Disk-Based Search

Stefan Edelkamp
University of Dortmund, Germany

Shahid Jabbar
University of Dortmund, Germany

INTRODUCTION

The need to deal with large data sets is at the heart of many real-world problems. In many organizations the data size has already surpassed Petabytes (10^{15}). It is clear that to process such an enormous amount of data, the physical limitations of RAM is a major hurdle. However, the media that can hold huge data sets, i.e., hard disks, are about a 10,000 to 1,000,000 times slower to access than RAM. On the other hand, the costs for large amounts of disk space have considerably decreased. This growing disparity has led to a rising attention to the design of *external memory algorithms* (Sanders et al., 2003) in recent years.

In a hard disk, random disk accesses are slow due to disk latency in moving the head on top of the data. But once the head is at its proper position, data can be read very rapidly. External memory algorithms exploit this fact by processing the data in the form of blocks. They are more informed about the future accesses to the data and can organize their execution to have minimum number of block accesses.

Traditional graph search algorithms perform well as long as the graph can fit into the RAM. But for large graphs these algorithms are destined to fail. In the following, we will review some of the advances in the field of search algorithms designed for large graphs.

BACKGROUND

Most modern operating systems provide a general-purpose memory management scheme called *Virtual Memory* to compensate for the limited RAM. Unfortunately, such schemes pay off only when the algorithm's memory accesses are local, i.e., it works on a particular memory address range for a while, before switching the attention to another range. Search algorithms, especially those that order the nodes on some particular node property, do not show such behaviour. They jump back and forth to pick the best node, in a spatially unrelated way for only marginal differences in the node property.

External memory algorithms are designed with a hierarchy of memories in mind. They are analyzed on an *external memory model* as opposed to the traditional von Neumann RAM model. We use the two-level memory model by Vitter and Shriver (1994) to describe the search algorithms. The model provides the necessary tools to analyze the asymptotic number of block accesses (I/O operations) as the input size grows. It consists of

- M: Size of the internal memory in terms of the number of elements,
- $N >> M$: Size of the input in terms of the number of elements, and
- B: Size of the data block that can be transferred between the internal memory and the hard disk; transferring one such block is called as a single I/O operation.

The complexity of external memory algorithms is conveniently expressed in terms of predefined I/O operations, such as, $scan(N)$ for scanning a file of size N with a complexity of $\Theta(N/B)$ I/Os, and $sort(N)$ for external sorting a file of size N with a complexity of $\Theta(N/B \log_{M/B} (N/B))$ I/Os. With additional parameters the model can accommodate multiple disks and multiple processors too.

In the following, we assume a graph as a tuple (V, E, c), where V is the set of nodes, E the set of edges, and c the weight function that assigns a non-zero positive integer to each edge. If all edges have the same weight, the component c can be dropped and the graphs are called as *unweighted*. Given a start node s and a goal node g, we require the search algorithm to return an optimal path wrt. the weight function.

EXTERNAL MEMORY SEARCH ALGORITHMS

External Memory Breadth-First Search

Breadth-first search (BFS) is one of the basic search algorithms. It explores a graph by first expanding the nodes that are closest to the start node. BFS for external memory has been proposed by Munagala and Ranade (1999). It only considers undirected and explicit (provided beforehand in the form of adjacency lists) graphs. The working of the algorithm is illustrated on a graph in Fig. 1. Let *Open*(*i*) be the set of nodes at BFS level i residing on disk. The algorithm builds *Open*(*i*) from *Open*(*i*-1) as follows. Let *Succ*(*Open*(*i*-1)) be the multi-set of successors of nodes in *Open*(*i*-1); this set is created by concatenating all adjacency lists of nodes in *Open*(*i*-1). As there can be multiple copies of the same node in this set the next step is to remove these duplicate nodes. In an internal memory setting this can be done easily using a hash table. Unfortunately, in an external setting a hash-table is not affordable due to random accesses to its contents. Therefore, we rely on alternative methods of duplicates' removal that are well-suited for large data on disk. The first step is to sort the successor set using external sorting algorithms resulting in duplicate nodes lying adjacent to each other. By an external scanning of this sorted set, all duplicates are removed. Still, there can be nodes in this set that have already been expanded in the previous layers. Munagala and Ranade proved that for undirected graphs, it is sufficient to subtract only two layers, *Open*(*i*–1) and *Open*(*i*–2), from *Open*(*i*). Since all three lists are sorted, this can be done by a parallel external scanning. The accumulated I/O complexity of this algorithm is $O(|V| + sort(|E|))$ I/Os, where $|V|$ is for the unstructured access to the adjacency lists, and $sort(|E|)$ for duplicates removal.

An implicit graph variant of the above algorithm has been proposed by Korf (2003). It applies $O(sort(|Succ(Open(i-1))|) + scan(|Open(i-1)| + |Open(i-2)|))$ I/Os in each iteration. Since no explicit access to the adjacency list is needed (as the state space is generated on-the-fly), by using $\Sigma_i |Succ(Open(i))| = O(|E|)$ and $\Sigma_i |Open(i)| = O(|V|)$, the total execution time is bounded by $O(sort(|E|) + scan(|V|))$ I/Os.

To reconstruct a solution path, we may store predecessor information with each node on disk (thus doubling the state vector size). Starting from the goal node, we recursively search for its predecessor in the previous layer through external scanning. The process continues until the first layer containing the start node is reached. Since the Breadth-first search preserves the shortest paths in a uniformly weighted graph, the

Figure 1. An example graph (left); Stages of External Breadth-First Search (right). Each horizontal bar corresponds to a file. The grey-shaded A(2) and A'(2) are temporary files.