An Experimental Analysis of a Compact Graph Representation

Author(s): Daniel K. Blandford, Guy E. Blelloch, Ian A. Kash

18.1 Paper review

There is a clear need for compact graph representations. The more of a graph that can be fit into physical memory, the better the access patterns for graph processing algorithms and the better the cache characteristics of the graph, both of which make processing the graph significantly faster. There are also benefits for handheld devices, which may need to work quickly in offline settings.

The authors draw off previous work in which they presented a compact graph representation based on graph separators, where the separators are defined as sets of nodes that form the boundary between two equally-sized sections of the graph. The method is especially effective for graphs like street grids, which have small separators and strong local structure, and less effective for random graphs, which may have very large separators. This paper presents an experimental analysis of this representation for real-world sparse graphs (FEM, circuits, maps, router connectivity, link graphs, etc.), performed on different hardware systems, and modified to accommodate dynamic graphs.

The separator algorithm is based on a separator tree. This is formed as a byproduct of partitioning the graph by its separators until there are only single nodes as leaves, a process which can be performed in linear time using a bottom-up approach. This approach leverages graph contraction algorithms, beginning with individual nodes and collapsing based on a weighted priority metric. The contraction ends when only one node, the root of the separator tree, remains. The structure of the separator tree determines the vertex labels for storage. The authors also get into the weeds with label representation, expanding beyond the standard gamma code, which requires significant decoding, by introducing the novel byte, nibble, and snip codes. Other tricks, such as anticipating repeated access by decoding a queried vertex's neighbors and storing them in cache as a temporary FIFO adjacency list, additionally contribute to the efficiency of the algorithm.

For the experimental portion of the paper, the authors draw 11 graphs real-world graphs from various sources, including both directed and undirected graphs. Two machines (Pentium III and Pentium 4) are used for evaluation, both of which have 32-bit processors. However, Pentium 4 has four times the effective cache-line size of Pentium III and supports hardware prefetching and quadruple loads. This reduces the latency associated with memory access, an improvement which is especially helpful in analyzing graphs with spatial locality. The experiments themselves are an examination of the impact of different parameters on the processing speed, including prefix codes, block size, cache usage, and edge encoding for out-edges.

One of the most promising findings is that ordering the vertices according to the graph separation algorithm described above has a significant impact on processing time compared to random ordering, up to a factor of 8 for low-degree graphs and on average 3.5x across all graphs. This is promising for the mobile application the authors foresee, as street grids tend to be spatially localized and low-degree. The authors also find that their novel byte prefix code performs excellently, largely because it leverages the native byte instructions of the machines and can be easily compressed. The speed-up is especially noticeable on the Pentium III, which has a smaller cache-line size and benefits from greater throughput owing to code compression. The authors also find meaningful accelerations in the dynamic case, especially when larger block-sizes are used, therefore accelerating the hash computation and reducing the likelihood of a time-intensive cache miss. Finally, they demonstrate its practical application to real-world algorithms on real-world graphs, e.g. PageRank on Google's internet link graph. PageRank is especially interesting because the access pattern is very different than DFS, which is the workhorse of this paper, so demonstrating the usefulness of their compression format on a is interesting.

This paper is a rigorous and thorough extension to the previous work. I appreciate the careful identification of parameters that are critical to the appropriate tuning of their implementation, and the causal relationships drawn between the findings and the hardware the authors tested on, though it would have been nice to get statistical bounds/some sense of reproducibility in the results. The algorithms evaluated (DFS, PageRank, Bipartite matching) are still somewhat limited and largely read-only, so it would be interesting to see if algorithms with more writing components (that may induce race conditions) would perform differently. Additionally, although I appreciate that two machines with different cache architectures (i.e. ensuring that there is a controlled experimental variable) were selected, it would be nice to see a larger-scale study on a range of machines that are used in real datacenters. Complementing that critique, this algorithm has not been demonstrated on a distributed system, since both evaluated machines are local single-core, in-memory, and it would be cool to see an extension of the separator-based system to distributed systems and work partitioning.