Graph

1. Introduction

A graph is made up of *vertices*/nodes and *edges*/lines that connect those vertices.

A graph may be undirected (meaning that there is no distinction between the two vertices associated with each bidirectional edge) or a graph may be directed (meaning that its edges are directed from one vertex to another but not necessarily in the other direction).

A graph may be weighted (by assigning a weight to each edge, which represent numerical values associated with that connection) or a graph may be unweighted (either all edges have unit weight 1 or all edges have the same constant weight).

1-1. Simple Graph

Most graph problems that we discuss in VisuAlgo involves *simple* graphs.

In a simple graph, there is no *(self-)loop* edge (an edge that connects a vertex with itself) and no *multiple*/*parallel* edges (edges between the same pair of vertices). In another word: There can only be up to one edge between a pair of distinct vertices.

The number of edges **E** in a simple graph can only range from 0 to $O(\mathbf{V}^2)$.

Graph algorithms on simple graphs are easier than on non-simple graphs.

1-2. Terminologies, Part 1

An undirected edge **e**: (**u**, **v**) is said to be incident with its two end-point vertices: **u** and **v**. Two vertices are called adjacent (or neighbor) if they are incident with a common edge. For example, edge (0, 2) is incident to vertices 0+2 and vertices 0+2 are adjacent.

Two edges are called adjacent if they are incident with a common vertex. For example, edge (0, 2) and (2, 4) are adjacent.

The degree of a vertex **v** in an undirected graph is the number of edges incident with **v**. A vertex of degree 0 is called an isolated vertex. For example, vertex 0/2/6 has degree 2/3/1, respectively.

A subgraph **G'** of a graph **G** is a (smaller) graph that contains subset of vertices and edges of **G**. For example, a triangle {0, 1, 2} is a subgraph of the currently displayed graph.

1-3. Terminologies, Part 2

A path (of length **n**) in an (undirected) graph **G** is a sequence of vertices $\{v_0, v_1, ..., v_{n-1}, v_n\}$ such that there is an edge between v_i and v_{i+1} $\forall i \in [0..n-1]$ along the path.

If there is no repeated vertex along the path, we call such path as a simple path.

For example, {0, 1, 2, 4, 5} is one simple path in the currently displayed graph.

1-4. Terminologies, Part 3

An undirected graph **G** is called connected if there is a path between every pair of distinct vertices of **G**. For example, the currently displayed graph is not a connected graph.

An undirected graph **C** is called a connected component of the undirected graph **G** if 1). **C** is a subgraph of **G**; 2). **C** is connected; 3). no connected subgraph of **G** has **C** as a subgraph and contains vertices or edges that are not in **C** (i.e. **C** is the maximal subgraph that satisfies the other two criteria).

For example, the currently displayed graph have {0, 1, 2, 3, 4} and {5, 6} as its two connected components.

1-5. Terminologies, Part 4

In a directed graph, some of the terminologies mentioned earlier have small adjustments.

If we have a directed edge **e**: (**u** → **v**), we say that **v** is adjacent to **u** but not necessarily in the other direction. For example, 1 is adjacent to 0 but 0 is not adjacent to 1 in the currently displayed directed graph.

In a directed graph, we have to further differentiate the degree of a vertex **v** into in-degree and out-degree. The in-degree/out-degree is the number of edges coming-into/going-out-from **v**, respectively. For example, vertex 1 has in-degree/out-degree of 2/1, respectively.

In a directed graph, we define the concept of *Strongly* Connected Component (SCC). In the currently displayed directed graph, we have {0}, {1, 2, 3}, and {4, 5, 6, 7} as its three SCCs.

1-6. Terminologies, Part 5

A cycle is a path that starts and ends with the same vertex.

An acyclic graph is a graph that contains no cycle.

In an undirected graph, each of its undirected edge causes a *trivial* cycle although we usually will not classify it as a cycle.

A directed graph that is also acyclic has a special name: Directed Acyclic Graph (DAG), as shown above.

