How graphs model internet routes

Graphs are data structures made up of nodes and edges. Nodes represent entities such as people while edges represent the relationship between these entities. The design and analysis of internet and computer networks relies on graphs. In these networks , devices like routers and switches re nodes and the connections between them are vertices. Graphs help in understanding the flow of data, ensuring efficient and reliable communication. Network administrators use graphs to detect vulnerabilities, optimize data flow, and plan network extensions.

Graphs model internet routing systems by representing routers as nodes and their connections as edges, allowing algorithms to find optimal paths for data transmission. These edges may have weights associated to them that represent the bandwidth, distance or the latency that aid routing algorithms in making decisions. Dijkstra’s algorithm can be used for this since it is efficient for finding the shortest path through a graph. This would determine the best path for data packets to travel from one node to another.

Discoveries: The weights are not on the nodes but the edges.

Graph structure in internet routing:

Nodes: routers, ISP, end devices such as your computer.

Edges: Physical links like fibre optic cables, satellite connections and wireless links, Logical links like virtual connections between large networks. The edges can be directed or undirected.

Internet routing is a graph traversal problem of finding the best path from a source node for the data packets:

* INTRA-DOMAIN ROUTING(THIS HAPPENS WITHIN A NETWORK);

Protocols: Open Shortest Path First(OSPF),Routing Information Protocol(RIP).

Routers within a single network form a graph edges have weights based on link costs. For example bandwidth.

OSP uses Dijkstra’s algorithm while RIP uses Bellman-Ford Algorithm but is less efficient.

**b) Inter-Domain Routing (Between Networks)**

* **Protocol**: BGP (Border Gateway Protocol).
* **Graph Model**: Nodes are autonomous systems, and edges represent peering or transit relationships. Weights might reflect AS path length, policy preferences, or business agreements rather than just physical metrics.
* **Algorithm**: BGP uses a **path-vector approach**, akin to a modified DFS. Each AS advertises paths to destinations, and routers choose the “best” path based on attributes like:
  + Shortest AS path (fewer hops between ASes).
  + Policy rules (e.g., prefer a cheaper provider).
* **Example**: To reach a Google server (AS15169) from your ISP (AS7018), BGP might select a path through AS3356 (Level 3) if it’s the shortest AS path or cheapest transit.

**c) Traversal Behavior**

* **DFS-Like**: BGP’s path exploration resembles DFS as it follows advertised paths deeply through ASes, but it stops at the best path rather than exhausting all possibilities.
* **BFS-Like**: OSPF’s shortest-path computation spreads out level by level from a source, finding optimal routes to all destinations.

**3. Graph Representation in Practice**

The internet’s scale (millions of routers, thousands of ASes) influences how graphs are represented:

* **Adjacency List**: Preferred because the internet graph is **sparse**—each router or AS connects to only a few neighbors. Space complexity is O(V + E), where V is nodes and E is edges.
  + Example: A router in Chicago might list neighbors in New York, Denver, and Toronto.
* **Adjacency Matrix**: Rarely used due to O(V²) space, impractical for sparse, large graphs like the internet.
* **Dynamic Updates**: The graph changes constantly (links fail, new connections form), so routing tables (derived from the graph) are updated via protocol messages.

**4. Key Features of Internet Graphs**

The internet graph has unique properties:

* **Scale-Free Nature**: Follows a power-law distribution—few nodes (e.g., Tier 1 ISPs like Level 3) have many connections, while most (e.g., small ISPs) have few. This affects routing efficiency and resilience.
* **Hierarchical Structure**:
  + Tier 1 ISPs (e.g., AT&T) form the backbone, peering with each other.
  + Tier 2 ISPs connect to Tier 1 and smaller networks.
  + Stub ASes (e.g., a university) connect to one or two providers.
* **Redundancy**: Multiple paths exist between nodes, modeled as multiple edges or parallel paths, ensuring fault tolerance.
* **Weighted Edges**: Beyond physical distance, weights include latency, bandwidth, jitter, or monetary cost, influencing path selection.

**4. Key Features of Internet Graphs**

The internet graph has unique properties:

* **Scale-Free Nature**: Follows a power-law distribution—few nodes (e.g., Tier 1 ISPs like Level 3) have many connections, while most (e.g., small ISPs) have few. This affects routing efficiency and resilience.
* **Hierarchical Structure**:
  + Tier 1 ISPs (e.g., AT&T) form the backbone, peering with each other.
  + Tier 2 ISPs connect to Tier 1 and smaller networks.
  + Stub ASes (e.g., a university) connect to one or two providers.
* **Redundancy**: Multiple paths exist between nodes, modeled as multiple edges or parallel paths, ensuring fault tolerance.
* **Weighted Edges**: Beyond physical distance, weights include latency, bandwidth, jitter, or monetary cost, influencing path selection.

