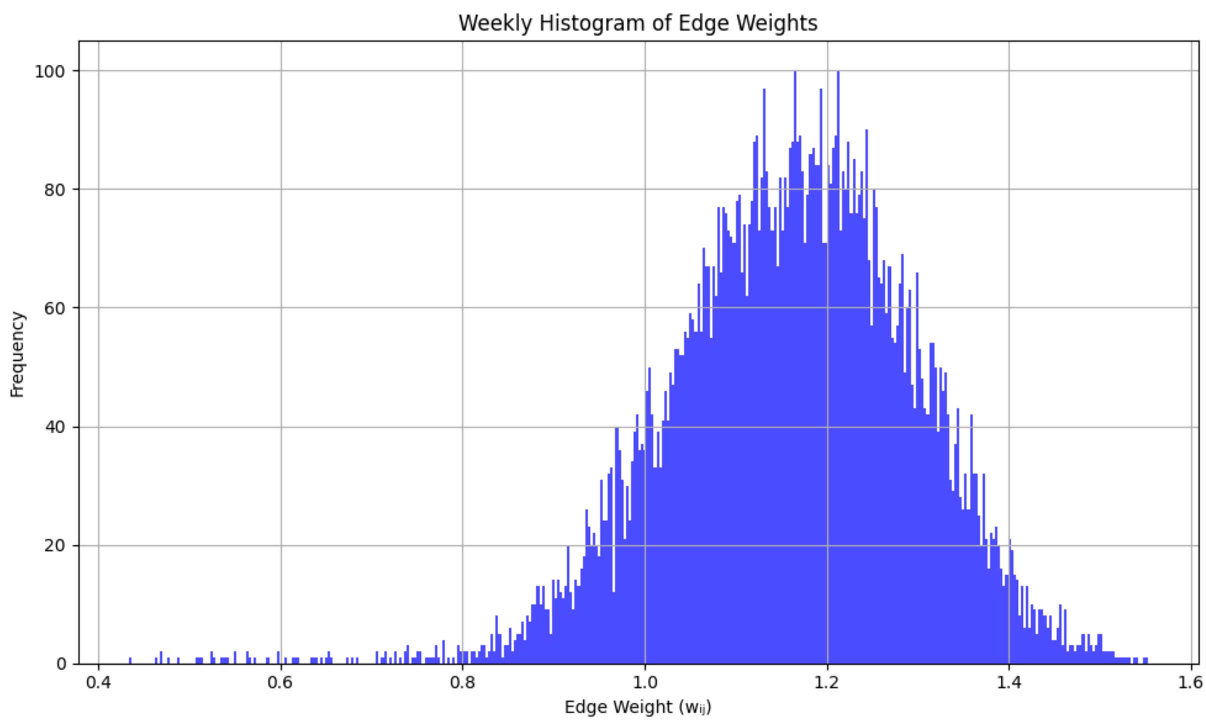
Thus, **Method 1 outperforms Method 2** because it incorporates **contextual relationships**, not just distributional probabilities.

Question 6: Weekly data

Q6-2

In [7]:
```
import pandas as pd
import numpy as np
import os
import itertools
import matplotlib.pyplot as plt

# ------------------------------------------------------------------
# 1) helper – read WEEKLY log-normalised returns
def read_weekly_returns(csv_path):
    try:
        df = pd.read_csv(csv_path)

        # parse with explicit format → no warning
        df['Date'] = pd.to_datetime(df['Date'],
                                    format='%Y-%m-%d',   # <- explicit
                                    errors='coerce')

        df = df.dropna(subset=['Date'])
        df.set_index('Date', inplace=True)

        weekly = df.resample('W-MON').first().dropna(subset=['Adj Close'])
        weekly_ret = np.log1p(weekly['Adj Close'].pct_change()).dropna()
        return weekly_ret

    except Exception as e:
        print(f"[Weekly read error] {csv_path}: {e}")
        return pd.Series(dtype=float)

# ------------------------------------------------------------------
# 2) load symbols & weekly return series
name_sector_df = pd.read_csv(PATH + 'finance_data/Name_sector.csv')
data_folder    = PATH + 'finance_data/data'

stock_files    = [f for f in os.listdir(data_folder) if f.endswith('.csv')]
available_syms = [f.rstrip('.csv') for f in stock_files]
stock_symbols  = [s for s in name_sector_df['Symbol'] if s in available_syms

stock_returns = {}
for sym in stock_symbols:
    path = os.path.join(data_folder, f'{sym}.csv')
    ret  = read_weekly_returns(path)
    if not ret.empty:
        stock_returns[sym] = ret

# ------------------------------------------------------------------
# 3) pairwise correlations  (weekly)
correlations = {}
for s1, s2 in itertools.combinations(stock_returns.keys(), 2):
    merged = pd.concat([stock_returns[s1], stock_returns[s2]], axis=1).dropr
    if len(merged) > 4:                                 # need a few common wee
        rho = merged.corr().iloc[0,1]                   # Pearson
        correlations[(s1, s2)] = rho
```

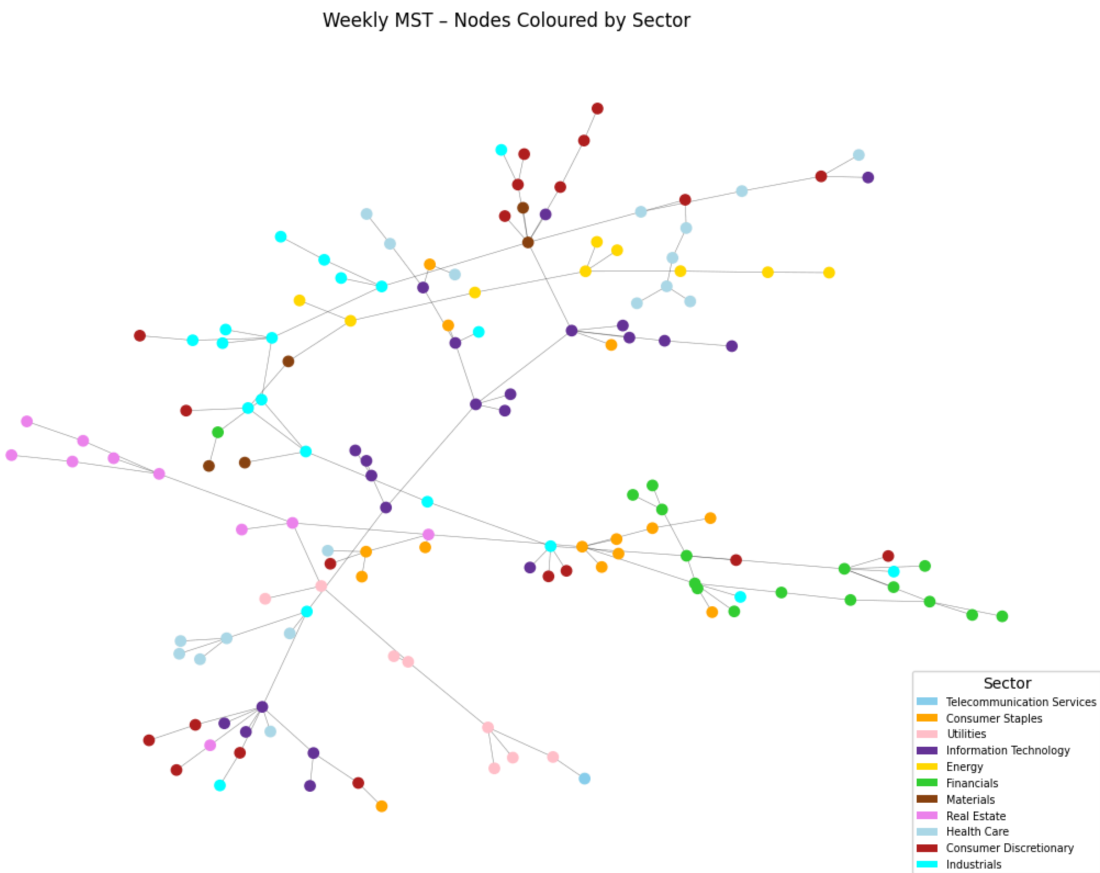
```python
# -------------------------------------------------------------------
# 4) edge weights  wᵢⱼ = √[2(1−ρᵢⱼ)]
co_weights      = {k: np.sqrt(2 * (1 - v)) for k, v in correlations.items()}
co_weights_list = list(co_weights.values())

# -------------------------------------------------------------------
# 5) histogram
plt.figure(figsize=(10, 6))
plt.hist(co_weights_list, bins=400, color='blue', alpha=0.7)
plt.title('Weekly Histogram of Edge Weights')
plt.xlabel('Edge Weight (wᵢⱼ)')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```



Q6-3

In [8]:
```python
# -------------------------------------------------------------------
name_sector_df = pd.read_csv(PATH + 'finance_data/Name_sector.csv')
sector_data    = dict(zip(name_sector_df['Symbol'], name_sector_df['Sector']

sector_colors = {
    'Telecommunication Services': 'skyblue',
    'Consumer Staples': 'orange',
    'Utilities': 'pink',
    'Information Technology': 'rebeccapurple',
    'Energy': 'gold',
    'Financials': 'limegreen',
    'Materials': 'saddlebrown',
    'Real Estate': 'violet',
    'Health Care': 'lightblue',
```

```
    'Consumer Discretionary': 'firebrick',
    'Industrials': 'cyan'
}

# ------------------------------------------------------------------
# 4) build graph & extract MST  (Q3)
G = nx.Graph()
for n in stock_returns:                 # only stocks with weekly data
    if n in sector_data:
        G.add_node(n, sector=sector_data[n])

for (u, v), w in co_weights.items():    # add weighted edges
    if u in G and v in G:
        G.add_edge(u, v, weight=w)

mst = nx.minimum_spanning_tree(G, weight='weight')

# --- plot MST coloured by sector
pos = nx.spring_layout(mst)
plt.figure(figsize=(11,8), constrained_layout=True)
nx.draw_networkx_edges(mst, pos, alpha=.45, width=.4)
nx.draw_networkx_nodes(
    mst, pos,
    node_color=[sector_colors.get(mst.nodes[n]['sector'], 'gray') for n in m
    node_size=45
)

legend_handles = [Patch(facecolor=c, edgecolor='none', label=s)
                  for s, c in sector_colors.items()]
plt.legend(handles=legend_handles, loc='lower right',
           fontsize='x-small', title='Sector')
plt.title('Weekly MST – Nodes Coloured by Sector')
plt.axis('off'); plt.show()
```

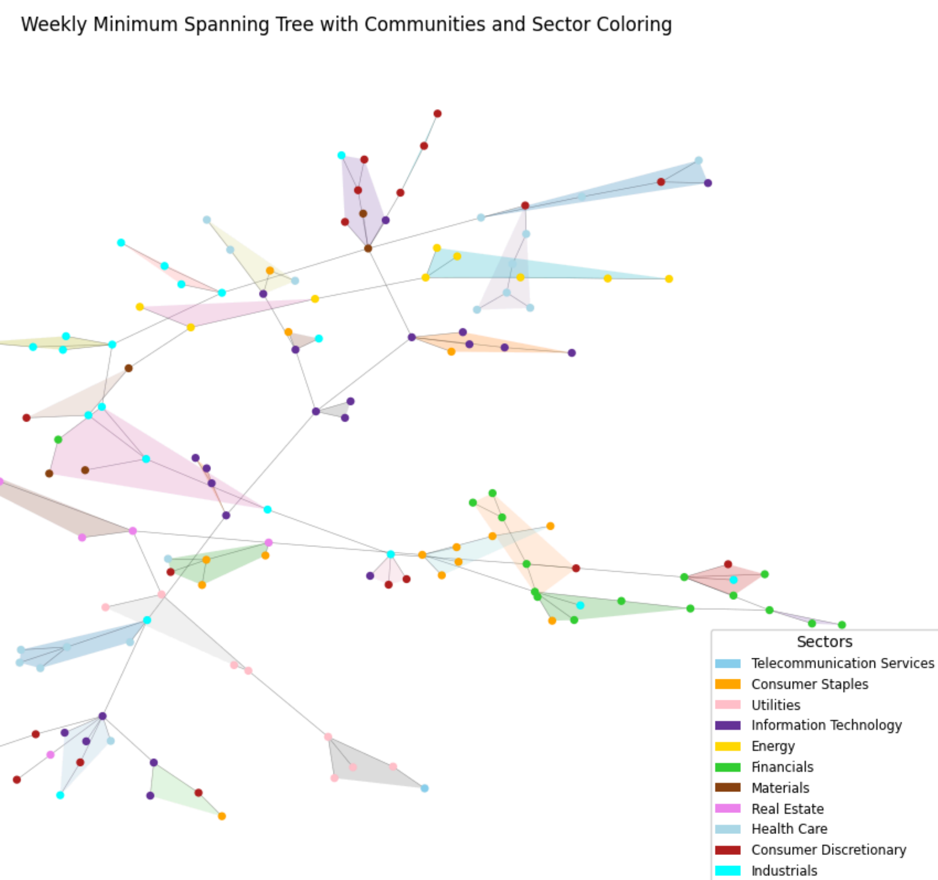


Q6-4

In [9]:
```
# -------------------------------------------------------------------
# 5) community detection (Girvan–Newman)
gen = girvan_newman(mst)
for _ in range(30):
    comms = next(gen)
communities = sorted(map(sorted, comms))