There are interesting algorithms that we can perform on acyclic graphs that will be explored in this visualization page and in other graph visualization pages in VisuAlgo.

1-7. Special Graphs

A graph with specific properties involving its vertices and/or edges structure can be called with its specific name, like Tree (like the one currently shown), Complete Graph, Bipartite Graph, Directed Acyclic Graph (DAG), and also the less frequently used: Planar Graph, Line Graph, Star Graph, Wheel Graph, etc.

In this visualization, we will highlight the first four special graphs later.

2. Graph is Pervasive

Graph appears very often in various form in real life. The most important part in solving graph problem is thus the **graph modeling** part, i.e., reducing the problem in hand into graph terminologies: vertices, edges, weights, etc.

2-1. Examples - Easier to See, 1

Social Network: Vertices can represent people, Edges represent connection between people (usually undirected and unweighted).

For example, see the undirected graph that is currently shown. This graph shows 7 vertices (people) and 8 edges (connection/relationship) between them. Perhaps we can ask questions like these:

1. Who is/are the friend(s) of people no 0?
2. Who has the most friend(s)?
3. Is there any isolated people (those with no friend)?
4. Is there a common friend between two strangers: People no 3 and people no 5?
5. Etc...

2-2. Examples - Easier to See, 2

Transportation Network: Vertices can represent stations, edges represent connection between stations (usually weighted).

For example, see the directed weighted graph that is currently shown. This graph shows 5 vertices (stations/places) and 6 edges (connections/roads between stations, with positive weight travelling times as indicated). Suppose that we are driving a car. We can perhaps ask what is the path to take to go from station 0 to station 4 so that we reach station 4 using the least amount of time?

Discussion: Think of a few other real life scenarios which can be modeled as a graph.

2-3. Examples - Harder to See

[This is a hidden slide]

3. Modes

To toggle between the graph drawing modes, select the respective header. We have:

1. U/U = Undirected/Unweighted,
2. U/W = Undirected/Weighted,
3. D/U = Directed/Unweighted, and
4. D/W = Directed/Weighted.

We restrict the type of graphs that you can draw according to the selected mode.

4. Visualization

You can click any one of the example graphs and visualize the graph above.

You can also draw a graph directly in the visualization area:

1. Click on empty space to add vertex,

2. Click a vertex, hold, drag the drawn edge to another vertex, and drop it there to add an edge (PS: This action is not available for mobile users; you need a mouse),
3. Select a vertex/edge and press 'Delete' key to delete that vertex/edge,
4. Select an edge and press 'Enter' to change the weight of that edge [0..99],
5. Press and hold 'Ctrl', then you can click and drag a vertex around.

4-1. Visualization Constraints

We limit the graphs discussed in VisuAlgo to be **simple** graphs. Refer to its discussion in this slide.

We also limit the number of vertices that you can draw on screen to be up to 10 vertices, ranging from vertex 0 to vertex 9. This, together with the simple graph constraint above, limit the number of undirected/directed edges to be 45/90, respectively.

5. Example Graphs

All example graphs can be found here. We provide seven example graphs per category (U/U, U/W, D/U, D/W).

Note that after loading one of these example graphs, you can further modify the currently displayed graph to suit your needs.

6. Special Graphs

Tree, Complete, Bipartite, Directed Acyclic Graph (DAG) are properties of special graphs. As you modify the graph in the visualization/drawing area above, these properties are checked and updated instantly.

There are other less frequently used special graphs: Planar Graph, Line Graph, Star Graph, Wheel Graph, etc, but they are not currently auto-detected in this visualization when you draw them.

6-1. Special Graph - Tree, Part 1

Tree is a connected graph with **V** vertices and **E = V-1** edges, acyclic, and has **one unique path** between any pair of vertices. Usually a Tree is defined on undirected graph.

An undirected Tree (see above) actually contains trivial cycles (caused by its bidirectional edges) but it does not contain non-trivial cycle. A directed Tree is clearly acyclic.

