CS 5968/6968: Data Str Alg Scalable Comp Spring 2023

Lecture 18 — March 27, 2023

Prof. Prashant Pandey *Scribe: Sarabjeet Singh*

1 Overview

So far, in this course, we have talked about the following topics:

1. Search Trees on integers
2. Data Structures (DS) for strings: Implicit DS, Succinct DS, Compact DS
3. Hashes:
    - Universal, K-wise independent, Simple tabulation hashing
    - Hash tables: Chaining hash functions, Perfect hashing, Linear probing, Cuckoo hashing, 2-choice hashing, Frontyard-Backyard hashing, Robinhood hashing
4. Sketches
5. Filters: Bloom filter, Quotient filter
6. Locality Sensitive Hashing, MinHash
7. Similarity Search, Cardinality, HyperLogLog

In this lecture, we will talk about graphs, its applications, its challenges, and common representation methods.

2 Introduction to Graphs

A graph is an abstract data type that is used to represent relationship between a set of nodes. Graph is represented by:

1. Vertices: models objects
2. Edges: models the relationship between the above objects

For instance, in Figure 1, v_1 & v_2 represent the vertices while e_1 represents the edge connecting the two vertices.

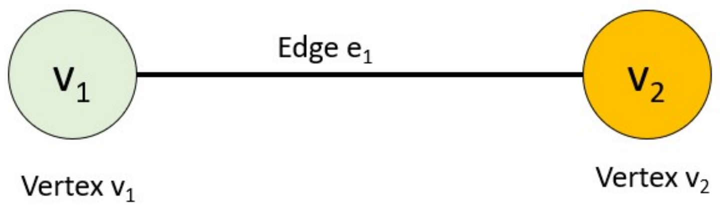


Figure 1: Graph with 2 vertices and an edge connecting them.

2.1 Applications of Graphs

Graphs are used everywhere in the real world. A few such applications are:

- Social Networking [1]
- Collaboration Network
- Transportation Network
- Computer Networks
- Genomics [2]
- Finance Transactions

2.2 Types of Graphs

While there exists numerous ways to represent graphs, the type of graph is determined by the properties of edge and vertices:

- Edges can be directed or undirected. Example: Family tree is a directed graph, while mutual friends is an undirected graph. Note that in directed graphs, the relationship can go one-way or both.
- Edges can be weighted, where the weight might denote the strength/distance/etc. Example: shortest distance graph.
- Edges and vertices can have metadata.

In this lecture, we will focus only on static graphs, where no insertions/deletions to the vertices would be made. An example of such a graph would be the SRA graph of a reference human genome. In the next lecture, we will extend our discussion to dynamic graphs.

3 Queries

In order to understand what we want from an ideal graph representation, we look at practical queries that a graph would get in various applications:

- Social Network Queries:
  - Find all your friends who went to the same high school as you.
  - Find cliques.
  - Mutual friend search.
  - Recommend people who you might know.
- Find good clusters:
  - finding groups of vertices that are *well connected* internally and *poorly connected* externally.
- Subgraph Finding:
  - finding or counting specific sub-groups inside a graph. Example: finding all the people that are live in a specific city, within the network of a person's mutual friends.
  - Finding the current subgraph.
  - Finding important nodes. Example: finding accounts in twitter that have high visibility of information.
  - Biological networks.

From the limited set of examples, we understand the the below set of operations are required from a *static* graph structures:

- $IsNeighbor(s, d)$: returns true if there exist an edge from vertex s to vertex d.
- $GetNeighbors(v)$: returns a set of all neighbors of vertex v.
- $AddEdge(s, d)$: adds an edge from vertex s to d.
- $DeleteEdge(s, d)$: remove the edge from vertex s to d.