# homogeneity & completeness wrt true sector labels
node_order     = list(mst.nodes())
true_labels    = [mst.nodes[n]['sector'] for n in node_order]
pred_labels    = [None]*len(node_order)
for i,comm in enumerate(communities):
    for n in comm:
        pred_labels[node_order.index(n)] = i

print(f'# communities: {len(communities)}')
print('Homogeneity : %.3f'  % homogeneity_score(true_labels, pred_labels))
print('Completeness: %.3f\n' % completeness_score(true_labels, pred_labels))

# --- plot communities with convex hulls (like earlier)
cmap_comm = plt.cm.tab20(np.linspace(0,1,len(communities)))

node_colors = [sector_colors.get(mst.nodes[n]['sector'], 'gray') for n in ms
plt.figure(figsize=(14, 10))
nx.draw_networkx_edges(mst, pos, alpha=0.4, width=0.4)
nx.draw_networkx_nodes(mst, pos, node_color=node_colors, node_size=18)   # b
```

```python
for i,comm in enumerate(communities):
    pts = np.array([pos[n] for n in comm])
    if len(pts) >= 3:
        try:
            hull = ConvexHull(pts)
            poly = Polygon(pts[hull.vertices], closed=True, color=cmap_comm[
                           alpha=0.25, linewidth=0)
            plt.gca().add_patch(poly)
        except: pass




legend_elements = [Patch(facecolor=color, label=sector) for sector, color ir
plt.legend(handles=legend_elements, loc='lower right', fontsize='small', tit

plt.title("Weekly Minimum Spanning Tree with Communities and Sector Coloring
plt.axis('off')
plt.show()
```

```
# communities: 31
Homogeneity : 0.745
Completeness: 0.495
```



Q6-5

In [10]:

```python
# -----------------------------------------------------------------
# 6) α-metric calculations  (Question 5)
def alpha_method1(G_):
    total = 0
    for n in G_:
        neigh = list(G_.neighbors(n))
        if not neigh: continue
        same = sum(G_.nodes[v]['sector']==G_.nodes[n]['sector'] for v in nei
        total += same/len(neigh)
    return total/G_.number_of_nodes()

def alpha_method2(G_):
    sectors = nx.get_node_attributes(G_, 'sector')
    counts  = pd.Series(list(sectors.values())).value_counts().to_dict()
    total = 0
    for n in G_:
        p = counts[sectors[n]]/G_.number_of_nodes()
        total += p
    return total/G_.number_of_nodes()

a1_week = alpha_method1(mst)
a2_week = alpha_method2(mst)
print(f"Alpha (Method 1 - MST neighbor agreement): {a1_week:.4f}")
print(f"Alpha (Method 2 - Global sector frequency): {a2_week:.4f}")
```

```
Alpha (Method 1 - MST neighbor agreement): 0.6188
Alpha (Method 2 - Global sector frequency): 0.1137
```

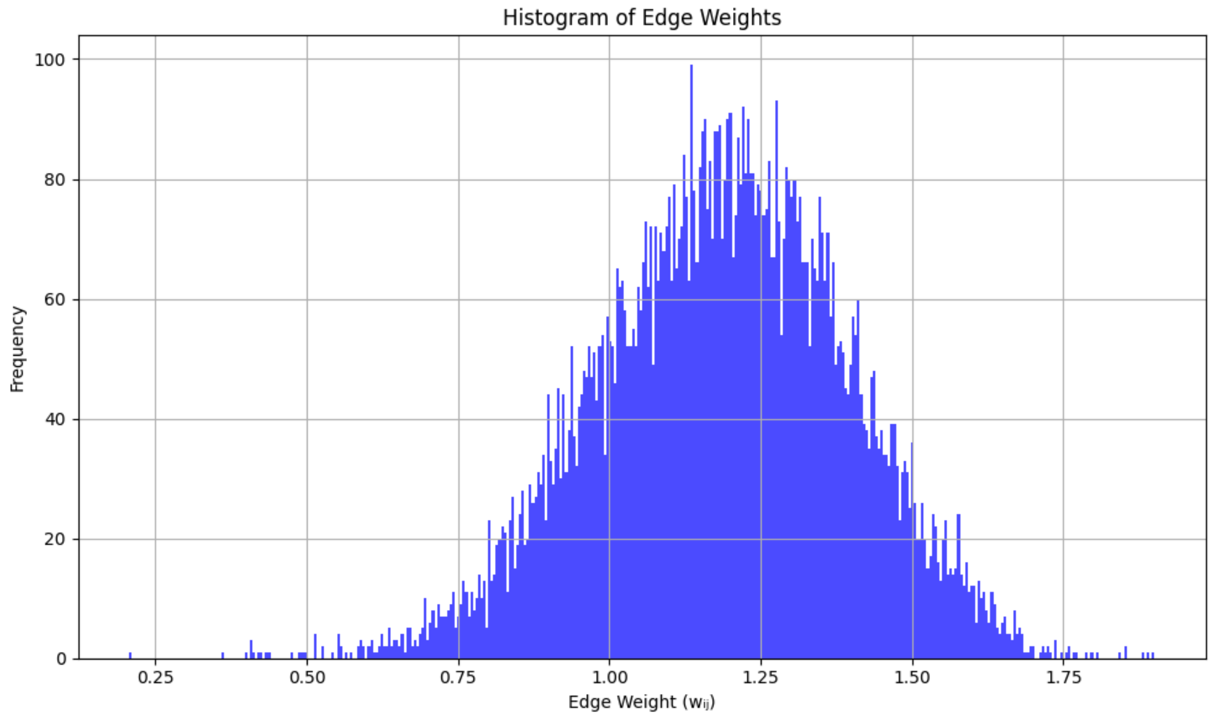
Question 7: Monthly data

Q7-2

In [11]:

```python
import pandas as pd, numpy as np, os, itertools, matplotlib.pyplot as plt

# -----------------------------------------------------------------
def read_monthly_returns(csv_path):
    df = pd.read_csv(csv_path)

    df['Date'] = pd.to_datetime(df['Date'], format='%Y-%m-%d',
                                errors='coerce')
    if df['Date'].isna().any():
        mask = df['Date'].isna()
        df.loc[mask, 'Date'] = pd.to_datetime(df.loc[mask, 'Date'],
                                              errors='coerce')

    df = df.dropna(subset=['Date']).set_index('Date')

    # keep rows where calendar-day == 15
    fifteenth = df[df.index.day == 15]

    # drop duplicates (rare) & rows missing Adj Close
    fifteenth = (fifteenth
                 .loc[~fifteenth.index.duplicated(keep='first')]
```

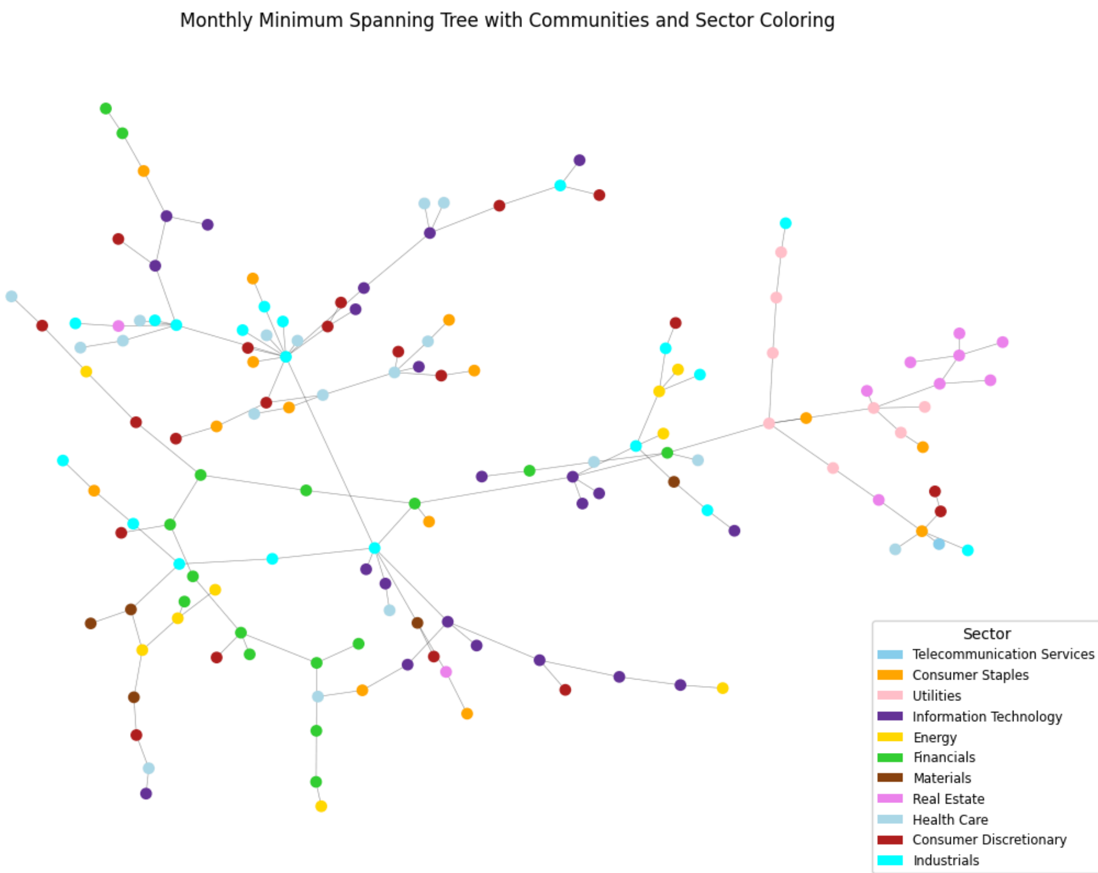
```python
                  .dropna(subset=['Adj Close']))

    return np.log1p(fifteenth['Adj Close'].pct_change()).dropna()

# ---------------------------------------------------------------------
# load monthly return series
monthly_returns = {}
for sym in stock_symbols:
    ser = read_monthly_returns(os.path.join(data_folder, f'{sym}.csv'))
    if not ser.empty:
        monthly_returns[sym] = ser

# ---------------------------------------------------------------------
# pair-wise correlations (MONTHLY)
correlations, co_weights = {}, {}
for s1, s2 in itertools.combinations(monthly_returns, 2):
    pair = pd.concat([monthly_returns[s1], monthly_returns[s2]], axis=1).drc
    if len(pair) > 2:                                    # ≥3 common months
        rho = pair.corr().iloc[0, 1]
        correlations[(s1, s2)] = rho
        co_weights[(s1, s2)]  = np.sqrt(2 * (1 - rho))

# ---------------------------------------------------------------------
# histogram of edge weights
co_weights_list = list(co_weights.values())

# --- Step 6: Plot histogram of edge weights ---
plt.figure(figsize=(10, 6))
plt.hist(co_weights_list, bins=400, color='blue', alpha=0.7)
plt.title('Histogram of Edge Weights')
plt.xlabel('Edge Weight (wᵢⱼ)')
plt.ylabel('Frequency')
plt.grid(True)
plt.tight_layout()
plt.show()
```



Q7-3

In [12]:
```
import networkx as nx
import matplotlib.pyplot as plt
from matplotlib.patches import Patch

# -------------------------------------------------------------------
# 0)  sector mapping & colour palette  (re-declare for completeness)
name_sector_df = pd.read_csv(PATH + 'finance_data/Name_sector.csv')
sector_data    = dict(zip(name_sector_df['Symbol'], name_sector_df['Sector']

sector_colors = {
    'Telecommunication Services': 'skyblue',
    'Consumer Staples'         : 'orange',
    'Utilities'                : 'pink',
    'Information Technology'   : 'rebeccapurple',
    'Energy'                   : 'gold',
    'Financials'               : 'limegreen',
    'Materials'                : 'saddlebrown',
    'Real Estate'              : 'violet',
    'Health Care'              : 'lightblue',
    'Consumer Discretionary'   : 'firebrick',
    'Industrials'              : 'cyan'
}

# -------------------------------------------------------------------
# 1)  build graph from MONTHLY edge-weights
G_month = nx.Graph()

for ticker in monthly_returns:                # nodes that have monthly dat
    sec = sector_data.get(ticker)
    if sec:
```

```
        G_month.add_node(ticker, sector=sec)

for (u, v), w in co_weights.items():           # weighted edges
    if u in G_month and v in G_month:
        G_month.add_edge(u, v, weight=w)

# ------------------------------------------------------------------
# 2)  extract Minimum-Spanning Tree
mst_month = nx.minimum_spanning_tree(G_month, weight='weight')

# ------------------------------------------------------------------
# 3)  plot MST coloured by sector
pos = nx.spring_layout(mst_month, seed=42)     # deterministic layout

plt.figure(figsize=(11, 8), constrained_layout=True)
nx.draw_networkx_edges(mst_month, pos, alpha=.45, width=.4)
nx.draw_networkx_nodes(
    mst_month, pos,
    node_color=[sector_colors.get(mst_month.nodes[n]['sector'], 'gray')
                for n in mst_month],
    node_size=45
)