As a Tree only have **V-1** edges, it is usually considered a **sparse** graph.

We currently show our **U/U: Tree example**. You can go to 'Exploration Mode' and draw your own trees.

6-2. Special Graph - Tree, Part 2

Not all Trees have the same graph drawing layout of having a special root vertex at the top and leaf vertices (vertices with degree 1) at the bottom. The (star) graph shown above is also a Tree as it satisfies the properties of a Tree.

Tree with one of its vertex designated as root vertex is called a rooted Tree.

We can always transform any Tree into a rooted Tree by designating a specific vertex (usually vertex 0) as the root, and run a DFS or BFS algorithm from the root.

6-3. Special Graph - Tree, Part 3

In a rooted tree, we have the concept of hierarchies (parent, children, ancestors, descendants), subtrees, levels, and height. We will illustrate these concepts via examples as their meanings are as with real-life counterparts:

1. The parent of 0/1/7/9/4 are none/0/1/8/3, respectively,
2. The children of 0/1/7 are {1,8}/{2,3,6,7}/none, respectively,
3. The ancestors of 4/8 are {3,1,0}/{0}, respectively,
4. The descendants of 1/8 are {2,3,4,5,6,7}/{9}, respectively,
5. The subtree rooted at 1 includes 1, its descendants, and all associated edges,
6. Level 0/1/2/3 members are {0}/{1,8}/{2,3,6,7,9}/{4,5}, respectively,
7. The height of this rooted tree is its maximum level = 3.

6-4. Special Graph - Tree, Part 4

For rooted tree, we can also define additional properties:

A binary tree is a rooted tree in which a vertex has at most two children that are aptly named: left and right child. We frequently see this form during the discussion of Binary Search Tree and Binary Heap.

A full binary tree is a binary tree in which each non-leaf (also called the internal) vertex has exactly two children. The binary tree shown above fulfils this criteria.

A complete binary tree is a binary tree in which every level is completely filled, except possibly the last level may be filled as far left as possible. We frequently see this form especially during discussion of Binary Heap.

6-5. Special Graph - Complete

Complete graph is a graph with **V** vertices and **E = V*(V-1)/2** edges (or **E** = O(**V**^{2})), i.e., there is an edge between any pair of vertices. We denote a Complete graph with **V** vertices as $\mathbf{K_V}$.

Complete graph is the **most dense** simple graph.

We currently show our **U/W: $\mathbf{K_5}$** example. You can go to 'Exploration Mode' and draw your own complete graphs (a bit tedious for larger **V** though).

6-6. Special Graph - Bipartite

Bipartite graph is an undirected graph with **V** vertices that can be partitioned into two disjoint set of vertices of size **m** and **n** where **V = m+n**. There is no edge between members of the same set. Bipartite graph is also free from odd-length cycle.

We currently show our **U/U: Bipartite example**. You can go to 'Exploration Mode' and draw your own bipartite graphs.

A Bipartite Graph can also be complete, i.e., all **m** vertices from one disjoint set are connected to all **n** vertices from the other disjoint set. When **m = n = V/2**, such Complete Bipartite Graphs also have **E** = O(**V**^{2}).

6-7. Special Graph - DAG

Directed Acyclic Graph (DAG) is a directed graph that has no cycle, which is very relevant for Dynamic Programming (DP) techniques.

Each DAG has at least one Topological Sort/Order which can be found with a simple tweak to DFS/BFS Graph Traversal algorithm. DAG will be revisited again in DP technique for SSSP on DAG.

We currently show our **D/W: Four 0→4 Paths** example. You can go to 'Exploration Mode' and draw your own DAGs.

7. Three Graph Data Structures

There are many ways to store graph information into a graph data structure. In this visualization, we show three graph data structures: Adjacency Matrix, Adjacency List, and Edge List — each with its own strengths and weaknesses.

