# Network and Graph Visualization

# - Principles of Network Visualization

# - Tools and Techniques for Graph Visualization

## key principles of effective network visualization

https://rhumbl.com/blog/top-5-design-principles-for-network-visualization

Effective network visualization follows principles such as clarity, node and edge distinction, minimizing overlap, emphasizing important nodes (like hubs), and maintaining an intuitive layout to reflect relationships accurately.

## Force-directed layouts work in graph visualization.

Force-directed layouts use physics-based algorithms where nodes repel each other and edges act like springs. This distributes nodes naturally in space, reducing clutter and highlighting structural patterns in the network.

A force-directed layout is a type of network layout algorithm used for visualizing networks or graphs. It uses a physics-based simulation to arrange the nodes and edges of the network in a way that minimizes the energy or "force" between them. The purpose of using a force-directed layout for network visualization is to create an aesthetically pleasing and informative representation of the network.

The main benefit of a force-directed layout is that it helps to reveal the underlying structure and relationships within the network. Nodes that are connected to each other are drawn closer together, while nodes that are not connected are pushed farther apart. This creates clusters or communities within the network that can be easily identified and analyzed. Force-directed layouts can also reveal important nodes that act as "hubs" or connectors between different parts of the network.

In addition to providing insights into the structure of the network, force-directed layouts are also visually appealing and easy to understand. They allow the viewer to quickly see which nodes are most central or important, and which nodes are more peripheral. This can be useful for identifying

patterns or anomalies in the data and for communicating the results of network analysis to a broader audience.

Overall, the purpose of using a force-directed layout for network visualization is to create a clear and intuitive representation of the network that highlights its key features and properties. By using this layout algorithm, it is possible to create a visualization that is both informative and visually engaging.

### graph visualization can help in solving a real-world problem.

In social network analysis, graph visualization helps identify influential users, community structures, and information flow. For instance, it can be used to track the spread of information or misinformation on social media.

## Tools used for graph visualization.

Gephi – A desktop tool for interactive exploration of large networks with built-in layouts and analytics.

Cytoscape – Originally for biological networks, it now supports any graph data with plugins for enhanced functionality.

**Gephi**:

It is an open-source, interactive graph and network visualization software designed for exploring and analyzing complex networks. Often referred to as the "Photoshop of graph visualization," it is highly popular among data scientists, journalists, and social scientists.

🛠️ Key Features

Graph layout algorithms: Includes several algorithms like ForceAtlas, Fruchterman-Reingold, and Yifan Hu for arranging graph nodes in an intuitive layout.

Real-time visualization: Allows dynamic filtering, interaction, and manipulation of graphs in real-time.

Network statistics: Computes metrics such as degree centrality, betweenness centrality, modularity (community detection), etc.

Customization: Offers customizable visual properties like node size, color, and edge thickness based on attribute values.

Import/Export: Supports CSV, GEXF, GraphML, and other formats; exports visuals as PNG, SVG, or PDF.

🎁 Use Cases

Social network analysis (e.g., Twitter user interactions)

Citation and co-authorship networks

Biological networks (e.g., protein-protein interaction)

Web graph and hyperlink analysis

✅ Strengths

Intuitive GUI for non-programmers

Powerful for large networks (up to ~100,000 nodes)

Good for exploratory and presentation-ready visuals

**2. Cytoscape**

🔍 Overview

Cytoscape is another open-source platform, mainly used for visualizing molecular interaction networks and biological pathways, but it's flexible enough to be applied to any kind of graph data.

🛠️ Key Features

Specialized in biological data: Originally designed for genomics and systems biology, supports integration with databases like UniProt, STRING, and KEGG.

Data integration: Allows mapping of additional data (e.g., gene expression levels, annotations) onto networks.

Extensible via apps: Supports over 300 plugins (called "apps") for advanced functions such as clustering, enrichment analysis, and 3D visualization.

Automation support: Can be scripted using Cytoscape Automation with Python or R through RESTful APIs.

🎁 Use Cases

Gene regulatory and metabolic pathway visualization

Disease-gene networks

Drug-target interaction mapping

Any complex data where nodes and edges represent entities and relationships

✅ Strengths

Rich plugin ecosystem for biology-specific needs

Supports large-scale network visualization and manipulation

Strong integration with other bioinformatics tools and databases

# Application of graph visualization

Example :

detect fraud in financial transactions

**1. Data Modeling: Building the Graph**

Transform financial transaction data into a graph structure.

| Data Element | Graph Component |
|---|---|
| Bank account, person | Node |
| Transaction | Edge |
| Transaction amount | Edge weight/attribute |
| Account type, status | Node attribute |

Example:

If Person A sends money to Person B, you draw a directed edge from Node A to Node B with attributes like amount and timestamp.

**2. Identify Fraud Patterns with Graph Structures**

Graph structures can reveal known fraudulent behaviors, such as:

🔁 Circular Transactions

Money moves in a loop across multiple accounts to obscure origin (e.g., for money laundering).

🕸️ Many-to-One or One-to-Many Relationships

Many accounts funneling money to a single account → Possible scam or money mule.

One account sending money to many unknown accounts → May indicate fraudulent disbursement.

👥 Account Clustering

Accounts that share personal details (e.g., same phone number or IP) may be part of a fraud ring.

📊 Unusual Transaction Behavior

Spikes in transaction frequency or size can be seen easily in visual networks.

3. Apply Visual and Analytical Techniques

Use a graph visualization tool like Gephi, Neo4j Bloom, or Cytoscape to:

🧭 Layout Algorithms

Apply force-directed or hierarchical layouts to highlight clusters and connections.

🎨 Visual Cues

Color nodes by risk score, transaction volume, or role.

Use edge thickness to represent transaction amounts.

Use node size to represent frequency of transactions.

📊 Compute Graph Metrics

Centrality (degree, betweenness, closeness): Fraudsters often act as intermediaries or hubs.

Community detection: Reveals tight groups potentially involved in fraud rings.

Anomaly detection: Outliers in the network (e.g., isolated accounts suddenly transacting) can be suspicious.

**4. Real-Time Monitoring**

Integrate visualization tools with live transaction systems:

Stream data into graph databases like Neo4j.

Trigger alerts when suspicious graph patterns emerge (e.g., circular transactions or high centrality nodes).

**5. Investigation and Audit**

Investigators can:

Visually explore paths between entities.

Trace money flow through different accounts.

Zoom into suspicious subnetworks for deeper analysis.

5

# Time Series Data Visualization
# - Techniques for Visualizing Time Series Data
# - Tools and Libraries (e.g., matplotlib, Plotly

## Important to visualize time series data

Definition

Time series data is a sequence of data points collected or recorded at specific, equally spaced intervals over time. Each data point is typically represented as a pair:

(Time, Value)

The key aspect is that time is the independent variable, and values change depending on it.

📃 Examples

Finance: Stock prices recorded every second or minute.

Weather: Temperature measured every hour or day.

Health: Heart rate data captured every few seconds via wearable devices.

Sales: Monthly revenue or product sales figures.

IoT/Smart Devices: Sensor readings every few milliseconds.

✅ 1. Reveals Trends

Visualization helps identify long-term upward or downward trends that aren't obvious in raw data.

E.g., A line chart of stock prices over a year reveals growth or decline.

✅ 2. Highlights Seasonality and Cycles

Many datasets have regular repeating patterns (daily, weekly, yearly).

E.g., Energy consumption peaks during summer afternoons due to AC usage.

✅ 3. Detects Anomalies and Outliers

Sudden spikes, dips, or flat lines are easier to detect visually.

E.g., A spike in website traffic may indicate a viral event or a bot attack.

✅ 4. Helps Compare Multiple Series

Plotting multiple time series (e.g., different products or locations) helps compare their behavior over time.

✅ 5. Aids Forecasting

Visualization helps in evaluating the performance of predictive models, especially when overlaid on historical data.

✅ 6. Enhances Communication

A well-designed time series chart helps stakeholders (e.g., managers, clients) understand patterns quickly without needing to interpret raw numbers.

## Importance of Time series data:

Time series data is vital in many fields because it captures trends, patterns, and anomalies that evolve over time. It supports:

Forecasting future values (e.g., sales, demand, prices)

Monitoring system behavior (e.g., server performance, patient vitals)

Detecting anomalies (e.g., fraud, faults, spikes)

Understanding seasonality (e.g., increased sales during holidays)

## techniques for visualizing time series data.

Line Plot – Connects data points over time, useful for showing trends.

Area Chart – Like a line plot but filled below the line, useful for comparing magnitudes over time.

## matplotlib (time series data in Python)

1. What You Need

To plot time series data with Matplotlib, you typically need:

A list or array of datetime objects for the x-axis (time)

A corresponding list or array of numeric values for the y-axis (observations)

We often use pandas alongside Matplotlib, as it simplifies handling dates and data.

🛠️ 2. Step-by-Step Guide

✅ Step 1: Install Required Libraries

If not already install

pip install matplotlib pandas

✅ Step 2: Import Libraries

```
import pandas as pd
import matplotlib.pyplot as plt
```

✅ Step 3: Prepare Time Series Data

Example 1: Using Pandas DataFrame with DateTime Index

```
# Sample data
data = {
    'Date': pd.date_range(start='2023-01-01', periods=10, freq='D'),
    'Temperature': [30, 32, 31, 29, 28, 30, 33, 34, 32, 31]
}

# Convert to DataFrame
df = pd.DataFrame(data)
df.set_index('Date', inplace=True)
```

✅ Step 4: Plot with Matplotlib

```
# Create the plot
plt.figure(figsize=(10, 5))  # Optional: set figure size
plt.plot(df.index, df['Temperature'], marker='o', linestyle='-', color='teal')

# Add labels and title
plt.title('Daily Temperature Over Time')
plt.xlabel('Date')
plt.ylabel('Temperature (°C)')
plt.grid(True)
plt.xticks(rotation=45)  # Rotate x-axis dates for readability

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

🎨 3. Customization Tips

| Feature | Code Example | Purpose |
|---|---|---|
| Line style | linestyle='--' | Dotted, dashed, or solid lines |
| Marker | marker='o' | Show data points |

Color color='red' Set line color

Date formattingplt.gca().xaxis.set_major_formatter(...) Customize date appearance

Add annotations plt.annotate() Highlight key points

🗓️ 4. Handling Date Formats

Sometimes, you need to parse date strings:

df['Date'] = pd.to_datetime(df['Date']) # Converts 'YYYY-MM-DD' to datetime

Or set a column as index:

df.set_index('Date', inplace=True)

🔄 5. Plot Multiple Time Series

# Multiple columns: Temperature and Humidity

df['Humidity'] = [70, 68, 65, 72, 75, 73, 69, 66, 64, 67]

plt.plot(df.index, df['Temperature'], label='Temperature', color='orange')

plt.plot(df.index, df['Humidity'], label='Humidity', color='blue')

plt.title('Weather Over Time')

plt.legend()

plt.grid(True)

plt.show()

## Advantages does Plotly offer over matplotlib for time series visualization

| Feature | Plotly | Matplotlib |
|---|---|---|
| Interactivity | ✅ Fully interactive out-of-the-box | ❌ Static (unless combined with widgets) |
| Zoom & Pan | ✅ Built-in zoom, pan, hover, drag | ❌ Manual setup or requires extra tools |
| Hover Tooltips | ✅ Hover to see exact values & time | ❌ Not supported without extra code |
| Responsive Layouts | ✅ Auto-resizes in web environments | ❌ Fixed size unless manually adjusted |
| Web Integration | ✅ Designed for web dashboards (Dash, HTML) | ❌ Not web-native |
| Ease of Animation | ✅ Built-in animation support | ❌ Complex to animate manually |
| Customization | ✅ High, via JSON-like dicts | ✅ Very high with code control |

🔍 Plotly:

Automatically includes hover tooltips showing time and value.

Users can zoom in, pan, and reset view without writing any extra code.

Hovering over a line shows precise timestamps and values.

import plotly.express as px

fig = px.line(df, x='Date', y='Temperature', title='Temperature Over Time')

fig.show()

🎯 Use case: Great for exploratory data analysis and dashboards.

〰️ Matplotlib:

Plots are static by default.

Interactivity is possible, but it requires Jupyter widgets, mpld3, or third-party tools.

🌐 2. Web and Dashboard Integration

Plotly:

Built for web: renders in HTML, integrates with Dash, and works in Jupyter notebooks seamlessly.

Easy to export plots to interactive HTML files.

Matplotlib:

Primarily designed for static output (PDFs, PNGs).

Limited web dashboard support.

🎨 3. Visual Appeal

Plotly provides:

Sleek, modern visuals with minimal effort.

Automatic styling: grid lines, tooltips, font scaling, etc.

Matplotlib:

Requires more customization for polished visuals.

Default style is more "scientific" and less polished out-of-the-box.

📽️ 4. Animations and Time Series Playback

Plotly makes it easy to animate time series data:

px.line(df, x='Date', y='Temperature', animation_frame='Date')

You can create sliders, play/pause buttons, or animated transitions with minimal code.

Matplotlib:

Requires use of FuncAnimation and more boilerplate code.

🎹 5. Multiple Series & Legends

Plotly:

Click-to-hide or isolate a line in a multi-series plot.

Interactive legends improve readability and exploration.

Matplotlib:

Legends are static, and exploring individual series visually is harder.

💻 6. Exporting & Sharing

| Task | Plotly | Matplotlib |
| --- | --- | --- |
| Shareable interactive chart | ✅ Export as standalone HTML | ❌ Static only |
| Save image | ✅ PNG, SVG, JPEG, HTML, PDF | ✅ PNG, SVG, PDF |
| Dashboards | ✅ Native with Dash | ⚠️ Requires external tools (e.g., Bokeh) |

## Suppose you are analyzing website traffic data. How would you visualize it to detect weekly trends.

🧰 Step-by-Step Strategy

✅ 1. Collect and Prepare the Data

Assume your dataset looks like this:

| Date | Visits |
| --- | --- |
| 2024-01-01 | 1234 |
| 2024-01-02 | 1456 |
| ... | ... |

Load and parse your data in Python:

```
import pandas as pd

# Load your traffic data
df = pd.read_csv('website_traffic.csv')

# Ensure 'Date' column is in datetime format
df['Date'] = pd.to_datetime(df['Date'])


# Optional: sort by date
df = df.sort_values('Date')
```

✅ 2. Extract Day of the Week

You need to group data by day names (e.g., Monday, Tuesday…)

```
df['DayOfWeek'] = df['Date'].dt.day_name()
```

✅ 3. Aggregate by Day of the Week

Now group by the weekday to find average visits per day:

```
weekly_trend = df.groupby('DayOfWeek')['Visits'].mean().reindex(
    ['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday', 'Saturday', 'Sunday']
)
```

✅ 4. Visualize Weekly Trends

Option A: Using Matplotlib

```
import matplotlib.pyplot as plt

plt.figure(figsize=(10, 5))
weekly_trend.plot(kind='bar', color='skyblue')

plt.title('Average Website Visits by Day of the Week')
plt.xlabel('Day of the Week')
plt.ylabel('Average Visits')
plt.grid(axis='y')
plt.tight_layout()
plt.show()
```

Option B: Using Plotly (for interactivity)

```
import plotly.express as px

fig = px.bar(
    x=weekly_trend.index,
    y=weekly_trend.values,
    labels={'x': 'Day of Week', 'y': 'Average Visits'},
    title='Average Website Visits by Day of the Week'
)
fig.show()
```

✅ 5. Optional: Heatmap to Show Trends Over Weeks

You can also visualize trends using a calendar heatmap or pivot table.

```
df['Week'] = df['Date'].dt.isocalendar().week
```

```
pivot = df.pivot_table(index='Week', columns='DayOfWeek', values='Visits')


import seaborn as sns

plt.figure(figsize=(12, 6))

sns.heatmap(pivot.reindex(columns=['Monday', 'Tuesday', 'Wednesday', 'Thursday', 'Friday',
'Saturday', 'Sunday']),

        cmap='Blues', linewidths=.5, annot=True, fmt=".0f")

plt.title('Website Traffic Heatmap by Weekday')

plt.xlabel('Day of the Week')

plt.ylabel('Week Number')

plt.tight_layout()

plt.show()
```

🔍 Interpretation of Weekly Trends

By analyzing the visual output:

- ▲ See which days get more traffic (e.g., Mondays and Thursdays).

- ▼ Spot lower traffic on weekends (Saturdays and Sundays).

- 🌀 Identify anomalies (e.g., spikes from marketing campaigns or drops due to downtime).