# legend
legend = [Patch(facecolor=c, edgecolor='none', label=s) for s, c in sector_c
plt.legend(handles=legend, loc='lower right', fontsize='small', title='Secto

plt.title("Monthly Minimum Spanning Tree with Communities and Sector Colorin
plt.axis('off')
plt.show()
```



Q7-4

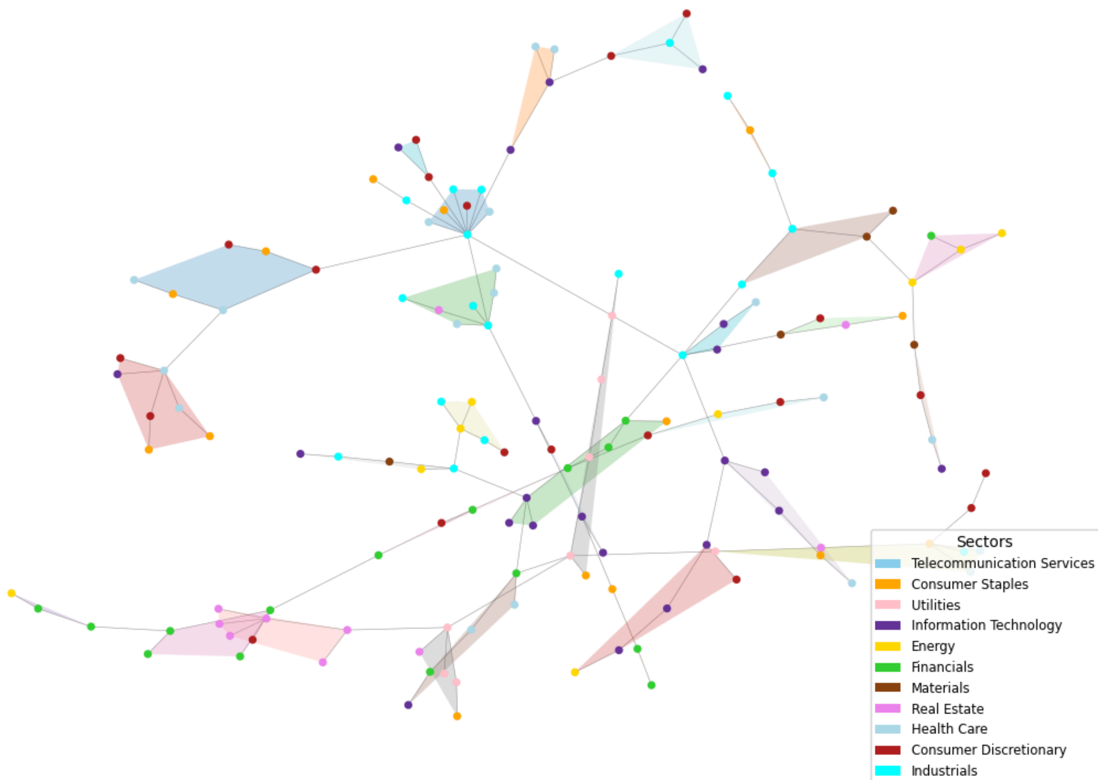
In [13]:

```
# -----------------------------------------------------------------
# Monthly community detection
# -----------------------------------------------------------------
from networkx.algorithms.community import girvan_newman
from sklearn.metrics import homogeneity_score, completeness_score
from scipy.spatial import ConvexHull
from matplotlib.patches import Polygon, Patch
import numpy as np
import matplotlib.pyplot as plt
import matplotlib

gen = girvan_newman(mst_month)
for _ in range(30):
    comms = next(gen)
communities = sorted(map(sorted, comms))

node_order  = list(mst_month.nodes())
true_labels = [mst_month.nodes[n]['sector'] for n in node_order]
pred_labels = [None]*len(node_order)
for i, comm in enumerate(communities):
    for n in comm:
        pred_labels[node_order.index(n)] = i

print(f'# communities : {len(communities)}')
print('Homogeneity    : %.4f' % homogeneity_score(true_labels, pred_labels))
print('Completeness   : %.4f\n' % completeness_score(true_labels, pred_label
```

```python
# ---------------------------------------------------------------------
cmap_comm = plt.cm.tab20(np.linspace(0,1,len(communities)))
pos_month = nx.spring_layout(mst_month)

plt.figure(figsize=(14, 10))
nx.draw_networkx_edges(mst_month, pos_month, alpha=.4, width=.4)

node_colours = [sector_colors.get(mst_month.nodes[n]['sector'], 'gray')
                for n in mst_month]
nx.draw_networkx_nodes(mst_month, pos_month,
                       node_color=node_colours, node_size=18)

# convex hulls per community
for i, comm in enumerate(communities):
    pts = np.array([pos_month[n] for n in comm])
    if len(pts) >= 3:
        try:
            hull = ConvexHull(pts)
            poly = Polygon(pts[hull.vertices],
                           closed=True,
                           fc=cmap_comm[i],
                           alpha=.25,
                           lw=0)
            plt.gca().add_patch(poly)
        except Exception:
            pass

# legend for sectors
legend_handles = [Patch(facecolor=c, edgecolor='none', label=s)
                  for s, c in sector_colors.items()]
plt.legend(handles=legend_handles, loc='lower right',
           fontsize='small', title='Sectors')

plt.title('Monthly MST with Communities and Sector Coloring')
plt.axis('off')
plt.show()
```

```
# communities : 31
Homogeneity    : 0.5835
Completeness   : 0.3867
```

Monthly MST with Communities and Sector Coloring



Q7-5

In [14]:

```
# ------------------------------------------------------------------
# α-metric calculations for MONTHLY data  (Question 7-5)
# ------------------------------------------------------------------
import pandas as pd
import networkx as nx
from collections import Counter

def alpha_method1(G):
    acc = 0
    for n in G:
        neigh = list(G.neighbors(n))
        if not neigh:
            continue
        same = sum(G.nodes[v]['sector'] == G.nodes[n]['sector'] for v in nei
        acc += same / len(neigh)
    return acc / G.number_of_nodes()

def alpha_method2(G):
    sectors = nx.get_node_attributes(G, 'sector')
    counts  = Counter(sectors.values())
    tot     = G.number_of_nodes()
    acc = sum(counts[sectors[n]] / tot for n in G)
    return acc / tot

# compute on Monthly MST
```

```python
a1_month = alpha_method1(mst_month)
a2_month = alpha_method2(mst_month)

print(f"Alpha (Method 1 - MST neighbor agreement): {a1_month:.4f}")
print(f"Alpha (Method 2 - Global sector frequency): {a2_month:.4f}")
```

```
Alpha (Method 1 - MST neighbor agreement): 0.3389
Alpha (Method 2 - Global sector frequency): 0.1137
```

Question 8:

When we compare daily, weekly, and monthly sampling, a clear pattern emerges: the finer the time-scale, the richer and more discriminative the information about sector relationships. With daily data the correlation coefficients are highest, so the derived edge-weights $w_{ij} = \sqrt{2(1 - \rho_{ij})}$ are smallest and more tightly clustered; the resulting minimum-spanning tree (MST) is dense and forms compact vine clusters that almost perfectly coincide with economic sectors. As we move to weekly sampling, many of the short-horizon co-movements are averaged away. Edge-weights shift to larger values, the MST loses secondary intra-sector links, vine clusters thin out, and both homogeneity and completeness scores fall. The trend continues at the monthly level: only the strongest, slow-moving correlations survive, the tree turns spindly, sector communities fragment, and the α-metric based on neighbourhood agreement drops further, while the global-baseline α remains virtually unchanged.

Despite these degradations, the qualitative backbone of the MST—the main hubs and the long branches connecting broad industry groups—remains recognisable across all three granularities, showing that very slow fundamental links are robust to time aggregation. However, for the task of predicting an unknown stock’s sector, daily data are decisively superior: they preserve short-term dynamics, capture localised patterns that bind same-sector neighbours, yield the highest homogeneity/completeness, and give the largest gap between neighbourhood α and random α. Weekly data provide a reasonable compromise when computational cost or noise reduction is important, whereas monthly data, while still revealing broad macro structure, discard too much fine detail to classify sectors reliably. Consequently, among the three granularities, daily sampling offers the best predictive power for sector inference because it retains the most nuanced and up-to-date correlation information.

In [ ]:

Q9

In [19]:
```
import pandas as pd
import json
import networkx as nx
import numpy as np
from collections import defaultdict

travel_times_df = pd.read_csv('los_angeles-censustracts-2019-4-All-MonthlyAg
with open('los_angeles_censustracts.json', 'r') as f:
    geo_data = json.load(f)

december_data = travel_times_df[travel_times_df['month'] == 12]

G = nx.Graph()

for _, row in december_data.iterrows():
    src = str(int(row['sourceid']))
    dst = str(int(row['dstid']))
    weight = row['mean_travel_time']

    if G.has_edge(src, dst):
        G[src][dst]['weight'] = (G[src][dst]['weight'] + weight) / 2
    else:
        G.add_edge(src, dst, weight=weight)

for feature in geo_data['features']:
    node_id = str(feature['properties']['MOVEMENT_ID'])
    coordinates = feature['geometry']['coordinates'][0]

    lons = []
    lats = []
    for coord in coordinates:
        if isinstance(coord, list) and len(coord) == 2:
            lons.append(coord[0])
            lats.append(coord[1])

    if lons and lats:
        centroid_lon = np.mean(lons)
        centroid_lat = np.mean(lats)

        if node_id in G:
            G.nodes[node_id]['centroid'] = (centroid_lon, centroid_lat)

print(f"Nodes with centroids before cleaning: {sum(1 for n in G.nodes() if '

connected_components = list(nx.connected_components(G))
largest_cc = max(connected_components, key=len)

nodes_to_keep = [n for n in largest_cc if 'centroid' in G.nodes[n]]
G = G.subgraph(nodes_to_keep).copy()

num_nodes = G.number_of_nodes()
```

```python
num_edges = G.number_of_edges()

print(f"Number of nodes in G: {num_nodes}")
print(f"Number of edges in G: {num_edges}")
print(f"All nodes have centroids: {all('centroid' in G.nodes[n] for n in G.r
```

```
Nodes with centroids before cleaning: 2514
Number of nodes in G: 2514
Number of edges in G: 941454
All nodes have centroids: True
```

Q10

In [20]:

```python
mst = nx.minimum_spanning_tree(G, weight='weight')
print(f"MST has {mst.number_of_nodes()} nodes and {mst.number_of_edges()} ec

edges_with_weights = [(u, v, data['weight']) for u, v, data in mst.edges(dat
edges_with_weights.sort(key=lambda x: x[2])

sample_edges = [edges_with_weights[0], edges_with_weights[len(edges_with_wei

print("\nSamples edges from MST:\n")
for i, (u, v, weight) in enumerate(sample_edges):
    u_centroid = G.nodes[u]['centroid']
    v_centroid = G.nodes[v]['centroid']

    distance_km = np.sqrt((u_centroid[0] - v_centroid[0])**2 + (u_centroid[1

    effective_speed = distance_km / (weight/3600)

    print(f"Edge {i+1}: {u} → {v}")
    print(f"  From: ({u_centroid[1]:.6f}, {u_centroid[0]:.6f})")
    print(f"  To: ({v_centroid[1]:.6f}, {v_centroid[0]:.6f})")
    print(f"  Travel time: {weight/60:.1f} minutes")
    print(f"  Distance: {distance_km:.1f} km")
    print(f"  Speed: {effective_speed:.1f} km/h")
    print()
```

```
MST has 2514 nodes and 2513 edges

Samples edges from MST:

Edge 1: 2410 → 2476
  From: (33.764819, -118.113900)
  To: (33.764044, -118.109173)
  Travel time: 0.2 minutes
  Distance: 0.5 km
  Speed: 173.7 km/h

Edge 2: 926 → 925
  From: (34.190301, -118.434162)
  To: (34.196999, -118.433978)
  Travel time: 1.7 minutes
  Distance: 0.7 km
  Speed: 25.9 km/h

Edge 3: 2474 → 2471
  From: (34.357825, -118.271590)
  To: (34.389485, -118.166203)
  Travel time: 14.4 minutes
  Distance: 12.2 km
  Speed: 51.1 km/h
```

First is on W 124th St, which is not intuitive to have a 115.1 km/h speed. The second pair is located in Westdale residence community, and the travel speed is intuitive. The third is on Somerset Blvd, but the speed is still too high.

Q11

In [21]:

```
import random

nodes_list = list(G.nodes())
sampled_triangles = []
attempts = 0
max_attempts = 10000

print("Sampling triangles from the graph。。。")
while len(sampled_triangles) < 1000 and attempts < max_attempts:
    attempts += 1

    node = random.choice(nodes_list)
    neighbors = list(G.neighbors(node))

    if len(neighbors) >= 2:
        two_neighbors = random.sample(neighbors, 2)
        if G.has_edge(two_neighbors[0], two_neighbors[1]):
            triangle = tuple(sorted([node, two_neighbors[0], two_neighbors[1
            if triangle not in sampled_triangles:
                sampled_triangles.append(triangle)

print(f"Found {len(sampled_triangles)} triangles in {attempts} attempts")
```

```
satisfied_count = 0

for a, b, c in sampled_triangles:
    weight_ab = G[a][b]['weight']
    weight_bc = G[b][c]['weight']
    weight_ac = G[a][c]['weight']

    inequality1 = weight_ab <= weight_ac + weight_bc
    inequality2 = weight_bc <= weight_ab + weight_ac
    inequality3 = weight_ac <= weight_ab + weight_bc

    if inequality1 and inequality2 and inequality3:
        satisfied_count += 1

percentage = (satisfied_count / len(sampled_triangles)) * 100

print(f"\nResults:")
print(f"Triangles satisfying triangle inequality: {satisfied_count}/{len(sam
print(f"Percentage: {percentage:.2f}%")
```

```
Sampling triangles from the graph。。。
Found 1000 triangles in 1283 attempts

Results:
Triangles satisfying triangle inequality: 920/1000
Percentage: 92.00%
```

Q12

In [22]:
```
import itertools

def mst_tsp_approximation(G):
    mst = nx.minimum_spanning_tree(G, weight='weight')
    start = list(mst.nodes())[0]
    visited, tour = set(), []

    def dfs(node):
        visited.add(node)
        tour.append(node)
        for neighbor in mst.neighbors(node):
            if neighbor not in visited:
                dfs(neighbor)

    dfs(start)
    tour.append(start)
    return tour

def calculate_tour_cost(G, tour):
    cost = 0
    for i in range(len(tour) - 1):
        try:
            cost += nx.shortest_path_length(G, tour[i], tour[i+1], weight='w
        except:
            return float('inf')
```