7-1. Adjacency Matrix (AM)

Adjacency Matrix (AM) is a square matrix where the entry AM[i][j] shows the edge's weight from vertex i to vertex j. For unweighted graphs, we can set a unit weight = 1 for all edge weights. An 'x' means that that vertex does not exist (deleted).

We simply use a C++/Python/Java native 2D array/list of size **VxV** to implement this data structure.

7-2. AM - Continued

Space Complexity Analysis: An AM unfortunately requires a big space complexity of $O(\mathbf{V}^2)$, even when the graph is actually sparse (not many edges).

Discussion: Knowing the large space complexity of AM, when is it beneficial to use it? Or is AM always an inferior graph data structure and should not be used at all times?

7-3. The Answer

[This is a hidden slide]

7-4. Adjacency List (AL)

Adjacency List (AL) is an array of **V** lists, one for each vertex (usually in increasing vertex number) where for each vertex i, **AL[i]** stores the list of i's neighbors. For weighted graphs, we can store pairs of (neighbor vertex number, weight of this edge) instead.

We use a Vector of Vector pairs (for weighted graphs) to implement this data structure.
In C++: `vector<vector<pair<int, int>>> AL;`
In Python: `AL = [[] for _ in range(N)]`
In Java: `Vector<Vector<IntegerPair>> AL;`

```
// class IntegerPair in Java is like pair<int, int> in C+
+
```

7-5. Class IntegerPair (in Java)

```
class IntegerPair implements Comparable<IntegerPair> {
  Integer _f, _s;
  public IntegerPair(Integer f, Integer s) { _f = f; _s = s; }
  public int compareTo(IntegerPair o) {
    if (!this.first().equals(o.first())) // this.first() != o.first()
      return this.first() - o.first();   // is wrong as we want to
    else                                 // compare their values,
      return this.second() - o.second(); // not their references
  }
  Integer first() { return _f; }
  Integer second() { return _s; }
}
// IntegerTriple is similar to IntegerPair, just that it has 3 fields
```

7-6. Why Vector of Vector Pairs?

We use pairs as we need to store pairs of information for each edge: (neighbor vertex number, edge weight) where weight can be set to 0 or unused for unweighted graph.

We use Vector of Pairs due to Vector's auto-resize feature. If we have **k** neighbors of a vertex, we just add **k** times to an initially empty Vector of Pairs of this vertex (this Vector can be replaced with Linked List).

We use Vector of Vectors of Pairs for Vector's indexing feature, i.e., if we want to enumerate neighbors of vertex **u**, we use AL[u] (C++/Python) or AL.get(u) (Java) to access the correct Vector of Pairs.

7-7. AL - Continued

Space Complexity Analysis: AL has space complexity of O(**V**+**E**), which is much more efficient than AM and usually the default graph DS inside most graph algorithms.

Discussion: AL is the most frequently used graph data structure, but discuss several scenarios when AL is actually **not** the best graph data structure?

7-8. The Answer

[This is a hidden slide]

7-9. Edge List (EL)

Edge List (EL) is a collection of edges with both connecting vertices and their weights. Usually, these edges are sorted by increasing weight, e.g., part of Kruskal's

algorithm for Minimum Spanning Tree (MST) problem. However in this visualization, we sort the edges based on increasing first vertex number and if ties, by increasing second vertex number. Note that Bidirectional edges in undirected/directed graph are listed once/twice, respectively.

We use a Vector of triples to implement this data structure.
In C++: `vector<tuple<int, int, int>> EL;`
In Python: `EL = []`
In Java: `Vector<IntegerTriple> EL;`
`// class IntegerTriple in Java is like tuple<int, int, int> in C++`

7-10. EL - Continued

Space Complexity Analysis: EL has space complexity of O(**E**), which is much more efficient than AM and as efficient as AL.

Discussion: Elaborate the potential usage of EL other than inside Kruskal's algorithm for Minimum Spanning Tree (MST) problem!

7-11. The Answer

[This is a hidden slide]

8. Simple Applications

After storing our graph information into a graph DS, we can answer **a few** simple queries.