4 Real world graphs

Graphs provide a good way to represent a list of objects while defining relationship between them. However, in practice, there are multiple issues that real use cases of graphs have to address. These include, but may not be limited to,:

- **Massive graphs:** Real use cases of graphs typically have massive amount of objects and/or edges that the framework wants to support. For instance: Twitter's social network graph consists 41.7 M vertices (users) and 1.47 B edges (social relations), as of 2010 [3], totalling

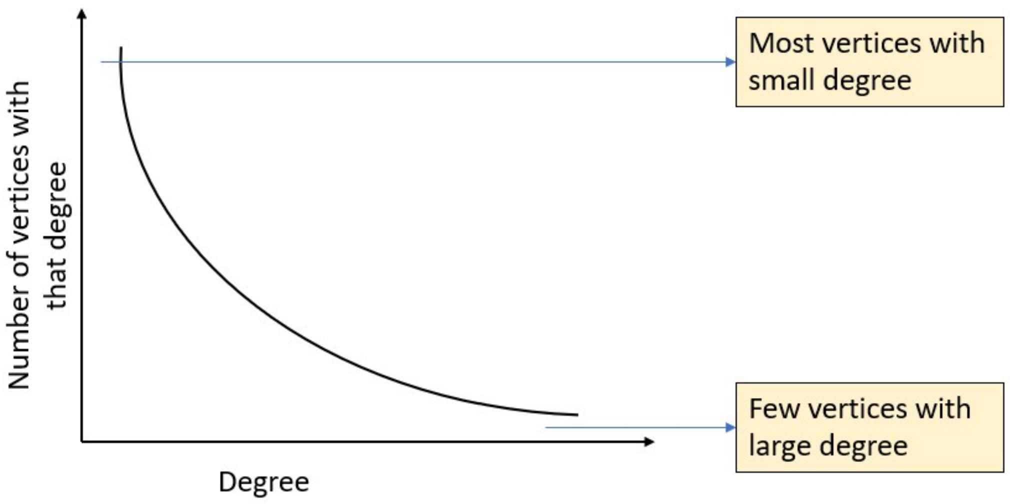


Figure 2: Skew in real graphs. Most vertices have very small degree, while a small portion of vertices have very large degrees.

6.3 GB of size. Web graphs have been reported to contain over 3.5 B web pages and 128.7 B links (Common Crawl Foundation 2012) [4], which results in a massive 540 GB graph. Similarly, SRA graphs in genomics can reach in petabytes in size. Therefore, for a practical implementation, space considerations are critical.

- **High Sparsity:** Real world graphs are typically very sparse. For instance, the Twitter's social graph has 41.1 M users. A fully connected graph over these users would have trillions of edges, but Kwak et al. report only 1.5 billion edges, depicting the high amounts of sparsity. Therefore, researchers should also take advantage of sparse representations to reduce the operation complexity and storage requirements.
- **Skewed Degrees:** Typical real applications of such graphs observe a skew in the operations that are performed over these structures. For instance, Figure 2 denotes a typical skew in the amount of activity seen on Twitter's network. There's a small subset of people that display a majority of tweets, while the majority of accounts sees little to no activity. Degree of skew impacts the load imbalance which must be addressed in order to maintain high resource utilization.

5 Graph Representations

For an efficient implementation of graphs, we must address the challenges mentioned in Section 4. In this section, we study a few popular graph representations. Let's consider a graph with n vertices and m edges.

5.1 Adjacency Matrix

Adjacency matrix ($Adj[n][n]$) is a two-dimensional bit array of size $n \times n$, where n is the total number of vertices. Bit $Adj[s][d]$ is set as 1 if there exists an edge from vertex s to vertex d, otherwise $A[s][d] = 0$. Figure 3a depicts this representation. Below are the properties of Adjacency Matrix:

$$Adj[s][d] = \begin{cases} 1, & \text{if there exists an edge from vertex } s \text{ to vertex } d \\ 0, & \text{otherwise} \end{cases}$$

- Storage size requirement: $O(n^2)$.
- Fast for querying $IsNeighbor(s, d)$. Return $Adj[s][d]$ in constant time.
- Fast for querying $GetNeighbors(v)$. Return $Adj[v][*]$ in $O(n)$ time.
- Fast $AddEdge(s, d)$ and $DeleteEdge(s, d)$, in constant time.

Overall, Adjacency Matrix is good for finding existence of a neighbor and insertion/deletion of relation, but requires large storage. This storage requirement is good if the graph is dense. However, if the graph is sparse, which is typically the case, the storage requirement is an overkill.

