EECS3101 notes:: Introduction to graph based algorithms

Jerry Wu

2022-03-16

Graph algorithms

Abstract

A graph, in essence, is a pair of sets. One containing vertices, and another with edges defined the following way::

$$G = (V, E)|V = \{a, b, c, d, \ldots\}, E = \{(v_1, v_2), (v_2, v_3), \ldots\}$$

Types of graphs

We have different kinds of graphs, each with their own applications.

- Undirected graphs:: Each $\epsilon \in E$ is unordered, i.e. $(a, b) = (b, a)$
- Directed graphs:: Each $\epsilon \in E$ is ordered, i.e. $(a, b) \neq (b, a)$
- Weighted:: Each edge will have a value/weight attached to it. This could represent distance between two points on a map.
- Unweighted:: No value/weight attached to edges.
- Simple graphs have exactly one edge joining any two $v \in V$
- Non simple graphs can have self loops and multiple edges joining two vertices.
- Cyclic graphs contain cycles, i.e. following a walk in the graph can end up back at the starting point.
- Acyclic graphs do not contain cycles.
- Connected graphs have a possible path between every pair of vertices in the graph, disconnected graphs have stray vertices.

  "Simple graphs aren't completely useless"-Larry YL Zhang 2023

When we talk about the length of a path in a graph, we refer to the number of edges we need to traverse, disregarding the weights (if there are any).

Data structures for the graph ADT

To represent a graph in code, we need two data structures::

- Adjacency matrix
  A $|V| \times |V|$ matrix A. If $V = \{v_1, v_2, v_3, \ldots, v_n\} \implies$

$$A_{i,j} = \begin{cases} 1 & (v_i, v_j) \in V \\ 0 & else \end{cases}$$

  Size of matrix is $|V|^2$. For an undirected graph, the adjacency matrix is **symmetric** with respect to the diagonal (slide 19). This means that we realistically only need the upper half of the matrix on th diagonal to retain all data we have about the graph.

- Adjacency list
  Each vertex v_i stores a list A_i of v_j that satisfies $(v_i, v_j) \in E$. This is similar to a hash table. The property in essence is saying that each vertex is mapped to a list containing all vertices that it is pointing to with directed edges (slide 22). The space complexity is $|V|+|E|$ in a directed graph, since we need to count all vertices, along with each edge that is pointing to neighbouring vertices. In an undirected graph, we would count each edge twice, so we have that the space complexity is $|V| + 2|E|$ now.

  "Today I had 3 office hours and 2 lectures, so I've officially been talking too much"-Larry YL Zhang 2023

  "If you don't mind, I have to take a phone call for a moment."-Larry YL Zhang 2023

In summary

Which one is more space efficient?

- Adjacency matrix is $\Theta(|V|^2)$
- Adjacency list is $\Theta(|V| + |E|)$

It would depend on the density of the graph. An adjacency list is good for graphs that are not very dense, i.e. $|E| \lll |V|^2$.

When is matrix better?

Checking some edge $\epsilon = (v_1, v_2) \in G$

- In a matrix, we can simply check if $A_{i,j} = 1$, which is $O(1)$
- In a list, we would have to look through the entire list to see if j is in the list, which is $O(n)$.

  *"*COUGH COUGH*, oh sorry"-Larry YL Zhang 2023"*

BFS and DFS

In essence, we want to visit every $v \in V$ exactly once starting from a given vertex. To do this, we invoke either BFS or DFS depending on the scenario. Consider a tree, which is a special type of graph. If we invoke BFS, we go from left to right, then top to bottom, whereas with DFS, we go top to bottom, then left to right.

Review

Recall that BFS in a tree is level by level traversal (not preorder). So we can use a queue to implement this algorithm. (slide 35)

```
1 NOT_QUITE_BFS(root):
2     Q = Queue()
3     Enqueue(Q, root)
4     while Q not empty:
5         x = Dequeue(Q)
6         print(x)
7         for each child c of x:
8             Enqueue(Q,c)
```

This will not work for regular graphs, because we might visit a vertex twice (which would cause infinite loops). We can avoid this by adding a label at each vertex that has already been visited (colouring). For each vertex when traversing, we want to remember some values::

- π_v is the vertex from which v is encountered, i.e. its "parent"
- d_v which is the distance of the current vertex from the source vertex (number of edges in unweighted, sum of weights in weighted).

Full example is on slides 40 to 48. The values we get at each vertex after running BFS are the distance from the source node to each node. But not only that. It's also the **shortest** distance. We can find a single path by using π_v.

If the graph is not connected, i.e. $\exists v \in V$ such that v is not connected to the rest of the graph by any edge, we can say that this node is **unreachable** by way of common sense. A good example of this is garbage collection in Java, where the JRE will check which functions are not used/called from the main function, whereby we can get rid of them.

> *"It is effectively a pile of garbage"-Larry YL Zhang 2023*

> *"Next class we will talk about BFS's evil twin DFS, so stay tuned"-Larry YL Zhang 2023*