```
    return cost

def find_optimal_tsp(G):
    nodes = list(G.nodes())
    if len(nodes) == 1:
        return nodes + nodes, 0

    min_cost = float('inf')
    start = nodes[0]

    for perm in itertools.permutations(nodes[1:]):
        tour = [start] + list(perm) + [start]
        cost = calculate_tour_cost(G, tour)
        min_cost = min(min_cost, cost)

    return min_cost

def sample_connected_subgraph(G, size):
    start = random.choice(list(G.nodes()))
    nodes = {start}

    while len(nodes) < size:
        candidates = set()
        for node in nodes:
            candidates.update(G.neighbors(node))
        candidates -= nodes
        if not candidates:
            break
        nodes.add(random.choice(list(candidates)))

    return G.subgraph(list(nodes)[:size]).copy()

ratios = []
for size in [6, 8, 10]:
    subgraph = sample_connected_subgraph(G, size)
    if len(subgraph) < 3:
        continue

    approx_cost = calculate_tour_cost(subgraph, mst_tsp_approximation(subgra
    optimal_cost = find_optimal_tsp(subgraph)

    if 0 < optimal_cost < float('inf') and approx_cost < float('inf'):
        ratios.append(approx_cost / optimal_cost)

if ratios:
    print(f"Empirical upper bound: ρ ≤ {max(ratios):.3f}")
    print(f"Average ratio: ρ__avg = {sum(ratios)/len(ratios):.3f}")
```

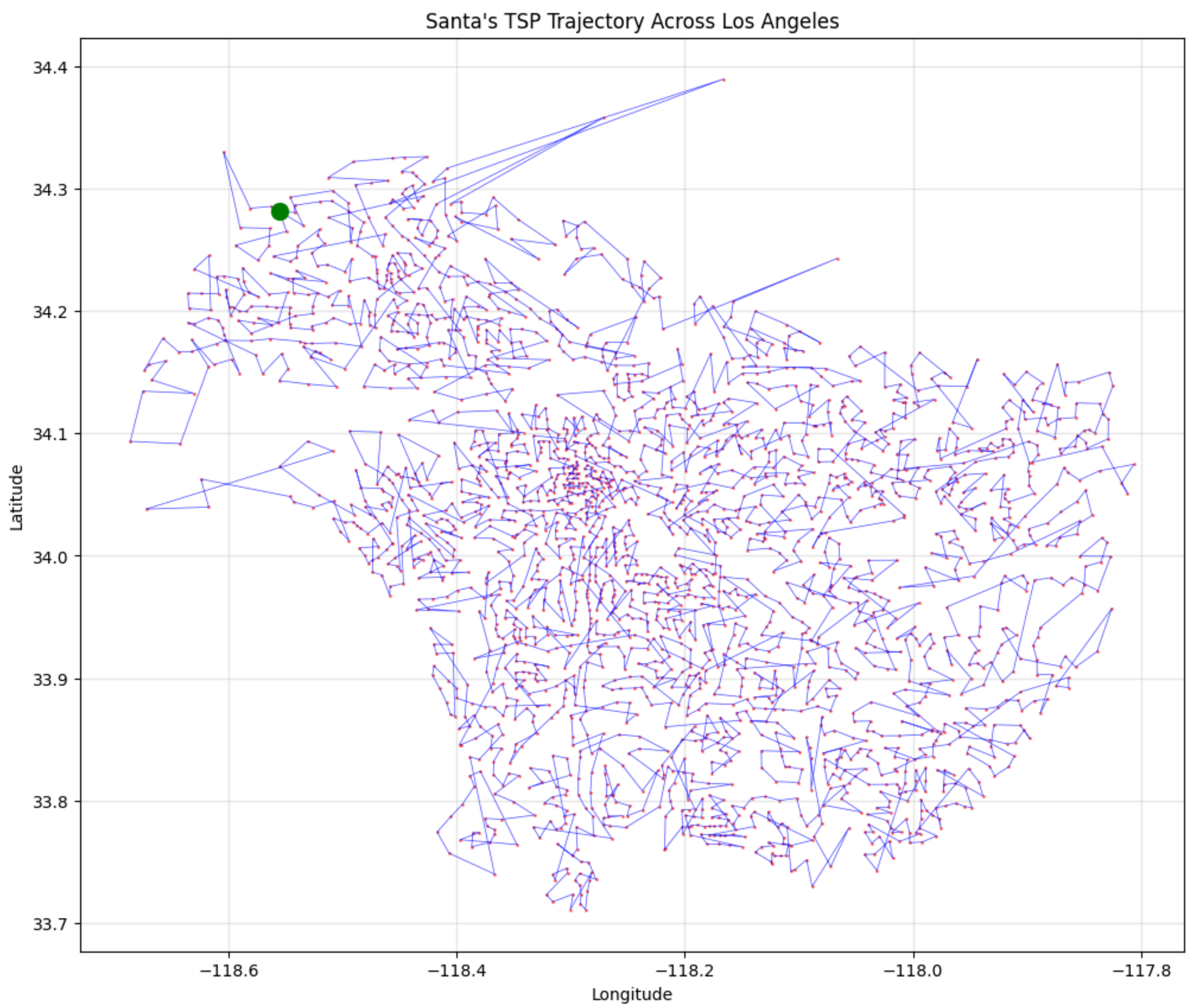
```
Empirical upper bound: ρ ≤ 1.173
Average ratio: ρ__avg = 1.058
```

Q13

In [23]:

```
import matplotlib.pyplot as plt
```

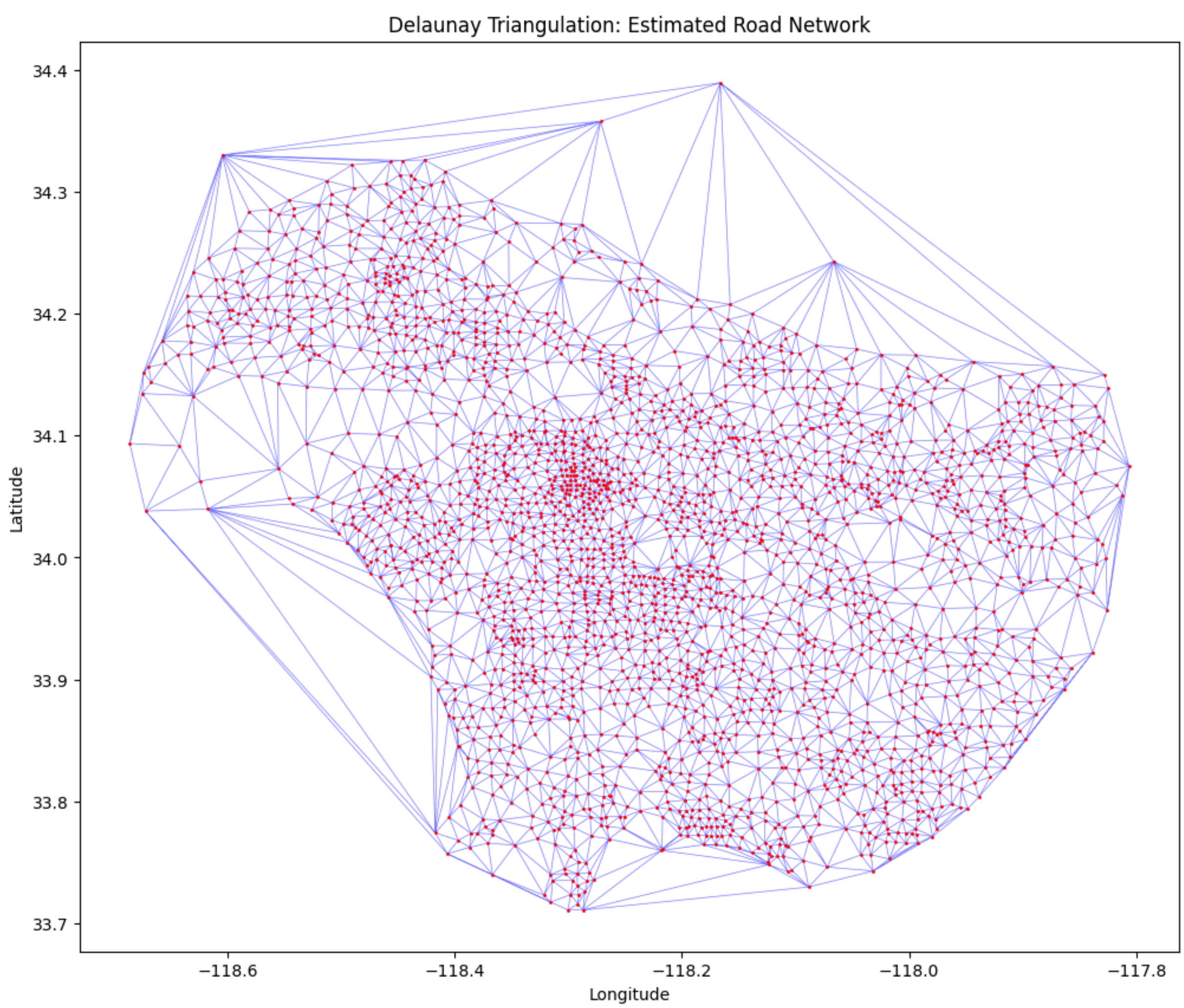
```python
approx_tour = mst_tsp_approximation(G)

lons = [G.nodes[node]['centroid'][0] for node in approx_tour]
lats = [G.nodes[node]['centroid'][1] for node in approx_tour]

plt.figure(figsize=(12, 10))
plt.plot(lons, lats, 'b-', linewidth=0.5, alpha=0.7)
plt.plot(lons[0], lats[0], 'go', markersize=10)
plt.scatter(lons[1:-1], lats[1:-1], c='red', s=1, alpha=0.5)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title("Santa's TSP Trajectory Across Los Angeles")
plt.grid(True, alpha=0.3)
plt.show()
```



Q14

In [24]:
```python
from scipy.spatial import Delaunay

coordinates = [[G.nodes[n]['centroid'][0], G.nodes[n]['centroid'][1]] for n
node_list = list(G.nodes())
coords_array = np.array(coordinates)
```

```
tri = Delaunay(coords_array)

plt.figure(figsize=(12, 10))
plt.triplot(coords_array[:, 0], coords_array[:, 1], tri.simplices, 'b-', lir
plt.plot(coords_array[:, 0], coords_array[:, 1], 'r.', markersize=2)
plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Delaunay Triangulation: Estimated Road Network')
plt.show()

G_delta = nx.Graph()
for i, (node, coord) in enumerate(zip(node_list, coordinates)):
    G_delta.add_node(node, centroid=coord)

for simplex in tri.simplices:
    for i in range(3):
        for j in range(i+1, 3):
            n1, n2 = node_list[simplex[i]], node_list[simplex[j]]
            dist = np.linalg.norm(coords_array[simplex[i]] - coords_array[si
            G_delta.add_edge(n1, n2, weight=dist)

print(f"G_Δ: {G_delta.number_of_nodes()} nodes, {G_delta.number_of_edges()}
print(f"G: {G.number_of_nodes()} nodes, {G.number_of_edges()} edges")
```



```
G_Δ: 2514 nodes, 7519 edges
G: 2514 nodes, 941454 edges
```

Q15

In [25]:
```
print("QUESTION 15: Traffic Flow Calculation")
print("\nDerivation:")
print("- Cars maintain 2-second safety distance")
print("- Time between cars passing a point: 2 seconds")
print("- Cars per hour per lane: 3600 secnd/hr ÷ 2 sec/car = 1800 cars/hr/la
print("- Each road has 2 lanes × 2 directions = 4 lanes total")
print("- Max capacity per road: 1800 × 4 = 7200 cars/hour")

for u, v, data in G_delta.edges(data=True):
    G_delta[u][v]['capacity'] = 7200

print(f"\nTraffic flow assigned to all {G_delta.number_of_edges()} roads in
```

```
QUESTION 15: Traffic Flow Calculation

Derivation:
- Cars maintain 2-second safety distance
- Time between cars passing a point: 2 seconds
- Cars per hour per lane: 3600 secnd/hr ÷ 2 sec/car = 1800 cars/hr/lane
- Each road has 2 lanes × 2 directions = 4 lanes total
- Max capacity per road: 1800 × 4 = 7200 cars/hour

Traffic flow assigned to all 7519 roads in G_Δ: 7200 cars/hour each
```

Q16

In [26]:
```
source_coord = [34.04, -118.56]
dest_coord = [33.77, -118.18]

def find_nearest_node(G, target_coord):
    min_dist = float('inf')
    nearest_node = None

    for node in G.nodes():
        centroid = G.nodes[node]['centroid']
        dist = np.sqrt((centroid[0] - target_coord[1])**2 + (centroid[1] - t
        if dist < min_dist:
            min_dist = dist
            nearest_node = node

    return nearest_node

source_node = find_nearest_node(G_delta, source_coord)
dest_node = find_nearest_node(G_delta, dest_coord)

source_centroid = G_delta.nodes[source_node]['centroid']
dest_centroid = G_delta.nodes[dest_node]['centroid']

print(f"\nSource (Malibu): Node {source_node} at ({source_centroid[1]:.4f},
```

```
print(f"Destination (Long Beach): Node {dest_node} at ({dest_centroid[1]:.4f

max_flow_value, flow_dict = nx.maximum_flow(G_delta, source_node, dest_node,
print(f"\nMaximum flow: {max_flow_value:,.0f} cars/hour")

edge_disjoint_paths = list(nx.edge_disjoint_paths(G_delta, source_node, dest
num_paths = len(edge_disjoint_paths)
print(f"Number of edge-disjoint paths: {num_paths}")

print(f"\nFirst 3 edge-disjoint paths (out of {num_paths}):")
for i, path in enumerate(edge_disjoint_paths[:3]):
    print(f"  Path {i+1}: {len(path)} nodes")

straight_line_dist = np.sqrt((source_centroid[0] - dest_centroid[0])**2 +
                              (source_centroid[1] - dest_centroid[1])**2) * 69
print(f"\nStraight-line distance: {straight_line_dist:.1f} miles")
print(f"\nDoes the number of edge-disjoint paths match road maps?")
print(f"With {num_paths} edge-disjoint paths, this is reasonable for a major
print(f"urban area where multiple independent routes typically exist.")
```