5.2 Edge List

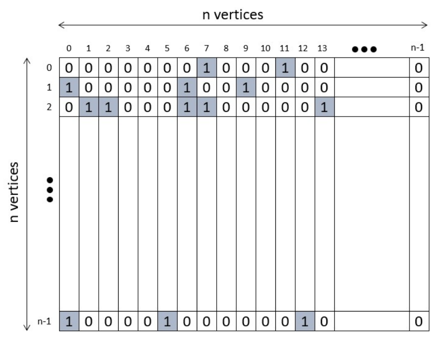
Adjacency Matrix defines a similar structure to bitmap of all possible edges, coming at a high storage cost. Instead, an Edge List only stores the existing edges between the nodes. Below are the properties of an Edge List:

- A relation/edge from vertex s to d exists if and only if (s, d) exists in the Edge List.
- Space requirement is $O(m)$, where m is the total number of edges.
- Finding neighbors, $IsNeighbor(s, d)$ and $GetNeighbors(v)$, both take $O(m)$ time as the function need to iterate all the entries of the edge list in worst case.
- $AddEdge(s, d)$ takes constant time, but $DeleteEdge(s, d)$ takes $O(m)$ time as we need to iterate the list to find position of the edge's entry.

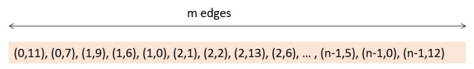
An Edge List requires low storage for sparse graphs as only existing edges are stored. However, it takes $O(m)$ time to find neighbors or relation between two nodes.

5.3 Adjacency List

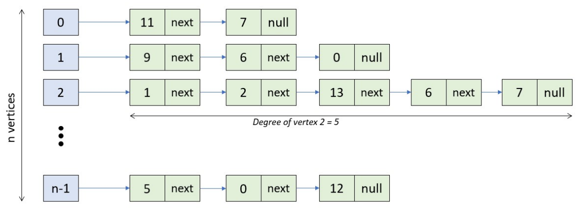
An adjacency List is a collection of list of edges, for each vertex. In other words, an array ($AdjList[n]$) of all vertices where $AdjList[v]$ points to a linked list of all vertices that are neighbors of vertex v. Figure 3c depicts an implementation of an Adjacency List. Below are the properties of such a representation:



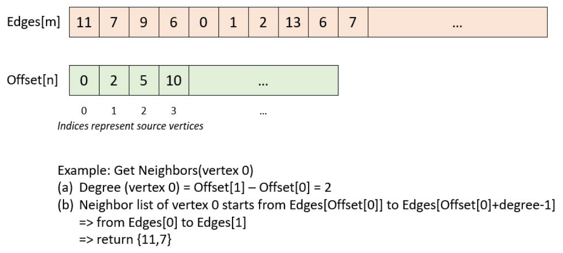
(a) Adjacency Matrix


m edges
(0,11), (0,7), (1,9), (1,6), (1,0), (2,1), (2,2), (2,13), (2,6), … , (n-1,5), (n-1,0), (n-1,12)

(b) Edge List



(c) Adjacency List


Edges[m]
Offset[n]
Indices represent source vertices
Example: Get Neighbors(vertex 0)
(a) Degree (vertex 0) = Offset[1] – Offset[0] = 2
(b) Neighbor list of vertex 0 starts from Edges[Offset[0]] to Edges[Offset[0]+degree-1]
=> from Edges[0] to Edges[1]
=> return {11,7}

(d) Compressed Sparse Rows

Figure 3: Various Graph representations. All four representations use the same graph example (same vertices and edges in all four representation) for easier comparison.

- Storage requirement is $O(n+m)$, as for each node the number of neighbors that are to be stored in the list can be m in the worst case.
- Since the list is unordered, $AddEdge(s,d)$ takes constant time, as the function only needs to insert vertex d in $AdjList[s]$ at the end.
- To delete an edge from a vertex s to vertex d, we need to scan the $AdjList[s]$'s list, which can be of length m in the worst case. However, in real applications, the actual number of neighbors of any vertex v, also known as $degree(v)$, is much smaller than total number of edges m. In short, $degree(v) \ll m$. Therefore, we say that $DeleteEdge(s,d)$ takes $O(degree(s))$ time.
- Similar to Edge List, the time for neighbor search depends on the length of the list. In Adjacency List, list of a vertex v is of size $degree(v)$. Hence, $IsNeighbor(s,d)$ takes $(O(degree(s))$ time and $GetNeighbors(v)$ take $(O(degree(v))$ time.
- Note that deletion and neighbor search above would take $O(\log(degree(v))$ instead of $O(degree(v))$ if the list was ordered, but so would adding a new edge.
- Another alternative to implement fast neighbor search is by maintaining a Hash Table of neighbors instead of a list. In such an implementation, $IsNeighbor(s,d)$ would take constant time.