**5. Real-World Example**

Imagine sending a packet from your home in New York to a server in Los Angeles:

1. **Local Network**: Your router (node) sends the packet to your ISP’s gateway (edge: Wi-Fi link).
2. **Intra-Domain**: The ISP’s routers use OSPF to find a path to its border router, e.g., New York → Chicago → Denver (edges weighted by latency).
3. **Inter-Domain**: BGP selects a path across ASes:
   * AS7018 (AT&T) → AS3356 (Level 3) → AS16591 (Google).
   * The graph here is AS-level, with edges based on peering agreements.
4. **Destination**: Google’s internal routers (another graph) deliver the packet to the server in Los Angeles.

The full path might be a subgraph: New York (home) → Chicago (ISP hub) → Denver (border router) → Los Angeles (Google data center).

**6. Challenges in Modeling Internet Routing with Graphs**

* **Scale**: The internet has ~1 billion routers and ~70,000 ASes (as of 2025 estimates), making full graph computation impractical. Protocols use partial views.
* **Dynamics**: Links fail (e.g., cable cuts) or change (e.g., new peering), requiring real-time updates to the graph.
* **Policy**: BGP paths aren’t always shortest; business decisions (e.g., avoiding a rival AS) override graph optimality.
* **Loops**: Misconfigurations can create routing loops, which graph algorithms like BGP’s AS path check prevent.
* **Security**: Attacks (e.g., BGP hijacking) manipulate the graph by advertising false paths.

**7. Comparison to Flight Network**

Like the flight network (New York → Chicago → Denver), internet routing uses graphs, but:

* **Flights**: Fixed routes, fewer nodes (~thousands of airports), simpler weights (distance/time).
* **Internet**: Dynamic routes, massive scale (~billions of nodes), complex weights (latency, cost, policy).

Both use BFS-like (shortest path) or DFS-like (path exploration) traversals, but internet routing adds layers of abstraction (AS vs. router level) and policy-driven decisions.

**Conclusion**

Graphs model internet routing by representing routers or ASes as nodes and connections as edges, with traversal algorithms (Dijkstra’s, path-vector) determining data paths. The sparse, hierarchical, and dynamic nature of the internet favors adjacency lists and distributed protocols like BGP and OSPF. This abstraction ensures scalability, resilience, and flexibility, making the internet a living, breathing graph traversal problem.

Let me know if you’d like a deeper dive into a specific protocol or a coded simulation of internet routing!

**COMPARE GRAPH REPRESEATIONS (ADJACENCY MATRIX VS ADJACENCY LIST):**

Adjacency list is an array of consisting of the address of all the linked lists. The first node of the linked list represents the vertex and the remaining lists connected to this node represent the vertices to which the node is connected. This linked list can also be adjusted to store the weight of the edge.

Adjacency matrix is a 2D array of size V X V where V is the number of vertices in a graph. A slot adj[i][j]=1 indicates that there is an edge from vertex I to vertex j. The adjacency matrix for undirected graph is always symmetric.

**Comparison:**

Storage space: adjacency matrix requires space in worst case of O(|V^2|) while overall space complexity is O(|V|+|E|) for adjacency list. Since for every vertex we store its neighbors, in the worst case, if a graph is connected O(V) is required for a vertex and O(E) is equired for storing neighbors corresponding to every vertex.

Adding a vertex: for an adjacency matrix, we need to copy the whole matrix to adda a vertex and therefore the complexity is O(|V|^2) while for adjacency list the insertion of a vertex can be done directly in O(1) time since there are 2 pointers, the first pointing to the front node and the other pointing to the rear node.

Adding an edge: For an adjacency matrix and an adjacency list itrequires O(1) time

Removing a vertex: in adjacency matrix, we need to copy the whole matrix so complexity is O(|V|^2) since the storage is decreased from (|V|+1)^2 to |V|^2 while in adjacency list to remove a vertex, it has to first be searched for which requires O(|V|) time in the worst case followed by traversing the edges which requires O(|E|) time in worst case.So its total time complexity is O(|V|+|E|).

Removing an edge: in adjacency matrix it requires O(1) time while in an adjacency list it requires O(E) time since we traverse through the edges and in worst case all the edges.

Querying: in adjacency matrix, to find an existing edge would require O(1) time since the content of the matrix needs to be checked while in an adjacency list, time complexity is O(|V|) since to check for an edge we need to check for vertices adjacent to a given vertex. So we must search the list of neighbors for the source vertex.

Advantages of adjacency matrix: quick edge lookup O(1), good for dense graphs: graphs with many edges.

Disadvantages of adjacency matrix: it takes u space f O(V^2) even with sparse graphs, adding vertices requires resixing the matrix.

TIME COMPLEXITY: O(V+E)

Advantages of adjacency list: It is space efficient O(V+E) for sparse graphs, it is easy to add vertices and edges.

Disadvantages of adjacency list: Its edge look up is O(V) which is slower than for an adjacency matrix, it is less efficient for dense graphs

TIME CMPLEXITY: O(V^2)

RECURSION

 **Without Memoization**: A naive recursive Fibonacci (fib(n) = fib(n-1) + fib(n-2)) recomputes the same values repeatedly, leading to exponential calls.

 **With Memoization**: We use a dictionary (memo) to store previously computed Fibonacci numbers. Before recursing, we check if n is in memo. If so, we return the cached result; if not, we compute and store it.

**Time Complexity Analysis**

* **Naive Recursion**:
  + Each call to fib(n) splits into two recursive calls: fib(n-1) and fib(n-2).
  + This forms a binary tree with height ~n, and the number of nodes is approximately 2^n.
  + Time complexity: O(2^n) — exponential and impractical for large n.
* **With Memoization**:
  + Each Fibonacci value (0 to n) is computed exactly once and stored in memo.
  + Subsequent calls for the same n are O(1) lookups.
  + There are n + 1 unique subproblems (fib(0) to fib(n)), and each is computed in O(1) time after memoization (excluding recursive overhead).
  + Total time: O(n) — linear, as we fill the memo table with n + 1 entries.
* **Space Complexity**: O(n) for storing the memoization table.

**Comparison**: For n = 40, naive recursion takes ~2^40 ≈ 1 trillion operations, while memoized takes ~40 operations—a massive improvement.

**2. Recursive Subset Sum Solution**

**Implementation**

The subset sum problem asks: given a set of numbers and a target sum, is there a subset that adds up to the target? Here’s a recursive solution:

The **subset sum problem** asks: Given a set of numbers and a target sum, can we find a subset of those numbers that adds up to the target? For example:

* Numbers: [3, 34, 4, 12, 5, 2]
* Target: 9
* Answer: Yes, because the subset [4, 5] sums to 9.

The recursive solution explores all possible subsets by making decisions at each step: include or exclude each number.

**Why It’s Recursive**

* **Divide and Conquer**: At each step, we reduce the problem:
  + Smaller target (if we include a number).
  + Fewer numbers to consider (move to index - 1).
* **Backtracking**: If a path fails (e.g., target < 0 or index < 0), it backtracks to try the other option (exclude).

**3. Real-World Applications of Recursive Algorithms**

Recursive algorithms are widely used in real-world applications due to their elegance in breaking complex problems into smaller, self-similar subproblems. Here’s a detailed exploration:

**a) Computer Graphics and Geometry**

* **Fractals**: Recursive algorithms generate self-similar shapes like the Mandelbrot set or Sierpinski triangle. Each recursive call draws a smaller version of the pattern.
  + **Example**: In game design (e.g., terrain generation in Minecraft), recursive subdivision creates natural-looking landscapes.
* **Ray Tracing**: Recursive ray tracing simulates light reflection and refraction by recursively tracing rays until they hit a light source or max depth.
  + **Use Case**: Pixar’s RenderMan uses recursion for realistic lighting in animated films.

**b) Graph and Tree Algorithms**

* **DFS (Depth-First Search)**: As seen earlier, DFS recursively explores graphs, used in:
  + **Network Routing**: Finding paths in internet routing (e.g., BGP exploration).
  + **Maze Solving**: Games like Pac-Man use recursive DFS to find escape routes.
* **Binary Tree Operations**: Recursive traversal (inorder, preorder, postorder) is common in databases (e.g., B-trees in SQL engines).

**c) Divide-and-Conquer Algorithms**

* **Sorting**:
  + **Merge Sort**: Recursively splits an array, sorts halves, and merges them (O(n log n)).
  + **Quick Sort**: Recursively partitions around a pivot.
  + **Use Case**: Python’s sorted() and NumPy’s sorting leverage these.
* **Fast Fourier Transform (FFT)**: Recursively decomposes signals, used in audio processing (e.g., MP3 compression) and image analysis (JPEG).

**d) Artificial Intelligence and Games**

* **Minimax Algorithm**: Recursively explores game trees (e.g., chess) to find optimal moves.
  + **Use Case**: Deep Blue (chess) and AlphaGo used recursive tree search with pruning.
* **Backtracking**: Recursive subset sum is a backtracking example, used in:
  + **Sudoku Solvers**: Try numbers, recurse, backtrack if invalid.
  + **Scheduling**: Assign tasks recursively (e.g., Google Calendar optimization).

Conclusively;

* **Fibonacci with Memoization**: O(n) time, a vast improvement over O(2^n), showing recursion’s power with optimization.
* **Subset Sum**: O(2^n) time, brute-force but intuitive, with real-world parallels in combinatorial problems.
* **Applications**: Recursion shines in hierarchical, divide-and-conquer, and exploratory tasks across graphics, AI, networking, and more.