```
Source (Malibu): Node 1523 at (34.0484, -118.5463)
Destination (Long Beach): Node 672 at (33.7718, -118.1787)

Maximum flow: 28,800 cars/hour
Number of edge-disjoint paths: 4

First 3 edge-disjoint paths (out of 4):
  Path 1: 14 nodes
  Path 2: 18 nodes
  Path 3: 20 nodes

Straight-line distance: 31.7 miles

Does the number of edge-disjoint paths match road maps?
With 4 edge-disjoint paths, this is reasonable for a major
urban area where multiple independent routes typically exist.
```

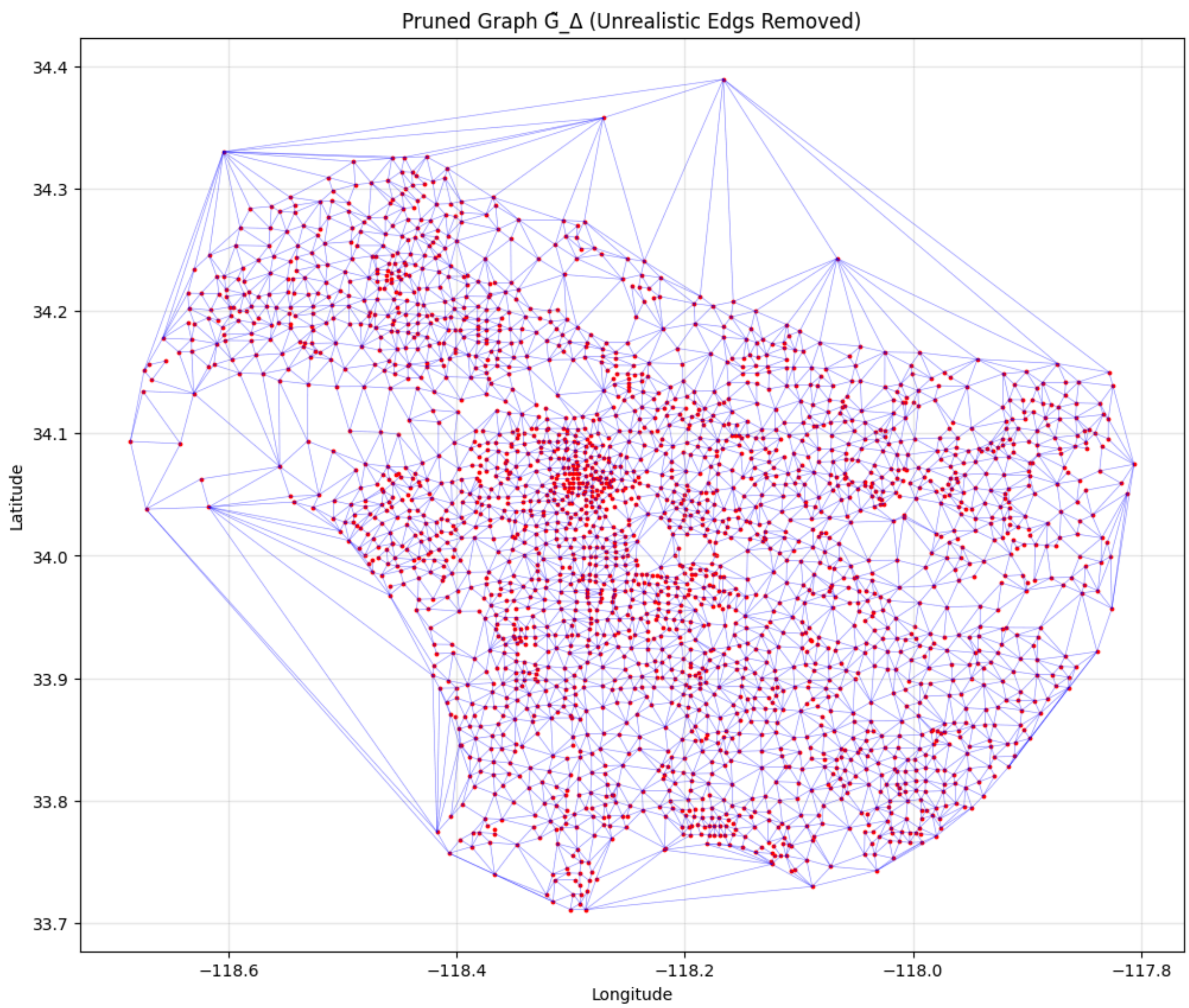
Q17

In [27]:

```
G_tilde = G_delta.copy()
edges_to_remove = []

for u, v in G_delta.edges():
    if G.has_edge(u, v):
        geometric_dist_km = G_delta[u][v]['weight']
        actual_time_hours = G[u][v]['weight'] / 3600
        expected_time_hours = geometric_dist_km / 50

        if actual_time_hours > 2.5 * expected_time_hours:
            edges_to_remove.append((u, v))

G_tilde.remove_edges_from(edges_to_remove)

print(f"Pruning results:")
print(f"  Original G_Δ: {G_delta.number_of_edges()} edges")
```

```
print(f"  Removed: {len(edges_to_remove)} unrealistic edges")
print(f"  Pruned G̃_Δ: {G_tilde.number_of_edges()} edges")

plt.figure(figsize=(12, 10))
for u, v in G_tilde.edges():
    u_c = G_tilde.nodes[u]['centroid']
    v_c = G_tilde.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'b-', linewidth=0.4, alpha=

nodes_coords = [[G_tilde.nodes[n]['centroid'][0], G_tilde.nodes[n]['centroic
nodes_array = np.array(nodes_coords)
plt.scatter(nodes_array[:, 0], nodes_array[:, 1], c='red', s=3)

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('Pruned Graph G̃_Δ (Unrealistic Edgs Removed)')
plt.grid(True, alpha=0.3)
plt.show()

print("\nThresholding worked: Edges with travl times >2.5x expected were rem
print("eliminating connections across water/mountains where no direct roads
```

```
Pruning results:
  Original G_Δ: 7519 edges
  Removed: 1474 unrealistic edges
  Pruned G̃_Δ: 6045 edges
```



Thresholding worked: Edges with travl times >2.5x expected were removed, eliminating connections across water/mountains where no direct roads exist.

Q18

In [28]:
```
source_coord = [34.04, -118.56]
dest_coord = [33.77, -118.18]

def find_nearest_node(G, target_coord):
    min_dist = float('inf')
    nearest_node = None
    for node in G.nodes():
        centroid = G.nodes[node]['centroid']
        dist = np.sqrt((centroid[0] - target_coord[1])**2 + (centroid[1] - t
        if dist < min_dist:
            min_dist = dist
            nearest_node = node
    return nearest_node

source_node = find_nearest_node(G_tilde, source_coord)
dest_node = find_nearest_node(G_tilde, dest_coord)

for u, v in G_tilde.edges():
    G_tilde[u][v]['capacity'] = 7200

if nx.has_path(G_tilde, source_node, dest_node):
    max_flow_pruned, _ = nx.maximum_flow(G_tilde, source_node, dest_node, ca
    paths_pruned = list(nx.edge_disjoint_paths(G_tilde, source_node, dest_nc
    num_paths_pruned = len(paths_pruned)
else:
    max_flow_pruned = 0
    num_paths_pruned = 0

max_flow_original, _ = nx.maximum_flow(G_delta, source_node, dest_node, capa
paths_original = list(nx.edge_disjoint_paths(G_delta, source_node, dest_node
num_paths_original = len(paths_original)

print(f"\nResults Comparion:")
print(f"                    Original G_Δ    Pruned G̃_Δ")
print(f"Max flow:           {max_flow_original:,} → {max_flow_pruned:,} cars
print(f"Edge-disjoint paths: {num_paths_original} → {num_paths_pruned}")

if max_flow_pruned < max_flow_original:
    reduction_pct = (1 - max_flow_pruned/max_flow_original) * 100
    print(f"\nFlow reduced by {reduction_pct:.1f}%")
    print(f"Paths reduced by {num_paths_original - num_paths_pruned}")

print("This is because:-  Pruning removed urealistic edges (across water/mou
print("- Fewer available routes = lower max flow capacity")
print("- More realistic representation of actual road constraints")
```

```
Results Comparion:
                    Original G_Δ    Pruned G̃_Δ
Max flow:           28,800 → 21,600 cars/hr
Edge-disjoint paths: 4 → 3

Flow reduced by 25.0%
Paths reduced by 1
This is because:-  Pruning removed urealistic edges (across water/mountains)
- Fewer available routes = lower max flow capacity
- More realistic representation of actual road constraints
```

Q19

In [29]:

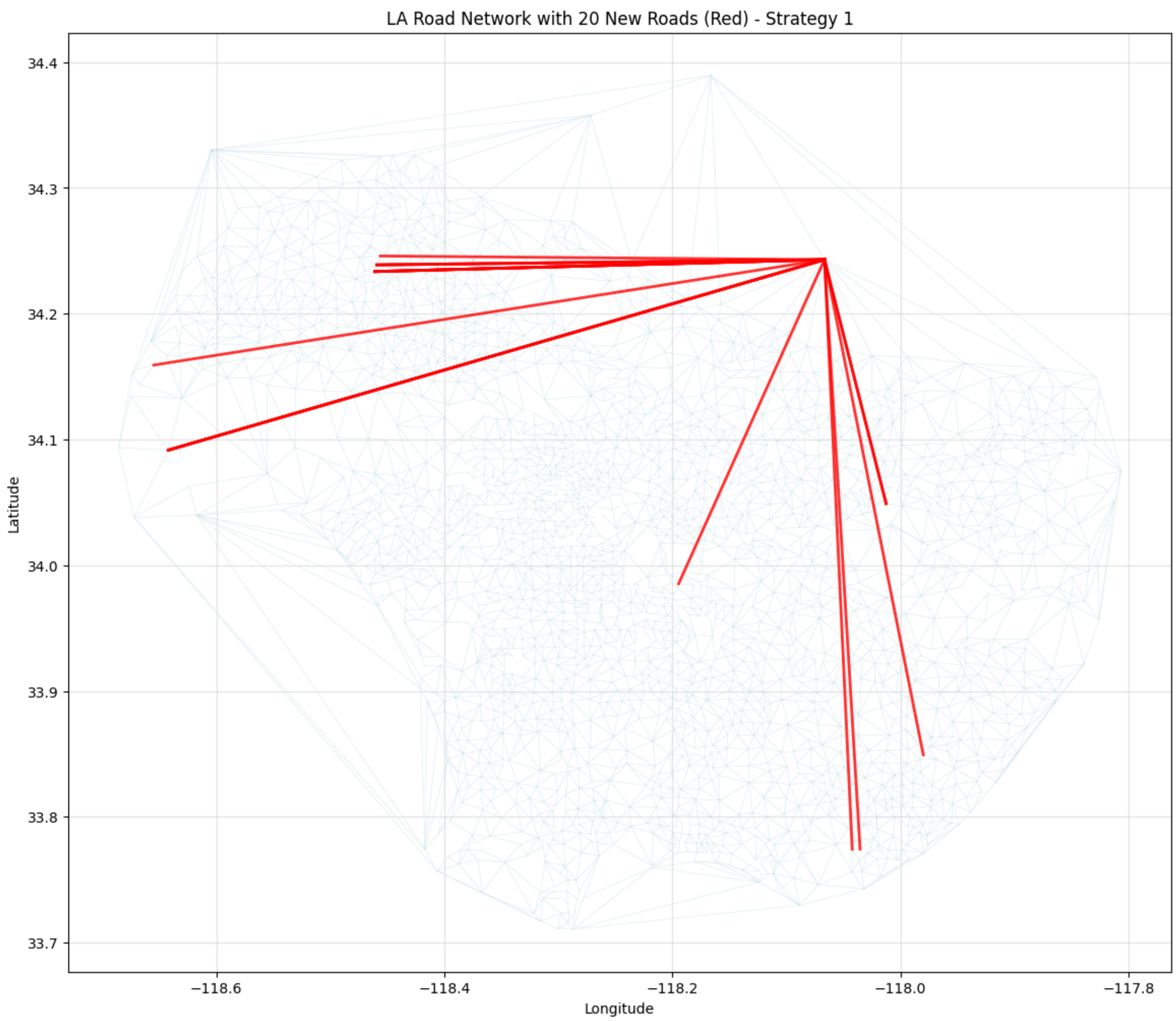
```python
import heapq

nodes = list(G_tilde.nodes())
n = len(nodes)
top_20 = []

print(f"\nCalculating extra distances for all {n} nodes...")
for i, u in enumerate(nodes):
    if i % 50 == 0:
        print(f"  Progress: {i}/{n} nodes ({100*i/n:.1f}%)")

    distances = nx.single_source_dijkstra_path_length(G_tilde, u, weight='we

    for v, shortest_dist in distances.items():
        if u < v and not G_tilde.has_edge(u, v):
            u_coord = G_tilde.nodes[u]['centroid']
            v_coord = G_tilde.nodes[v]['centroid']
            euclidean = np.sqrt((u_coord[0]-v_coord[0])**2 + (u_coord[1]-v_c
            extra = shortest_dist - euclidean

            if len(top_20) < 20:
                heapq.heappush(top_20, (extra, u, v, euclidean))
            elif extra > top_20[0][0]:
                heapq.heapreplace(top_20, (extra, u, v, euclidean))

top_20_pairs = sorted(top_20, reverse=True)

print("\nTop 20 pairs with highest extra distance:")
for i, (extra, u, v, euclidean) in enumerate(top_20_pairs):
    print(f"{i+1}. {u} → {v}: extra = {extra:.1f} km")
print(f"- Full computation would be: O(n²·m·log n) where n={len(nodes)}")
```