1. Counting **V**,
2. Counting **E**,
3. Enumerating neighbors of a vertex **u**,
4. Checking the existence of edge **(u, v)**, etc.

8-1. Counting V

In an AM and AL, **V** is just the number of rows in the data structure that can be obtained in O(**V**) or even in O(**1**) — depending on the actual implementation.

Discussion: How to count **V** if the graph is stored in an EL?

PS: Sometimes this number is stored/maintained in a separate variable so that we do not have to re-compute this every time — especially if the graph never/rarely changes after it is created, hence O(**1**) performance, e.g., we can store that there are 7 vertices (in our AM/AL/EL data structure) for the example graph shown above.

8-2. The Answer

[This is a hidden slide]

8-3. Counting E

In an EL, **E** is just the number of its rows that can be counted in O(**E**). Note that depending on the need, we may store a bidirectional edge just once in the EL but on other case, we store both directed edges inside the EL.

In an AL, **E** can be found by summing the length of all **V** lists and divide the final answer by 2 (for undirected graph). This requires O(**V**+**E**) computation time as each vertex and each edge is only processed once.

Discussion: How to count **E** if the graph is stored in an AM?

PS: Sometimes this number is stored/maintained in a separate variable for efficiency, e.g., we can store that there are 8 undirected edges (in our AM/AL/EL data structure) for the example graph shown above.

8-4. The Answer

[This is a hidden slide]

8-5. Enumerating Neighbors of a Vertex u

In an AM, we need to loop through all columns of AM[u][j] $\forall j \in [0..\mathbf{V}-1]$ and report pair of (j, AM[u][j]) if AM[u][j] is not zero. This is O(**V**) — slow.

In an AL, we just need to scan AL[u]. If there are only **k** neighbors of vertex **u**, then we just need O(**k**) to enumerate them — this is called an **output-sensitive** time complexity and is already the best possible.

We usually list the neighbors in increasing vertex number. For example, neighbors of vertex 1 in the example graph above are {0, 2, 3}, in that increasing vertex number order.

Discussion: How to do this if the graph is stored in an EL?

8-6. The Answer

[This is a hidden slide]

8-7. Checking the Existence of Edge (u, v)

In an AM, we can simply check if AM[u][v] is non zero. This is O(**1**) — the fastest possible.

In an AL, we have to check whether AL[u] contains vertex **v** or not. This is O(**k**) — slower.

For example, edge (2, 4) exists in the example graph above but edge (2, 6) does not exist.

Note that if we have found edge (u, v), we can also access and/or update its weight.

Discussion: How to do this if the graph is stored in an EL?

8-8. The Answer

[This is a hidden slide]

8-9. Discussion

Quiz: **So, what is the best graph data structure?**

- It Depends
- Adjacency Matrix
- Edge List
- Adjacency List

Discussion: Why?

8-10. The Answer

[This is a hidden slide]

9. Extras

You have reached the end of the basic stuffs of this relatively simple Graph Data Structures and we encourage you to explore further in the **Exploration Mode** by drawing **your own** graphs.

However, we still have a few more interesting Graph Data Structures challenges for you that are outlined in this section.

Note that graph data structures are usually just the necessary but not sufficient part to solve the harder graph problems like MST, SSSP, MF, Matching, MVC, ST, or TSP.

9-1. Online Quiz

For a few more interesting questions about this data structure, please practice on Graph Data Structures training module.

9-2. Implementation Examples

Please look at the following C++/Python/Java/OCaml implementations of the three graph data structures mentioned in this e-Lecture: Adjacency Matrix, Adjacency List, and Edge List: graph_ds.cpp | py | java | ml.

9-3. Online Judge Exercises

Try to solve two basic programming problems that somewhat requires the usage of graph data structure without any fancy graph algorithms:

1. UVa 10895 - Matrix Transpose and,
2. Kattis - flyingsafely.

9-4. Discussion

[This is a hidden slide]