In conclusion, Adjacency List reduces the neighbor search and deletion time to degree of the vertex as compared to iterating over all edges in Edge List. However, this performance requires trading off

Table 1: Trade-offs in graph representation

	Adj. Matrix	Edge List	Adj. List	Compressed Sparse Rows	Adj. List + Hash Tables
Storage	$O(n^2)$	$O(m)$	$O(n+m)$	$O(n+m)$	$O(n+m)$
$AddEdge(s,d)$	$O(1)$	$O(1)$	$O(1)$	$O(n+m)$	$O(1)$
$DeleteEdge(s,d)$	$O(1)$	$O(m)$	$O(degree(s))$	$O(n+m)$	$O(1)$
$GetNeighbor(v)$	$O(n)$	$O(m)$	$O(degree(v))$	$O(degree(v))$	$O(degree(v))$
$IsNeighbor(s,d)$	$O(1)$	$O(m)$	$O(degree(s))$	$O(degree(s))$	$O(1)$

storage. Also, note that this structure is based on pointers and requires pointer-chasing to perform operations.

5.4 Compressed Sparse Rows (CSR)

The CSR representation stores edges and vertices in two separate arrays: (a) $Edges[m]$ array is a sorted array that stores the destination of each edge. This array is sorted as per the source vertices of each edge. (b) The $Offset[n]$ array stores the offsets into the edge array, denoting the offset of the first edge going outgoing from each vertex. In other words, $Offset[i]$ denotes the offset of where vertex i's edges start in $Edges$ array. For instance, in Figure 3d, $degree(vertex0) = Offset[1] - Offset[1] = 2$ denotes that vertex 0 have exactly 2 edges. Looking at the $Edges[0]$ & $Edges[1]$, we say that those 2 edges go to vertices 11 and 7. Below are the properties of CSR:

- Storage requirement is $O(n+m)$ as we need to store the 2 arrays of size n and m.
- Degree of a vertex v is calculated by: $degree(v) = Offset[v+1] - Offset[v]$.
- Neighbor search for a vertex v is performed by indexing into $Edges$ array based on its offset and degree. To be precise, $\{Edges[v], Edges[v+1], ..., Edges[v + degree(v) - 1]\}$ is the set of all neighbors of v. Therefore, $GetNeighbor(v)$ and $IsNeighbor(s,d)$ takes $O(degree(v)$ and $O(degree(s)$ time, respectively.
- $AddEdge(s,d)$ and $DeleteEdge(s,d)$ take $O(n+m)$ time as both the arrays need to be updated.

CSR requires less storage than Adjacency List as it doesn't need pointers, and same asymptotic time complexity for neighbor search. However, note that Adjacency List requires pointer chasing, while CSR can exploit spatial locality in arrays. Therefore, in actual implementation, CSR outperforms Adjacency List. Table 1 compares all the four graph representations and suggests that CSR is typically the best implementation.

6 Conclusion

In this lecture, we study Graphs and its types and applications. We focus on static graphs and understand the typical operations that real use cases need from such graphs. Next, we look at a few real graph implementations and observe properties that one must take into account while developing a scalable graph representation. Finally, we discuss four popular graph representations and study their trade-offs in terms of storage requirement and query time. Table 1 summarizes the costs of each representation. We conclude that for static graph implementations, CSR is typically the best choice, unless we have a very dense graph.

References

[1] Wasserman, Stanley, and Philippa Pattison. Logit models and logistic regressions for social networks: I. An introduction to Markov graphs and p. *Psychometrika*, 61.3 (1996): 401-425.

[2] Compeau, Phillip EC, Pavel A. Pevzner, and Glenn Tesler. How to apply de Bruijn graphs to genome assembly. *Nature biotechnology*, 29.11 (2011): 987-991.

[3] Kwak, Haewoon, Changhyun Lee, Hosung Park, and Sue Moon. What is Twitter, a social network or a news media? *Proceedings of the 19th international conference on World wide web*, pp. 591-600. 2010.

[4] Meusel, Robert, Sebastiano Vigna, Oliver Lehmberg, and Christian Bizerr. Graph structure in the web—revisited: a trick of the heavy tail. *Proceedings of the 23rd international conference on World Wide Web*, pp. 427-432. 2014.