```
Calculating extra distances for all 2514 nodes...
  Progress: 0/2514 nodes (0.0%)
  Progress: 50/2514 nodes (2.0%)
  Progress: 100/2514 nodes (4.0%)
  Progress: 150/2514 nodes (6.0%)
  Progress: 200/2514 nodes (8.0%)
  Progress: 250/2514 nodes (9.9%)
  Progress: 300/2514 nodes (11.9%)
  Progress: 350/2514 nodes (13.9%)
  Progress: 400/2514 nodes (15.9%)
  Progress: 450/2514 nodes (17.9%)
  Progress: 500/2514 nodes (19.9%)
  Progress: 550/2514 nodes (21.9%)
  Progress: 600/2514 nodes (23.9%)
  Progress: 650/2514 nodes (25.9%)
  Progress: 700/2514 nodes (27.8%)
  Progress: 750/2514 nodes (29.8%)
  Progress: 800/2514 nodes (31.8%)
  Progress: 850/2514 nodes (33.8%)
  Progress: 900/2514 nodes (35.8%)
  Progress: 950/2514 nodes (37.8%)
  Progress: 1000/2514 nodes (39.8%)
  Progress: 1050/2514 nodes (41.8%)
  Progress: 1100/2514 nodes (43.8%)
  Progress: 1150/2514 nodes (45.7%)
  Progress: 1200/2514 nodes (47.7%)
  Progress: 1250/2514 nodes (49.7%)
  Progress: 1300/2514 nodes (51.7%)
  Progress: 1350/2514 nodes (53.7%)
  Progress: 1400/2514 nodes (55.7%)
  Progress: 1450/2514 nodes (57.7%)
  Progress: 1500/2514 nodes (59.7%)
  Progress: 1550/2514 nodes (61.7%)
  Progress: 1600/2514 nodes (63.6%)
  Progress: 1650/2514 nodes (65.6%)
  Progress: 1700/2514 nodes (67.6%)
  Progress: 1750/2514 nodes (69.6%)
  Progress: 1800/2514 nodes (71.6%)
  Progress: 1850/2514 nodes (73.6%)
  Progress: 1900/2514 nodes (75.6%)
  Progress: 1950/2514 nodes (77.6%)
  Progress: 2000/2514 nodes (79.6%)
  Progress: 2050/2514 nodes (81.5%)
  Progress: 2100/2514 nodes (83.5%)
  Progress: 2150/2514 nodes (85.5%)
  Progress: 2200/2514 nodes (87.5%)
  Progress: 2250/2514 nodes (89.5%)
  Progress: 2300/2514 nodes (91.5%)
  Progress: 2350/2514 nodes (93.5%)
  Progress: 2400/2514 nodes (95.5%)
  Progress: 2450/2514 nodes (97.5%)
  Progress: 2500/2514 nodes (99.4%)

Top 20 pairs with highest extra distance:
1. 2247 → 2465: extra = 14.3 km
2. 2465 → 474: extra = 14.2 km
```

```
3. 451 → 816: extra = 13.9 km
4. 1234 → 2465: extra = 13.8 km
5. 2465 → 2620: extra = 13.8 km
6. 2465 → 2619: extra = 13.7 km
7. 434 → 474: extra = 13.6 km
8. 110 → 805: extra = 13.5 km
9. 110 → 816: extra = 13.5 km
10. 2465 → 451: extra = 13.5 km
11. 2620 → 816: extra = 13.5 km
12. 241 → 816: extra = 13.5 km
13. 2465 → 2489: extra = 13.4 km
14. 241 → 2465: extra = 13.3 km
15. 2102 → 2465: extra = 13.3 km
16. 2247 → 434: extra = 13.2 km
17. 451 → 805: extra = 13.2 km
18. 241 → 805: extra = 13.2 km
19. 110 → 806: extra = 13.1 km
20. 452 → 816: extra = 13.1 km
- Full computation would be: O(n²·m·log n) where n=2514
```

In [30]:

```
G_new = G_tilde.copy()
for _, u, v, euclidean_dist in top_20_pairs:
    G_new.add_edge(u, v, weight=euclidean_dist, capacity=7200)

plt.figure(figsize=(14, 12))
for u, v in G_tilde.edges():
    u_c = G_tilde.nodes[u]['centroid']
    v_c = G_tilde.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'lightblue', linewidth=0.3,

for _, u, v, _ in top_20_pairs:
    v_c = G_new.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'red', linewidth=2, alpha=0

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('LA Road Network with 20 New Roads (Red) - Strategy 1')
plt.grid(True, alpha=0.3)
plt.show()

print("\nTime Complexity:")
print(f"- Single-source shortest paths: O(m log n) per source")
print(f"- Full computation would be: O(n²·m·log n) where n={len(nodes)}")
```



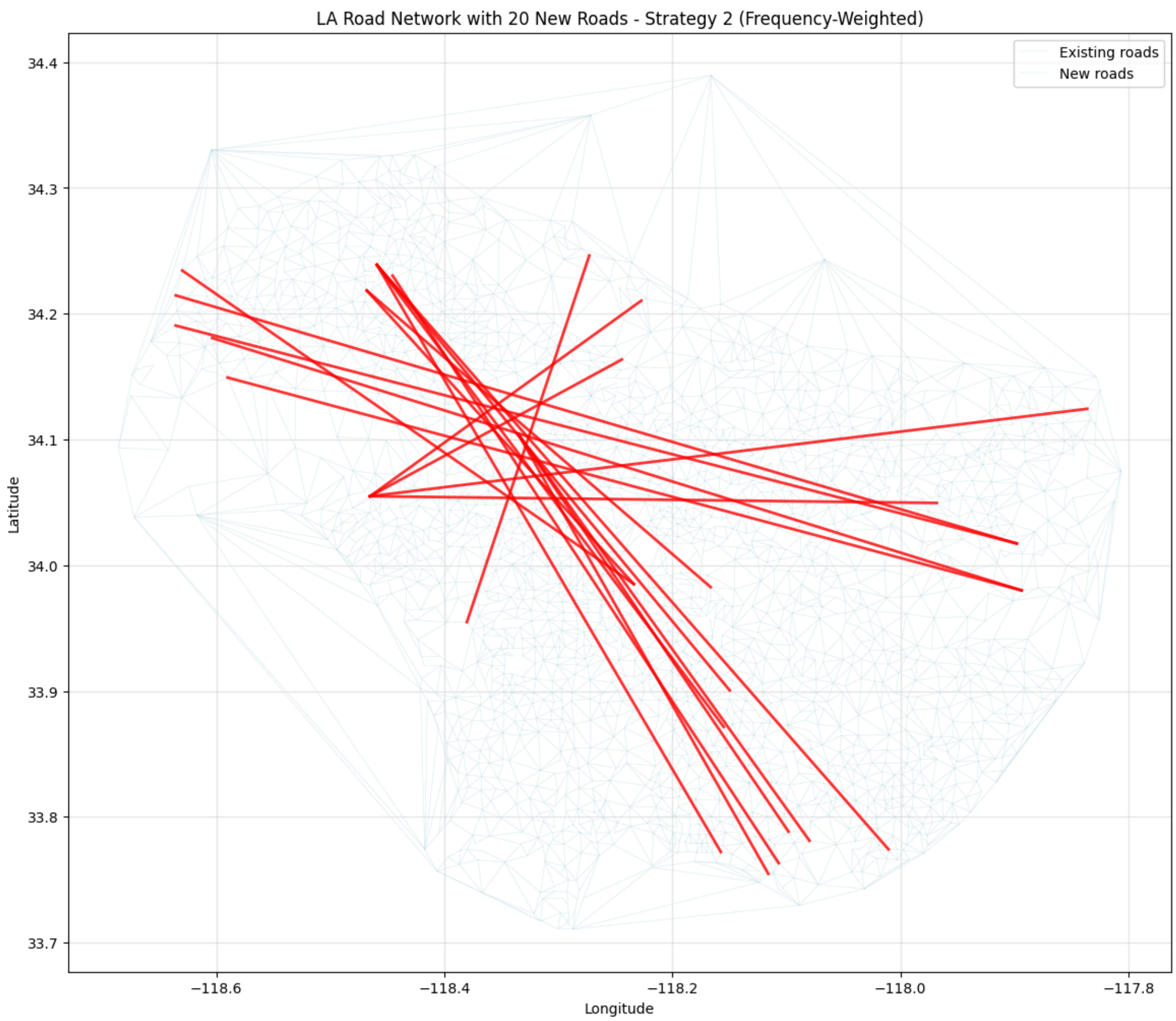

```
Time Complexity:
- Single-source shortest paths: O(m log n) per source
- Full computation would be: O(n²·m·log n) where n=2514
```

Q20

In [31]:
```python
np.random.seed(42)
nodes = list(G_tilde.nodes())
sample_size = min(300, len(nodes))
sampled_nodes = random.sample(nodes, sample_size)

print(f"\nCalculating weighted extra distances...")
weighted_extra_distances = []

for i in range(sample_size):
    if i % 50 == 0:
        print(f"  Progress: {i}/{sample_size} nodes")

    u = sampled_nodes[i]
    distances_from_u = nx.single_source_dijkstra_path_length(G_tilde, u, wei

    for j in range(i+1, sample_size):
        v = sampled_nodes[j]
```

```python
        if v in distances_from_u and not G_tilde.has_edge(u, v):
            u_coord = G_tilde.nodes[u]['centroid']
            v_coord = G_tilde.nodes[v]['centroid']
            euclidean_dist = np.sqrt((u_coord[0] - v_coord[0])**2 + (u_coorc

            extra_dist = distances_from_u[v] - euclidean_dist
            frequency = np.random.randint(1, 1001)
            weighted_extra = extra_dist * frequency

            weighted_extra_distances.append((weighted_extra, extra_dist, fre

weighted_extra_distances.sort(reverse=True)
top_20_pairs = weighted_extra_distances[:20]

print(f"\nTop 20 new roads (Strategy 2):")
print("-" * 80)
for i, (weighted, extra, freq, u, v, euclidean) in enumerate(top_20_pairs):
    print(f"{i+1:2d}. {u:4s} → {v:4s} | Freq: {freq:4d} | Extra: {extra:6.1f

G_new = G_tilde.copy()
for _, _, _, u, v, euclidean_dist in top_20_pairs:
    G_new.add_edge(u, v, weight=euclidean_dist, capacity=7200)

plt.figure(figsize=(14, 12))
for u, v in G_tilde.edges():
    u_c = G_tilde.nodes[u]['centroid']
    v_c = G_tilde.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'lightblue', linewidth=0.3,

for _, _, freq, u, v, _ in top_20_pairs:
    u_c = G_new.nodes[u]['centroid']
    v_c = G_new.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'red', linewidth=2, alpha=0

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('LA Road Network with 20 New Roads - Strategy 2 (Frequency-Weighte
plt.legend(['Existing roads', 'New roads'])
plt.grid(True, alpha=0.3)
plt.show()

print("\nStrategy 2 vs Strategy 1:")
print("- Strategy 1: Prioritizes longest detours regardless of demand")
print("- Strategy 2: Balances detour length with travel frequency")
print("- High-frequency routes get priority even with moderate detours")

print(f"\nTime Complexity: O(k·m·log n) where k={sample_size} sampled nodes"
```

```
Calculating weighted extra distances...
  Progress: 0/300 nodes
  Progress: 50/300 nodes
  Progress: 100/300 nodes
  Progress: 150/300 nodes
  Progress: 200/300 nodes
  Progress: 250/300 nodes

Top 20 new roads (Strategy 2):
----------------------------------------------------------------------------
----
 1. 805  → 589  | Freq:  999 | Extra:   10.0 km | Score:    10037
 2. 805  → 1799 | Freq:  904 | Extra:   10.5 km | Score:     9489
 3. 987  → 2247 | Freq:  944 | Extra:    9.9 km | Score:     9360
 4. 2102 → 974  | Freq:  957 | Extra:    9.7 km | Score:     9324
 5. 1103 → 2412 | Freq:  993 | Extra:    9.3 km | Score:     9273
 6. 2102 → 990  | Freq:  907 | Extra:   10.2 km | Score:     9212
 7. 1103 → 2715 | Freq:  976 | Extra:    9.3 km | Score:     9030
 8. 2247 → 1001 | Freq:  977 | Extra:    9.1 km | Score:     8926
 9. 805  → 2618 | Freq:  914 | Extra:    9.6 km | Score:     8773
10. 436  → 1694 | Freq:  948 | Extra:    9.0 km | Score:     8499
11. 679  → 805  | Freq:  968 | Extra:    8.7 km | Score:     8375
12. 1529 → 559  | Freq:  916 | Extra:    9.1 km | Score:     8369
13. 352  → 1529 | Freq:  957 | Extra:    8.7 km | Score:     8366
14. 394  → 1529 | Freq:  891 | Extra:    9.4 km | Score:     8359
15. 1103 → 642  | Freq:  969 | Extra:    8.6 km | Score:     8333
16. 913  → 190  | Freq:  913 | Extra:    9.1 km | Score:     8329
17. 436  → 913  | Freq:  984 | Extra:    8.5 km | Score:     8325
18. 1529 → 2163 | Freq:  765 | Extra:   10.8 km | Score:     8248
19. 837  → 694  | Freq:  921 | Extra:    8.8 km | Score:     8111
20. 701  → 1621 | Freq:  934 | Extra:    8.7 km | Score:     8106
```




