Compressed Sparse Row: A Dynamic Graph Representation

Andrea Bosisio
Computer Science and Engineering
andrea2.bosisio@mail.polimi.it
Politecnico di Milano

Salvatore Gabriele Karra
Computer Science and Engineering
salvatoregabriele.karra@mail.polimi.it
Politecnico di Milano

Abstract

In the Data Structure track of the HPGDA contest there is the chance to play with different sparse graph representations and explore their runtime performance to improve. In particular, the challenge requires the creation of an appropriate data structure for the implementation of a graph, a population algorithm and a neighbor search algorithm for each node. The data structure interfaces with real data from the LDBC Graphanalytics Benchmark suite dataset. The evaluation is based on: graph population time, graph size in memory and execution time of the BFS and DFS algorithms.

I. Introduction

Many real-world graphs such as the Facebook social network [UKBM11] are sparse in that the number of edges present in the graphs are much smaller than the possible number of edges. In practice, sparse matrices and graphs are often stored in compressed sparse row (CSR) format, which packs edges into an array and takes space proportional number of vertices and edges. Sparse storage formats pay for these space savings with the cost of updates. CSR format supports fast queries such as membership or finding all neighbors of a vertex, but may require changing the entire data structure to add or remove an edge. Social networks such as Facebook and Twitter are highly dynamic graphs since new users and connections are added constantly. Twitter averages about 500 million tweets a day [Say] and Facebook has about 41,000 posts (2.5Mb of data) per second [WKF+15]. Compressed sparse row (CSR) is a popular format for storing sparse graphs and matrices. It efficiently packs all the entries together in arrays, allowing for quick traversals of the data structure. CSR uses three arrays to store a sparse graph: a node array, an edge array, and a values array. In our implementation we did not memorized three arrays, but we used instead 1 pointer to an array and another one to a pair struct to store the edge arrays and the value array. Each entry in the node array contains the starting index in the edge array where the edges from that node are stored in sorted order by destination. The edge array stores the destination vertices of each edge. CSR stores a graph $G = (NumEdges, NumVertex)$ in size $O(|NumEdges| + |NumVertex|)$.

II. Data pre-processing

To create a general structure that could be populated by any type of list of edges, even if there were nodes without arcs or nodes that do not exist sequentially (e.g. graphs that do not start from node 0 or that have missing nodes) we have preprocessed the data, ordering them sequentially and mapping them by key "new index" value "old index". The new index is then stored sequentially and this also increases the chache hint. The map is then reused to decipher the old indexes.

III. Methods

We present a CSR data structure with the following methods: **Populate, Get Neighborns**

I. Populate

For the populate method we started with the analysis of the edge list that we receive as input. For CSR we must receive this list in an orderly manner. We recognized a similarity between the structure we chose and the counting sort algorithm. Below, the implementation of how we mixed the sorting algorithm with our structure to optimize time and space complexity.

```
// Re-arranged counting sort
// Count number of neighbors for each vertex
for (uint64_t i = 0; i < num_edges; i++)
  row_ptr[std::get<0>(e_list[i]) + 1]++;
...
//Cumulative sum
for (uint64_t i = 1; i <= num_vertices; i++)
{
    row_ptr[i] += row_ptr[i - 1];
    count[i - 1] = row_ptr[i];
}
...
// Sorting col_idx and weights
for (uint64_t i = 0; i < num_edges; i++){
    curr_edge = e_list[i];
    curr_vertex = std::get<0>(curr_edge);
    new_pos = count[curr_vertex] - 1;
    col_idx[new_pos] = std::get<1>(curr_edge);
    col_idx_weight[new_pos] = std::make_pair(
    std::get<1>(curr_edge), std::get<2>(curr_edge));
    count[curr_vertex]--;
}
```

Thanks to this similarity we have obtained a time complexity equal to *O(NumEdges)*.

II. Get Neighborns

For the Get Neighborns method we used a customized iterator to scroll through the array and the pair struct to return the start and end positions of the neighbors of each node and the weight of each edge. We used an Input Iterator, because we need only to read-only once each value pointed, in this way we dont need to keep trace of the size of the arrays and we can possible add or remove elements efficently.

IV. Results

System: We ran our experiments on an laptop with Intel Core i7-8550U, and 1.8GHz clock speed, 4 cores. The machine had 8GB of RAM, 256KB of L1 cache, 1M of L2 cache, and 8MB of L3 cache. Programs were written in c++ and compiled with GCC 9.4.0.

wiki Talks - 5.021.410 edges, 2.394.385 nodes
cit Patents - 16.518.947 edges, 3.774.768 nodes
dota league - 50.870.313 edges, 61.170 nodes

Table 1: *Compered result average populating time and memory usage for AL and CSR - Serial*

CSR vs AL		
AL	CSR	Graph
$695ms - 454MB$	$135ms - 1MB$	wiki-Talks
$8967ms - 2333MB$	$1867ms - 872MB$	dota-league
$5443ms - 1182MB$	$1789ms - 58MB$	cit-Patents

Table 2: *Compered result average BFS and DFS time for AL and CSR - Serial*

CSR vs AL (BFS-DFS)		
AL	CSR	Graph
$(767ms - 696ms)$	$(142ms - 219ms)$	wiki-Talks
$(197ms - 217ms)$	$(1ms - 1ms)$	dota-league
$(186ms - 188ms)$	$(40ms - 35ms)$	cit-Patents

V. Parallelization

For parallelization we used the openMP library which makes concurrent programming possible in a simple way and a recution in terms of times and space.

Table 3: *Compered result average populating time for AL and CSR - Parallel*

CSR VS AL	
CSR	Graph
$135ms$	wiki-Talks
$1867ms$	dota-league
$1789ms$	cit-Patents