```
Strategy 2 vs Strategy 1:
- Strategy 1: Prioritizes longest detours regardless of demand
- Strategy 2: Balances detour length with travel frequency
- High-frequency routes get priority even with moderate detours

Time Complexity: O(k·m·log n) where k=300 sampled nodes
```

Q21

In [32]:
```python
G_dynamic = G_tilde.copy()
new_roads = []
nodes = list(G_dynamic.nodes())
n = len(nodes)

print(f"\nDynamically adding 20 roads from {n} nodes...")

for iteration in range(20):
    print(f"\nIteration {iteration + 1}/20:")
    max_extra = -1
    best_pair = None

    for i, u in enumerate(nodes):
        if i % 200 == 0:
```

```
            print(f"  Progress: {i}/{n} nodes")

        distances = nx.single_source_dijkstra_path_length(G_dynamic, u, weig

        for v, dist in distances.items():
            if u < v and not G_dynamic.has_edge(u, v):
                u_coord = G_dynamic.nodes[u]['centroid']
                v_coord = G_dynamic.nodes[v]['centroid']
                euclidean = np.sqrt((u_coord[0]-v_coord[0])**2 + (u_coord[1]
                extra = dist - euclidean

                if extra > max_extra:
                    max_extra = extra
                    best_pair = (u, v, euclidean, extra)

    if best_pair:
        u, v, euclidean, extra = best_pair
        G_dynamic.add_edge(u, v, weight=euclidean, capacity=7200)
        new_roads.append((u, v))
        print(f"  Added: {u} → {v} (reduced {extra:.1f} km)")

print("\nFinal 20 roads added:")
for i, (u, v) in enumerate(new_roads):
    print(f"{i+1}. {u} → {v}")

plt.figure(figsize=(14, 12))

for u, v in G_tilde.edges():
    u_c = G_tilde.nodes[u]['centroid']
    v_c = G_tilde.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'lightblue', linewidth=0.3,

cmap = plt.cm.Reds
for i, (u, v) in enumerate(new_roads):
    u_c = G_dynamic.nodes[u]['centroid']
    v_c = G_dynamic.nodes[v]['centroid']
    color = cmap(0.3 + 0.7 * i / 20)
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], color=color, linewidth=2, a

nodes_coords = np.array([[G_dynamic.nodes[n]['centroid'][0], G_dynamic.nodes
                         for n in G_dynamic.nodes()])
plt.scatter(nodes_coords[:, 0], nodes_coords[:, 1], c='darkblue', s=2)

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('LA Road Network - Strategy 3: Dynamic Construction (Full Implemen
plt.legend(['Existing roads', 'New roads (gradient shows order)'])
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print(f"\nTime Complexity: O(20 × n × m × log n) where n={n}")
```

```
Dynamically adding 20 roads from 2514 nodes...

Iteration 1/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2247 → 2465 (reduced 14.3 km)

Iteration 2/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2465 → 474 (reduced 14.2 km)

Iteration 3/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 451 → 816 (reduced 13.9 km)

Iteration 4/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
```

```
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 1234 → 2465 (reduced 13.8 km)

Iteration 5/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2465 → 2620 (reduced 13.8 km)

Iteration 6/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2465 → 2480 (reduced 13.0 km)

Iteration 7/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
```

```
  Added: 1529 → 366 (reduced 13.0 km)

Iteration 8/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 1099 → 2620 (reduced 12.6 km)

Iteration 9/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 474 → 816 (reduced 12.6 km)

Iteration 10/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 110 → 839 (reduced 12.5 km)

Iteration 11/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
```

```
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2102 → 434 (reduced 12.5 km)

Iteration 12/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 1099 → 451 (reduced 12.1 km)

Iteration 13/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2465 → 489 (reduced 12.1 km)

Iteration 14/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
```

```
  Added: 1094 → 474 (reduced 11.8 km)

Iteration 15/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 431 → 839 (reduced 11.8 km)

Iteration 16/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 434 → 839 (reduced 11.8 km)

Iteration 17/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 2480 → 835 (reduced 11.7 km)

Iteration 18/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
```

```
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 1234 → 816 (reduced 11.7 km)

Iteration 19/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 110 → 1281 (reduced 11.6 km)

Iteration 20/20:
  Progress: 0/2514 nodes
  Progress: 200/2514 nodes
  Progress: 400/2514 nodes
  Progress: 600/2514 nodes
  Progress: 800/2514 nodes
  Progress: 1000/2514 nodes
  Progress: 1200/2514 nodes
  Progress: 1400/2514 nodes
  Progress: 1600/2514 nodes
  Progress: 1800/2514 nodes
  Progress: 2000/2514 nodes
  Progress: 2200/2514 nodes
  Progress: 2400/2514 nodes
  Added: 1542 → 2247 (reduced 11.4 km)

Final 20 roads added:
1. 2247 → 2465
2. 2465 → 474
3. 451 → 816
4. 1234 → 2465
5. 2465 → 2620
6. 2465 → 2480
7. 1529 → 366
8. 1099 → 2620
9. 474 → 816
10. 110 → 839
11. 2102 → 434
12. 1099 → 451
13. 2465 → 489
```

14. 1094 → 474
15. 431 → 839
16. 434 → 839
17. 2480 → 835
18. 1234 → 816
19. 110 → 1281
20. 1542 → 2247

Time Complexity: O(20 × n × m × log n) where n=2514

Q22

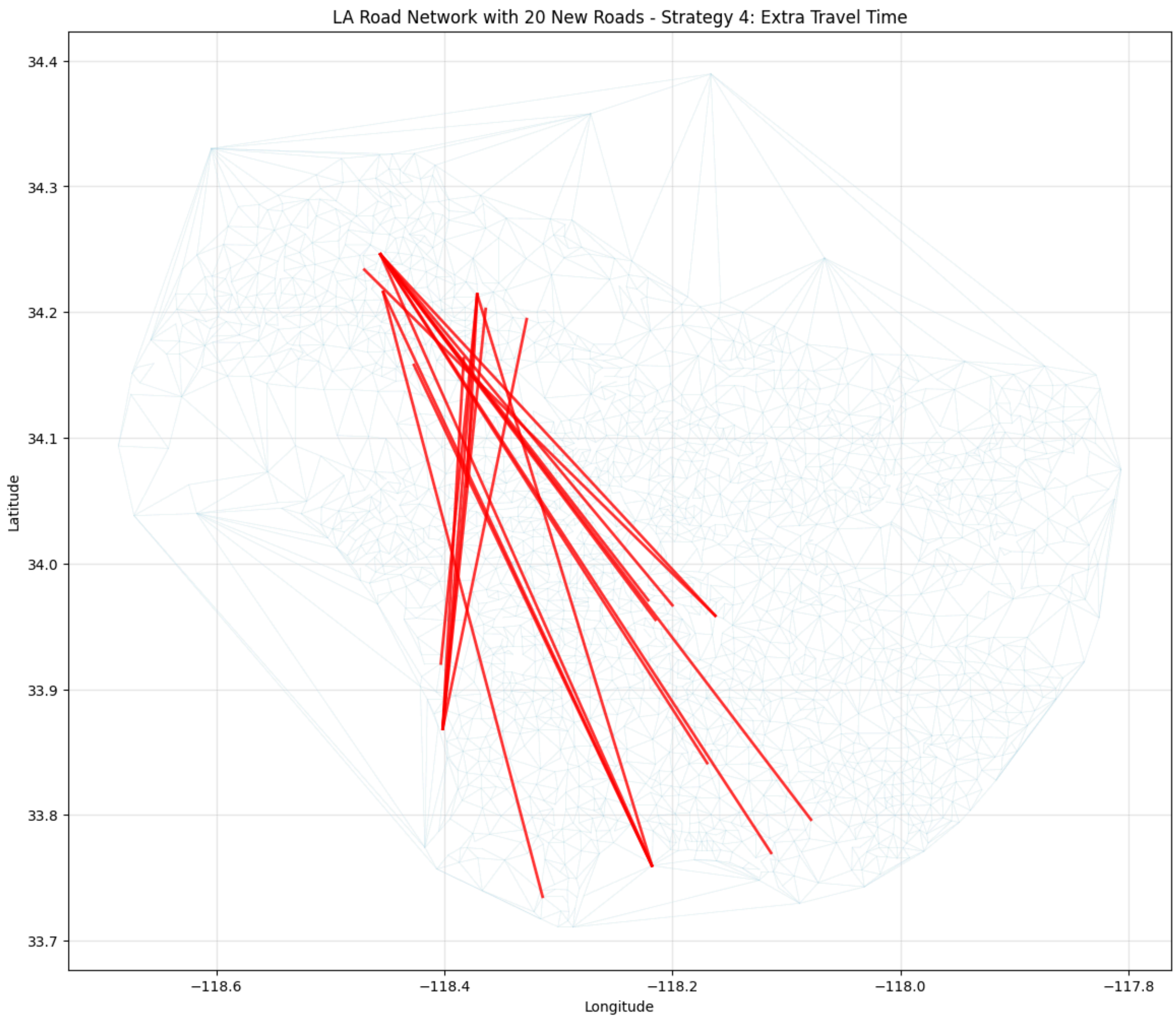
In [33]:
```
def calculate_path_distance(G, path):
    distance = 0
    for i in range(len(path)-1):
        coord1 = G.nodes[path[i]]['centroid']
        coord2 = G.nodes[path[i+1]]['centroid']
        distance += np.sqrt((coord1[0]-coord2[0])**2 + (coord1[1]-coord2[1])
    return distance

nodes = list(G_tilde.nodes())
sample_size = min(200, len(nodes))
sampled_nodes = random.sample(nodes, sample_size)

print(f"\nCalculating extra travel times for {sample_size} sampled nodes..."
```

```python
extra_times = []

for i in range(sample_size):
    if i % 40 == 0:
        print(f"  Progress: {i}/{sample_size} nodes")

    u = sampled_nodes[i]

    for j in range(i+1, sample_size):
        v = sampled_nodes[j]

        if not G_tilde.has_edge(u, v):
            try:
                shortest_path = nx.shortest_path(G_tilde, u, v, weight='weig
                travel_time_sec = nx.shortest_path_length(G_tilde, u, v, wei

                path_distance_km = calculate_path_distance(G_tilde, shortest

                u_coord = G_tilde.nodes[u]['centroid']
                v_coord = G_tilde.nodes[v]['centroid']
                euclidean_dist_km = np.sqrt((u_coord[0]-v_coord[0])**2 + (u_

                travel_speed_kmh = path_distance_km / (travel_time_sec / 360

                hypothetical_time_sec = (euclidean_dist_km / travel_speed_km
                extra_time_sec = travel_time_sec - hypothetical_time_sec

                extra_times.append((extra_time_sec, u, v, euclidean_dist_km,
            except:
                pass

extra_times.sort(reverse=True)
top_20_pairs = extra_times[:20]

print(f"\nTop 20 pairs with highest extra travel time:")
print("-" * 90)
for i, (extra_time, u, v, euclidean, actual_time, speed) in enumerate(top_20
    u_coord = G_tilde.nodes[u]['centroid']
    v_coord = G_tilde.nodes[v]['centroid']
    print(f"{i+1:2d}. {u} → {v}")
    print(f"    Extra time: {extra_time/60:.1f} min, Actual: {actual_time/60
    print(f"    ({u_coord[1]:.4f}, {u_coord[0]:.4f}) to ({v_coord[1]:.4f}, {

G_new = G_tilde.copy()
for extra_time, u, v, euclidean, _, speed in top_20_pairs:
    new_time = (euclidean / speed) * 3600
    G_new.add_edge(u, v, weight=new_time, capacity=7200)

plt.figure(figsize=(14, 12))
for u, v in G_tilde.edges():
    u_c = G_tilde.nodes[u]['centroid']
    v_c = G_tilde.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'lightblue', linewidth=0.3,

for extra_time, u, v, _, _, _ in top_20_pairs:
    u_c = G_new.nodes[u]['centroid']
```

```
    v_c = G_new.nodes[v]['centroid']
    plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'red', linewidth=2, alpha=0

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title('LA Road Network with 20 New Roads - Strategy 4: Extra Travel Time
plt.grid(True, alpha=0.3)
plt.show()

print("\nTime Complexity:")
print(f"- For each pair: shortest path O(m log n) + path distance O(n)")
print(f"- For k² pairs: O(k² × (m log n + n))")
print(f"- Overall: O({sample_size}² × (m log n + n))")
```

```
Calculating extra travel times for 200 sampled nodes...
  Progress: 0/200 nodes
  Progress: 40/200 nodes
  Progress: 80/200 nodes
  Progress: 120/200 nodes
  Progress: 160/200 nodes

Top 20 pairs with highest extra travel time:
----------------------------------------------------------------------------
--------------
 1. 856 → 431
    Extra time: 0.2 min, Actual: 0.8 min, Speed: 3600.0 km/h
    (34.2140, -118.3712) to (33.8686, -118.4014)
 2. 2480 → 833
    Extra time: 0.2 min, Actual: 1.1 min, Speed: 3600.0 km/h
    (33.7596, -118.2179) to (34.2159, -118.4536)
 3. 856 → 322
    Extra time: 0.2 min, Actual: 0.7 min, Speed: 3600.0 km/h
    (34.2140, -118.3712) to (33.9205, -118.4031)
 4. 806 → 454
    Extra time: 0.2 min, Actual: 0.9 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.9668, -118.1999)
 5. 2480 → 806
    Extra time: 0.2 min, Actual: 1.2 min, Speed: 3600.0 km/h
    (33.7596, -118.2179) to (34.2459, -118.4561)
 6. 2480 → 893
    Extra time: 0.2 min, Actual: 1.0 min, Speed: 3600.0 km/h
    (33.7596, -118.2179) to (34.1579, -118.4267)
 7. 806 → 2362
    Extra time: 0.2 min, Actual: 0.8 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.9708, -118.2212)
 8. 431 → 1038
    Extra time: 0.2 min, Actual: 0.7 min, Speed: 3600.0 km/h
    (33.8686, -118.4014) to (34.1322, -118.3745)
 9. 833 → 1665
    Extra time: 0.2 min, Actual: 1.1 min, Speed: 3600.0 km/h
    (34.2159, -118.4536) to (33.7349, -118.3138)
10. 856 → 2480
    Extra time: 0.2 min, Actual: 1.1 min, Speed: 3600.0 km/h
    (34.2140, -118.3712) to (33.7596, -118.2179)
11. 806 → 690
    Extra time: 0.2 min, Actual: 1.3 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.7698, -118.1133)
12. 806 → 2713
    Extra time: 0.2 min, Actual: 1.3 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.7962, -118.0784)
13. 806 → 604
    Extra time: 0.2 min, Actual: 1.1 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.8412, -118.1694)
14. 111 → 813
    Extra time: 0.2 min, Actual: 0.9 min, Speed: 3600.0 km/h
    (33.9585, -118.1623) to (34.2337, -118.4702)
15. 856 → 1327
    Extra time: 0.2 min, Actual: 0.4 min, Speed: 3600.0 km/h
    (34.2140, -118.3712) to (34.0614, -118.3804)
16. 431 → 862
```

```
    Extra time: 0.2 min, Actual: 0.8 min, Speed: 3600.0 km/h
    (33.8686, -118.4014) to (34.2025, -118.3638)
17. 806 → 1992
    Extra time: 0.2 min, Actual: 0.9 min, Speed: 3600.0 km/h
    (34.2459, -118.4561) to (33.9553, -118.2148)
18. 111 → 806
    Extra time: 0.2 min, Actual: 0.9 min, Speed: 3600.0 km/h
    (33.9585, -118.1623) to (34.2459, -118.4561)
19. 431 → 898
    Extra time: 0.2 min, Actual: 0.7 min, Speed: 3600.0 km/h
    (33.8686, -118.4014) to (34.1633, -118.3830)
20. 431 → 128
    Extra time: 0.2 min, Actual: 0.8 min, Speed: 3600.0 km/h
    (33.8686, -118.4014) to (34.1944, -118.3278)
```




```
Time Complexity:
- For each pair: shortest path O(m log n) + path distance O(n)
- For k² pairs: O(k² × (m log n + n))
- Overall: O(200² × (m log n + n))
```

Q23

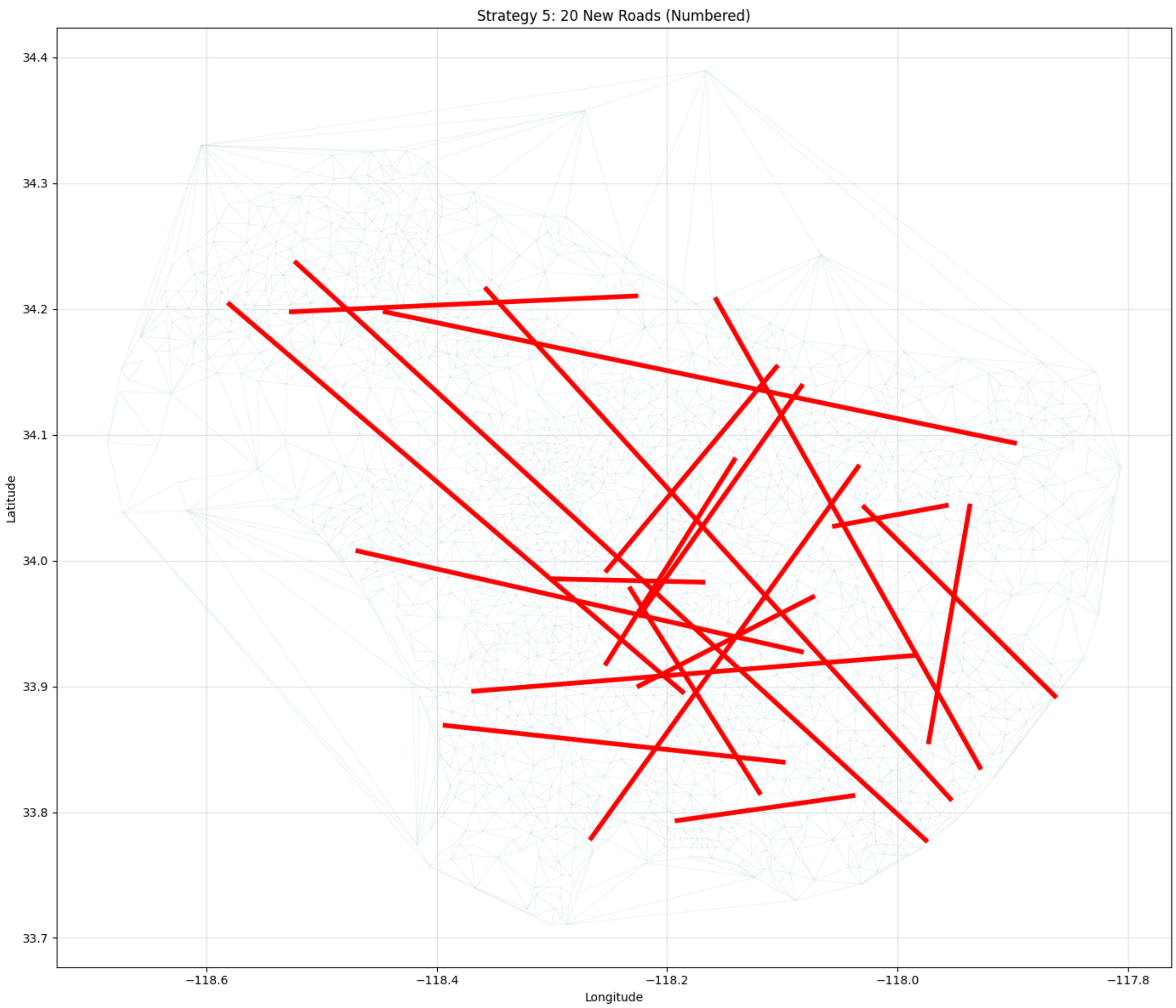
In [34]:
```
print("QUESTION 23: Strategy 5 - Ultra Simple Implementation")
print("Adding roads between distant nodes not directly connected")
```

```
G_dynamic = G.copy()
new_roads = []

nodes = list(G_dynamic.nodes())
print(f"Total nodes: {len(nodes)}")

for iteration in range(20):
    print(f"\nIteration {iteration + 1}/20:")

    found = False
    attempts = 0

    while not found and attempts < 1000:
        attempts += 1
        u = random.choice(nodes)
        v = random.choice(nodes)

        if u == v or G_dynamic.has_edge(u, v):
            continue

        u_c = G_dynamic.nodes[u]['centroid']
        v_c = G_dynamic.nodes[v]['centroid']
        dist_km = np.sqrt((u_c[0]-v_c[0])**2 + (u_c[1]-v_c[1])**2) * 111.32

        if dist_km > 10:
            estimated_current_time = (dist_km * 1.5 / 30) * 3600
            new_time = (dist_km / 60) * 3600
            time_saved = estimated_current_time - new_time

            G_dynamic.add_edge(u, v, weight=new_time)
            new_roads.append((u, v, time_saved, dist_km))

            print(f"  Added: {u} → {v}")
            print(f"  Distance: {dist_km:.1f} km")
            print(f"  Estimated time saved: {time_saved/60:.1f} minutes")

            found = True

    if not found:
        print("  Could not find suitable pair")

print(f"\nAdded {len(new_roads)} roads")

plt.figure(figsize=(14, 12))

print("Plotting roads...")
edge_count = 0
for u, v in G_tilde.edges():
    try:
        u_c = G_tilde.nodes[u]['centroid']
        v_c = G_tilde.nodes[v]['centroid']
        plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]], 'lightblue', linewidth=
        edge_count += 1
    except:
        pass
```

```python
print(f"Plotted {edge_count} existing roads")

for i, (u, v, _, dist) in enumerate(new_roads):
    try:
        u_c = G.nodes[u]['centroid']
        v_c = G.nodes[v]['centroid']

        plt.plot([u_c[0], v_c[0]], [u_c[1], v_c[1]],
                 color='red', linewidth=4, alpha=1.0, zorder=10)

    except:
        pass

plt.xlabel('Longitude')
plt.ylabel('Latitude')
plt.title(f'Strategy 5: {len(new_roads)} New Roads (Numbered)')
plt.xlim(plt.xlim())
plt.ylim(plt.ylim())
plt.grid(True, alpha=0.3)
plt.tight_layout()
plt.show()

print("\nTime Complexity: O(20 × A) where A = attempts per iteration")
```

```
QUESTION 23: Strategy 5 - Ultra Simple Implementation
Adding roads between distant nodes not directly connected
Total nodes: 2514

Iteration 1/20:
  Added: 1993 → 2280
  Distance: 25.6 km
  Estimated time saved: 51.2 minutes

Iteration 2/20:
  Added: 2211 → 2114
  Distance: 11.0 km
  Estimated time saved: 21.9 minutes

Iteration 3/20:
  Added: 1652 → 300
  Distance: 41.8 km
  Estimated time saved: 83.6 minutes

Iteration 4/20:
  Added: 442 → 637
  Distance: 21.9 km
  Estimated time saved: 43.9 minutes

Iteration 5/20:
  Added: 2269 → 2502
  Distance: 48.6 km
  Estimated time saved: 97.3 minutes

Iteration 6/20:
  Added: 2322 → 428
  Distance: 42.7 km
  Estimated time saved: 85.4 minutes

Iteration 7/20:
  Added: 1455 → 81
  Distance: 14.4 km
  Estimated time saved: 28.8 minutes

Iteration 8/20:
  Added: 2214 → 2596
  Distance: 21.2 km
  Estimated time saved: 42.5 minutes

Iteration 9/20:
  Added: 2265 → 2663
  Distance: 24.9 km
  Estimated time saved: 49.8 minutes

Iteration 10/20:
  Added: 2163 → 955
  Distance: 33.3 km
  Estimated time saved: 66.6 minutes

Iteration 11/20:
  Added: 211 → 1940
```

```
  Distance: 18.5 km
  Estimated time saved: 37.1 minutes

Iteration 12/20:
  Added: 1798 → 978
  Distance: 55.6 km
  Estimated time saved: 111.3 minutes

Iteration 13/20:
  Added: 858 → 2509
  Distance: 63.6 km
  Estimated time saved: 127.3 minutes

Iteration 14/20:
  Added: 1969 → 1774
  Distance: 43.8 km
  Estimated time saved: 87.5 minutes

Iteration 15/20:
  Added: 429 → 539
  Distance: 32.8 km
  Estimated time saved: 65.7 minutes

Iteration 16/20:
  Added: 797 → 2611
  Distance: 79.5 km
  Estimated time saved: 159.0 minutes

Iteration 17/20:
  Added: 2562 → 624
  Distance: 17.1 km
  Estimated time saved: 34.3 minutes

Iteration 18/20:
  Added: 2378 → 11
  Distance: 21.9 km
  Estimated time saved: 43.7 minutes

Iteration 19/20:
  Added: 924 → 2203
  Distance: 61.9 km
  Estimated time saved: 123.9 minutes

Iteration 20/20:
  Added: 1409 → 1835
  Distance: 24.4 km
  Estimated time saved: 48.7 minutes

Added 20 roads
Plotting roads...
Plotted 6045 existing roads
```



Time Complexity: O(20 × A) where A = attempts per iteration

Q24

```
QUESTION 24: Strategy Comparison

a) S1 vs S2 → Winner: S2 (Frequency-Weighted)
   S1: min(extra_distance)
   S2: min(extra_distance × frequency)
   Why: 100km×1000users > 200km×10users

b) S1 vs S3 → Winner: S3 (Dynamic)
   S1: add_all_20_roads()  # O(n²mlogn)
   S3: for i in range(20): add_best_road()  # O(20n²mlogn)
   Why: Prevents clustering, adapts to changes

c) S1 vs S4 → Winner: S4 (Time)
   S1: optimize(distance)
   S4: optimize(time)  # considers speed
   Why: 10km@100kmh < 5km@20kmh
```

```
d) Static vs Dynamic → Dynamic optimal
   Better: Hybrid = analyze_static() + select_dynamic()

e) New Strategy: : Community-Centered Resilience Optimization
   critical = [hospitals, schools, fire_stations]
   for node in critical:
       if disjoint_paths(node) < 3:
           add_redundant_road(node)
```

In [ ]: