# Traversing large graphs in realistic settings

**Deepak Ajwani**

**Max-Planck-Institut für Informatik**
**Saarbrücken, Germany**

Dissertation zur Erlangung des Grades
*Doktor der Ingenieurwissenschaften(Dr.-Ing.)*
der Naturwissenschaftlich-Technischen Fakultäten I
der Universität des Saarlandes

**Betreuender Hochschullehrer – Supervisor**
Prof. Dr. h. c. Kurt Mehlhorn,
Max-Planck-Institut für Informatik, Saarbrücken, Germany

**Gutachter – Reviewers**
Prof. Dr.-Ing. Ulrich Meyer,
Institut für Informatik,
Goethe-Universität, Frankfurt am Main, Germany

Gerth Brodal, Ph.D.,
Department of Computer Science,
Aarhus University, Århus, Denmark

**Vorsitzender des Prüfungsausschusses – Chairman of the Examination Board**
Prof. Dr. Raimund Seidel,
Universität des Saarlandes, Saarbrücken, Germany

**Beisitzer – Observer**
Dr. Stefan Canzar,
Max-Planck-Institut für Informatik, Saarbrücken, Germany

**Dekan – Dean**
Prof. Dr. Joachim Weickert,
Universität des Saarlandes, Saarbrücken, Germany

**Datum des Kolloquiums – Date of Defense**
21. Dezember 2008 – December 21st, 2008

Deepak Ajwani
MADALGO – Center for Massive Data Algorithmics,
IT-parken, Åbogade 34,
DK-8200 Århus N, Denmark
ajwani@madalgo.au.dk

# Abstract

The notion of graph traversal is of fundamental importance to solving many computational problems. In many modern applications involving graph traversal such as those arising in the domain of social networks, Internet based services, fraud detection in telephone calls etc., the underlying graph is very large and dynamically evolving. This thesis deals with the design and engineering of traversal algorithms for such graphs.

We engineer various I/O-efficient Breadth First Search (BFS) algorithms for massive sparse undirected graphs. Our pipelined implementations with low constant factors, together with some heuristics preserving the worst-case guarantees makes BFS viable on massive graphs. We perform an extensive set of experiments to study the effect of various graph properties such as diameter, initial disk layouts, tuning parameters, disk parallelism, cache-obliviousness etc. on the relative performance of these algorithms.

We characterize the performance of NAND flash based storage devices, including many solid state disks. We show that despite the similarities between flash memory and RAM (fast random reads) and between flash disk and hard disk (both are block based devices), the algorithms designed in the RAM model or the external memory model do not realize the full potential of the flash memory devices. We also analyze the effect of misalignments, aging, past I/O patterns, etc. on the performance obtained on these devices. We also consider I/O-efficient BFS algorithms for the case when a hard disk and a solid state disk are used together.

We present a simple algorithm which maintains the topological order of a directed acyclic graph with $n$ nodes under an online edge insertion sequence in $O(n^{2.75})$ time, independent of the number $m$ of edges inserted. For dense DAGs, this is an improvement over the previous best result of $O(\min\{m^{\frac{3}{2}}\log n, m^{\frac{3}{2}} + n^2 \log n\})$. While our analysis holds only for the incremental setting, our algorithm itself is fully dynamic.

We also present the first average-case analysis of online topological ordering algorithms. We prove an expected runtime of $O(n^2 \operatorname{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for various incremental topological ordering algorithms.

# Kurzfassung

Die Traversierung von Graphen ist von fundamentaler Bedeutung für das Lösen vieler Berechnungsprobleme. Moderne Anwendungen, die auf Graphtraversierung beruhen, findet man unter anderem in sozialen Netzwerken, internetbasierten Dienstleistungen, Betrugserkennung bei Telefonanrufen. In vielen dieser Anwendungen ist der zugrunde liegende Graph sehr gross und ändert sich kontinuierlich.

Wir entwickeln mehrere I/O-effiziente Breitensuch-Algorithmen für massive, dünnbesiedelte, ungerichtete Graphen. Im Zusammenspiel mit Heuristiken zur Einhaltung von Worst-Case-Garantien, ermöglichen unsere pipeline-basierten Implementierungen die Praktikabilität von Breitensuche auf massiven Graphen. Wir führen eine Vielfalt an Experimente durch, um die Wirkung unterschiedlicher Grapheigenschaften zu untersuchen, wie z.B. Graph-Durchmesser, anfängliche Belegung der Festplatte, Tuning-Parameter, Plattenparallelismus.

Wir charakterisieren die Leistung von NAND-Flash basierten Speichermedien, einschliesslich vieler solid-state Disks. Wir zeigen, dass trotz der Ähnlichkeiten von Flash-Speicher und RAM (schnelle wahlfreie Lese-Zugriffe) und von Flash-Platten und Festplatten (beide sind blockbasiert) Algorithmen, die für das RAM-Modell oder das Externspeicher-Modell entworfenen wurden, nicht das volle Potential der Flash-Speicher-Medien ausschöpfen. Zusätzlich analysieren wir die Wirkung von Ausrichtungsfehlern, Alterung, vorausgehenden I/O-Mustern, usw., auf die Leistung dieser Medien. Wir berücksichtigen auch I/O-effiziente Breitensuch-Algorithmen für die gleichzeitige Nutzung von Festplatten und solid-state Disks.

Wir stellen einen einfachen Algorithmus vor, der beim Online-Einfügen von Kanten die topologische Ordnung von einem gerichteten, azyklischen Graphen (DAG) mit $n$ Knoten beibehält. Dieser Algorithmus hat eine Laufzeitkomplexität von $O(n^{2.75})$ unabhängig von der Anzahl $m$ der eingefügten Kanten. Für dichte DAGs ist dies eine Verbesserung des besten, vorherigen Ergebnisses von $O(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$. Während die Analyse nur im inkrementellen Szenario gütlig ist,

ist unser Algorithmus völlständig dynamisch.

Ferner stellen wir die erste Average-Case-Analyse von Online-Algorithmen zur Unterhaltung einer topologischen Ordnung vor. Für mehrere inkrementelle Algorithmen, welche die Kanten eines kompletten DAGs in zufälliger Reihenfolge einfügen, beweisen wir eine erwartete Laufzeit von $O(n^2 \operatorname{polylog}(n))$.

# Acknowledgements

First of all, I would like to thank my two supervisors Kurt Mehlhorn and Ulrich Meyer for their generous support. In particular, I would like to thank Uli for his guidance, not only in scientific affairs, but also in practical matters of everyday life. At the same time, he gave me considerable scientific freedom to work on many different topics at the same time, thereby providing me ample opportunities to develop my own ideas and grow as an independent researcher. He has been very patient with me, as the engineering projects in this dissertation took much longer than expected.

I would also like to thank my co-authors Saurabh Ray, Roman Dementiev, Tobias Friedrich, Khaled Elbassioni, Hans Raj Tiwary, Sathish Govindarajan, Vitaly Osipov and Raimund Seidel. I learnt a lot while working with them. I am also grateful to Norbert Zeh for sharing valuable insights and many helpful discussions. Thanks are also due to Andreas Beckmann for his great help with organizing and debugging the external memory implementations.

I am also grateful to the members of my thesis examination committee who carefully read this thesis and raised many interesting questions during my defense.

My friends Shirley, Sky, Imran, Rali, Joisy and Chris made the life in Saarbruecken so much fun. I always looked forward to having discussions with the lunch-group of the Databases and Information Retrieval group, in particular Fabian, Gjergji, Julia, Gerard and Mouna.

I would also like to acknowledge the International Max-Planck Research School (IMPRS), Deutsche Forschungsgemeinschaft (DFG) and Max-Planck Society for their financial support in the past years. MPII has been an incredibly nice place to work. Its transparent building and the beautiful nature surrounding it add to its charms. The bureacracy here is almost hidden – the ease of procuring new hardware and getting it installed is truly amazing.

I am indebted to my parents, who made sure that I get the best possible education and let me pursue my career abroad.

And last but not the least, I would like to thank my wife Georgiana Ifrim for her unflinching love through the thick and thin. She had been encouraging and understanding, even as I had been mentally immersed in my work.

# Contents

# Chapter 1

# Introduction

*A theory must be tempered with reality.*

— Jawaharlal Nehru

A graph is one of the most useful objects in discrete mathematics. It can be used to represent physical networks such as electrical circuits, roadways or organic molecules as well as less tangible interactions as might occur in ecosystems, sociological relationships, databases or in the flow of control in a computer program. It therefore comes as no surprise that graph theory finds applications in physics, chemistry, communication science, computer science, electrical and civil engineering, architecture, operational research, genetics, psychology, sociology, economics, anthropology and linguistics. The theory is also intimately related to many branches of mathematics, including group theory, matrix theory, numerical analysis, probability, topology, and combinatorics. In fact, graph theory serves as a mathematical model for any system involving a binary relation.

The notion of graph traversal is nearly as old and as important as the notion of a graph itself. One of the most celebrated results in graph traversal dates back to 1736 when Leonhard Euler solved the famous *Seven Bridges of Königsberg problem* using a graph traversal technique called Euler tour. A surprisingly large number of optimization problems from many different domains can be reduced to traversing graphs in a structured way.

Graph traversal algorithms have therefore received considerable attention in the computer science literature. Simple linear time algorithms have been developed for Breadth-First Search (BFS), Depth-First Search (DFS), computing connected and strongly connected components on directed graphs, and topological ordering of directed acyclic graphs [51]. Also, there exist near-linear time algorithms for computing Minimum Spanning Trees (MST) [45, 88, 128] of undirected graphs. Dijkstra's algorithm [61] with Fibonacci heaps [71] can solve the Single-Source Shortest-Paths (SSSP) [61, 71] problem on directed graphs with non-negative weights in $O(m+n\log n)$, where $n$ is the number of nodes and $m$ is the number of edges in the graph. For All-Pair Shortest-Paths (APSP), the naïve algorithm of computing SSSP from all nodes takes $O(m\cdot n+n^2\log n)$. It has been improved to $O(m\cdot n+n^2\log\log n)$ [127] for sparse graphs and $O(n^3/\log n)$ [44] for dense graphs.

## 1.1 Large graphs

In many applications involving graph traversal, the underlying graph is too big to fit in the internal memory of the computing device. Consider the following examples:

- The World Wide Web (WWW) can be looked upon as a massive graph where each web-page is a node and the hyperlink from one page to another is a directed edge between the nodes corresponding to those pages. As of August 2008, it is estimated that the indexed web contains at least 27 billion webpages [53].

  Typical problems in the analysis (e.g., [35, 95]) of WWW graphs include computing the diameter of the graph, computing the diameter of the core of the graph, computing connected and strongly connected components and other structural properties such as computing the correct parameters for the power law modeling of WWW graphs. There has also been a lot of work on understanding the evolution of such graphs.

  Computing Page rank [36] (the basis of the Google search engine) is considered to be a very important problem with respect to webgraphs, owing to its immense usage in search engines, classification and many other machine learning applications. A key challenge here is that since the webgraph is continuously evolving, recomputing Page rank every time there is a minor modification in the webgraph is considered to be "increasingly infeasible" [59].

- Social networking websites such as Facebook, Orkut, MySpace, LinkedIn etc. also provide massive and continuously evolving graphs. The nodes here refer to the profiles of people and an edge refers to an acknowledgement of acquaintance between two people. Typical problems on these graphs are computing similarity based clustering to find communities of people.

- Citation graphs of scientific papers from specific domains, where nodes are the publications and a directed edge from one paper to the other reflects a citation is yet another such graph class. The main problem here is to understand the nature of scientific collaboration and identify communities.

- Automatic classification of data items, based on training samples, can be boosted by considering the neighborhood of data items in a graph structure [16]. This is particularly useful when the objects to be classified are images (in web-sites such as Flickr) or videos (in web-sites such as YouTube). The tags associated with these pictures and videos are hardly enough for their classification. The graph structure containing the likings and dislikings of different users provides important clues that can improve the classification accuracy significantly. Such graphs can often be quite huge. For example, the online photo sharing network Flickr that started in 2004 had more than two billion pictures as of November 2007 [70] and claims that three to five million photos are updated daily on its network.

- There have also been attempts (e.g., [98]) to improve the results of web search by using the implicit feedback obtained from query logs. The underlying assumption behind these approaches is that by clicking (or ignoring) the results provided by the search engines for a particular query, users mark the relevance of clicked (or ignored) pages with respect to their query. The graph here is huge as the set of nodes consists not only of the web-pages, but also of all the queries posted by users to the search engine. There is an edge between a node representing a query $q$ and a node representing a web-page $w$ if a user searches for the query $q$, the search engine shows him/her the web-page $w$ as a result, and he/she clicks on it.

- Telephone call graphs: Telephone call graphs have the telephone numbers operated by a company as nodes and there is a directed edge between two nodes if and only if there has been a call from one to another in a certain time-frame. The call graphs managed by telecom companies like AT&T can be massive. Typical problems on telephone call graphs are fraud detection and searching for local communities (e.g., by detecting maximum cliques [1]).

- GIS terrain data: Remote sensing has made massive amounts of high resolution terrain data readily available. Terrain analysis is central to a range of important geographic information systems (GIS) applications concerned with the effects of topography. Typical problems in this domain involve flow routing and flow accumulation [22].

- Route planning on small PDA devices [76, 141]: The PDA devices used to compute the shortest/fastest routes have very small main memory. Although the street maps of even continents are only a few hundred MBs in size, they are too large to fit into the small main memory of these devices.

- State space search in Artificial Intelligence [64]: In many applications of model checking, the state space that needs to be searched is too big to fit in the main memory. In these graphs, the different configuration states are the nodes and the edge between two nodes represent the possibility of a transition from one state to another in the course of the protocol/algorithm being checked. Typical problems in this domain are reachability analysis (to find out if a protocol can ever enter into a wrong state) [84], cycle detection (to search for liveness properties) [63], or just finding some path to eventually reach a goal state (action planning).

- Semantic graphs (e.g., [99]), where the nodes represent entities and the edges represent relationship between two entities can also be quite huge. Typical problems in the analysis of semantic graphs include determining the nature of the relationship between nodes in the graph. Such queries can be answered by finding shortest paths or computing Steiner trees. Another key area of interest on semantic graphs is community analysis.

- Purchase graph from electronic commerce (e-commerce) companies such as the online book-shop Amazon is yet another example of massive and continuously evolving graph. Here, the bipartite graph consists of the products (such as books) and the buyers as the nodes and a purchase as an edge. There has been a lot of work in building a personalized recommendation system to show those products to users that they may also like. This is typically based on their past purchases. The key idea here is to identify other users whose purchasing behavior is similar and recommend the appropriately weighted sum of their other purchases.

- Frameworks for keyword querying of relational databases (e.g., [28]) may also involve traversing large graphs.

- Many problems arising in VLSI design, XML query processing, querying

ontology DAGs, Delaunay triangulation of meshes in computer graphics, visualization of biological networks such as protein-protein interactions, and molecular data mining also involve traversing large graphs.

While some solutions for these problems are based on sparse-matrix dense-vector multiplications, or approximating the solution using integer linear programming, a large number of solutions rely on traversing graphs. For example, a community detection algorithm by Newman and Girman [119] uses all-pair BFS as a subroutine to identify edges with high "betweenness", where betweenness is some measure that favors edges that lie between communities and disfavors those that lie inside communities. Removal of these edges reveal the inherent "natural" division of the network into groups.

## 1.2 Realistic setting for traversing large graphs

Since, the standard linear or near-linear time algorithms for graph traversal are also reasonably simple, it is tempting to use them directly in real applications involving large and massive graphs as well. Unfortunately, the real world offers many more challenges than the ones for which our simple algorithms are designed.

First and foremost, these algorithms are designed and analyzed in the von Neumann or RAM model of computation. This model assumes a unit cost access to any memory location. In reality, the computer architecture is far more complex. There is a sophisticated memory hierarchy (cf. Section 2.4.2) and the cost of data access depends on the level of memory where the data is currently residing. In particular, the cost of accessing the data from the disk is about a million times more than that of accessing it from the L1 cache.

As the storage requirements of the input graph reaches and exceeds the size of the main memory available, the running time of the simple linear or near-linear time algorithms deviates significantly from their predicted asymptotic performance in the RAM model. Furthermore, on massive graphs (with a billion or more edges), these algorithms are simply non-viable as they require many *months or years* for the requisite graph traversal. The main cause for such a poor performance of these algorithms on massive graphs is the number of I/Os (transfer of data from/to the external memory) they incur.

Figure 1.1 displays the results of experiments with the commonly used BFS rou-

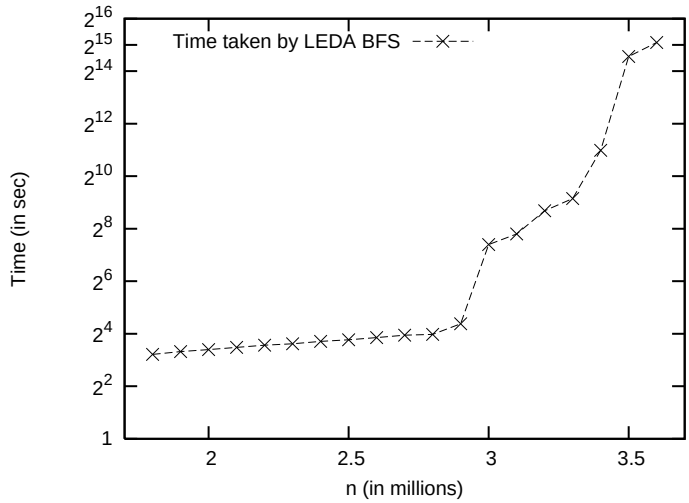



Figure 1.1: Time (in seconds) required by BFS from the LEDA graph package on random graphs with $m = 4n$ edges.

tine of the LEDA [107] graph package on random graph [66, 67] $G(n,m)$ with $m = 4n$. These experiments were done on a machine with Intel Xeon 2.0 GHz processor, 1 GB RAM and 2 GB swap space on a Seagate Baracuda hard-disk [142]. On random graphs with 3.6 million nodes (and 14.4 million edges), it takes around 10 *hours* as compared to just 10 *seconds* for graphs with 1.8 million nodes (and 7.2 million edges).

With the advent of solid state disks and other flash memory based storage mediums, the memory hierarchy is likely to become even more sophisticated as the read-write characteristics of these devices can be very different from the traditional hard disks. Since the storage devices significantly affect the practical performance of traversal algorithms when running on large graphs, we would like to exploit the I/O characteristics of these devices to design graph traversal algorithms that perform better in practice. For this, it is important to first properly characterize these disks.

Another important challenge when dealing with real world applications is that the input graphs are often dynamically changing. For instance, the World Wide Web graph, social networking graphs, purchase graphs, and scientific collaboration graphs are all continuously evolving. The telephone call graphs are also continuously changing. Even the street maps for route planning applications which may seem static most of the time are actually quite dynamic once the traffic jam and other road-block information is accounted in. The naïve way of recomputing all the information every time there is a minor modification in the original

graph is inefficient and for large and massive graphs, often impractical. Ideally, we would like to bound the amount of work needed to recompute the required traversal-related solution by a function of some measure of change done in the input graphs and the change in the output solution [132]. Since this is not always possible, our next best hope is to bound the required work in an amortized sense. In other words, while some updates may necessiate a lot of work, we would like to bound the sum of the time required for recomputing the solution over a sequence of graph updates.

While most of the algorithm design is done keeping the worst case complexity in mind, the worst case graphs for many of these algorithms are quite rare. Most application requirements are already met if an algorithm performs good on average.

In short, the simple graph traversal algorithms are often inappropriate for real applications involving massive graphs owing to the problems with the computation model, the noise and dynamicity of the input and the need for a different complexity measure (worst-case vs. average case).

Since most static algorithms analyzed for worst-case RAM complexity are impractical for massive graphs, one often relies on heuristics, pre-computations or exploiting special graph properties of underlying graphs. Such solutions are usually tailored for particular domains and are often application-specific. For each new application, one needs to design and implement different heuristics from the scratch. There is clearly a need for algorithms that will not only give nice theoretical guarantees for general graphs (without assuming any domain-specific knowledge), but also perform good in practise. This thesis focuses on the design, analysis and engineering of such algorithms.

## 1.3 Our contribution

The main contributions of this thesis are:

- We consider the problem of I/O-efficient Breadth-First Search (BFS) on massive sparse undirected graphs. We engineer the MR_BFS algorithm by Munagala and Ranade [115] into a practical implementation with low constant factors in the I/O complexity. Our pipelined implementation based on the external memory library STXXL can use multiple disks to further alleviate the I/O bottleneck. With this implementation, we are able to compute the BFS level decomposition of a web-crawl based graph of around 130 million nodes and 1.4 billion edges in less than 3 hours, using 4 disks.

We also engineer the $o(n)$-I/O MM_BFS algorithm [106] by Mehlhorn and Meyer. Our experiments suggest that while on small diameter graphs, MR_BFS performs quite well, MM_BFS performs significantly better on moderate to large diameter graphs. The usage of some heuristics further improves the running time of the faster variant of MM_BFS, while at the same time preserving the worst-case asymptotic I/O-complexity of MM_BFS. Demonstrating the viability of our BFS implementations [7, 9, 13] on various synthetic and real world benchmarks, we show that BFS level decompositions for large graphs (around a billion edges) can be computed on a cheap machine in a *few hours*, even if the underlying graph has large diameter.

We also present the design and engineering of simple I/O-efficient algorithms for generating large input graphs (of various graph classes) and a BFS decomposition verifier. As a part of our BFS implementations, we also look into the past engineering efforts on list ranking, Euler tour, minimum spanning forest and connected components, and adapt some of these implementations to the faster STXXL framework.

Furthermore, we compare the building blocks of our implementation with their corresponding cache-oblivious implementations and demonstrate that in the context of BFS on massive graphs, the cache-oblivious implementation is likely to be at least a factor of 4-5 slower than our implementation.

The key engineering ideas in our implementations also form the starting-point for implementing other I/O-efficient algorithms like Single-Source Shortest-Paths and Dynamic BFS. A significant chunk of our code is likely to be re-used for these implementations.

- Flash memory is fast becoming the dominant form of end-user storage in mobile computing. Since storage devices play a crucial role in the performance of (traversal) algorithms when the input (graph) data does not fit in the main memory, it is important to understand the I/O-characteristics of the storage devices to be able to predict the real running times of these algorithms. Such an understanding can also be exploited to design algorithms that are faster in practice. We characterize [10, 11] the performance of NAND flash based storage devices, including many solid state disks. We show that these devices have better random read performance than hard disks, but much worse random write performance. We also analyze the effect of misalignments, aging, past I/O patterns, etc. on the performance obtained on these devices. We show that despite the similarities between flash memory and RAM (fast random reads) and between flash disk and hard disk (both are block based devices), the algorithms designed in the

RAM model or the external memory model do not realize the full potential of the flash memory devices.

In the scenario when a solid state disk is used as an additional secondary storage rather than replacing the traditional hard disk, we engineer the I/O-efficient BFS implementation to exploit the comparative advantages of both the disks. We show that this is at least 25% faster than randomly striping the data on the two disks.

- We present a simple algorithm [8, 12] which maintains the topological order of a directed acyclic graph with $n$ nodes under an online edge insertion sequence in $O(n^{2.75})$ time, independent of the number $m$ of edges inserted. For dense DAGs, this is an improvement over the previous best result of $O(\min\{m^{\frac{3}{2}}\log n, m^{\frac{3}{2}} + n^2 \log n\})$ by Katriel and Bodlaender [91]. While our analysis holds only for the incremental setting, our algorithm itself is fully dynamic.

  We also provide an empirical comparison of our algorithm with other algorithms for dynamic topological sorting.

  The externalization of our algorithm provides interesting new results for dynamic topological ordering in external memory.

  We also present the first average-case analysis [5, 6] of online topological ordering algorithms. We prove an expected runtime of $O(n^2 \operatorname{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for the algorithms of Alpern et al. [15], Katriel and Bodlaender [91], and Pearce and Kelly [124].

## 1.4 Organization of the thesis

The rest of this thesis is organized as follows: Chapter 2 formally defines a graph and various notations used in the remaining chapters. It also shows the various computation models used to capture memory hierarchy and presents the basic tools and techniques for the design and engineering of I/O-efficient algorithms. Chapter 3 presents our work in engineering the I/O-efficient BFS algorithms. We also describe the related design and engineering of I/O-efficient algorithms for list ranking, Euler tour, directed breadth-first search, depth-first search, and topological ordering, and undirected connected components, minimum spanning forest, single-source shortest paths, dynamic BFS, and diameter approximation.

In Chapter 4, we show the characterization of flash memory devices including solid state disks. We also describe our efforts for tuning our I/O-efficient BFS algorithms to handle the case when the computing machine uses both the traditional hard disks as well as solid state disks for storage.

In Chapter 5, we present our $O(n^{2.75})$ algorithm for online topological ordering. We also show some open problems that can help tighten the analysis of our algorithm. Also, we show how to externalize our algorithm to obtain interesting new results on dynamic topological ordering in external memory. Furthermore, we present our results for the average-case analysis of the online topological ordering. We show that the algorithms by Alpern et al. [15], Katriel and Bodlaender [91], and Pearce and Kelly [124] require an expected runtime of $O(n^2 \cdot polylog(n))$ for maintaining the topological ordering, when edges of a complete DAG are inserted in a random order. We also briefly describe some recent advances in improving our bounds for this problem.

# Chapter 2

# Basic tools and techniques

*Intelligence is the faculty of making artificial objects, especially tools to make tools.*

— Henri Bergson

We start this chapter (Section 2.1) by giving the formal definitions and notations used in the remaining chapters. Section 2.2 provides some basic facts about probability theory and Section 2.3 presents some random graph models. Section 2.4 describes the real architecture and various computational models used to capture the memory hierarchy and Sections 2.5 and 2.6 present the tools and techniques used in the design and engineering of I/O-efficient algorithms. A reader familiar with the standard graph terminology, basic probability theory, random graph models, and the computation models capturing memory hierarchies may wish to skip sections 2.1, 2.2, 2.3 and 2.4, respectively.

## 2.1 Preliminary definitions

Formally, a graph $G$ is an ordered pair of disjoint sets $(V,E)$ such that $E$ is a subset of the set of unordered pairs of $V$. In this manuscript, we only consider finite graphs, that is $V$ and $E$ are always finite (though they are often very large). The set $V$ is the set of *vertices* and $E$ is the set of *edges*. If $G$ is a graph then

$V = V(G)$ is the vertex set of $G$ and $E = E(G)$ is the edge set. An edge $\{x,y\}$ is said to *join* the vertices $x$ and $y$. The vertices $x$ and $y$ are the *end-vertices* of this edge. If $\{x,y\} \in E$, then $x$ and $y$ are *adjacent* or *neighboring* vertices of $G$ and the vertices $x$ and $y$ are *incident* with the edge $\{x,y\}$. Two edges are *adjacent* if they have exactly one common end-vertex. We also use the notation $G(V,E)$ to refer to a graph $G = (V,E)$ and $G(V,E,w(\cdot))$ to refer to a weighted graph $G = (V,E)$, where each edge $e := \{x,y\} \in E$ is associated with a weight $w(e)$ (or $w(x,y)$).

The set of neighbors of a vertex $v$ in $G$ is denoted by $N_G(v)$, or briefly by $N(v)$. More generally for $U \subseteq V$, the neighbors in $V \setminus U$ of vertices in $U$ are called neighbors of $U$; their set is denoted by $N(U)$. The degree $d(v)$ of a vertex $v$ is the number $|E(v)|$ of edges at $v$; this is equal to the number of neighbors of $v$. A vertex of degree 0 is *isolated*. The number $\delta(G) := \min\{d(v)|v \in V\}$ is the *minimum degree* of $G$, the number $\Delta(G) := \max\{d(v)|v \in V\}$ denotes its *maximum degree*. The number $d(G) := \frac{1}{|V|}\sum_{v\in V} d(v) = \frac{2|E|}{|V|}$ is the *average degree* of $G$. Clearly, $\delta(G) \leq d(G) \leq \Delta(G)$.

An *independent set* in $G = (V,E)$ is a set of nodes $V' \subseteq V$ such that if $u,v \in V'$, $\{u,v\} \notin E$ i.e., no two nodes of $V'$ are adjacent in $G$. A *maximal independent set* is an independent set which is not contained in any larger independent set.

We say that $G' = (V',E')$ is a *subgraph* of $G = (V,E)$ if $V' \subseteq V$ and $E' \subseteq E$. In this case, we write $G' \subseteq G$. If $G'$ contains all edges of $G$ that join two vertices in $V'$ then $G'$ is said to be the subgraph induced by $V'$ and is denoted by $G[V']$. A subgraph $G'$ of $G$ is an *induced subgraph* if $G' = G[V(G')]$. $G' \subseteq G$ is a *spanning subgraph* of $G$ if $V'$ spans all of $G$, i.e. if $V' = V$. We say $G'$ *spans* $G$.

A *self-loop* is an edge that connects a vertex to itself. A *simple graph* is an undirected graph that has no self-loops and no more than one edge between any two different vertices. In this thesis, we will only be dealing with simple graphs. A complete graph is a simple graph in which every pair of distinct vertices is connected by an edge. An empty graph on $n$ nodes consists of $n$ isolated nodes with no edges.

A path $P$ from $u$ to $w$ in a graph $G$ is a node sequence $(v_0, v_1, \ldots, v_k)$ for some $k \geq 1$, such that the edges $\{v_0,v_1\},\{v_1,v_2\},\ldots,\{v_{k-1},v_k\}$ are part of $E$, $v_0 = u$, and $v_k = w$. If all nodes $v_i$ on $P$ are pairwise distinct then we say that the path is *simple*. Cycles are those paths where the starting point and the endpoint are identical. The *weight* of a path $P = (v_0,\ldots,v_k)$ from $u$ to $v$ in a weighted graph $G(V,E,w(\cdot))$ is defined to be $\sum_{i=0}^{k-1} w(v_i,v_{i+1})$.

A non-empty graph $G$ is called *connected* if any two of its vertices are linked

by a path in $G$. A maximal connected subgraph of $G$ is called a *component* or a *connected component* of $G$. An acyclic graph, one not containing any cycles, is called a *forest*. A connected forest is called a *tree*. The vertices of degree 1 in a tree are its *leaves*. The weight of a forest (tree) is defined to be the sum of the weights of all the edges in the forest (tree). A forest $F$ (tree $T$) that spans $G$ is a *spanning forest* (*spanning tree*) of $G$. A spanning forest (spanning tree) of minimum weight is called *minimum spanning forest* (*minimum spanning tree*).

The *distance* $d(x,y)$ in $G$ (also referred as $d_G(x,y)$) of two vertices $x,y$ is the minimum weight of a path from $x$ to $y$ in $G$; if no such path exists, we set $d(x,y) := \infty$. The greatest distance between any two vertices in $G$ is the *diameter* of $G$, denoted by $\mathrm{diam}(G)$. Sometimes it is convenient to consider one vertex of a tree as special; such a vertex is then called a *root* of this tree. A tree with a fixed root is a *rooted tree*.

An edge set $E$ of a *directed* graph consists of ordered pairs of nodes: an edge $e$ from node $u$ to node $v$ is denoted by $e = (u,v)$. Here $u$ is also called the *tail*, $v$ the *head*, and both nodes are called *endpoints* of $(u,v)$. Furthermore, $(u,v)$ is referred to as one of $u$'s *outgoing* edges or one of $v$'s *incoming* edges, as an edge *leaving* $u$ or an edge *entering* $v$. The number of edges leaving (entering) a node is called the *out-degree* (*in-degree*) of this node. The *degree* of a node is the sum of its in-degree and out-degree.

A path $P$ from $u$ to $w$ in a directed graph $G$ is a node sequence $(v_0, v_1, \ldots, v_k)$ for some $k \geq 1$, such that the edges $(v_0,v_1),(v_1,v_2),\ldots,(v_{k-1},v_k)$ are part of $E$, $v_0 = u$, and $v_k = w$. The nodes $v_0$ and $v_k$ are called the starting point and endpoint of $P$, respectively. If all nodes $v_i$ on $P$ are pairwise distinct then we say that the path is *simple*. Cycles are those paths where the starting point and the endpoint are identical. A graph is called *acyclic* if it does not contain any directed cycle.

A linear order is a relation that is reflexive, transitive, antisymmetric, and total. A topological order $T$ of a directed graph $G(V,E)$ is a linear ordering of its nodes such that for all directed paths from $x \in V$ to $y \in V$ $(x \neq y)$, it holds that $T(x) < T(y)$. A directed graph has a topological ordering if and only if it is acyclic.

A *walk* is an alternating sequence of vertices and edges, beginning and ending with a vertex, in which each vertex is incident to the two edges that precede and follow it in the sequence, and the vertices that precede and follow an edge are the end vertices of that edge. A walk is *closed* if its first and last vertices are the same, and *open* if they are different.

A *trail* is a walk in which all the edges are distinct. A closed trail is called a *tour*

or a *circuit*. *Euler tour* is a tour which contains all the edges exactly once. A graph that contains an Euler tour is an *Eulerian graph*.

*Graph traversal* refers to the problem of visiting all the nodes in a graph in a particular (structured) manner. Popular examples of graph traversal are Breadth-First Search, Depth-First Search, A*, and Dijkstra's algorithm. *Tree traversal* is a special case of graph traversal. Examples of tree traversal include pre-order, post-order, and in-order traversal. A pre-order traversal visits all nodes of a tree by processing the root, then recursively processing all subtrees rooted at its children from left to right. A post-order traversal first recursively processes all subtrees from left to right and then processes the node. An in-order traversal on binary trees first processes the left subtree, then the root and finally the right subtree.

## 2.2 Basic probability theory

In this section we review a few basic definitions and facts for the probabilistic analysis of algorithms.

The *sample space*, often denoted $\Omega$ of an experiment or random trial is the set of all possible outcomes. Any subset $\varepsilon \subseteq \Omega$ of the sample space is usually called an *event*. A *probability measure* $P$ is a function that satisfies the following three conditions: $0 \leq P[\varepsilon] \leq 1$ for each $\varepsilon \subseteq \Omega$, $P[\Omega] = 1$, and $P[\cup_i \varepsilon_i] = \sum_i P[\varepsilon_i]$ for pairwise disjoint events $\varepsilon_i$. A sample space together with its probability measure build a *probability space*. For a problem of size $n$, we say that an event $\varepsilon$ occurs *with high probability (w.h.p.)* if $P[\varepsilon] \geq 1 - O(n^{-\alpha})$ for an arbitrary but fixed constant $\alpha \geq 1$. The *conditional probability* $P[\varepsilon_1|\varepsilon_2] = \frac{P[\varepsilon_1 \cap \varepsilon_2]}{P[\varepsilon_2]}$ refers to the probability of an event $\varepsilon_1$ to occur when we already know that another event $\varepsilon_2$ happens. Two events $\varepsilon_1$ and $\varepsilon_2$ are called *independent* if $P[\varepsilon_1|\varepsilon_2] = P[\varepsilon_1]$.

Any real valued numerical function $X = X(\Omega)$ defined on a sample space $\Omega$ may be called a *random variable*. If $X$ maps elements in $\Omega$ to $\mathbb{R}_+ \cup \{0\}$ then it is called a *nonnegative* random variable. A *discrete* random variable only takes isolated values with nonzero probability. Typical representatives for discrete random variables are *binary* random variables, which map elements in $\Omega$ to $\{0,1\}$. Two random variables $X$ and $Y$ are called *independent* if, for all $x, y \in \mathbb{R}$, $P[X = x|Y = y] = P[X = x]$.

The *expectation* of a discrete random variable $X$ is given by $E[X] = \sum_{x \in \mathbb{R}} x \cdot P[X = x]$. Here are a few important properties of the expectation for arbitrary random variables $X$ and $Y$:

- If $X$ is nonnegative, then $E[X] \geq 0$.

- $|E[X]| \leq E[|X|]$.
- $E[c \cdot X] = c \cdot E[X]$ for any $c \in \mathbb{R}$.
- $E[X+Y] = E[X] + E[Y]$ (Linearity of expectation).
- If $X$ and $Y$ are independent, then $E[X \cdot Y] = E[X] \cdot E[Y]$.

Frequently, we are interested in the probability that random variables do not deviate too much from their expected values. The *Markov Inequality* for an arbitrary nonnegative random variable $X$ states that $P[X \geq k] \leq \frac{E[X]}{k}$ for any $k > 0$. The *Chebyshev Inequality* states that if a random variable $X$ has an expected value $\mu$ and finite variance $\sigma^2$, then for any real number $k > 0$,

$$P[|X - \mu| \geq k \cdot \sigma] \leq \frac{1}{k^2}.$$

In our average case analysis of online topological ordering algorithms, we will use an alternative formulation of this inequality:

$$P[|X - \mu| \geq v] \leq \frac{\sigma^2}{v^2}.$$

More powerful tail estimates exist for the sum of independent random variables. Here is one version of the well-known *Chernoff bound*: Let $X_1, \ldots, X_k$ be independent binary random variables and $\mu = E[\sum_{j=1}^{k} X_j]$. Then it holds for all $\delta > 0$ that

$$P[\sum_{j=1}^{k} X_j \geq (1+\delta) \cdot \mu] \leq e^{-\min\{\delta^2, \delta\} \cdot \mu/3}.$$

Furthermore, it holds for all $0 < \delta < 1$ that

$$P[\sum_{j=1}^{k} X_j \leq (1-\delta) \cdot \mu] \leq e^{-\delta^2 \cdot \mu/2}.$$

## 2.3 Random Graph Model

Random graph models are important tools for the average-case analysis of graph traversal algorithms. Furthermore, since most real-world phenomenon have a random component, many important properties of real-world graphs are similar to those of random graphs. For instance, our experiments suggests that the performance of various external memory BFS algorithms on webgraphs is similar to that on random graphs.

Erdős and Rényi [66, 67] introduced and popularized random graphs. They defined two closely related models: $G(n,p)$ and $G(n,m)$. The $G(n,p)$ model ($0 < p < 1$) consists of a graph with $n$ nodes in which each edge is chosen independently with probability $p$. On the other hand, the $G(n,m)$ model assigns equal probability to all graphs with $n$ nodes and exactly $m$ edges. Each such graph occurs with a probability of $1/\binom{N}{m}$, where $N := \binom{n}{2}$.

For our study of online topological ordering algorithms, we use the random DAG model of Barak and Erdős [26]. They obtain a random DAG by directing the edges of an undirected random graph from lower to higher indexed vertices. Depending on the underlying random graph model, this defines the $DAG(n,p)$ and $DAG(n,M)$ model.

The set of all DAGs with $n$ nodes is denoted by $DAG^n$. For a random variable $f$ with probability space $DAG^n$, $\mathbf{E}_M[f]$ and $\mathbf{E}_p[f]$ denotes the expected value in the $DAG(n,M)$ and $DAG(n,p)$ model, respectively.

The following theorem shows that in most investigations the models $G(n,p)$ and $G(n,m)$ are practically interchangeable, provided $m$ is close to $p \cdot N$.

**Theorem 1** *Given a function $f\colon G^n \to [0,a]$ with $a > 0$ and $f(G) \leq f(H)$ for all $G \subseteq H$ and functions $p$ and $m$ of $n$ with $0 < p < 1$, $q := 1-p$, $N := \binom{n}{2}$, and $m \in \mathbb{N}$,*

1. *If* $\lim\limits_{n\to\infty} pqN = \lim\limits_{n\to\infty} \dfrac{pN-m}{\sqrt{pqN}} = \infty$, *then* $\mathbf{E}_M[f] \leq \mathbf{E}_p[f] + o(1)$.
2. *If* $\lim\limits_{n\to\infty} pqN = \lim\limits_{n\to\infty} \dfrac{m-pN}{\sqrt{pqN}} = \infty$, *then* $\mathbf{E}_p[f] \leq \mathbf{E}_M[f] + o(1)$.

A closer look at the proof for it given by Bollobás [33] reveals that the probabilistic argument used to show the close connection between $G(n,p)$ and $G(n,M)$ can be applied in the same manner for the two random DAG models $DAG(n,p)$ and $DAG(n,M)$.

**Theorem 2** *Given a function $f\colon DAG^n \to [0,a]$ with $a > 0$ and $f(G) \leq f(H)$ for all $G \subseteq H$ and functions $p$ and $m$ of $n$ with $0 < p < 1$, $q := 1-p$, $N := \binom{n}{2}$, and $m \in \mathbb{N}$,*

1. *If* $\lim\limits_{n\to\infty} pqN = \lim\limits_{n\to\infty} \dfrac{pN-m}{\sqrt{pqN}} = \infty$, *then* $\mathbf{E}_M[f] \leq \mathbf{E}_p[f] + o(1)$.
2. *If* $\lim\limits_{n\to\infty} pqN = \lim\limits_{n\to\infty} \dfrac{m-pN}{\sqrt{pqN}} = \infty$, *then* $\mathbf{E}_p[f] \leq \mathbf{E}_M[f] + o(1)$.

## 2.4 Computation models capturing memory hierarchies

We start this section by describing the RAM model which is one of the most popular computation models for designing algorithms.

### 2.4.1 RAM model or von Neumann model

The running time of an algorithm is traditionally analyzed by counting the number of executed primitive operations or "instructions" as a function of the input size $n$. The implicit underlying model of computation is the one-processor, random-access machine (*RAM*) model. The RAM model or the "von Neumann model of computation" consists of a computing device attached to a storage device (or "memory"). The following are the key assumptions of this model:

- Every instruction takes the same amount of time, at least up to small constant factors.
- Unbounded amount of available memory.
- Memory stores words of size $O(\log n)$ bits where $n$ is the input size.
- Any desired memory location can be accessed in unit time.

The above assumptions greatly simplify the analysis of algorithms and allow for expressive asymptotic analysis.

### 2.4.2 Real Architecture

Unfortunately, modern computer architecture is not as simple. Rather than having an unbounded amount of unit-cost access memory, we have a hierarchy of storage devices (Figure 2.1) with very different access times and storage capacities. Modern computers have a microprocessor attached to a file of *registers*. The *first level (L1) cache* is usually only a few kilobytes large and incurs a delay of a few clock cycles. Often there are separate L1 caches for instructions and data. Nowadays, typical *second level (L2) cache* has a size of about 32-64 KB and access latencies around ten clock cycles. Some processors also have a rather expensive *third level (L3) cache* of up to 256 MB made of fast static random access memory cells. A

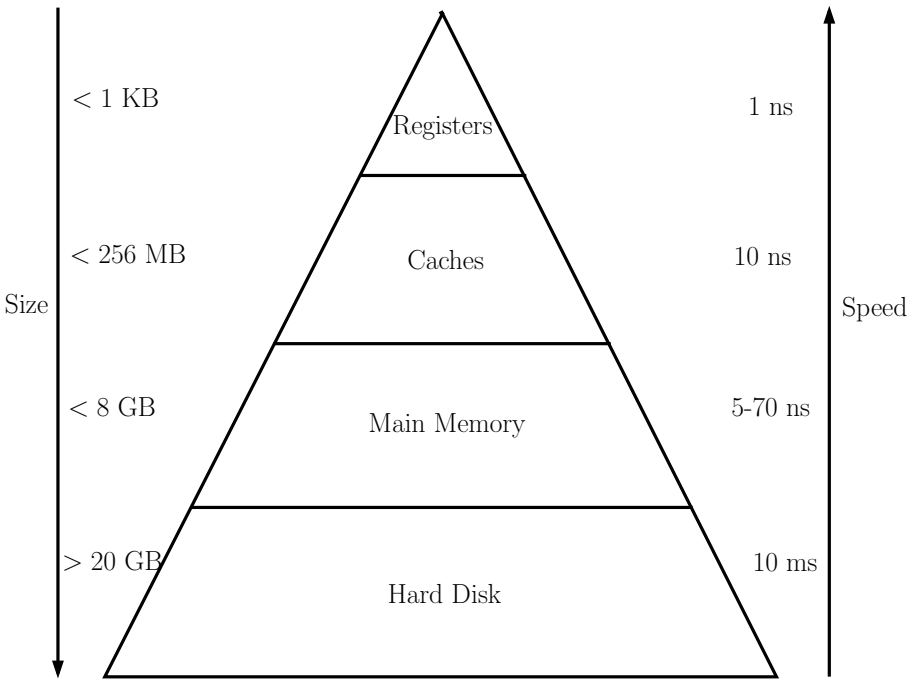



Figure 2.1: Memory Hierarchy in modern computer architecture.

cache consists of *cache lines* that each store a number of memory words. If an accessed item is not in the cache, it and its neighbor entries are fetched from the main memory and put into a cache line. These caches usually have limited associativity, i. e. an element brought from the main memory can be placed only in a restricted set of cache lines. In a *direct-mapped* cache the target cache line is fixed and only based on the memory address, whereas in a *full-associative* cache the item can be placed anywhere. Since the former is too restrictive and the latter is expensive to build and manage, a compromise often used is a *set-associative* cache. There, the item's memory address determines a fixed set of cache lines into which the data can be mapped, though within each set, any cache line can be used. The typical size of such a set of cache lines is a power of 2 in the range from 2 to 16. For more details about the structure of caches the interested reader is referred to [122] (in particular its Chapter 7).

The *main memory* is made out of dynamic random access memory cells. These cells store a bit of data as a charge in a capacitor rather than storing it as the state of a flip-flop which is the case for most static random access memory cells. It requires practically the same amount of time to access any piece of data stored in the main memory, irrespective of its location, as there is no physical movement (e. g. of a reading head) involved in the process of retrieving data. Main memory

is usually volatile, which means that it loses all data when the computer is powered down. At the time of writing this thesis, the main memory size is usually between 512 MB and 8 GB and a typical RAM memory has an access time of 5 to 70 nanoseconds.

Magnetic *hard disks* offer cheap non-volatile memory with an access time of 10 ms, which is $10^6$ times slower than a register access. This is because it takes very long to move the access head to a particular track of the disk and wait until the disk rotates into the seeked position. However, once the head starts reading or writing, data can be transfered at the rate of 35-105 MB/s [80]. Hence, reading or writing a contiguous block of hundreds of KB takes only about twice as long as accessing a single byte, thereby making it imperative to process data in large chunks.

Apart from the above mentioned levels of a memory hierarchy, there are instruction pipelines, an instruction cache, logical/physical pages, the translation look-aside buffer (TLB), magnetic tapes, optical disks and the network, which further complicate the architecture.

The reasons for such a memory hierarchy are mainly economical. The faster memory technologies are costlier and, as a result, fast memories with large capacities are economically prohibitive. The memory hierarchy emerges as a reasonable compromise between the performance and the cost of a machine.

### Disadvantages of the RAM Model

The beauty of the RAM model lies in the fact that it hides all the messy details of computer architecture from the algorithm designer and at the same time, it encapsulates the comparative performance of algorithms remarkably well. It strikes a fine balance by capturing the essential behavior of computers while being simple to work with. The performance guarantees in the RAM model are not architecture-specific and therefore robust. However, this is also the limiting factor for the success of this model. In particular, it fails significantly when the input data or the intermediate data structure is too large to reside completely within the internal memory.

For most (traversal) problems on large (graph) data sets, the dominant part of the running time of algorithms is not the number of “instructions”, but the time these algorithms spend waiting for the data to be brought from the hard disk to internal memory. The I/Os or the movement of data between the memory hierarchies (and in particular between the main memory and the disk) are not captured by the RAM

model and hence, the predicted performance on the RAM model may increasingly deviate from the actual performance.

### Future Trends

The problem is likely to aggravate in the future. In following with the Moore's law, the number of transistors double every 18 months. As a result, the CPU speed continues to improve at nearly the same pace, i.e., an average performance improvement of 1% per week. Besides, the usage of parallel processors and multi-cores makes the computations even faster. On the other hand, random access memory speeds and hard drive seek times improve at best a few percentages per year. Although the capacity of the random access memory doubles about every two years, users double their data storage every 5 months. The Internet applications like social networks and e-commerce companies (cf. Section 1.1) are also extending their user and product base at a very fast pace.

## 2.4.3 External Memory Model

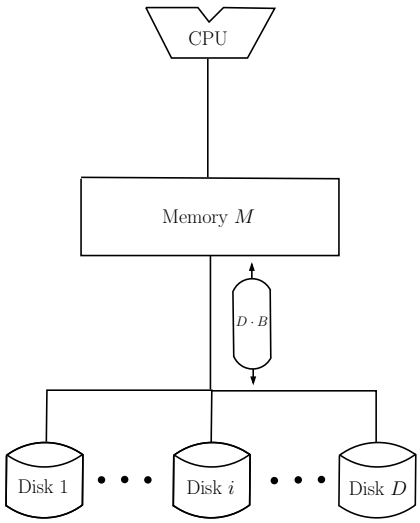



Figure 2.2: The external memory model

The I/O model or the external memory (*EM*) model (depicted in Figure 2.2) as introduced by Aggarwal and Vitter [3] assumes a single central processing unit and two levels of memory hierarchy. The internal memory is fast, but has a limited size of $M$ words. In addition, we have an external memory which can only be accessed

using I/Os that move $B$ contiguous words between internal and external memory. For graph traversal problems, the notation is slightly altered: we assume that the internal memory can have up to $M$ *data items of a constant size* (e.g., vertices or edges), and in one I/O operation, $B$ contiguous data items move between the two memories. At any particular time, the computation can only use the data already present in the internal memory. The measure of performance of an algorithm is the number of I/Os it performs. An algorithm $A$ is better than another algorithm $A'$ if $A$ requires less I/Os than $A'$.

Although we mostly use the sequential variant of the external memory model, it also has an option to express disk parallelism. There can be $D$ parallel disks and in one I/O, $D$ arbitrary blocks can be accessed in parallel from the disks. The usage of parallel disks helps us alleviate the I/O bottleneck.

### 2.4.4 Parallel Disk Model

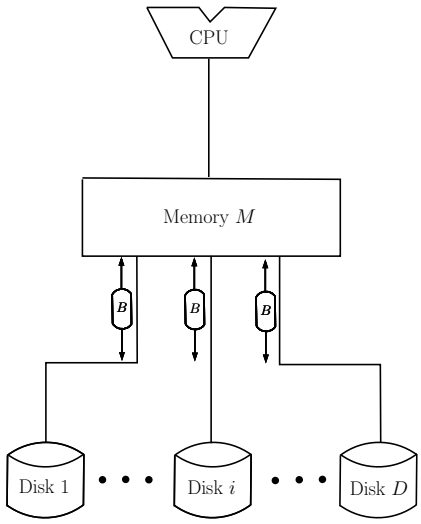



Figure 2.3: Parallel Disk Model

The parallel disk model (depicted in Figure 2.3) by Vitter and Shriver [152] is similar to the external memory model, except that it adds a realistic restriction that only one block can be accessed per disk during an I/O, rather than allowing $D$ *arbitrary* blocks to be accessed in parallel. The parallel disk model can also be extended to allow parallel processing by allowing $P$ parallel identical processors each with $M/P$ internal memory and equipped with $D/P$ disks.

Sanders et al. [140] gave efficient randomized algorithms for emulating the external memory model of Aggarwal and Vitter [3] on the parallel disk model.

### 2.4.5 Ideal Cache Model

In the external memory model we are free to choose any two levels of the memory hierarchy as internal and external memory. For this reason, external memory algorithms are sometimes also referred to as cache-aware algorithms ("aware" as opposed to "oblivious"). There are two main problems with extending this model to caches: limited associativity and automated replacement. As shown by Sen and Chatterjee [143], the problem of limited associativity in caches can be circumvented at the cost of constant factors. Frigo et al. [73] showed that a regular algorithm causes asymptotically the same number of cache misses with LRU or FIFO replacement policy as with optimal off-line replacement strategy. Intuitively, an algorithm is called regular if the number of incurred cache misses (with an optimal off-line replacement) increases by a constant factor when the cache size is reduced to half.

Similar to the external memory model, the ideal cache model assumes a two level memory hierarchy, with the faster level having a capacity of storing at most $M$ elements and data transfers in chunks of $B$ elements. In addition, it also assumes that the memory is managed automatically by an optimal off-line cache-replacement strategy, and that the cache is fully associative.

### 2.4.6 Cache-Oblivious Model

In practice, the model parameters $B$ and $M$ need to be finely tuned for an optimal performance. For different architectures and memory hierarchies, these values can be very different. This fine-tuning can be at times quite cumbersome. Besides, we can optimize only one memory hierarchy level at a time. Ideally, we would like a model that would capture the essence of the memory hierarchy without knowing its specifics, i.e. values of $B$ and $M$, and at the same time be efficient on all hierarchy levels simultaneously. Yet, it should be simple enough for a feasible algorithm analysis. The cache oblivious model introduced by Frigo et al. [73] promises all of the above. In fact, the immense popularity of this model lies in its innate simplicity and its ability to abstract away the hardware parameters.

The cache-oblivious model also assumes a two level memory hierarchy with an internal memory of size $M$ and block transfers of $B$ elements in one I/O. The performance measure is the number of I/Os incurred by the algorithm. However, the algorithm does not have any knowledge of the values of $M$ and $B$. Consequently, the guarantees on I/O-efficient algorithms in the cache-oblivious model hold not only on any machine with multi-level memory hierarchy but also on all levels of

the memory hierarchy at the same time. In principle, they are expected to perform well on different architectures without the need of any machine-specific optimization.

The cache-oblivious model assumes full associativity and optimal replacement policy. However, as we argued for the ideal cache model (cf. Section 2.4.5), these assumptions do not affect the asymptotics on realistic caches.

However, note that cache-oblivious algorithms are usually more complicated than their cache-aware I/O-efficient counterparts. As a result, the constant factors hidden in the complexity of cache-oblivious algorithms are usually higher and on large external memory inputs, they are slower in practice.

### 2.4.7 Various streaming models

In the data stream model [116], input data can only be accessed sequentially in the form of a data stream, and needs to be processed using a working memory that is small compared to the length of the stream. The main parameters of the model are the number $p$ of sequential passes over the data and the size $s$ of the working memory (in bits). Since the classical data stream model is too restrictive for graph algorithms and even the undirected connectivity requires $s \times p = \Omega(n)$ [82] (where $n$ is the number of nodes in a graph), less restrictive variants of streaming models have also been studied. These include stream-sort model [4] where sorting is also allowed, W-stream model [58] where one can use intermediate temporary streams and semi-streaming model [68], where the available memory is $O(n \cdot polylog(n))$ bits.

### 2.4.8 Other memory hierarchy models

Recently, Arge et al. [23] have proposed Parallel External-Memory model as a natural parallel extension of the external-memory model of Aggarwal and Vitter [3], to private-cache chip multiprocessors.

There are still a number of issues not addressed by these models that can be critical for performance in practical settings, e. g. branch mispredictions [87], TLB misses etc. For other models on memory hierarchies, refer to [4, 25, 94, 114, 131].

## 2.5 Basic tools for designing external memory graph traversal algorithms

Many different tools and techniques have been developed for graph algorithms in external memory in the last couple of decades. In this Section, we describe some of the commonly used building blocks for the design of I/O-efficient graph traversal algorithms.

### 2.5.1 Parallel scanning

Scanning many different streams (of data from the disk) simultaneously is one of the most basic tools used in I/O-efficient algorithms. This can be used, for example, to copy some information from one stream to the other. Sometimes, different streams represent different sorted sets and parallel scanning can be used to compute various operations on these sets such as union, intersection, or difference.

Given $k = O(M/B)$ streams containing a total of $O(n)$ elements, we can scan them "in parallel" in $\text{scan}(n) = O(n/B + k)$ I/Os. This is done by simply keeping $O(1)$ blocks of each stream in the internal memory. When we need a block not present in the internal memory, we remove (or write back to the disk) the existing block from the corresponding stream and load the required block from the disk.

### 2.5.2 Sorting

Sorting is fundamental to many I/O-efficient graph traversal algorithms. In particular, sorting can be used to rearrange the nodes on the disk so that a graph traversal algorithm does not have to spend $\Omega(1)$ I/Os for loading the adjacency list of each node into the internal memory.

Sorting $n$ elements in the external memory requires $\text{sort}(n) = \Theta(\frac{n}{B} \log_{\frac{M}{B}} \frac{n}{B})$ I/Os [3]. There exist many different algorithms for I/O-efficient sorting. The most commonly used external memory sorting algorithm is based on $(M/B)$-way merge sort. It first scans through the input data, loading $M$ elements at a time, sorting them internally and writing them back to disk. In the next round, we treat each of these chunks as a stream and merge $O(M/B)$ streams at a time using "parallel scanning" to produce sorted chunks of size $O(M^2/B)$. By repeating this process for $O(\log_{\frac{M}{B}} \frac{n}{B})$ rounds, we get all the elements sorted.

External memory libraries such as STXXL [56, 57] and TPIE [21] provide fast implementations of external memory sorting routines. STXXL also has specialized functions for sorting elements with integer keys and sorting streams.

In the cache-oblivious setting, funnel-sort [73] and lazy funnel-sort [39], also based on a merging framework, lead to sorting algorithms with the same I/O complexity of $\Theta(\frac{n}{B}\log_{\frac{M}{B}}\frac{n}{B})$ I/Os. Brodal et al. [41] show that a careful implementation of this algorithm outperforms several widely used library implementations of quick-sort on uniformly distributed data. For the largest instances in the RAM, this implementation outperforms its nearest rival std::sort from the STL library included in GCC 3.2 by 10-40% on many different architectures like Pentium III, Athlon and Itanium 2.

### 2.5.3 PRAM simulation

A Parallel Random Access Machine (PRAM) is a basic model of computation that consists of a number of sequential processors, each with its own memory, working synchronously and communicating between themselves through a common shared memory.

Simulating a PRAM algorithm [48] on the external memory model is an important tool in the design of I/O-efficient graph algorithms. A PRAM algorithm that uses $p$ processors and $O(p)$ (shared memory) space and runs in time $T(p)$ can be simulated in $O(T(p) \cdot \text{sort}(p))$ I/Os.

Each step taken by a PRAM involves each processor independently reading a data element, computing on it and writing some output. In order to simulate it on the external memory model, the read requests of all the processors are sorted according to the location of the required data. Afterwards, one scan of the entire data of the shared memory is enough to fetch all the requisite data. This is then sorted back according to the processor ids. Thereafter, in one scan of the fetched data, we perform all the computations by all the processors and collect the output data (together with its location) that would have been produced by each processor. This is then sorted according to the memory location and written back to the disk. Thus, each step of the $O(p)$-processor PRAM algorithm requiring $O(p)$ space can be simulated by a constant number of sorts and scans, i.e., $O(\text{sort}(p))$ I/Os.

PRAM simulation is particularly appealing as it translates a large number of PRAM-algorithms into I/O-efficient and sometimes I/O-optimal algorithms.

Even without directly using the simulation, I/O-efficient algorithms can be obtained by appropriately translating PRAM algorithms, as many of the ideas ap-

plied in parallel computing for reducing a problem into many independent subproblems are also useful for designing external memory algorithms. For many problems, the bounds obtained by appropriately translating PRAM algorithms are much better than those obtained by direct simulation.

### 2.5.4 Algorithms on trees

Efficient external memory algorithms are known for many different problems on undirected trees. These include rooting a tree, computing pre-order, post-order or in-order traversal, computing the depth of each node, least common ancestor queries, etc. Most of these algorithms (e.g., the tree traversal algorithms in [48]) are efficient translations of their PRAM counterparts.

### 2.5.5 Priority queues

A priority queue is an abstract data structure that stores an ordered set of keys and allows efficient insertion, search of the minimum element (find_min) and deletion of the minimum element (delete_min). Sometimes operations such as deleting an arbitrary key and decreasing the value of the key are also supported. Priority queues are fundamental to many graph traversal algorithms, particularly for computing single-source shortest-paths.

One way of implementing efficient external memory priority queues is using buffer trees [17]. Buffer trees are useful for batched operations, i.e., when the answers to the queries are not required immediately but eventually.

A buffer tree has degree $\Theta(M/B)$. Each internal node is associated with a buffer containing a sequence of up to $\Theta(M)$ updates and queries to be performed in its subtree. Leaves contain $\Theta(B)$ keys. Updates and queries are simply performed by inserting the appropriate signal in the root node buffer. If the buffer is full, the signal buffer is flushed to its children. This process may need to be repeated all the way down to the leaves. Since flushing the buffer requires $\Theta(M/B)$ I/Os (which is done after inserting $\Theta(M)$ signals) and the tree has $O(\log_{M/B} n/B)$ levels, the amortized cost of the update and query operations is $O((1/B)\cdot\log_{M/B}(n/B))$ I/Os. It can be shown that the re-balancing operations for maintaining the tree can also be done within the same bounds.

In order to use buffer trees as a priority queue, the entire buffer of the root node together with the $O(M/B)$ leftmost leaves (all the leaves of the leftmost internal

node) is kept in internal memory. We maintain the invariant that all buffers on the path from the root to the leftmost leaf are empty. Thus, the element with the smallest priority always remains in internal memory. The invariant is maintained by flushing out all buffers in the leftmost path whenever the root buffer is flushed, at a total cost of $O((M/B) \cdot \log_{M/B}(n/B))$ I/Os. The amortized cost of updates and queries still remains $O((1/B) \cdot \log_{M/B}(n/B))$ I/Os.

Note that the buffer tree based priority queue can not efficiently perform a decrease_key of an element, if we do not know its old key. For efficient but lazy decrease_key operations, we can use tournament trees [93]. On an I/O-efficient tournament tree with $n$ elements, any sequence of $z$ operations each of them being either a delete, delete_min or an update, requires at most $O((z/B) \cdot \log_2(n/B))$ I/Os. The update operation referred here is a combined insert and decrease_key operation.

Cache-oblivious priority queues with amortized $O((1/B) \cdot \log_{M/B}(n/B))$ I/O insertion, deletion and delete_min operations have also been developed [20, 38]. The cache-oblivious bucket heap based priority queue [40] provides amortized $O((1/B) \cdot \log_2(n/B))$ update, delete and delete_min operations, where the update operation is similar to the one provided by tournament trees.

### 2.5.6 Time forward processing

Time forward processing [17, 48] is an elegant technique for solving problems that can be expressed as a traversal of a directed acyclic graph (DAG) from its sources to its sinks. Let $G$ be a DAG and $\phi(v)$ be a label associated with the node $v$. The goal is to compute another labelling $\psi(v)$ for all nodes $v \in G$, given that $\psi(v)$ can be computed from labels $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_k)$, where $u_1, \ldots, u_k$ are the in-neighbors of $v$.

Time forward processing on an $n$-node DAG can be solved in external memory in $O(\text{sort}(n))$ I/Os if the following conditions are met:

1. The nodes of $G$ are stored in topologically sorted order.
2. $\psi(v)$ can be computed from $\phi(v)$ and $\psi(u_1), \ldots, \psi(u_k)$ in $O(\text{sort}(k))$ I/Os.

This bound is achieved by processing the nodes in the topologically sorted order and letting each node pass its label $\psi$ to its out-neighbors using a priority queue. Each node $u$ inserts $\psi(u)$ in the priority queue for each out-neighbor $v$ with the key being the topological number of $v$, $T(v)$. We ensure that before we process $v$,

we extract all the nodes with priority $T(v)$ and therefore, get all the necessary information to compute $\psi(v)$.

### 2.5.7 Graph contraction

The key idea in graph contraction is to reduce the size of the input graph $G$ while preserving the properties of interest. Such a procedure is often applied recursively till either the number of edges or the number of nodes are reduced by a factor of $O(B)$ or the number of nodes is reduced to $O(M)$. In the first case, the algorithm can afford to spend $O(1)$ I/Os per remaining node to solve the problem. In the latter case, an efficient semi-external algorithm is used to solve the problem.

Graph contraction is particularly useful for problems like connected components and minimum spanning forests, where the connectivity information is preserved (see e.g. [18]) during the edge contraction steps.

### 2.5.8 Graph clustering

Clustering a graph refers to decomposing the graphs into disjoint clusters of nodes. Each cluster contains the adjacency lists of a few nodes. These nodes should be close in the original graph. Since each cluster is connected and small, if a node of the cluster is visited during BFS, SSSP or APSP, the other nodes of the cluster will also be visited "shortly". This fact can be exploited to design better algorithms (see e.g. [106], [112]) for these problems.

### 2.5.9 Ear decomposition

An ear decomposition $\varepsilon = (P_0, P_1, P_2, \ldots, P_k)$ of a graph $G = (V, E)$ is a partition of $E$ into an ordered collection of edge-disjoint simple paths $P_i$ with endpoints $s_i$ and $t_i$. Ear $P_0$ is an edge. For $1 \leq i \leq k$, ear $P_i$ shares its two endpoints $s_i$ and $t_i$, but none of its internal nodes, with the union $P_0 \cup \ldots P_{i-1}$ of all previous ears. A graph has an ear decomposition if and only if it is two-edge connected, i.e., removing any edge still leaves a connected subgraph.

An ear decomposition of a graph can be computed in $O(\text{sort}(n))$ I/Os in external memory [103].

## 2.6 Tools for engineering external memory graph traversal algorithms

In the last decade, many techniques have evolved for engineering external memory graph traversal algorithms. Libraries specifically containing fundamental algorithms and data structures for external memory have been developed. Techniques such as pipelining can save some constant factors from the I/O complexity of the external memory implementations, which can be significant for making the implementation viable. In this section, we describe some of these tools and techniques.

### 2.6.1 External memory libraries

External memory libraries play a crucial role in engineering algorithms running on large data-sets. These libraries not only reduce the development time for external memory algorithms, but also speed up the implementations themselves. The former is done by abstracting away the details of how an I/O is performed and providing ready-to-use building blocks including algorithms such as sorting and data structures such as priority queues. The latter is done by offering frameworks such as pipelining (described ahead in this section) that can reduce the constant factors in the I/O complexity of an implementation. Furthermore, the algorithms and data structures provided are optimized and perform less internal memory work.

#### STXXL

STXXL [56, 57] is an implementation of the C++ standard template library STL [147] for external memory computations. Since the data-structures and algorithms in STXXL have a well known generic interface similar to STL interface, it is easy to use and the existing applications based on STL can be easily made to work with STXXL. STXXL supports parallel disks, overlapping between disk I/O and computation and the *pipelining* technique that can save a significant fraction of the I/Os. It provides I/O-efficient implementations of various containers (stack, queue, deque, vector, priority queue, $B^+$-tree, etc.) and algorithms (scanning, sorting using parallel disks, etc.). It is being used both in academic and industrial environments for a range of problems including text processing, graph algorithms, computational geometry, Gaussian elimination, visualization, and analysis of microscopic images, differential cryptographic analysis, etc.

### TPIE

TPIE [21] or "Transparent Parallel I/O Environment" is another C++ template library supporting out-of-core computations. The goal of the TPIE project has been to provide a portable, extensible, flexible, and easy to use programming environment for efficiently implementing I/O-efficient algorithms and data structures. Apart from supporting algorithms with a sequential I/O pattern (i.e., algorithms using primitives such as scanning, sorting, merging, permuting and distributing) and basic data structures such as $B^+$-tree, it supports many more external memory data structures such as $(a,b)$-tree, persistent $B$-tree, $Bkd$-tree, $K$-$D$-$B$-tree, $R$-tree, $EPS$-tree, $CRB$-tree etc. It is used for many geometric and GIS implementations.

## 2.6.2 Pipelining

Conceptually, pipelining is a partitioning of the algorithm into practically independent parts that conform to a common interface, so that the data can be streamed from one part to the other without any intermediate external memory storage. This may reduce the constant factors in the I/O complexity of the algorithm. It leads to better structured implementations, as different parts of the pipeline only share a narrow common interface. On the other hand, it may also increase the computational costs as in a stream, searching an element can't be done by exponential or binary search, but by going through potentially all the elements in the stream. This means that the correct extent of pipelining needs to be carefully determined.

Usually, a pipelined code requires more debugging efforts and hence, significantly more development time. For more details on the usage of pipelining as a tool to save I/Os, refer to [55].

# Chapter 3

# Breadth first search on massive graphs

*Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it.*

–Brian W. Kernighan

Breadth-First Search (BFS) is an archetype for many important graph problems. Many real world problems involve BFS (and some of its generalizations like shortest paths or $A^*$) traversal on large graphs. These applications (cf. Section 1.1 for more details) include crawling and analyzing the WWW [118, 144], route planning using small navigation devices with flash memory cards [76], state space exploration [64], and community detection [119].

Given a large undirected graph $G(V,E)$ ($n := |V|, m := |E|$) and a source node $s$, the goal of BFS is to decompose the set of nodes $V$ into disjoint subsets called BFS levels, such that the level $i$ comprises of all nodes that can be reached from $s$ via $i$ edges, but no less. The problem of computing the BFS level decomposition can also be viewed as computing single source shortest paths on unweighted graphs.

BFS is well-understood in the RAM model. There exists a simple linear time algorithm [51] (hereafter referred as IM_BFS) for the BFS traversal in a graph. However, as discussed in Section 1.2, this algorithm (as implemented in LEDA) performs quite badly when the input graph does not fit in the main memory. Fur-

thermore, on massive graphs (with a billion or more edges), these algorithms are simply non-viable as they require many *months or years* for the requisite graph traversal.

External memory algorithms for computing BFS have therefore been studied. For general undirected graphs, Munagala and Ranade proposed a simple algorithm (MR_BFS) that incurs $O(n + \text{sort}(m))$ I/Os. Mehlhorn and Meyer proposed the first $o(n)$ I/O algorithm (MM_BFS) that improves the results for sparse graphs.

In this chapter, we focus on engineering these external memory BFS algorithms. Since most of the large real world graphs are sparse, we mainly concentrate on the problem of computing a BFS level decomposition for massive sparse undirected graphs. Demonstrating the viability of our BFS implementations on various synthetic and real world benchmarks, we show that BFS level decompositions for large graphs (around a billion edges) can be computed on a cheap machine in a *few hours*.

The rest of the chapter is organized as follows: We review some related work in Section 3.1. In Section 3.2, we describe the external memory algorithms for list ranking, computing Euler tours on trees, minimum spanning forests and connected components on general undirected graphs. These form the building blocks in the external memory BFS algorithms presented in Section 3.3. Sections 3.4 – 3.7 present our implementations of MR_BFS and MM_BFS. We also designed and engineered I/O-efficient frameworks for generating massive graphs and checking if the BFS decomposition is correct. These are discussed in Section 3.8 and Section 3.9, respectively. Section 3.10 describes the evolution of our BFS codes into a software package. Our detailed empirical study is presented in Section 3.11. Section 3.12 discusses the extensions of BFS to SSSP and dynamic BFS in external memory. It also describes the recent advances in approximating the diameter of the graph that can help us decide which BFS algorithm to use. Section 3.13 concludes with related open problems.

Note that in this chapter, the term "adjacency list" refers to the set of all adjacent edges of a node, and not to some list data structure containing this set.

## 3.1 Related prior work

External-memory BFS algorithms are known for special graphs classes like trees, grid graphs [19], planar graphs [101], outer-planar graphs [100], and graphs of bounded tree width [102]. These algorithms use special graph properties such as

planar separators, planar and outerplanar embeddings, and tree-decompositions. For graphs with small separators (not necessarily planar), we can represent the graph [30, 31] in a more compact way that minimizes the I/Os required by the standard algorithms.

Very little is known for traversing general directed graphs in external memory. The main result known in this direction is the $O((n+m/B)\log_2 \frac{n}{B} + \text{sort}(m))$ I/O algorithm [43] for computing Breadth-First Search (BFS), and Depth-First Search (DFS) on general directed graphs and topological ordering on general directed acyclic graphs. These algorithms crucially rely on a data structure called buffered repository tree [43] for removing edges leading to visited nodes.

### 3.1.1 Engineering Directed DFS in external memory

Owing to the $O(n\log_2 \frac{n}{B})$ term in the I/O complexity, these algorithms are considered impractical for general sparse directed graphs. Since real world graphs are usually sparse, it is unlikely that these algorithms will improve the running time significantly as compared to the internal memory traversal algorithms. As such, there has been no engineering attempt (up to the best of our knowledge) for these algorithms.

Sibeyn et al. [146] showed an implementation of semi-external DFS (i.e., computing DFS when $M \geq c \cdot n$ for some small constant $c$) based on the batched processing framework. We assume that the internal memory can contain up to $2n$ edges. We maintain a tentative DFS tree throughout the algorithm in the internal memory and proceed in rounds. In each round, all the edges of the graph are processed in cyclic order. A round consists of $m/n$ phases and in each phase we load a batch of $n$ edges and compute the DFS of the $2n$ edges in the internal memory. The DFS computation can be made faster by the following heuristics [146]:

- Rearrange the tree after every round so as to find the global DFS tree more rapidly. For each node, we visit its children (in the tree) in descending order of their sub-tree sizes. Thus, after rearrangement the leftmost child of any node has more descendants than any other child, thereby heuristically reducing the number of forward (left to right) cross edges.

- Reduce the number of nodes and edges "active" in any round so as to leave more space in the internal memory for loading new edges. Since nodes on the leftmost path are not going to change their place in the tree anymore (unless they are rearranged), they can be marked "passive" and removed

from consideration. Furthermore, we can mark all nodes $u$ that satisfy the following conditions passive:

- All nodes on the path from root node to $u$ are already marked passive.
- There is no edge from any node with smaller pre-order number (in the current tree) to any node with pre-order number equal to or larger than that of $u$.

Together with these heuristics, the batched processing framework manages to compute DFS on a variety of directed graphs (such as random graphs and 2-dimensional random geometric graphs) with very few (3–10) average accesses per edge (and hence few I/Os). It can compute strongly connected components (using the DFS) of an AT&T call graph with around 9.9 million nodes and 268.4 million edges in around 4 hours on a Pentium III machine with a 1 GHz processor.

### 3.1.2 Engineering external memory A*

A* [81] is a goal-directed graph traversal strategy that finds the least-cost path from a given source node to a target node. A* is similar to Dijkstra's famous shortest path algorithm [61], except that it visits the node with the minimum sum of distance from the source node and the heuristic distance to the target node rather than the node with the minimum distance from the source.

A* can be solved using external memory priority queues in $O(n+m/B\cdot\log_2(m/B))$ I/Os. For implicit unweighted graphs, a suitably modified version of the external memory BFS algorithm MR_BFS by Munagala and Ranade [115] (cf. Section 3.3 for more details) helps computing A* in $O(\text{sort}(m))$ I/Os [64]. This is because in implicit graphs accessing the adjacency list of a node does not require I/Os to fetch it from the disk, but only internal memory computation to generate it.

The practical performance of A* crucially depends on the heuristic estimate of the distance between target node and a given node. This estimate in turn is heavily application-dependent.

Edelkamp et al. [85] engineered the variant of external memory A* for implicit undirected unweighted graphs and used it for many different model checking applications. They improved the practical performance of external memory A* for their applications further by the following heuristics:

- Delayed duplicate detection: Unlike MR_BFS, duplicates are not removed

till the nodes are actually visited.

- The nodes with equal value of the sum of distances from the source and the target node, are visited in increasing order of their distance from the source node.

External A* as incorporated in the External SPIN model checker software was used to detect the optimal path to a deadlock situation in an Optical Telegraph protocol involving 14 stations. This problem required 3 Terabytes of hard disk space (with 3.6 GB RAM) and took around 8 days with 4 instances of Parallel External SPIN running on 4 AMD Opteron dual processor machines with NFS shared hard disk. In model checking applications involving a massive state space, finding such deadlocks can be critical for the correct performance of the protocol and hence, even running-times of weeks are considered acceptable.

However, this implementation as well as the heuristics used are specific to A* on implicit graphs and are unlikely to yield good results for BFS on general graphs.

## 3.2 Basic building blocks

In the RAM model, graph problems like connected components etc. can be efficiently solved by graph traversal strategies such as Depth-First Search (DFS) and Breadth-First Search (BFS). However, the picture is very different in the memory hierarchy models. Algorithms for connected components, minimum spanning tree, Euler tour and list ranking are asymptotically faster than the currently best ones for BFS and DFS. Hence, many algorithms for graph traversal strategies like BFS and DFS use connected components, minimum spanning forests, Euler tour and list ranking as sub-routines. In this section, we review the algorithms for these building blocks.

### 3.2.1 Euler tour of a bi-directional tree

An Euler tour of a (bi-directional) tree $T = (V, E)$ traverses every edge exactly twice, once in each direction. Such a traversal produces a linear list of edges or vertices capturing the structure of the tree. In order to compute such a tour, we choose an order of the edges $\{v, w_1\}, \ldots, \{v, w_k\}$ incident to each node $v$ of $T$. Then, we mark the successor of $\{w_i, v\}$ to be $\{v, w_{i+1}\}$ and the successor of $\{w_k, v\}$ to be $\{v, w_1\}$. We break the resulting circular list at some node $r$ by choosing an

edge $\{v,r\}$ with successor $\{r,w\}$, setting the successor of $\{v,r\}$ to be null, and choosing $\{r,w\}$ to be the first edge of the traversal.

An Euler tour of a (bi-directional) tree can be computed in $O(\text{sort}(n))$ I/Os.

### 3.2.2 List ranking

A list $L$ is a collection of elements $x_1,\ldots,x_n$ such that each element $x_i$, except the last element of the list, stores a pointer to its successor, no two elements have the same successor and every element can reach the last element by following successor pointers. Given a list $L$ of elements kept in an arbitrary order on the disk and a pointer to the first element and weights $w$ on all edges, the list ranking problem is that of computing for every element $x_i$, its distance from the first element.

The external memory list ranking algorithm [48] computes an independent set $I$ of size $\Omega(n)$. All elements $x_i \in I$ are removed from $L$ by marking $succ(x_i)$ as the successor of $pred(x_i)$, where $succ(x_i)$ and $pred(x_i)$ are the successor and predecessor of $x_i$ in $L$. The weight of the new edge $\{pred(x_i), succ(x_i)\}$ is the sum of the weights of $\{pred(x_i), x_i\}$ and $\{x_i, succ(x_i)\}$. The problem on the compressed list is recursively solved. For each node $x_i \in I$, its distance from the head is equal to the sum of the distance of $pred(x_i)$ (computed for the compressed list) and the weight of the edge $\{pred(x_i), x_i\}$. All operations for compressing the list incur $O(\text{sort}(n))$ I/Os and thus the total cost of list ranking is $I(n) = I(\alpha \cdot n) + O(\text{sort}(n)) = O(\text{sort}(n))$ I/Os, for some constant $0 < \alpha < 1$.

Note that any maximal independent set of a list has size at least $n/3$. Thus in order to compute the independent set $I$ of size $\Omega(n)$, we just need to compute a maximal independent set. A maximal independent set $I$ of a graph $G(V,E)$ can be computed simply by a greedy algorithm in which the nodes are processed in an arbitrary order. When a node $v \in V$ is visited, we add it to the set $I$ if none of its neighbors is already in $I$. This can be done in $O(\text{sort}(n+m))$ I/Os using time forward processing (cf. Section 2.5.6). A list of length $n$ can thus be ranked in $O(\text{sort}(n))$ I/Os.

### 3.2.3 Minimum Spanning Forest

Given an undirected connected graph $G$, a spanning tree of $G$ is a subgraph which is a tree and connects all the nodes. A minimum spanning tree is a spanning tree with minimum weight. For a general undirected graph (not necessarily con-

nected), we define a minimum spanning forest (MSF) to be the union of the minimum spanning trees for its connected components. Computing a minimum spanning forest of a graph $G$ is a well-studied problem in the RAM model.

The first algorithm for this problem is due to Boruvka [34]. This algorithm runs in phases; in each phase we find the lightest edge incident to each node. These edges are output as a part of the MSF. Contracting these edges leads to a new graph with at most half of the nodes. Since the remaining MSF edges are also in the MSF of the contracted graph, we recursively output the MSF edges of the contracted graph.

The most popular algorithms for MSF in the RAM model are Kruskal's and Prim's algorithms. Kruskal's algorithm [92] looks at the edges in increasing order of their weight and maintains the minimum spanning forest of the edges seen so far. A new edge is output as a part of the MSF if its two endpoints belong to different components in the current MSF. The necessary operations can be performed efficiently using a disjoint set (union-find) data structure [75]. The resultant complexity for this algorithm is $O(n \cdot \alpha(n))$ [149], where $\alpha(\cdot)$ is the inverse Ackermann function.

Unlike Kruskal's algorithm which maintains potentially many different MSTs at the same time, Prim's algorithm [86, 130] works by "growing" one MST at a time. Starting with an arbitrary node, it searches for the lightest edge incident to the current tree and outputs it as a part of the MST. The other end-point of the edge is then added to the current tree. The candidate edges are maintained using Fibonacci heaps [71], leading to an asymptotic complexity of $O(m + n \log n)$. If there is no edge between the nodes in and outside the current MST, we "grow" a new MST from an arbitrarily chosen node outside the MSF "grown" so far.

### Semi-external Kruskal's algorithm

In the semi-external version of Kruskal's algorithm, an external memory sorting algorithm is used to sort the edges according to their edge weights. The minimum spanning forest and the union-find data structure are kept in the internal memory (as both require $O(n)$ space). The I/O complexity of this algorithm is $O(\text{sort}(m))$.

### External memory Prim's algorithm

In order to externalize Prim's algorithm, we use an external memory priority queue (cf. Section 2.5.5) for maintaining the set of candidate edges to grow the

current minimum spanning tree. This results in an I/O complexity of $O(n + \text{sort}(m))$. The $O(n)$ term comes from the unstructured accesses to the adjacency lists, as we spend $O(1 + d(v)/B)$ ($d(v)$ being the degree of $v$) I/Os to get hold of edges incident to the node $v$ that need to be inserted into the priority queue.

### External memory Boruvka steps

In most external memory algorithms, a Boruvka step like contraction method is used to reduce the number of nodes to either $O(M)$ or $O(m/B)$. In the first case, semi-external Kruskal's algorithm or other semi-external base cases are used. In the latter case, any external algorithm like Prim's algorithm or MR_BFS can be used as we can afford one I/O per node in the contracted graph.

We initialize the adjacency lists of all nodes by sorting the edges first according to their tail node and that being equal, by their weight. In each EM Boruvka phase, we find the minimum weight edge for each node and output it as a part of MSF. This can easily be done by scanning the sorted adjacency lists. This partitions the nodes into pseudo-trees (a tree with one additional edge). The minimum weight edge in each pseudo-tree is repeated twice, as it is the minimum weight edge incident to both its end-points. Such edges can be identified in $O(\text{sort}(m))$ I/Os. By removing the repeated edges, we obtain a forest. We select a leader for each tree in the forest and let each node $u \in V$ know the leader $L(u)$ of the tree containing it. This can be done by variants of external memory list ranking algorithm (cf. Section 3.2.2) or by using time forward processing (cf. Section 2.5.6) and can be done in $O(\text{sort}(n))$ I/Os. We then replace each edge $(u,v)$ in $E$ by an edge $(L(u), L(v))$. At the end of the phase, we remove all isolated nodes, parallel edges and self loops. Again, this requires a constant number of sorts and scans of the edges.

The Boruvka steps as described here reduce the number of nodes by at least a factor of 2 in one phase and costs $O(\text{sort}(m))$ I/Os. Thus, it takes $\log \frac{n \cdot B}{m}$ phases to reduce the number of nodes to $O(m/B)$, after which the externalized version of Prim's algorithm or BFS algorithm can be used. This gives a total I/O complexity of $O(\text{sort}(m) \cdot \log \frac{n \cdot B}{m})$. Alternatively, we can have $O(\log \frac{n}{M})$ phases of Boruvka algorithm to reduce the number of nodes to $O(M)$ in order to apply semi-external Kruskal's algorithm afterwards. This will result in a total I/O complexity of $O(\text{sort}(m) \cdot \log \frac{n}{M})$.

### An $O(\text{sort}(m) \cdot \log\log(\frac{n \cdot B}{m}))$ I/O algorithm

Arge et. al. [18] improved the asymptotic complexity of the above algorithm by dividing the $O(\log \frac{n \cdot B}{m})$ phases of Boruvka steps into $O(\log\log \frac{n \cdot B}{m})$ super-phases requiring $O(\text{sort}(m))$ I/Os each. The idea is that rather than selecting only one

edge per node, we select $\sqrt{S_i}$ lightest edges for contraction in each super-phase, where $S_i := 2^{(3/2)^i} (= S_{i-1}^{3/2})$. If a node does not have that many adjacent edges, all its incident edges are selected and the node becomes inactive. The selected edges form a graph $G_i$. We apply $\log \sqrt{S_i}$ phases of Boruvka steps on $G_i$ to compute a leader $L(u)$ for each node $u \in V$. At the end of the super-phase, we replace each edge $(u,v)$ in $E$ by an edge $(L(u), L(v))$ and remove isolated nodes, parallel edges and self loops.

The number of active nodes after phase $i$ is at most $n/(S_i \cdot S_{i-1} \cdots S_0) = n/(S_i \cdot S_i^{2/3} \cdots S_0) \leq n/S_i^{5/3} \leq n/S_{i+1}$ and thus, $O(\log\log \frac{n \cdot B}{m})$ super-phases suffice to reduce the number of nodes to $O(n \cdot B/m)$.

Note that in super-phase $i$, there are $\log \sqrt{S_i}$ phases of Boruvka steps on $G_i$. Since $G_i$ has at most $n/S_i$ nodes at the beginning of phase $i$ and $n\sqrt{S_i}/S_i$ edges (as each of the $n/S_i$ nodes selects $\sqrt{S_i}$ edges around it), total cost of all these Boruvka phases is $O(\text{sort}(n/\sqrt{S_i}) \cdot \log \sqrt{S_i}) = O(\text{sort}(n))$ I/Os. The cost of replacing the edges by the contracted edges and other post-processing is $O(\text{sort}(m))$ I/Os.

Since each super-phase takes $O(\text{sort}(m))$ I/Os, the total I/O complexity of the algorithm is $O(\text{sort}(m) \cdot \log\log(\frac{n \cdot B}{m}))$.

Arge et al. [20] propose a cache-oblivious minimum spanning tree algorithm that uses a cache-oblivious priority queue to achieve the I/O complexity of $O(\text{sort}(m) \cdot \log\log n)$.

**Connected components** Minimum spanning forest also contains the information regarding the connected components of the graph. For directly computing connected components, one can use the above algorithm by modifying the comparator function for edge weights (since the weights on the edges can be ignored for connected components computation) – an edge is smaller than the other edge if either the head node has a smaller index or the two head nodes are equal, but the tail node has a smaller index.

### Randomized CC and MSF

Abello et. al. [2] proposed a randomized algorithm for computing connected components and minimum spanning tree of an undirected graph in external memory in $O(\text{sort}(m))$ expected I/Os. Their algorithm uses Boruvka steps together with edge sampling and batched least common ancestor (LCA) queries in a tree.

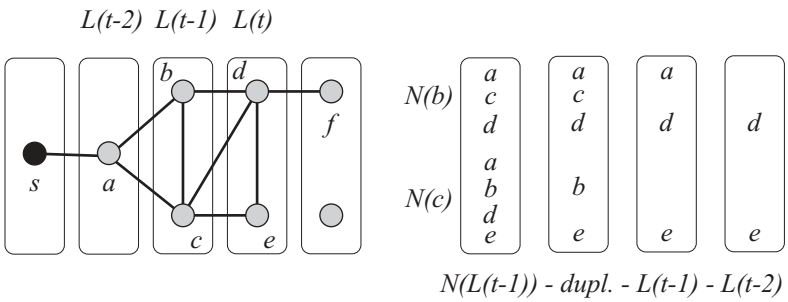



Figure 3.1: A phase in the BFS algorithm of Munagala and Ranade.

## 3.3 Algorithms

There are two main problems associated with running an internal memory BFS algorithm for computation on an externally stored graph:

- Remembering visited nodes needs $\Theta(m)$ I/Os in the worst case
- Unstructured access to adjacency lists, i.e., random I/Os to fetch adjacent edges may result in $\Theta(n)$ I/Os

### 3.3.1 Munagala and Ranade's algorithm

The algorithm (MR_BFS) by Munagala and Ranade [115] (as depicted in Figure 3.1) solves the first problem by exploiting the fact that in an undirected graph, the edges from a node in BFS level $t$ lead to nodes in BFS levels $t-1$, $t$ or $t+1$ only. Thus, in order to compute the nodes in BFS level $t+1$, one just needs to collect all neighbors of nodes in level $t$, remove duplicates and remove the nodes visited in levels $t-1$ and $t$. Except the unstructured accesses to the adjacency lists, all steps can be done in $\Theta(\text{sort}(m))$ I/Os. The total number of I/Os required by this algorithm is $\Theta(n+\text{sort}(m))$ as it may incur $\Omega(n)$ random I/Os (for reading the adjacency lists) in the worst-case.

### 3.3.2 Mehlhorn and Meyer's algorithm

In order to solve the problem of unstructured accesses to adjacency lists, Mehlhorn and Meyer [106] (MM_BFS) propose a pre-processing step in which the input graph is rearranged on the disk. The preprocessing phase involves clustering the

input graph into small disjoint groups of nodes that are close in the input graph. The edges incident to all nodes of a cluster are contiguously stored on the disk. This is useful as once a node from the cluster is visited, other nodes in the cluster will also be visited soon (owing to their proximity in the original graph). By spending only one random access (and possibly, some sequential accesses depending on the cluster size) for loading the whole cluster and then keeping the cluster data in some efficiently accessible data structure (hot pool) until it is all used up, the total number of I/Os can be reduced by a factor of up to $\sqrt{B}$ on sparse graphs. The neighboring nodes of a BFS level can be computed simply by scanning the hot pool and not the whole graph. Though some edges may be scanned multiple times in the hot pool, unstructured I/Os for fetching adjacency lists are considerably reduced, thereby decreasing the total number of I/Os.

The input graph is decomposed into $O(n/\mu)$ clusters of diameter $\tilde{O}(\mu)$[1] for some parameter $\mu$ to be fixed later. This can be done in two ways – "parallel cluster growing" and "Euler tour chopping".

### "Parallel cluster growing" variant

This variant (hereafter referred as MM_BFS_R) works by randomly choosing $\frac{n}{\mu}$ master nodes. The source node $s$ is also chosen to be a master node. Thereafter, we run a local BFS from all master nodes "in parallel". In each round, each master node tries to capture all unvisited neighbors of its current sub-graph. The ties can be resolved arbitrarily.

Capturing new nodes on the fringes of all clusters can be done by sorting the neighbors of the nodes captured in the previous round and then scanning the adjacency lists of the input graph. Each round $i$ thus takes $O(\text{sort}(m_i) + \text{scan}(m))$ I/Os, where $m_i$ is the number of edges adjacent to nodes captured in round $i-1$. The total number of clusters is at most $1+n/\mu$ and the number of rounds (number of edges in a shortest path between any node and its cluster center) is $O(\log n \cdot \mu)$ with high probability (w.h.p.). Thus the total complexity for this clustering is $O(\text{sort}(n+m) + \text{scan}(m) \cdot \mu \cdot \log n)$ w.h.p. and it produces $O(n/\mu)$ clusters of diameter $O(\log n \cdot \mu)$ w.h.p.

[1] Just as $O$ notation hides constant factors in the complexity, $\tilde{O}$ hides the polylogarithmic factors

### Euler tour based clustering

In this variant (MM_BFS_D), we first use the connected components (CC) algorithm to identify the component of the graph containing the source node $s$. The nodes outside this component are output with BFS level $\infty$ and can be safely ignored, as they do not affect the BFS level of any other node. Then, we compute a spanning tree of nodes in this connected component. Considering the undirected edges of this tree as bi-directional edges, we compute an Euler tour on these (up to $2n-2$) edges. We then employ the list ranking algorithm to store the nodes on the disk in the order of their appearance in the Euler tour. Note that the internal nodes of the spanning tree may appear multiple times in this tour. The nodes arranged in this way are then chopped into $\frac{2n-2}{\mu}$ clusters of size $\mu$. After removing the duplicates from this node sequence, we get the requisite clustering of nodes.

Since CC/MST can be computed in $O((1+\log\log\frac{n\cdot B}{m})\cdot \text{sort}(n+m))$ I/Os and the Euler tour and list ranking of $O(n)$ elements can both be done in $O(\text{sort}(n))$ I/Os, the total complexity of this preprocessing is $O((1+\log\log\frac{n\cdot B}{m})\cdot \text{sort}(n+m))$ I/Os. If the randomized expected $O(\text{sort}(m))$ I/O algorithm for CC/MST is used instead, we get a total expected I/O complexity of $O(\text{sort}(m))$ for the Euler tour based clustering.

### BFS phase

The actual BFS computation is similar to MR_BFS, but with one crucial difference: the adjacency lists of nodes in the current level $t$ are no longer accessed directly from the input graph using random I/Os. Instead, the nodes in BFS level $t$ are scanned in parallel with the nodes in the hot pool $H$ to compute the cluster indices of all nodes in BFS level $t$ whose adjacency lists are not already there in $H$. The multi-set of these cluster indices is then sorted and duplicates are removed from the sorted multi-set. The clusters corresponding to the resultant set of indices are then merged into $H$. A next round of scanning the nodes in BFS level $t$ in parallel with the hot pool $H$ fetches all the required adjacency lists.

Since each cluster is merged exactly once, it requires $O(n/\mu+\text{scan}(m))$ I/Os to load these clusters into $H$. For the Euler tour based approach, each adjacency list in $H$ is scanned for at most $O(\mu)$ rounds as the distance between any two nodes in the cluster is $O(\mu)$. Thus the total number of I/Os required for the BFS phase by the Euler tour based variant of MM_BFS is $O(n/\mu+\mu\cdot\text{scan}(n+m)+\text{sort}(n+m))$. By choosing $\mu=\max\left\{1,\sqrt{\frac{n}{\text{scan}(n+m)}}\right\}$, we get a total I/O

complexity (including the pre-processing) of $O(\sqrt{n \cdot \text{scan}(n+m)} + \text{sort}(n+m) + ST(n,m))$ I/Os for MM_BFS_D, where $ST(n,m)$ is the number of I/Os required to compute a spanning tree (of the connected component containing the source node) of a graph with $n$ nodes and $m$ edges. Using the randomized algorithm for MST/CC with $O(\text{sort}(n+m))$ expected I/O complexity, MM_BFS_D requires expected $O(\sqrt{n \cdot \text{scan}(n+m)} + \text{sort}(n+m))$ I/Os.

For the "parallel cluster growing" variant, an adjacency list stays in $H$ for $O(\mu \cdot \log n)$ levels w.h.p. Since there are at most $1 + \frac{n}{\mu}$ clusters and each cluster is loaded at most once, loading them into $H$ requires $O(\frac{n}{\mu} + \text{scan}(m))$ I/Os. The total complexity for MM_BFS_R is thus $O(n/\mu + \mu \cdot \log n \cdot \text{scan}(n+m) + \text{sort}(n+m))$ I/Os w.h.p. Choosing $\mu = \max\left\{1, \sqrt{\frac{n}{\text{scan}(n+m) \cdot \log n}}\right\}$, we get an I/O complexity of $O(\sqrt{n \cdot \text{scan}(n+m) \cdot \log n} + \text{sort}(n+m))$ I/Os w.h.p. for MM_BFS_R.

## 3.4 Engineering MR_BFS

One of the first decisions in designing any external memory implementation is to decide whether or not to use an external memory library (cf. Section 2.6.1). The advantage of using these libraries is that they reduce the development time by abstracting away the details of how an I/O is performed and providing ready-to-use efficient implementations of basic algorithms and data structures. We decided to work with STXXL [56, 57] because of the geographic proximity of its development[2] and its easy to use STL interface. Over the course of this project, many bugs were discovered and fixed in STXXL and quite a few additional features were requested and added. These bug-fixes and features have helped making the library more usable.

Although certain special features of STXXL are crucial to deal with some extreme graph classes, we believe that modulo some constant factors, the performance of our implementation should be the same on most graphs even with other external memory libraries, such as TPIE [21].

[2]The development of STXXL started in 2002 at Max Planck Institut für Informatik, Saarbrücken

### 3.4.1 STXXL

The key component of STXXL used by us is the stream sorter, which runs in two phases – the **Runs Creator (RC) Phase** and the **Runs Merger (M) Phase**. In the runs creator phase, the input vector/stream is divided into chunks of $M$ elements and each chunk is sorted within itself. These chunks are thereafter written to the disk space. In the runs merger phase, the first blocks of all the sorted chunks are brought to internal memory and merged there to produce the output stream which does not necessarily have to be stored on the disk. In case the sorting requires more than two rounds, the runs merger phase merges the sorted chunks recursively. For better efficiency, it is recommended to choose the block size and the internal memory available in such a way that the sorting does not require more than one round of merging.

Our data structures are implemented using the STXXL vector data-type. A vector in STXXL is organized as a collection of blocks residing on the parallel disks (or any other external storage). Each vector maintains a fully associative cache in internal memory. The vector cache consists of some fixed amount of pages. Each page in turn consists of $P$ external blocks. A random access to an element in the vector involves $P$ I/Os and therefore, in order to make full use of the disk parallelism, it is recommended that $P$ be some multiple of the number of parallel disks.

When accessing an element, if the page which the requested element belongs to, is in the vector cache, a reference to the element in the cache is returned. Otherwise, the page is first brought into the cache. If there is no free space in the cache, some page needs to be evicted. Each vector maintains its own paging strategy that decides which page is to be evicted. STXXL currently provides LRU and random paging strategies.

Each vector also has its own allocation strategy that decides how the vector will be stored across multiple disks. STXXL supports many different allocation strategies that stripe the data across disks (usually in some randomized way).

We also developed our own allocation strategies to deal with the case of heterogenous disks (e.g., when a hard disk and a solid state disk are used in parallel). This has been particularly useful when we ran our BFS implementation in such a setting (cf. Section 4.4).

STXXL vector also maintains a dirty flag with each page in the cache. The purpose of the flag is to track whether any element of the page is modified and therefore the page needs to be written to the disk(s) when it has to be evicted from the

cache. STXXL distinguishes between constant and non-constant accesses to the element, as the dirty flag is set when non-constant reference to one of the page's elements is returned.

### 3.4.2 Graph representation

Our main consideration in choosing our graph representation was to keep it as compact as possible. This is important as the I/O volume of our BFS implementations involves scanning the graph representation multiple times and a compact representation can save significant constant factors in I/Os. At the same time, we want to be able to access the adjacency list of an arbitrary node $v$ in $O(1+d(v)/B)$ I/Os and able to scan all the edges of the graph in $O(m/B)$ I/Os.

In our graph representation, nodes are assumed to have implicit unsigned integer labels in the range from 0 to $n-1$. The representation consists of two STXXL vectors – $N$ and $E$. The $i$th entry in $N$ contains the index to the beginning of the adjacency list of node $i$ in $E$. Note that this index is not the same as keeping a pointer to the appropriate location on disk, which may require up to 12 bytes of storage. Each edge $\{u,v\}$ is stored twice in $E$ – once as $v$ in the adjacency list of $u$ and once as $u$ in the adjacency list of $v$. Note that an element of the node vector $N$ contains only the index of an element in $E$. In particular, it does not contain the node label itself. An element of the edge vector $E$ contains only the node label of the adjacent node and not of the node itself. If a node label is 4-bytes (number of nodes less than $2^{32}-1$) and an index in $E$ is 8-bytes (number of edges less than $2^{63}-1$), the total storage requirement of our graph representation for MR_BFS is $8n+4m$ bytes.

Although we minimize the amount of information kept with node and edge elements in this data-structure, our implementation is still generic: it can handle graphs with arbitrary number of nodes (by appropriately modifying the data-type of a node label) and the graph template is basic and can be used for other graph algorithms as well.

In order to get the adjacency list of node $i$, we first load the page containing the $i$th and $(i+1)$th entry in $N$ into its vector cache. This gives us the necessary indexes in $E$. We then load all the pages containing elements in this range into $E$'s cache one by one and output the required adjacency list. Note that in order to efficiently handle the last node, $N$ contains a dummy node at the end that marks the end of $E$.

From an unordered list of edges, we can obtain the above graph representation in $O(\text{sort}(m))$ I/Os as follows: For each edge $\{u,v\}$ we enter two entries – $(u,v)$ and $(v,u)$ into a STXXL vector. We then sort this vector with respect to the first node in the ordered tuple, remove duplicates (if there are any) and initialize the node vector $N$ with the correct indexes of adjacency lists in this vector. The edge vector $E$ is then obtained by removing the label of the first node from each edge element.

Our output format that stores the BFS decomposition is similar. The two vectors, referred as $L$ and $NL$, represent the BFS levels and the nodes in those levels, respectively. The $i$th entry in vector $L$ merely contains the index to a location in vector $NL$ where the nodes in the $i$th BFS level are stored. This ensures that we do not spend one I/O per level when storing the output which is a major performance consideration for large diameter graphs (cf. Section 3.4.5).

### 3.4.3 Implementing MR_BFS

We present the details of our software using flow-charts. The circular or elliptical blocks in these flow-charts represent storage on the external media, the arrows leading to these blocks correspond to write I/Os and the arrows leading away from these blocks correspond to read I/Os. Figure 3.2 shows the flow-chart of MR_BFS. Let $L(t)$ denote the set of nodes in BFS level $t$, $E(t)$ be the adjacency lists of the nodes in $L(t)$, $A(t)$ be the multi-set of neighbors of nodes in $L(t)$ and $N(S)$ denote the set of neighbors of nodes in a set $S$. Given $L(t-1)$, $L(t)$ and $N(L(t))$ computed in the previous iteration, we compute $L(t+1)$ in the current iteration. This is done by reading the nodes in sorted sets $L(t-1)$, $L(t)$ and $N(L(t))$ from the disk and scanning them in parallel to compute $L(t+1) = \{N(L(t)) \setminus (L(t-1) \cup L(t))\}$. The set $L(t+1)$ so produced is also sorted. It is then written back to the disk. We collect the adjacency lists of all nodes in $L(t+1)$ from the disk (using potentially random I/Os) as $E(t+1)$. Note that since $L(t+1)$ is sorted, this step requires $O(\text{scan}(m))$ I/Os. Using $E(t+1)$, we compute the multi-set $A(t+1)$ which is written to the disk. $A(t+1)$ is then passed as an input to the runs creator (the first phase of STXXL sorting) which produces sorted runs (sorted chunks of $M$ elements). These runs are read from the disk and merged (second phase of STXXL sorting). Duplicates are removed from this sorted set to compute $N(L(t+1))$ which is written to the disk. This forms the set $N(L(t))$ for the next BFS level ($t := t+1$) or the next iteration.

Summing over all BFS levels, the worst case number of I/Os for this implementation of MR_BFS (assuming a single merge pass in sorting) is given by the follow-

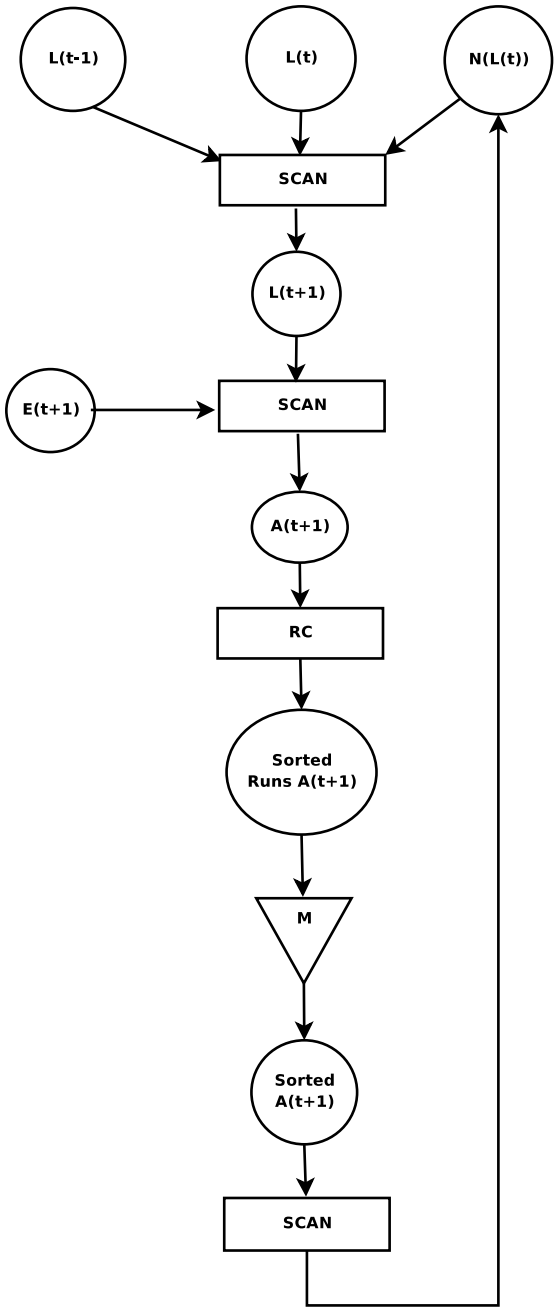



Figure 3.2: Flow-chart of MR_BFS implementation.

ing expression:

$$n+\text{scan}(\sum_t(|L(t)|+|L(t-1)|+2\cdot|L(t+1)|+|E(t+1)|+6\cdot|A(t+1)|+2\cdot|N(L(t))|))$$

The factor 2 for scanning $L(t+1)$ and $N(L(t))$ stems from summing the reading and writing costs and the factor 6 for $A(t+1)$ comes from reading and writing $A(t+1)$, sorted runs of $A(t+1)$ and sorted $A(t+1)$. Since $\sum_t |L(t)| \leq n$, $\sum_t |E(t)| \leq 2m$, $\sum_t |A(t)| \leq 2m$, and $\sum_t |N(L(t))| \leq \sum_t(|L(t-1)|+|L(t)|+|L(t+1)|) \leq 3n$, the worst case total number of I/Os is $n+\text{scan}(10n+14m)$.

### 3.4.4 Pipelined MR_BFS

Recall from Section 2.6.2 that an engineering technique called pipelining is often employed in external memory algorithms to save constant factors in the I/O complexity. The key idea behind pipelining is to connect a given sequence of algorithmic steps with an interface so that the data can be passed-through from one algorithm to another without needing any external memory intermediate storage. Figure 3.3 shows the flow-chart of a coalesced MR_BFS algorithm.

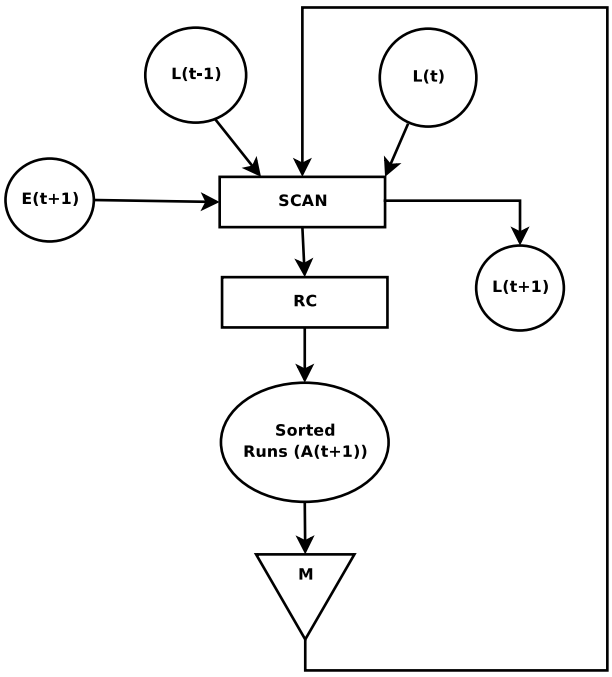



Figure 3.3: Flow-chart of pipelined MR_BFS implementation.

The complexity of the pipelined MR_BFS mainly lies in its scanner. The scanner receives the stream of sorted multi-set $A(t)$. While looking at the elements one at a time, it determines if it is a duplicate by checking with the stored previous element of the stream. If not, it checks if this element is in $L(t)$ or $L(t-1)$ reading these sets from the disk as sorted streams. If not, it collects this element into the $L(t+1)$ buffer and reads its adjacency list from the disk (represented as $E(t+1)$ in the figure) to form the stream $A(t+1)$. The stream $A(t+1)$ is passed directly to runs creator and sorted runs are written on the disk. These are later merged and passed to the scanner as sorted multi-set $A(t)$ for the next level.

In this case, the worst case number of I/Os (again assuming a single merge pass) is given by the following expression:

$$n + \text{scan}(\sum_t (|L(t-1)| + |L(t)| + |L(t+1)| + |E(t+1)| + 2 \cdot |A(t)|))$$

Since $\sum_t |L(t)| \leq n$, $\sum_t |E(t)| \leq 2m$, and $\sum_t |A(t)| \leq 2m$, the worst case total number of I/Os for pipelined MR_BFS is $n + \text{scan}(3n + 6m)$. Thus, for MR_BFS, pipelining reduces the worst case number of I/Os from $n + \text{scan}(10n + 14m)$ to $n + \text{scan}(3n + 6m)$. This is particularly significant for graphs that do not force MR_BFS to incur $n$ I/Os for reading adjacency lists.

### 3.4.5 Dealing with large diameter graphs

Consider the case of large diameter graphs with a good layout on disk such as a list where the nodes are stored on the disk in the order the BFS algorithm needs to traverse them. Theoretically, MR_BFS should require $O(n/B)$ I/Os on these graphs as reading the graph stored in this way and storing the output can both be done in $O(n/B)$ I/Os. Our preliminary implementation however took $\Omega(n)$ I/Os. We discovered that the reason for this has been that the initialization of the runs creator for sorting $N(L(t))$ (even if it contained only one node) and converting a vector into stream (even if the vector contained only one element) both required $\Omega(1)$ I/Os. Since for each level, a new instance of runs creator is initialized and a vector is converted into a stream, this causes $\Omega(n)$ I/Os for the list graphs. The reason for this behavior is that STXXL was designed to handle external memory data and it was not conceived that in the course of it, it may also have to sort streams with $k < B$ elements without incurring any I/Os.

New features were added to STXXL to handle these problems. The STXXL stream sorter (from version 0.75 onwards) does not need any I/O (with the appropriate flag) if $k < B$. Also, for this case, the internal work is proportional to

$n \log n$, independent of $B$. Converting a vector into a stream or initialization of runs creator or runs merger do not cause any I/O.

While these new features helped reduce the I/O time, the computation time remained quite high. This was because of the overhead associated with initializing the external sorters, which involved allocating appropriate amount of memory. In the pipelined version of MR_BFS, we do not know in advance the exact number of elements to be sorted and hence, we can't switch between the external and the internal sorter so easily. In order to get around this problem, we first buffer the first $B$ elements and initialize the external sorter only when the buffer is full. Otherwise, we sort it internally.

Overall, these add-ons reduced the I/O and the computation cost for running MR_BFS on large diameter graphs significantly and helped achieving the theoretical bounds for this case. The BFS phase of MM_BFS inherits these optimizations and hence, does not suffer from $\Omega(1)$ I/O and high computation cost per level.

## 3.5 Engineering MM_BFS_R

In this section, we first present the graph representation that we use both for MM_BFS_R and MM_BFS_D. We then describe our pipelined implementation of MM_BFS_R.

### 3.5.1 Graph representation

We consider here the graph representation to store the preprocessed input graph. Together with the nodes and edges, we also need to store the clustering information. From this representation, we should be able to collect all nodes in an arbitrary cluster in $O(1) + \frac{\text{cluster size}}{B}$ I/Os. Each edge needs to keep not only the labels of both the adjacent nodes, but also their cluster indices, so that we can efficiently determine whether or not the cluster of the adjacent node is in the hot pool.

Rather than having each cluster consist of an adjacency array containing nodes and edges belonging to it, we store the partitioned input graph as three vectors $F$, $N$, and $E$ (as shown in Figure 3.4). Vectors $N$ and $E$ contain the nodes and adjacency lists, respectively. Vector $N$ is kept sorted according to cluster indices of the nodes and that being equal, according to the node labels. Edges in vector

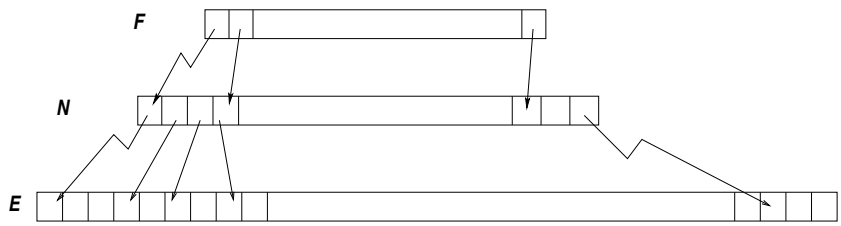



Figure 3.4: I/O-efficient data structure to represent a partitioned graph.

$E$ are kept sorted with respect to the cluster index of the first node and that being equal, according to the first node label. The $i$th entry in vector $F$ contains only an index of vector $N$ representing the beginning of the set of nodes in the $i$th cluster. Elements in $N$ contain the node label as well as an index of vector $E$ where the adjacency list of a particular node starts. Each edge in $E$ contains the labels of both the adjacent nodes as well as their cluster indices.

In order to facilitate accessing all nodes in the last cluster and the adjacency list of the last node, we keep dummy nodes at the end of vectors $F$ and $N$ to mark the last element of $N$ and $E$, respectively.

### 3.5.2 Pipelined MM_BFS

Figure 3.5 shows the flow-chart of the pipelined version for the "parallel cluster growing" phase of MM_BFS_R. This phase begins with randomly selecting $n/\mu$ nodes to be master nodes. The main scanner (SCAN 1) of this phase takes the stream of the sorted sequence of the nodes on the fringe of expanding clusters and stores the cluster index (by including the fringe nodes into their corresponding clusters) with these nodes. It also reads the adjacency lists of these nodes to compute the new sequence of fringe nodes to be sent to the two-phase sorter. After the partitioning of nodes into clusters is complete, SCAN 2 stores the cluster index of the tail node with each edge. We then sort $E$ with respect to the head node label. The next scanner (SCAN 3) then stores the cluster index of the head node with each edge. We then sort $N$ and $E$ with respect to the cluster index (of the tail node) and that being equal, according to the (tail) node label. SCAN 4 then adjusts the cluster and the node iterators (which are indexes in $N$ and $E$ respectively) appropriately. Since the diameter of any cluster is less than $\mu \cdot \log n$ w.h.p., the total number of I/Os for this phase is bounded by $\text{scan}\left(16m + 6n + \frac{n}{\mu} + 2 \cdot (m+n) \cdot \log n \cdot \mu\right)$ w.h.p.

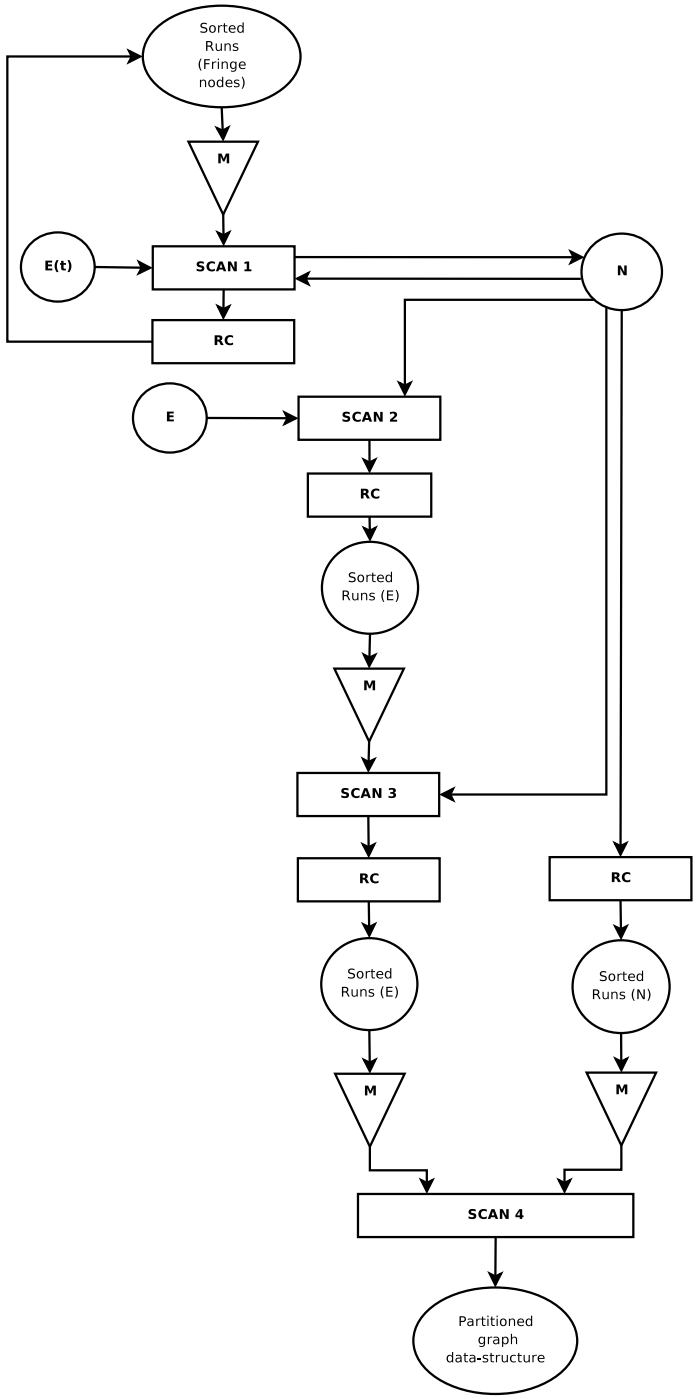



Figure 3.5: Flow-chart for the “parallel cluster growing” phase of MM_BFS_R.

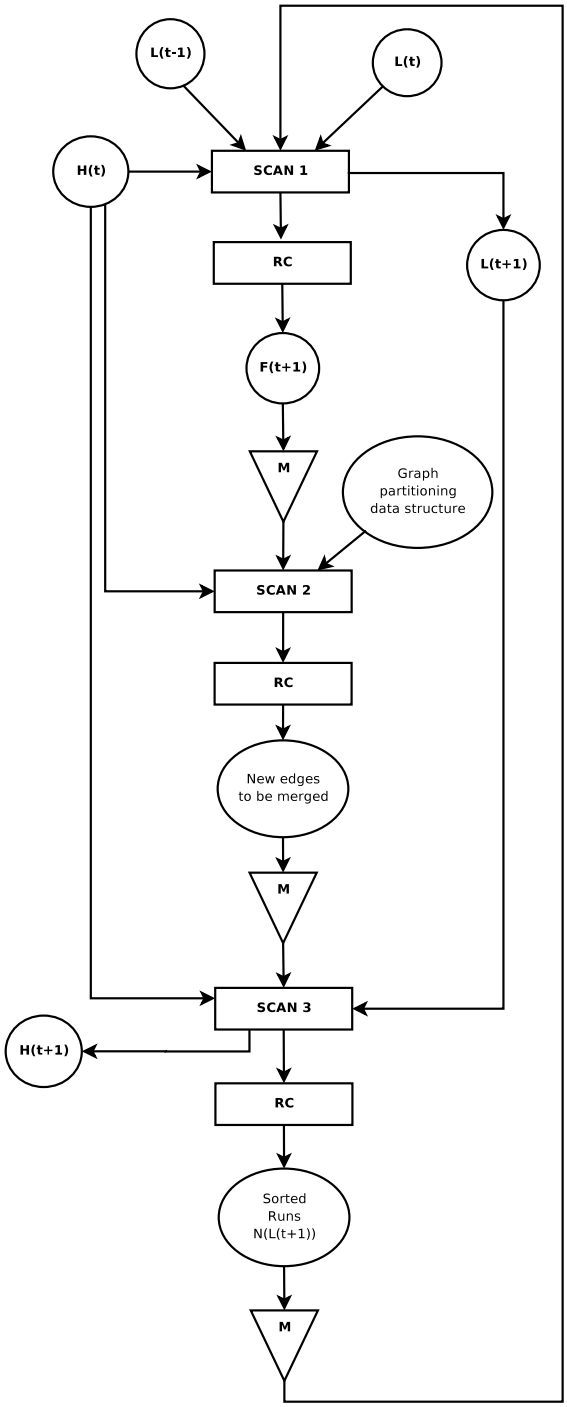



Figure 3.6: Flow-chart for the BFS phase of MM_BFS.

In the pipelined BFS phase of MM_BFS (which is common to both MM_BFS_R and MM_BFS_D) shown in Figure 3.6, the first scanner (SCAN 1) receives the sorted sequence $N(L(t))$ of neighbor nodes of $L(t)$ from the merger stream computed in the previous iteration. It reads $L(t-1)$ and $L(t)$ from the disk and the adjacency lists of nodes in $L(t)$ from the hot pool $H(t)$ (at level $t$) and computes $F(t+1)$ – the multi-set of cluster indices of nodes in $L(t+1)$ – and in the process, also writes (sorted) $L(t+1)$ to disk. The second scanner (SCAN 2) takes the sorted stream $F(t+1)$ and eliminates duplicates from it. It then checks if the cluster (corresponding to the element in $F(t+1)$) is already loaded into the hot pool $H(t)$. If not, it reads the cluster edges from the graph partitioning data structure and outputs these edges as the stream to the two-phase sorter. The next scanner (SCAN 3) reads the sorted sequence of edges that need to be merged into the hot pool, removes the adjacent edges of nodes in $L(t+1)$ and computes $H(t+1) := (H(t) \cup \text{Merged cluster edges}) \setminus \text{Adj}(L(t+1))$, where $\text{Adj}(S)$ represents the edges adjacent to nodes in a set $S$. In the process, this third scanner also outputs the multi-set $N(L(t+1))$. This is then sorted and passed on to the next round as sorted $N(L(t))$.

The total number of I/Os for this phase is bounded by $\#clusters + \text{scan}(8m \cdot \text{cluster\_diameter} + 10m + 6n)$. Since for MM_BFS_R, each cluster diameter is bounded by $\mu \cdot \log m$ w.h.p. and the number of clusters is $1 + \frac{n}{\mu}$, the total I/O complexity is bounded by $1 + \frac{n}{\mu} + \text{scan}(10m + 6n + 8m \log m \cdot \mu)$ w.h.p. For MM_BFS_D, the cluster diameter is bounded by $\mu$ and the number of clusters is at most $\frac{2n}{\mu}$. So, the total number of I/Os required by MM_BFS_D is bounded by $\frac{2n}{\mu} + \text{scan}(10m + 6n + 8m \cdot \mu)$.

## 3.6 Engineering MM_BFS_D

As discussed in Section 3.3, the key components of the Euler tour based preprocessing of MM_BFS include minimum spanning tree, list ranking and the Euler tour of a tree. In this section, we discuss the various design choices for each of these components.

### 3.6.1 Engineering minimum spanning forest

Dementiev et al. [54] carefully engineered an external memory MSF algorithm. Their implementation is based on a sweeping paradigm to reduce the number of

nodes to $O(M)$ and then running the semi-external Kruskal's algorithm. The node contraction phase consists of repeatedly choosing a node at random and contracting the lightest edge incident to it. In external memory, selecting random nodes can be done by using I/O-efficient random permutation (e.g., [139]) and looking at the nodes in that order. In contracting the edges, one needs to "inform" all the other neighbors of the non-leader node about the leader node. This can be done by time-forward processing (cf. Section 2.5.6) using external memory priority queues or using a bucket structure. This MSF implementation uses STXXL for sorting, priority queue and other basic data structures.

Dementiev et al. [54] showed that with their tuned implementation, massive minimum spanning tree problems filling several hard disks can be solved "overnight" on a low cost PC-server with 1 GB RAM. They experimented with many different graph classes – random graphs, random geometric graphs and grid graphs. In general, they observed that their implementation of semi-external Kruskal's algorithm only loses a factor of 2 in the execution time per edge as compared to the internal memory algorithm. Running on disks, their external memory implementation merely loses an additional factor of 2.

Our experiments confirmed that this implementation is quite fast in practice and despite the fact that the underlying graph representation in it is different than ours, it is well-suited for our application.

### 3.6.2 Engineering List Ranking

The list ranking algorithm by Sibeyn [145] has low constant factors (for realistic input size) in its I/O complexity and is therefore, more practical than the algorithm [48] (described in Section 3.2.2) based on independent set removal. The algorithm splits the input list into sublists of size $O(M)$ and goes through the data in a wave-like manner. For all elements of the current sublist, it follows the links leading to the elements of the same sublist and updates the information on their final element and the number of links to it. For all elements with links running outside the current sublist, the required information is requested from the sublists containing the elements to which they are linked. The algorithm uses *bucketing* and *lazy processing* of the requests and the answers to the sublists, i.e., it stores them in one common stack and processes them only when the wave through the data hits the corresponding sublist.

Unfortunately, Sibeyn's implementation relies on the operating system for I/Os and does not guarantee that the top blocks of all the stacks remain in the internal

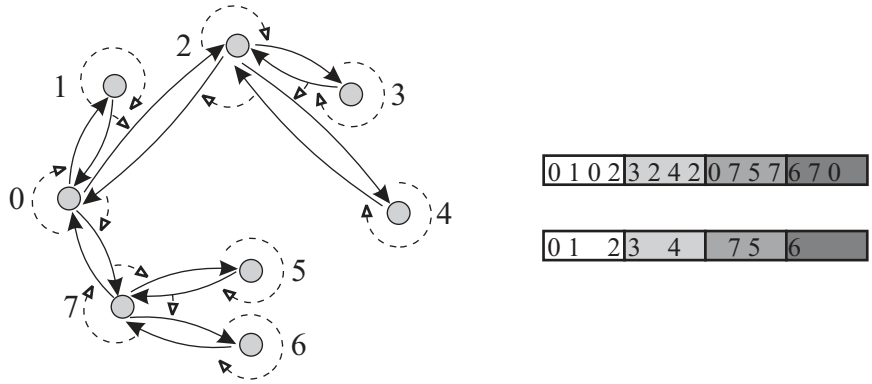



Figure 3.7: The bi-directed tree (shaded circles and solid lines) and the closed linked list of its edges (dashed lines) on the left. The order of the vertices and their partitioning before and after the duplicates removal on the right.

memory, which is a necessary assumption for the asymptotic analysis of the algorithm. Besides, its reliance on internal arrays and swap space puts a restriction on the size of the lists it can rank. We re-implemented this algorithm using STXXL stacks and vectors. The deeper integration of the algorithm in the STXXL framework makes it possible to obtain a scalable solution, which could handle graph instances of the size we require while keeping the theoretical worst case bounds.

Our implementation of this algorithm in the STXXL framework is quite fast in practice and takes only around 20 minutes for a list of $2^{29}$ elements.

### 3.6.3 Euler tour

Recall from Section 3.2 that in order to construct the Euler tour around the bi-directional minimum spanning tree (Figure 3.7), each node chooses a cyclic order of its neighbors. For every edge $(u,v)$, its successor is defined to be the edge $(v,w)$ ($u$ may be the same as $w$) such that in the cyclic order of neighors of $v$, $u$ is followed by $w$. In one scan of the edges of the bi-directional tree, each edge is linked to its successor. The linear ordering induced by the successor function constitutes the Euler tour. This tour is then split at the source node $s$ by marking an edge leading away from $s$ in the circuit as the starting edge of the tour.

The position of an edge in the Euler tour is computed using list ranking. These edges are then sorted such that they are stored on the disk in the order of their position in the tour. While scanning the nodes in the order they appear in the tour (some nodes may be repeated), we subdivide the tour into chunks of size

$\max\{1, \sqrt{\frac{n \cdot B}{n+m}}\}$ nodes. Thereafter, we remove the duplicate nodes using the sorting routine of STXXL and get the partitioning of the input graph.

## 3.7 A heuristic for maintaining the pool

In this section, we propose a heuristic for efficient management of the hot pool. This heuristic is aimed at speeding up the practical performance of MM_BFS_D, particularly for large diameter graphs. At the same time, it preserves the worst case I/O bounds of MM_BFS.

For many large diameter graphs, the pool fits into the internal memory most of the time. Although in this case, the number of edges in the pool is not so large, scanning all the edges in the pool for each level can still be computationally quite expensive. Hence, we keep a portion of the pool that fits in the internal memory as a multi-map hash table. Given a node as a key, it returns all the nodes adjacent to the current node. Thus, to get the neighbors of a set of nodes we just query the hash table for those nodes and then delete them from the hash table. For loading the cluster, we just insert all the adjacency lists of the cluster in the hash table, unless the hash table has already $\Theta(M)$ elements.

Recall that after the deterministic preprocessing, the adjacency lists are stored on the disk in the order in which their corresponding nodes appear on the Euler tour around a spanning tree of the input graph. The Euler tour is chopped into clusters with $\max\left\{1, \sqrt{\frac{n \cdot B}{n+m}}\right\}$ nodes (before the duplicate removal) ensuring that the maximum distance between any two nodes in the cluster is at most $\max\left\{1, \sqrt{\frac{n \cdot B}{n+m}}\right\} - 1$. However, the fact that the contiguous adjacency lists on the disk have their corresponding nodes closer in terms of BFS levels is not restricted to intra-cluster nodes. The adjacency lists that come alongside the requisite cluster will also be required soon and by caching these other adjacency lists, we can save some I/Os in the future. This caching is particularly beneficial when the pool fits in the internal memory. Note that we still load the $\max\left\{1, \sqrt{\frac{n \cdot B}{n+m}}\right\}$ node clusters in the pool, but keep the remaining elements of the block in the pool-cache. For MM_BFS_D on linked lists, this means that we load $O(\sqrt{B})$ nodes in the internal pool, while keeping the remaining $O(B)$ adjacency lists which we get in the same block, in the pool-cache, thereby reducing the I/O complexity for the BFS traversal on linked lists to that of scanning a list stored in the ranked order.

Recall that we represent the adjacency lists of nodes in the graph as a STXXL

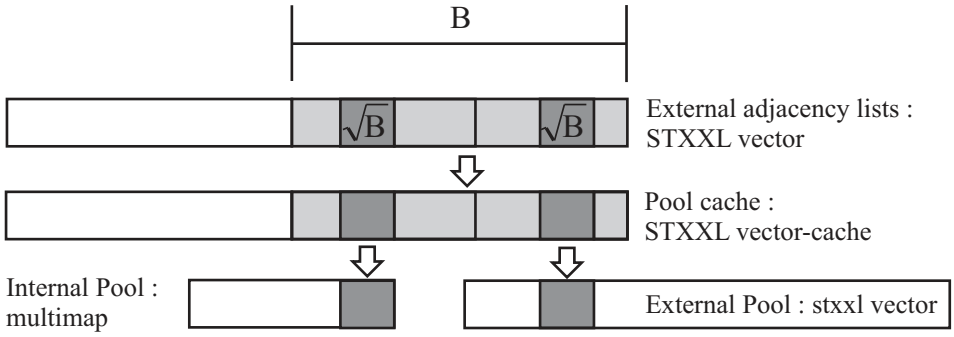



Figure 3.8: Scheme depicting an example run of the implementation of our heuristic. The dark regions denote the clusters that need to be loaded into the hot pool. The entire block containing the two clusters is first loaded into the vector-cache. At this juncture, the internal hot-pool (a multi-map hash table) can only hold one more cluster. Therefore, one of the clusters goes into the internal pool and the other cluster is stored on the external hot-pool.

vector. STXXL already provides a fully associative cache with every vector. Before doing an I/O for loading a block of elements from a vector, it first checks if the block is already there in the vector-cache. If so, it avoids the I/O and provides the elements from the cache instead. Increasing the vector-cache size of the adjacency list vector with a layout computed by the Euler tour based preprocessing and choosing the replacement policy to be LRU provides us with an implementation of the pool-cache. Figure 3.8 depicts the implementation of our heuristic.

## 3.8 External memory graph generator

For the purpose of this study, we designed and implemented a pipelined version of an I/O efficient framework for generating large graphs of many different classes. Our graph generator can be easily de-coupled from our graph representation and is therefore, of independent interest. Since it can generate massive graphs quickly, it was one of the few graph generators recommended for the DIMACS implementation challenge on shortest paths [62].

Our graph generator first produces a stream of edges (each undirected edge is represented as a pair of directed edges, one in each direction), randomly permutes the node labels if required by the graph class, sorts the edge-sequence, removes duplicates, and converts it into our graph representation. For an I/O-efficient random permutation needed in the generation process of many graphs, we use [139].

### 3.8.1 Graph classes.

We consider the following graph classes covering a broad spectrum of different characteristics influencing the performance of external memory BFS algorithms:

#### Random Graph

The random graph model $G(n,p)$ [66, 67] (cf. Section 2.3) refers to graphs with $n$ nodes in which each edge is chosen independently with probability $p$. Generating such a graph by considering whether or not an edge exists between every pair would take $\Omega(n^2)$ time. So, we consider a different notion of random graphs in which all the $m$ edges are chosen with having tail and head nodes picked randomly, i.e., on $n$ node graphs, we randomly select $m$ edges with replacement. We make sure that the randomly chosen tail and head nodes are not the same to avoid self loops. From the multi-set of edges so generated, we remove duplicates to avoid parallel edges. The random graph so obtained is equivalent to a random graph $G(n,p)$ with $p = 1 - \left(1 - \frac{1}{\binom{n}{2}}\right)^m$. For our experiments, we mostly work with $m = 4n$ which corresponds to $p \sim \frac{4}{n-1}$.

A random graph $G(n,p)$ has a giant connected component with a small diameter w.h.p. if $p = \Omega(\frac{\log n}{n})$. In conformity with the theoretical results, we observed that on large random graphs with $m$ around $4n$, there is a big connected component (containing more than $0.99n$ nodes) with 10–15 BFS levels starting from a random node.

#### $B$-level random graph

Given $n$, $m$ and $B$, consider the graph in Figure 3.9. The graph consists of $n$ nodes, and with the exception of the source node $s$ they are spread over $B$ levels of $\frac{n-1}{B}$ nodes each. These $B$ levels approximate the BFS levels, as edges in this graph only connect nodes between consecutive levels. The source node is connected to all nodes in the first BFS level. The $\frac{m-\frac{n-1}{B}}{B-1}$ edges between any two consecutive levels $i$ and $i+1$ have their one end-point from level $i$ and the other end-point from level $i+1$ chosen randomly with a uniform probability distribution.

The following layout of this graph on the disk causes MR_BFS to incur its worst case of $\Omega(n)$ I/Os: For each level, the nodes are arranged in the node vector

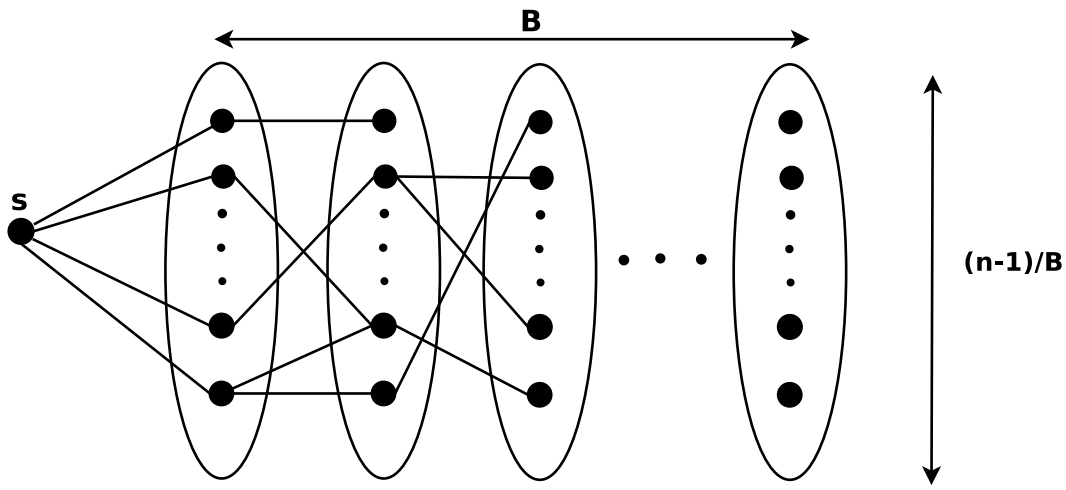



Figure 3.9: B-level random graph

such that each node in the level resides in a different block. For this, we choose the node labels such that the $i$th level $L_i = \{u | u \bmod B = i\}$. Since these levels approximate BFS levels and MR_BFS involves accessing these nodes in the node vector together, it will cause MR_BFS to incur $\sim \frac{n-1}{B}$ I/Os for every BFS level. Summing over all $B$ BFS levels, it will cause MR_BFS to have $\Omega(n)$ I/Os.

We consider $B$-level random graphs with $m = 4n$. They have a giant connected component and the levels correspond very well with the actual BFS levels.

One can also generate the above graph with a random layout on the disk. The performance of external memory BFS algorithms on the two layouts is similar.

### $B$-level spider web graph

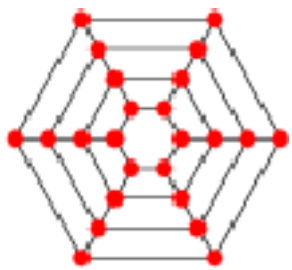


Figure 3.10: $B$-level spider web graph

This graph class (as shown in Figure 3.10) is a specialization of web graph (not to

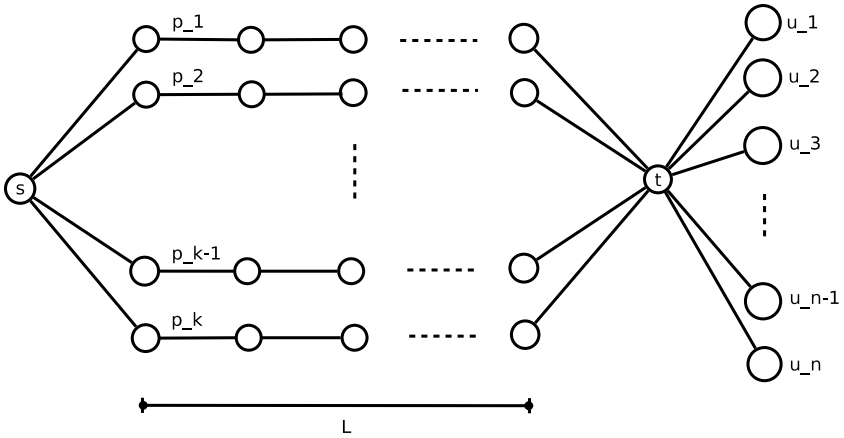



Figure 3.11: MM BFS worst graph

be confused with the power-law graphs used to simulate WWW or WWW crawls) defined in [153]. It also consists of $B$ levels, each having $\frac{n-1}{B}$ nodes. All nodes in a level are connected in a cyclic fashion and a node has an edge to its corresponding node in the level before and after. The initial layout of the nodes on the disk is random. A similar graph with $\sqrt{B}$ levels is also supported by our generator.

### MM BFS worst graph

Given two parameters $n$ and $\mu$ (closely related to the MM_BFS parameter with the same notation), this graph [37] shown in Figure 3.11 consists of a source node $s$ and a node $t$ connected by $k := \sqrt{n}$ independent paths of length $L := \log_{1-1/\mu}(1/\sqrt{n})$. Furthermore, $t$ is connected to $n$ independent nodes $u_1, \ldots, u_n$ by an edge. The total number of nodes and edges in this graph is $O(n)$. This graph is so named as it causes MM_BFS_R to incur its worst case of $\Theta\left(n \cdot \sqrt{\frac{\log n}{B}} + \text{sort}(n)\right)$ I/Os w.h.p. on sparse graphs.

### Grid Graph

Given $x$, $y$ and $p$, a grid graph consists of an $x \times y$ grid where each edge of this grid is chosen independently with a probability $p$. The layout of this graph on the disk is random. We mostly consider the case with $p = 1$, $x = \lceil\sqrt{n}\rceil$ and $y = \lfloor\sqrt{n}\rfloor$. For this case, the grid graph has a diameter of $\lceil\sqrt{n}\rceil + \lfloor\sqrt{n}\rfloor$.

We also consider long and narrow grids in two dimensions as well as grids in three and four dimensions as examples of large diameter graphs.

### List graphs

A list graph consists of $n$ nodes and $n-1$ edges such that there exist two nodes $u$ and $v$, with the path from $u$ to $v$ consisting of all the $n-1$ edges. We consider three different initial layouts – simple, in which consecutive nodes in the list appear contiguous on the disk; B-interleaved in which consecutive nodes are all in different but consecutive blocks; random in which the arrangement of nodes on disk is given by a random permutation.

### Webgraph

As an instance of a real world graph, we consider an actual crawl of a part of the world wide web in 2001 [150], where an edge represents a hyperlink between two sites. Although this is a directed graph, we treat it as undirected. This graph has around 130 million nodes and 1.4 billion edges. It has a core which consists of most of its nodes and behaves like a random graph.

Our graph generator also includes a translator to read this webgraph, make it undirected (by inserting an edge in the other direction) and convert it into our graph representation.

### Other graph classes

There are many other graph classes supported by our generator such as geometric graphs where the nodes are associated with points in some space and the probability of an edge to exist between two nodes in the graph is inversely proportional to the Euclidean distance between their corresponding points.

## 3.9 External memory BFS decomposition verifier

As another side tool, we designed an I/O efficient verifier routine to determine whether or not a BFS level decomposition is correct for a given graph.

For an undirected connected graph, the following are necessary and sufficient conditions for a BFS level labeling to be correct:

1. BFS level 0 contains the source node $s$ only.
2. Every node $v \in V$, $v \neq s$ has a unique BFS level $bfs\_level(v) > 0$.
3. $\forall (u,v) \in E, |bfs\_level(u) - bfs\_level(v)| \leq 1$.
4. $\forall u \in V$ in BFS level $k$ ($k > 0$), $\exists\ edge(u,v)$ such that $v$ is in BFS level $k-1$.

Next, we show how to check all these conditions in $O(\text{sort}(n+m))$ I/Os in a pipelined way. Figure 3.12 shows the flow-chart of our pipelined implementation of the BFS checker. Recall that the representation of the BFS output consists of two vectors – $L$ and $NL$. In $NL$, the nodes of the graph are kept sorted according to their BFS levels. The $i$th entry in $L$ contains the index of an element in $NL$ from where the nodes in the $i$th BFS level begin.

The first scanner (SCAN 1) checks the first condition and forms tuples of the form $<$ node label, *bfs_level* $>$ from the BFS output representation. These tuples are then sorted according to node label and passed on to the second scanner (SCAN 2). SCAN 2 checks the second condition. It also does a parallel scan of the sorted (w.r.t. the tail node label) set of edges and stores the BFS level of the first end-point with each edge. The set of edges is then sorted according to the label of the other end-point (head node of each edge). SCAN 3 then scans this sorted edge set in parallel with the sorted tuple list and stores the BFS level of the head node with each edge. In the process, it also checks if the third condition is satisfied. The set of edges is then sorted according to the BFS level of the tail node and that being equal with the BFS level of the head node. The last scanner (SCAN 4) then checks the last condition on this sorted set of edges. A BFS level decomposition is correct only if it satisfies all the conditions checked by the scanners.

## 3.10 BFS software package

*The software isn't finished until the last user is dead.*

–Anonymous

Our code for the experimental study of external memory BFS algorithms has now evolved into a software package that can be used as a black-box for many applications. We eventually plan to integrate this code into a library of external memory

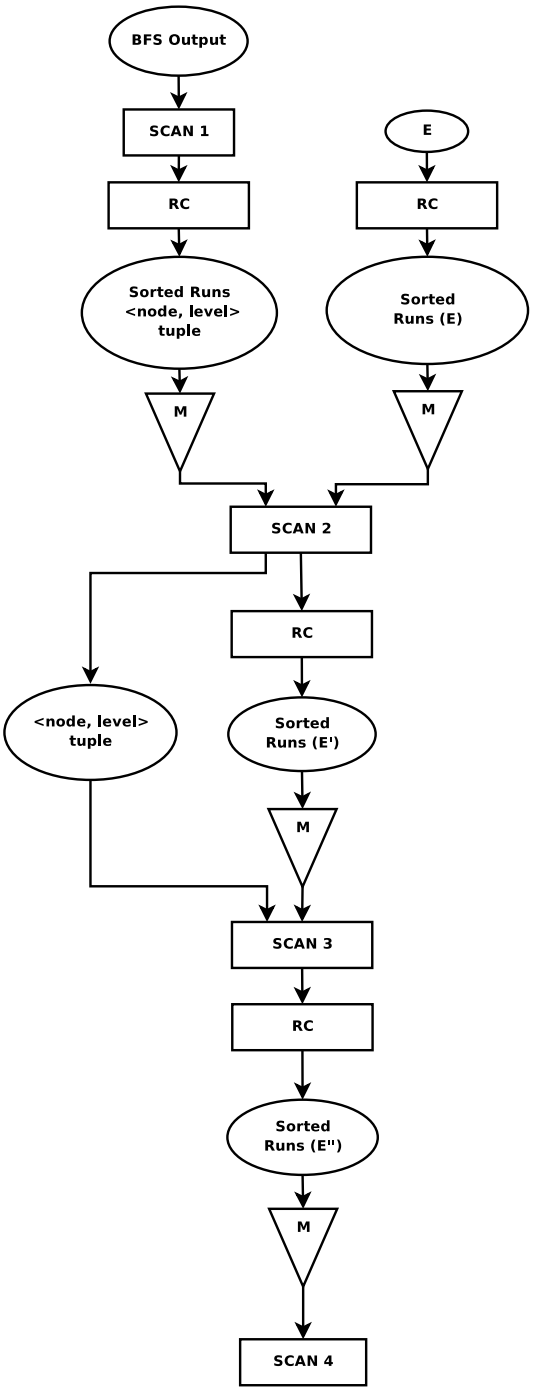



Figure 3.12: Flow-chart of the pipelined BFS checker.

algorithms dealing with massive graphs.

Many features for easing the usability of the code (both for a naïve and an expert user) have been integrated in this package. The software currently supports many different input graph formats such as a list of edges or the DIMACS shortest path challenge graph format (with adaptors to convert them into our graph representation in $O(\text{sort}(m))$ I/Os). It can also output the BFS results in many different formats such as a BFS tree, BFS levels of all nodes and all nodes in a particular BFS level (both in binary and ASCII format).

Our implementation can be used on many different 32-bit and 64-bit architectures with single or multi-core processors and single or multiple (homogenous or heterogenous) external disks. In order to efficiently use our external memory implementations on different machines, one needs to tune the values of block size, number of external disks, and available main memory size based on the underlying hardware.

This package has been continously evolving for the last four years. The latest stable version of our code is available from the SVN repository https://svn.mpi-inf.mpg.de/AG1/EM/ajwani/embfs/trunk. Apart from many bug-fixes, it includes many features requested by the users of our software package.

The download page of an earlier version of our code was visited more than 300 times in the last two years. We released this code under GNU General Public License (GPL) version 2 as freely downloadable and did not keep any statistics about our users. From the log of feature requests, we found that there have been attempts to use (an older version of) our code for at least the following applications:

- Processing large semantic graphs in order to build a scalable parallel data management system.
- Searching in social network graphs.
- A graph visualization project dealing with large graphs.

### 3.10.1 Goals

Our most important goal in engineering these BFS algorithms has been to make BFS viable on massive graphs. Constant factors in the I/O complexity are particularly important in an external memory setting as they can make the difference

between an implementation running overnight and one that takes a month. Whenever we had a trade-off between saving I/Os and more development time, we always chose to optimize our code by saving I/Os. Pipelining involves more development time, makes the code less readable and makes it difficult to debug, but since optimization has been our key priority we continue to rely on it heavily.

Our next goal has been reusability. The extensive use of templates provides a lot of flexibility with respect to using our code in different applications.

Last but not least, reliability is an important consideration. Not only the implementation should result in a correct output, but it should also not terminate before giving the output (e.g., with an error message or segmentation fault). This is particularly important for external memory implementations as they may take hours and days of running. In this context, errors that happen infrequently constitute the main problems. We have put a lot of effort to make this code as bug-free as possible.

## 3.11 Results of our experimental study

In this section, we present the main results of our extensive experimental study with external memory BFS algorithms. For comparing the different algorithms, we consider the total running time and the I/O wait time – the total time spent by an implementation waiting for an I/O to complete, and not I/O time – the total time spent by an implementation on I/Os. This distinction is necessary as STXXL maximizes the overlap of I/O with computation.

The external memory BFS algorithms require hours, days and sometimes even months for computing BFS on various graph classes. As such, some of the results presented in this section (specifically those requiring months) have been interpolated using the symmetry in the graph structure.

### 3.11.1 Configuration

We have implemented the algorithms in C++ using the g++ compiler (optimization level –O3) on the *Debian GNU/Linux* distribution with a Linux kernel and the external memory library STXXL. Table 3.1 summarizes the configuration of the three machines on which we ran our experiments. Note that for chronological reasons, Config A had only partial support for large diameter graphs, a 16-byte edge

| | Config A | Config B | Config C |
|---|---|---|---|
| Processor | Intel | Opteron | Opteron |
| Processor speed | 2.0 GHz | 2.0 GHz | 2.5 GHz |
| Cache | 512 KB | 1 MB | 1 MB |
| RAM | 1 GB | 1 GB | 2.5 GB |
| Disk Model | ST3250823A | ST3250823A | ST3500320AS |
| Disk capacity | 250 GB | 250 GB | 500 GB |
| Disk Buffer cache | 8 MB | 8 MB | 32 MB |
| Disk: Sustained data transfer rate (outer zone) | 65 MBps | 65 MBps | 105 MBps |
| Disk: Average latency | 4.16 msec | 4.16 msec | 4.16 msec |
| Disk: Spindle Speed | 7200 rpm | 7200 rpm | 7200 rpm |
| Disk: Random read seek time | <11.0 msec | <11.0 msec | < 8.5 msec |
| Disk: Random write seek time | <12.0 msec | <12.0 msec | < 9.5 msec |
| Disk: Connecting interface | PATA | PATA | SATA 3Gbps |
| g++ version | 3.3.2 | 4.0.2 | 4.1.2 |
| Linux kernel | 2.4 | 2.6 | 2.6 |
| STXXL version | 0.77 | 0.77 | 1.1.1 |
| STXXL support for large diameter graphs | Partial | Complete | Complete |
| EMBFS Heuristic | No | Yes | Yes |
| MR_BFS edge size | 16 | 8 | 4 |

Table 3.1: Configuration of different machines used for experimenting with EM BFS algorithms.

representation for MR_BFS and no heuristic included in MM_BFS. Also note that with the hard disks used in all of these machines, it takes many hundreds of hours for $2^{28}$ (most common value of $n$ in our experiments) random reads and writes.

The relative performance of different algorithms does not vary much across different architectures. In this section, we therefore present various performance measures on different configurations to illustrate the key features of our results.

### 3.11.2 Fine-tuning Parameters

Most practitioners of external memory algorithms know that the block size $B$ is a parameter that needs to be finely tuned for optimal performance. This is all the more relevant in the STXXL design framework, as the STXXL vector is organized as a collection of blocks (of size $B$) residing on the external storage media (parallel disks). Recall from Section 3.4.1 that access to the external blocks is organized through the fully associative *cache* which consists of a few (Pg_Nr) in-memory pages where a page is a collection of a few ($P$) logically consecutive blocks. Apart from Pg_Nr and $P$, another important parameter to be fine-tuned is the internal memory reserved for a runs creator and a runs merger. While tuning these parameters, a key constraint is that the internal memory allocated for all the vectors, runs creators and runs mergers active simultaneously, at any time, should be less than the main memory available for the user. Typically, half of the main memory is kept for OS requirements. The allocation strategy of blocks over disks in a multi-disk setting and the page replacement policy of a vector cache are some other parameters to be considered. For our implementations, we chose $B = 512$ KB/1 MB (depending on the machine), Pg_Nr $= 4$, $P =$ number of parallel disks in use, allocation strategy = randomized cyclic striping and LRU page replacement strategy.

Another important parameter to be optimized for MM_BFS is $\mu$ which is related to the diameter of the clusters. For worst case optimality, we choose $\mu := \max\left\{1, \sqrt{\frac{n}{\text{scan}(n+m)}}\right\}$ for MM_BFS_D and $\mu := \max\left\{1, \sqrt{\frac{n}{\text{scan}(n+m)\cdot\log(n)}}\right\}$ for MM_BFS_R. On the other hand, if some a priori information is available about the graph structure, one can use it to reduce the random or sequential accesses by appropriately modifying $\mu$. We consider both the cases – one in which we choose our $\mu$ value independent of the graph-structure (common $\mu$) and one in which we assume a priori knowledge of the graph diameter (graph-structure dependent $\mu$).

### 3.11.3 IM_BFS looses fast

Figure 3.13 shows the total running time of IM_BFS, MR_BFS, MM_BFS_R, and MM_BFS_D on random graphs of varying sizes (keeping $m = 4n$) on config A (cf. Table 3.1). An important point to note here (also see Figure 1.1) is that even when half of the graph fits in internal memory, the performance of IM_BFS is much worse than that of the external BFS algorithms. For this case ($2^{22}$ nodes and $2^{24}$ edges), the I/O wait time of IM_BFS (8.09 *hours*) dominates the total

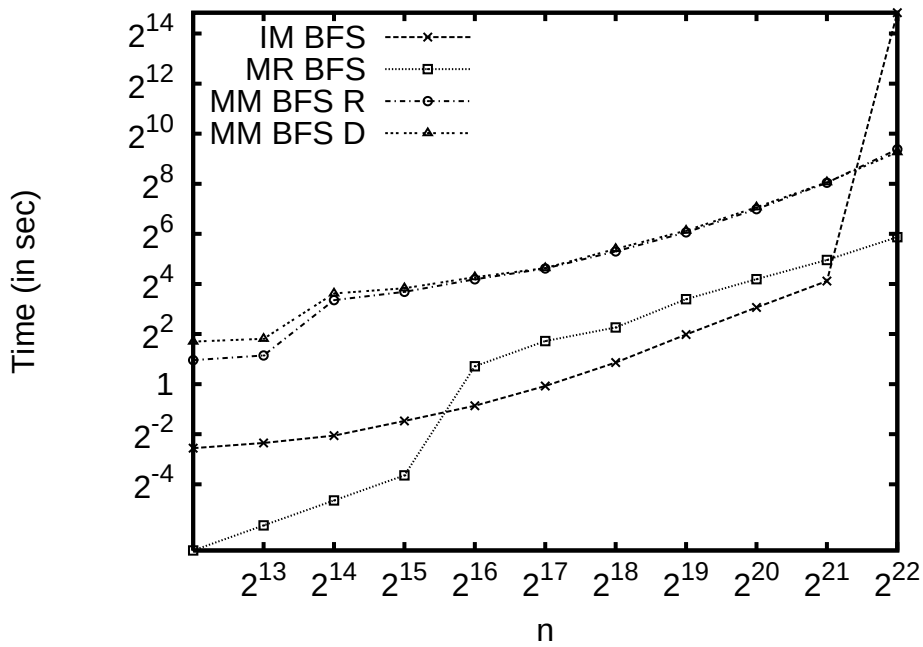



Figure 3.13: Variation of running time of IM_BFS, MR_BFS, MM_BFS_R, and MM_BFS_D (in logarithmic scale) on random graph with $n$ nodes (also in logarithmic scale) and $m = 4n$ edges.

running time (8.11 *hours*), thereby explaining the worse behavior of IM_BFS. On the other hand, MR_BFS, MM_BFS_R, and MM_BFS_D have much less I/O wait time (0.70, 5.15 and 4.36 *minutes* respectively) and consequently, the total running time (0.97, 11.11 and 10.23 *minutes* respectively) is also small. This further establishes the need for efficient implementations of external memory BFS algorithms.

### 3.11.4 Single disk – common $\mu$

Table 3.2 shows the I/O wait time and running time (in hours) for different graphs in the single disk common $\mu$ case. Note that MR_BFS does not use $\mu$ in any way.

First, observe that for these large graphs, even the efficient implementations of external memory algorithms are I/O dominant. This is particularly true for MR_BFS as the I/O wait time for MR_BFS on most graph classes accounts for most of its total running time.

| Graph class | $n$ | $m$ | MR_BFS | | MM_BFS_R | | MM_BFS_D | |
|---|---|---|---|---|---|---|---|---|
| | | | I/O wait Time | Total Time | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Random | $2^{28}$ | $2^{30}$ | 0.9 | 1.0 | 4.5 | 8.0 | 4.4 | 8.3 |
| Webgraph | $\sim 2^{27}$ | $1.4 \cdot 10^9$ | 1.7 | 1.8 | 5.2 | 8.4 | 3.0 | 6.4 |
| 2D-Grid | $2^{28}$ | $\sim 2^{29}$ | 3300 | 3300 | 30.9 | 34.9 | 11.6 | 16.0 |
| 4D-Grid | $2^{28}$ | $\sim 2^{30}$ | 23.5 | 23.6 | 21.1 | 24.9 | 12.4 | 16.5 |
| $B$-level random | $2^{28}$ | $2^{30}$ | 5000 | 5000 | 37.1 | 52.6 | 2.9 | 7.2 |

Table 3.2: I/O wait time and total running time (in hours) of MR_BFS, MM_BFS_R, and MM_BFS_D on various graph classes on Config C.

Let us first consider the case of random graphs. The total time for BFS traversal (particularly MR_BFS) on random graphs is much less than that for most other graph classes. This is explained by the fact that there are very few BFS levels in random graphs (typically 10–15 for the graph sizes we studied). In fact, it is known [135] that a random graph $G(n, c/n)$ has an expected diameter $O(\log n)$. Both MR_BFS and MM_BFS benefit from the low diameter of the graph, though to a different degree.

MR_BFS directly benefits from fewer BFS levels as it incurs $O(\text{sort}(n+m))$ I/Os per level, thus avoiding the expensive $O(n)$ factor. MM_BFS benefits from low diameter as the cluster diameters are small (at least smaller than the graph diameter) and consequently, nodes do not stay in the hot pool for too long. For MM_BFS_R, this also means that the preprocessing time is less. Furthermore, the clusters get loaded in fewer sort steps and as such MM_BFS need not incur $\Omega(1)$ I/Os for loading each cluster. Nonetheless, owing to its more compact data structures and its inherent simplicity, MR_BFS not only outperforms MM_BFS (on low diameter graphs) in terms of I/O wait time by a factor of around five, but also in terms of total running time by a factor of around eight.

While MR_BFS performs better than the other two on random graphs saving a few *hours*, MM_BFS_D with the heuristic outperforms MR_BFS and MM_BFS_R on moderate ($O(\sqrt{n})$ or $O(\sqrt{B})$ diameter) to large ($O(n)$) diameter graphs with a non-simple[3] layout on disk saving a few *months* and a few *days*, respectively. This performance behavior on large diameter graphs is mainly because of the different asymptotic I/O complexities of these algorithms. On ($\lceil\sqrt{n}\rceil \times \lfloor\sqrt{n}\rfloor$) 2D-grid graphs and $B$-level random graphs, MR_BFS incurs close to its worst

[3]By a simple layout of a graph, we mean that the adjacency lists of nodes are kept on the disk sorted according to the BFS level of these nodes.

case I/O complexity of $\Omega(n)$ I/Os for loading the adjacency lists.

Apart from diameter, another important consideration affecting the relative performance of the two algorithms is the initial graph layout on the disk. The preprocessing phase of MM_BFS neutralizes the impact of an adverse layout. So, while we observe that on Config A, the I/O wait time of MR_BFS (0.6 *hours*) is much less than 84.8 *hours* of MM_BFS_R (dominated by the 84.3 *hours* in the preprocessing phase) on a simple list graph, the I/O wait time of MR_BFS (167.6 and 177.7 *days*) is much more than that of MM_BFS (4.2 and 4.1 *days*) on random and $B$-interleaved layouts. Thus, preprocessing makes MM_BFS provide better worst case guarantees (saving *months*) at the cost of loosing out on simple layouts (loosing *days*).

### 3.11.5 Two phases of MM_BFS

Let's analyze the performance of MM_BFS in terms of its two phases. Tables 3.3 and 3.4 show the results of the preprocessing and the BFS phase of the two MM_BFS variants. The preprocessing time of MM_BFS_D only depends on the graph size and not its structure. The I/O wait time for the Euler tour based preprocessing of graphs with around $2^{29}$ edges is around 2 hours, while that for graphs with $2^{30}$ edges is around 2.7 hours. This is because Euler tour computation followed by list ranking only requires $O(\text{sort}(m))$ I/Os independent of the diameter of the graph.

| Graph class | $n$ | $m$ | MM_BFS_R | | MM_BFS_D | |
|---|---|---|---|---|---|---|
| | | | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Random | $2^{28}$ | $2^{30}$ | 2.3 | 3.0 | 2.6 | 3.7 |
| Webgraph | $\sim 2^{27}$ | $1.4 \cdot 10^9$ | 3.7 | 4.3 | 1.9 | 2.6 |
| 2D-Grid | $2^{28}$ | $\sim 2^{29}$ | 5.1 | 5.4 | 2.2 | 3.0 |
| 4D-Grid | $2^{28}$ | $\sim 2^{30}$ | 2.7 | 3.4 | 2.7 | 3.7 |
| $B$-level random | $2^{28}$ | $2^{30}$ | 2.3 | 3.0 | 2.7 | 4.0 |

Table 3.3: I/O wait time and total running time (in hours) of the preprocessing phase of the two MM_BFS variants on Config C.

On the other hand, the "parallel cluster growing" preprocessing in the worst case scans the graph $\Omega\left(\sqrt{\frac{n}{\text{scan}(n+m)}}\right)$ times, and thus incurring $\Omega(\sqrt{n \cdot \text{scan}(n+m)})$

| Graph class | $n$ | $m$ | MM_BFS_R | | MM_BFS_D | |
|---|---|---|---|---|---|---|
| | | | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Random | $2^{28}$ | $2^{30}$ | 2.2 | 5.0 | 1.8 | 4.6 |
| Webgraph | $\sim 2^{27}$ | $1.4 \cdot 10^9$ | 1.5 | 4.1 | 1.1 | 3.8 |
| 2D-Grid | $2^{28}$ | $\sim 2^{29}$ | 25.8 | 29.5 | 9.4 | 13.0 |
| 4D-Grid | $2^{28}$ | $\sim 2^{30}$ | 18.4 | 21.5 | 9.7 | 12.8 |
| $B$-level random | $2^{28}$ | $2^{30}$ | 34.8 | 49.6 | 0.2 | 3.2 |

Table 3.4: I/O wait time and total running time (in hours) of the BFS phase of the two MM_BFS variants on Config C.

| Graph class | $n$ | $m$ | MM_BFS_R | MM_BFS_D |
|---|---|---|---|---|
| Random graph | $2^{28}$ | $2^{30}$ | 500 | 630 |
| Random List | $2^{28}$ | $2^{28}-1$ | 10500 | 480 |

Table 3.5: I/O volume (in GB) required in the preprocessing phase by the two variants of MM_BFS on Config A.

I/Os w.h.p. But if the diameter of the graph is small, no two nodes in a cluster are further than the diameter and hence, MM_BFS_R needs to scan the graph fewer times. Thus, while the I/O volume of the "parallel cluster growing" preprocessing on random graphs is around 500 GB, it is more than 10.5 TeraBytes on a random list graph (cf. Table 3.5). As for MM_BFS_D preprocessing, the I/O volume is less for the random list graph because it has fewer number of edges. Therefore, while the preprocessing time increases for MM_BFS_R from 3.0 hours for random graphs to 4.8 hours on a $O(\sqrt{n})$ diameter square grid graph, it decreases from 3.7 hours to 2.6 hours for MM_BFS_D (cf. Table 3.3).

Except for some special cases, BFS phase dominates the running time of MM_BFS. The BFS phase itself is a balance between the random I/Os to load the clusters into the hot pool and the sequential I/Os to update and scan the hot pool. For small diameter graphs, we do not need $\Omega(1)$ random I/Os to load a cluster. All clusters are loaded in a span of a few BFS levels and the cost for this is thus, subsumed by I/Os required to scan the graph a few times. Hence in this case, the scanning of hot pools dominate the running time. For large diameter graphs, the hot pool almost always fits in the internal memory and no I/Os are required to scan it. MM_BFS also significantly benefits from our heuristics in this case. So for large diameter graphs, the random I/Os to load the clusters dominates the running time of the BFS phase. The moderate diameter graphs are the key challenge for the BFS phase as here, we need to incur I/Os both for loading the clusters and for

scanning the hot pool.

As compared to MM_BFS_R, MM_BFS_D provides dual advantages: First, the preprocessing itself is faster and second, for most graph classes, the partitioning is also more robust, thus leading to better worst-case running-times (cf. Table 3.4) in the BFS phase. The later is because the clusters generated by Euler tour based preprocessing are of diameter at most $\max\left\{1, \sqrt{\frac{n}{\text{scan}(n+m)}}\right\}$, while the ones generated by "parallel cluster growing" preprocessing can have a larger diameter of $O\left(\sqrt{\frac{n \cdot \log n}{\text{scan}(n+m)}}\right)$ causing adjacency lists to be scanned more often. Also, MM_BFS_D benefits much more from our caching heuristic than MM_BFS_R as Euler tour based preprocessing gathers neighboring clusters of the graph on contiguous locations in the disk.

### 3.11.6 Effect of Disk parallelism

| Graph class | $n$ | $m$ | MR_BFS | | MM_BFS_R | |
|---|---|---|---|---|---|---|
| | | | Single Disk | Four Disks | Single Disk | Four Disks |
| Random | $2^{28}$ | $2^{30}$ | 3.4 | 1.3 | 9.6 | 4.4 |
| $B$-level Random | $2^{28}$ | $2^{30}$ | 3994.8 | 2105.1 | 49.7 | 26.0 |
| $B$-level Spider Web | $2^{28}$ | $\sim 2^{29}$ | 3366.5 | 1497.9 | 39.8 | 17.1 |
| MM Worst | $2^{25}$ | $\sim 2^{25}$ | 25.4 | 13.7 | 32.4 | 10.5 |
| Random list | $2^{28}$ | $2^{28}$ | 4167.7 | 4156.2 | 283.3 | 239.9 |
| $B$-interleaved list | $2^{28}$ | $2^{28}$ | 4222.6 | 1258.7 | 280.8 | 239.9 |

Table 3.6: The running times (in hours) of MR_BFS and MM_BFS_R on Config A in the single-disk and multi-disk settings.

| Graph class | $n$ | $m$ | Phase 1 | | Phase 2 | |
|---|---|---|---|---|---|---|
| | | | Single Disk | Four Disks | Single Disk | Four Disks |
| Random | $2^{28}$ | $2^{30}$ | 5.1 | 2.5 | 4.5 | 1.9 |
| $B$-level random | $2^{28}$ | $2^{30}$ | 5.1 | 2.5 | 44.6 | 23.5 |
| $B$-level Spider Web | $2^{28}$ | $\sim 2^{29}$ | 7.3 | 3.2 | 32.5 | 13.9 |
| Random list | $2^{28}$ | $2^{28}$ | 80.4 | 50.5 | 200.4 | 189.4 |

Table 3.7: The running times (in hours) of the two phases of MM_BFS_R on Config A in the single-disk and multi-disk settings.

In the multi-disk setting, we ran our experiments with the same parameters, except that the vectors are randomly striped over four disks. Although the usage of multiple disks allows us to handle larger volumes of data, herein we restrict ourselves to smaller sizes for better comparison with the single disk case.

As Tables 3.6 and 3.7 show, the usage of parallel I/O channels alleviates the I/O problem further. In general, we see a performance improvement by a factor of two to three with four disks as compared to the single-disk case. For many graphs, the computation time starts becoming the bottleneck, in particular for MM_BFS, which seems to gain more from the parallel I/O channels. However, with some new features of STXXL like a SMP multi-processor version of sorting routines, we hope to bring down the total running time fairly close to the I/O wait time. Besides, the computation speed increases at a much faster rate than the external memory throughput, thereby reducing the computation time relative to the I/O wait time.

While MR_BFS on random list graphs hardly seems to have any benefit from the multiplicity of disks, it is almost four times better with four disks on $B$-interleaved list graphs. This is because a random access to a block brings the neighboring blocks on other disks automatically to the internal memory and therefore, the access to the adjacency lists of the next three nodes (located on the other three disks) comes without any extra I/Os.

### 3.11.7 Exploiting a priori information about graph diameter

Recall from Section 3.11.5 that the BFS phase of MM_BFS for small diameter graphs is dominated by sequential accesses to the hot pool and for large diameter graphs is dominated by the random I/Os for loading the clusters. Since we choose $\mu$ to balance the random I/Os to load the clusters and sequential accesses to the hot pool, it makes sense to choose a very low value of $\mu$ for small diameter graphs (to ensure that an adjacency list stays for a really short time in hot pool) and very high value of $\mu$ for large diameter graphs (as the hot pool stays internal and we want to reduce the random I/Os to load the clusters).

Tables 3.8 and 3.9 show the I/O wait time and running time for the two algorithms in the single disk case, where $\mu$ could be optimized based on the graph structure. With a low value of $\mu$ ($\mu \sim 1.5$), the I/O wait time and the total running-time of the BFS phase of MM_BFS_R on random graphs is less than that of MR_BFS on Config A. In general, with an appropriate $\mu$ value chosen to balance the I/O

| Graph class | $n$ | $m$ | MR_BFS | | MM_BFS_R | |
|---|---|---|---|---|---|---|
| | | | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Random | $2^{28}$ | $2^{30}$ | 2.4 | 3.4 | 5.5 | 7.9 |
| $B$-level Random | $2^{28}$ | $2^{30}$ | 3989.8 | 3994.8 | 10.0 | 16.6 |
| $B$-level Spider Web | $2^{28}$ | $\sim 2^{29}$ | 3364.2 | 3366.5 | 25.1 | 29.3 |

Table 3.8: Single Disk, Graph structure dependent $\mu$ – I/O wait time and running time (in hours) of MR_BFS and MM_BFS on Config A.

| Graph class | $n$ | $m$ | MM_BFS_R Phase 1 | | MM_BFS_R Phase 2 | |
|---|---|---|---|---|---|---|
| | | | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Random | $2^{28}$ | $2^{30}$ | 3.3 | 4.9 | 2.2 | 3.0 |
| $B$-level Random | $2^{28}$ | $2^{30}$ | 4.0 | 5.5 | 6.0 | 11.1 |
| $B$-level Spider Web | $2^{28}$ | $\sim 2^{29}$ | 12.9 | 13.7 | 12.2 | 15.6 |

Table 3.9: Single Disk, Graph structure dependent $\mu$ – I/O wait time and running time (in hours) of the two phases of MM_BFS on Config A.

time of the two phases of MM_BFS, one can save a significant factor in the I/O complexity. Our experiments with graph-dependent $\mu$ and disk parallelism suggest that when used together, they can significantly alleviate the I/O bottleneck for MM_BFS.

### 3.11.8 Results on the webgraph

| | MR_BFS | | MM_BFS_R Common $\mu$ | | MM_BFS_R Graph dep $\mu$ | |
|---|---|---|---|---|---|---|
| | I/O wait Time | Total Time | I/O wait Time | Total Time | I/O wait Time | Total Time |
| Single disk | 3.7 | 4.0 | 7.4 | 9.4 | 6.3 | 8.4 |
| Multiple disk | 2.0 | 2.3 | 2.7 | 4.8 | 2.3 | 4.5 |

Table 3.10: I/O wait time and running time (in hours) of the two algorithms on a web graph on Config A.

As an instance of a real world graph, we consider an actual crawl of the world wide web [150], where an edge represents a hyper-link between two sites. This

graph has around 130 million nodes and 1.4 billion edges. The bulk of the nodes are contained in the core of this web graph spanning 10–12 BFS levels (similar to random graphs). The remaining nodes are spread out over thousands of levels with 2–3 nodes per level (which behaves more like a list graph). However, the I/O wait time as well as the total running time for BFS traversal is dominated by the core of this graph and hence, the results are similar to the ones for random graphs. As Table 3.10 shows for Config A, both MR_BFS and MM_BFS can compute the BFS decomposition of this graph in a matter of a few *hours*. In fact as Table 3.2 shows, MR_BFS requires merely 1.8 hours on Config C with a single disk, owing to its more compact edge-representation there. Similar to random graphs, MR_BFS outperforms both MM_BFS_R and MM_BFS_D on webgraph.

### 3.11.9 Penalty for cache-obliviousness

Brodal et al. [40] gave a cache-oblivious undirected BFS algorithm (CO_BFS) that has a complexity of $O(\text{sort}(m) + (m/B) \cdot \log n + \sqrt{n \cdot m/B} + ST(n,m))$ I/Os, where $ST(n,m)$ is the complexity of computing a spanning tree of a graph with $n$ nodes and $m$ edges in a cache-oblivious way. The currently best cache-oblivious algorithms for computing a spanning tree require $O(\text{sort}(m) \cdot \log\log(n))$ I/Os deterministically and $O(\text{sort}(m))$ I/Os randomized.

Christiani [49] gave a prototypical cache-oblivious implementation of MR_BFS and the preprocessing phase of MM_BFS_R and MM_BFS_D. These implementations use cache-oblivious algorithms for sorting, minimum spanning tree and list ranking. In this subsection, we provide evidence that even though the cache-oblivious BFS algorithms have the same asymptotic I/O complexity as their external memory counterparts, they are slower in practice for graphs that do not fit in the main memory.

#### Sorting

While CO_SORT provides tight asymptotic guarantees on all levels of memory hierarchy, it is a factor three to four slower than STXXL_SORT in practice for data-sizes that do not fit in the main memory. Our results shown in Table 3.11 are in conformity with that of Brodal et al. [41], where it is shown that the external memory sorting algorithm in the library TPIE [151] is better than their carefully implemented cache-oblivious sorting algorithm, when run on disk.

| $n$ | CO_SORT | STXXL_SORT |
|---|---|---|
| $256 \times 10^6$ | 21 | 8 |
| $512 \times 10^6$ | 46 | 13 |
| $1024 \times 10^6$ | 96 | 25 |

Table 3.11: Timing in minutes for sorting $n$ elements using either CO_SORT or STXXL_SORT on Config B.

| Graph class | CO_MST | EM_MST |
|---|---|---|
| Random graph; $n = 2^{28}$, $m = 2^{30}$ | 107 | 35 |
| List graph with contiguous disk layout (Simple List); $n = 2^{28}$ | 38 | 16 |
| List graph with random disk layout (Random List); $n = 2^{28}$ | 47 | 22 |

Table 3.12: Timing in hours (on Config B) required by Euler tour based preprocessing of Christiani's implementation using either CO_MST or EM_MST.

### Spanning forest

The Euler tour based preprocessing of Christiani [49] uses the cache-oblivious MST (CO_MST) algorithm [2]. Table 3.12 shows the total time required by Christiani's MM_BFS_D preprocessing [49] using CO_MST and the one in which CO_MST is replaced by the external memory MST implementation (cf. Section 3.6.1).

### List ranking and Euler tour

The cache-oblivious implementation [49] uses the algorithm based on independent set removal [48] for list ranking. While it takes around 14.3 *hours* for ranking $2^{29}$ element random list using 3 GB RAM on Config B, our adaptation of Sibeyn's algorithm (cf. Section 3.6.2) takes less than 40 *minutes* in the same setting.

### MM_BFS_D comparison

We compared the performance of our implementation of MM_BFS_D with Christiani's implementation [49] based on cache-oblivious subroutines. Table 3.13 show the preprocessing time for the two extreme graph classes – random graphs

and list graphs with random layout on disk. We observe that on both graph classes, the preprocessing time required by our implementation is significantly less than the one by Christiani.

| Graph class | $n$ | $m$ | CO_BFS | MM_BFS_D |
|---|---|---|---|---|
| Random graph | $2^{28}$ | $2^{30}$ | 107 | 5.2 |
| Random List | $2^{28}$ | $2^{28}-1$ | 47 | 3.2 |

Table 3.13: Timing in hours for computing Euler tour based preprocessing of MM_BFS by the two implementations of MM_BFS_D on Config A.

We suspect that the performance losses in Christiani's CO_BFS implementations are inherent in cache-oblivious algorithms to a certain extent and will be carried over to any cache-oblivious BFS implementation.

### 3.11.10 Remark on the shape of the spanning tree

The shape of the computed spanning tree can have a significant impact on the clustering and the disk layout of the adjacency list after Euler tour based preprocessing, and consequently on the BFS phase. For instance, in the case of the square grid graphs, a spanning tree containing a list with elements in a snake-like row major order produces long and narrow clusters, while a "random" spanning tree is likely to result in clusters with low diameters. Such a "random" spanning tree can be obtained by assigning random weights to the edges of the graph and then computing a minimum spanning tree or by randomly permuting the indices of the nodes. The nodes in the long and narrow clusters tend to stay longer in the pool and therefore, their adjacency lists are scanned more often. This causes the pool to grow external and results in larger I/O volume. On the other hand, low diameter clusters are evicted from the pool sooner and are scanned less often reducing the I/O volume of the BFS phase. Consequently as Table 3.14 shows, the BFS phase of MM_BFS_D takes only 28 hours on Config B with clusters produced by "random" spanning tree, while it takes 51 hours with long and narrow clusters.

| Graph class | $n$ | $m$ | Long clusters | Random clusters |
|---|---|---|---|---|
| Grid($2^{14} \times 2^{14}$) | $2^{28}$ | $2^{29}$ | 51 | 28 |

Table 3.14: Time taken (in hours) by the BFS phase of MM_BFS_D with long and random clustering on Config B.

### 3.11.11 Summary

| Graph class | $n$ | $m$ | MR_BFS | MM_BFS_R | MM_BFS_D |
|---|---|---|---|---|---|
| Random | $2^{28}$ | $2^{30}$ | **1.4** | $7\times$ | $6\times$ |
| Webgraph | $\sim 2^{27}$ | $1.4 \cdot 10^9$ | **2.6** | $3.5\times$ | $2\times$ |
| Grid $(2^{14} \times 2^{14})$ | $2^{28}$ | $2^{29}$ | $2.5\times$ | $1.25\times$ | **21** |
| Grid $(2^{21} \times 2^{7})$ | $2^{28}$ | $\sim 2^{29}$ | $>100\times$ | $>10\times$ | **4.0** |
| Grid $(2^{27} \times 2)$ | $2^{28}$ | $\sim 2^{28} + 2^{27}$ | $>500\times$ | $>25\times$ | **3.8** |
| Simple List | $2^{28}$ | $2^{28} - 1$ | **0.4** | $7\times$ | $7\times$ |
| Random List | $2^{28}$ | $2^{28} - 1$ | $>1300\times$ | $>75\times$ | **3.6** |
| Max | | | $\sim 1/2$ **year** | $\sim 1$ **week** | $< 1$ **day** |

Table 3.15: The best total running time (in hours) for BFS traversal on different graphs on Config B with the best external memory BFS implementations; Entries like $> 25\times$ denote that this algorithm takes more than 25 times the time taken by the best algorithm for this input instance.

Table 3.15 points to the current state of the art implementations of external memory BFS on different graph classes (on Config B). Our MR_BFS implementation outperforms the other external memory BFS implementations on low diameter graphs or when the nodes of a graph are arranged on the disk in the order required for BFS traversal. For random graphs with 256 million nodes and a billion edges, MR_BFS performs BFS in just 1.4 hours. Similarly, MR_BFS takes only 2.6 hours on webgraphs (whose runtime is dominated by the short diameter core) and 0.4 hours on list graph with contiguous layout on disk. For large diameter graphs like random list graphs, MM_BFS_D along with our heuristic computes the BFS in just about 3.6 *hours*, which would have taken MR_BFS a few *months*, an improvement by a factor of more than 1300. In general, if there is no a priori information about the graph structure or its layout on the disk, one should use MM_BFS_D as it has better asymptotic worst case guarantee.

## 3.12 Recent work related to EM BFS

In this section, we review some recent work related to external memory BFS. Meyer and Osipov [111] have extended our work to external memory single-source shortest paths (SSSP). We briefly review this extension together with other known results on EM SSSP in Section 3.12.1. Meyer recently proposed algorithms for dynamic BFS [110] and approximating the diameter of an undirected

graph [109]. We review these algorithms in Section 3.12.2 and Section 3.12.3, respectively. We believe that the key ideas and the code in our work can also easily be extended to implement dynamic BFS and to approximate the diameter. Efficiently approximating the diameter of the graph can in turn help determine which BFS algorithm to use and with what parameters.

### 3.12.1 Single-Source Shortest Paths

The single-source shortest paths (SSSP) problem takes as an input a large weighted undirected graph $G(V,E)$ and a source node $s$ and computes the shortest path distance $d(s,v)$ for all nodes $v \in V$. It can be computed in $O(n \log n + m)$ in internal memory using Dijkstra's algorithm [61] with a Fibonacci heap [71] based priority queue. Dijkstra's algorithm relies heavily on the priority queue.

Kumar and Schwabe proposed an $O(n + m/B \cdot \log_2 (m/B))$ I/O algorithm [93] that relies on I/O-efficient tournament trees for priority queue operations. Once again, the $O(n)$ term comes from unstructured accesses to adjacency lists and because of it, this algorithm is unlikely to yield good results on real-world massive graphs, which are usually sparse. Furthermore, due to edge weights, there are typically many more "levels".

As regards resolving the problem of unstructured accesses to adjacency lists, Meyer and Zeh [112] proposed an algorithm MZ_SSSP that has a preprocessing phase where the adjacency lists are re-arranged on the disk. Unlike BFS where the edges are all unweighted, MZ_SSSP distinguishes between edges with different weights and separates the edges into categories based on their weights. The total I/O complexity of this algorithm is $O(\sqrt{(n \cdot m \cdot \log W)/B} + MST(n,m))$ I/Os, where $W$ is the ratio between the weights of the heaviest and the lightest edge and $MST(n,m)$ is the number of I/Os required to compute a minimum spanning tree of a graph with $n$ nodes and $m$ edges.

Meyer and Zeh [113] extended this framework to handle the case of unbounded edge-weights. Their algorithm for SSSP with unbounded edge-weights requires $O((\sqrt{n \cdot m/B}) \cdot \log n + MST(n,m))$ I/Os.

Brodal et al. [40] showed that SSSP can be computed in $O(n + \text{sort}(m))$ I/Os with a cache-oblivious algorithm relying on a cache-oblivious bucket heap for priority queue operations. Allulli et al. [14] gave a cache-oblivious SSSP algorithm improving the upper bound to $O(\sqrt{(n \cdot m \cdot \log W)/B} + (m/B) \cdot \log n + \text{sort}(m) + MST(n,m)$, where $W$ is the ratio between the smallest and the largest edge weight and $MST(n,m)$ is the I/O complexity of the cache-oblivious algorithm computing

a minimum spanning tree of a $n$ node and $m$ edge graph.

### Engineering EM SSSP

Recently, some external memory SSSP approaches (similar in nature to the one proposed in [93]) have been implemented [46, 138] and tested on graphs of up to 6 million nodes. However, in order to go external and still not produce huge running times for larger graphs, these implementations restrict the main memory size to rather unrealistic 4 to 16 MB.

Meyer and Osipov [111] extended our work to engineer a practical I/O-efficient single-source shortest-paths algorithm on general undirected graphs where the ratio between the largest and the smallest edge weight is reasonably bounded. Their implementation is semi-external as it assumes that the main memory is big enough to keep some constant bits of information per node. This assumption allows them to use a bit vector of size $n$ kept in the internal memory for remembering settled nodes.

In order to get around the lack of optimal decrease_key operation in current external memory priority queues, it allows up to $d(v)$ (degree of node $v$) many entries for a node $v$ in the priority queue at the same time and when extracting them, it discards all but the first one with the help of the bit vector. As regards accessing the adjacency lists in an unstructured way, they do a preprocessing similar to the Euler tour based variant of MM_BFS (i.e., without considering the edge weights at all) to form clusters of nodes. For integer edge weights from $\{1,\ldots,W\}$ and $k = \log_2 W$, the algorithm keeps $k$ "hot pools" where the $i$-th pool is reserved for edges of weight between $2^{i-1}$ and $2^i - 1$. It loads the adjacency lists of all nodes in a cluster into these "hot pools" as soon as the first node in the cluster is settled.

In order to relax the edges incident to settled nodes, the hot pools are scanned and all relevant edges are relaxed. The algorithm crucially relies on the fact that the relaxation of large weight edges can be delayed because for such an edge (even assuming that it is in the shortest path), it takes some time before the other incident node needs to be settled. The hot pools containing higher weight edges are thus touched less frequently than the pools containing short edges.

Similar to the implementation of MM_BFS, it partially maintains the pool in the internal memory hash table for efficient dictionary look up rather than computationally quite expensive scanning of all hot pool edges. The memory can be shared between "hot pools" either uniformly or in an exponentially decreasing way. The latter makes sense as the hot pools with lighter edges are scanned more often.

When the clusters are small enough, the algorithm caches all neighboring clusters that are anyway loaded into the main memory while reading $B$ elements from the disk.

For random edge weights uniformly distributed in $[1,\ldots,W]$, the expected number of I/Os incurred by this algorithm is $O(\sqrt{(n\cdot m\cdot \log W)/B}+MST(n,m))$, the same as that for MZ_SSSP.

Similar to our implementations, their pipelined implementation makes extensive use of STXXL algorithms and data structures such as stream sorting.

#### SSSP in practice

As predicted theoretically, this SSSP approach is acceptable on graphs with uniformly distributed edge weights. For random graphs ($2^{28}$ nodes and $2^{30}$ edges) with uniformly random weights in $[1,\ldots,2^{32}]$, it requires around 40 hours to compute SSSP (with 1 GB RAM). On a US road network graph with around 24 million nodes and around 29 million edges, it requires only around half an hour for computing SSSP, even when the node labels are randomly permuted before. On many difficult graph classes for BFS, the running time of this SSSP approach is within a factor of two to the BFS implementation [106].

The final performance of this algorithm has been shown to be significantly dependent on the quality of the spanning tree and the way space is allocated in the main memory among different "hot pools".

### 3.12.2 Dynamic BFS in external memory

In many real-world applications, the underlying input graph keeps on evolving continuously (cf. Section 1.1). Since even the best of the carefully tuned implementations of external memory graph algorithms usually take hours and days of time (for massive graphs), it is difficult to re-compute everything from scratch every time there is any modification in the input graph.

Very few results are known for dynamic graph algorithms in external memory. Meyer [110] shows an interesting result for computing BFS on general undirected graphs in incremental or decremental setting. They prove an amortized high-probability bound of $O(n/B^{2/3}+\text{sort}(n)\cdot \log B)$ I/Os per update under a sequence of either $\Theta(n)$ edge insertions, but no deletions or $\Theta(n)$ edge deletions, but no

insertions.

Recall that the deterministic preprocessing in the static BFS [106] works by computing an Euler tour around a spanning tree of the input graph and dividing it into chunks of size $\mu$ where $1 < \mu = O(\sqrt{B})$. The nodes belonging to different clusters can be assigned to any of them. This can potentially cause many clusters with $O(1)$ adjacency lists. For dynamic BFS, this is modified such that each cluster (except possibly the last) contains an expected $\Omega(\mu)$ nodes. This is done by exploiting the following observation: In the sequence of nodes in the Euler tour of a spanning tree, an intermediate visit of a node is directly preceded by the last visit to one of its children and followed by the first visit to some other child. This means that in any chunk half of the nodes are either the first or the last visit of a node. Thus, if rather than assigning nodes to different chunks arbitrarily (as in static BFS), we make a node belong to the chunk corresponding to its first and last visit each with probability one half, the expected number of adjacency lists per cluster will be at least $\mu/8$.

For the BFS phase, lets consider the insertion of the $i$th edge $(u,v)$ in incremental setting and refer to the graph (and the shortest path distances from the source in the graph) before and after the insertion of this edge as $G_{i-1}$ ($d_{i-1}$) and $G_i$ ($d_i$). We first run an external memory connected component algorithm in order to check if the insertion of $(u,v)$ enlarges the connected component $C_s$ of the source node $s$. If so, we run the MR_BFS algorithm on the nodes in the new component starting from node $v$ (assuming w.l.o.g. that $u \in C_s$) and add $d_i(u)+1$ ($d_i(u) = d_{i-1}(u)$ in this case) to all the distances obtained.

Otherwise, we run the BFS phase of $MM\_BFS$, with the difference that the adjacency list for $v$ is added to $H$ when creating BFS level $\max\{0, d_{i-1}(v) - \alpha\}$ of $G_i$, for a certain advance $\alpha > 1$. For nodes with $d_{i-1}(v) - d_i(v) > \alpha$, we import the whole clusters containing their adjacency lists into $H$ using random I/Os. If it requires more than $\alpha \cdot n/B$ random cluster accesses, we increase $\alpha$ by a factor of two, compute a new clustering for $G_{i-1}$ with larger chunk size and start a new attempt by repeating the whole approach with the increased parameters.

The decremental version is similar, except that rather than advancing the adjacency lists, we let them be in hot pool for $\alpha$ BFS levels. For nodes $v$ with $d_i(v) - d_{i-1}(v) > \alpha$, we use random I/Os to get the cluster containing $v$'s adjacency list later on.

The analysis relies on the fact that there can be only be very few updates in which the BFS levels change significantly for a large number of nodes. As such, most of the updates will require few random I/Os in early attempts with little advance.

We believe that our work can easily be extended to engineer an implementation of this algorithm.

### 3.12.3 External memory approximation graph algorithms

One of the major approximation challenges in external memory graph traversal has been to compute approximate diameter of an undirected sparse graph in $o(n/\sqrt{B})$ I/Os. For unweighted graphs, BFS from an arbitrary node already gives a 2-approximation to the diameter of a connected graph. As noted in Section 3.3, BFS on undirected sparse graphs ($m = O(n)$) can be computed in external memory in $O(n/\sqrt{B} + \text{sort}(n))$ I/Os.

Recently, Meyer [109] proposed an algorithm that computes an expected $O(\sqrt{k})$-approximation for the diameter of a sparse undirected and unweighted graph with $n$ nodes and $m = O(n)$ edges using $O(n \cdot \sqrt{\log k/(k \cdot B)} + k \cdot \text{scan}(n) + \text{sort}(n))$ I/Os. This is done by reducing this problem to that of computing exact shortest paths on a graph $G'$ with $O(n/k)$ nodes and $O(m)$ edges.

Graph $G'$ is computed using a preprocessing similar to the "parallel cluster growing" variant of MM_BFS as follows: We first choose each node to be a master node with a probability $1/k$. Then, we select every $k$-th node in the Euler-tour traversal around an arbitrary spanning tree of $G$, to also be a master node. Thereafter, we grow the clusters "in parallel". In each round, each master node tries to capture all unvisited neighbors of the current cluster. This is done by first sorting the nodes at the fringes of the clusters and then scanning the adjacency-lists of the nodes in the yet unexplored graph. Ties are broken arbitrarily.

Let $C(u)$ be the cluster containing $u$. An edge $\{u,v\} \in G$ results in an edge $\{C(u),C(v)\} \in G'$ if $C(u) \neq C(v)$. The weight of the created edge $\{C(u),C(v)\}$ is $d_c(u) + 1 + d_c(v)$, where $d_c(u)$ is the distance of $u$ from its cluster center. We remove the parallel edges by keeping only the lightest edge between $C(u)$ and $C(v)$.

We run single source shortest path from an arbitrary node $s$ in $G'$ and output the maximum distance from $s$ to any other node in $G'$. Note that this is a 2-approximation to the weighted diameter of $G'$. It can easily be shown that the weighted diameter of $G'$ $D_{G'}$ is more than $D_G$. Next, in order to show that $D_{G'}$ is a $O(\sqrt{k})$ approximation of the diameter of $G$ ($D_G$), we consider two cases:

- $D_G \leq 2\sqrt{k}$. Consider any edge $(u,v)$ replaced by $\{C(u),C(v)\}$ in $G'$. The shortest path between any two nodes in $G$ is at most $2\sqrt{k}$ and therefore,

$d_c(u) \leq 2\sqrt{k}-1$ for any node $u \in G$. The weight of $\{C(u), C(v)\}$ is at most $4\sqrt{k}-1$. The weighted diameter of $G'$ can thus be at most $(4\sqrt{k}-1) \cdot D_G$.

- $D_G > 2\sqrt{k}$. Consider any path $P \in G$ of length $p$ such that $\sqrt{k} \leq p \leq 2\sqrt{k}$ and consider $u \in P$ such that $d_c(u)$ is minimum. Note that for all $v \in P$, $d_c(v) \leq d_c(u) + d(u,v)$, as otherwise the master node of $u$ can also capture $v$ during the cluster growing phase. Also, if $C(u) \neq C(v)$, there have to be node disjoint paths from $u$ and $v$ to the cluster centers. Consider larger and larger neighborhoods around $P$ until we find the first level with a cluster center at distance $d_c(u)$. Since each node has been chosen to be a cluster center with uniform probability $1/k$, the expected number of nodes we have to check till reaching the first cluster center is $k$. Recall that each edge $\{u,v\} \in G$ leads to an edge $\{C(u), C(v)\} \in G'$ with weight $d_c(u) + d_c(v) + 1$. Thus, there should exist a path $P' \in G'$ with expected weight $O(k)$.
  For longer paths $P_l \in G$ of length $p_l$, we consider sub-paths of length $\Theta(\sqrt{k})$. For each such sub-path the corresponding path $P' \in G'$ has expected weight $O(k)$. Using linearity of expectation, we can show that the corresponding path $P'_l \in G'$ has expected weight $\sqrt{k} \cdot p_l$. This implies that the weighted diameter of $G'$ can be at most $\sqrt{k} \cdot D_G$.

Since each $k$-th node on the Euler tour is a master node, each node $u \in G$ is at most distance $k$ away from a master node and the clusters are grown for at most $k$ rounds. Each cluster growing round requires $O(\text{scan}(m))$ I/Os to scan the adjacency lists of the unexplored graph and each node appears only once as a fringe node of some cluster leading to a total of $O(\text{sort}(n))$ I/Os. Thus, the total complexity of computing $G'$ is $O(k \cdot \text{scan}(n+m) + \text{sort}(n+m) + ST(n,m))$ I/Os, where $ST(n,m)$ is the I/O complexity of computing a spanning tree of an $n$ node and $m$ edge undirected graph. Computing single source shortest path on a graph with $O(n/k)$ nodes and $O(m)$ edges with the ratio between maximum and minimum edge weight being $k$ requires $O(\sqrt{(n \cdot m \cdot \log_2 k)/(k \cdot B)} + \text{sort}(n+m) + ST(n,m))$ I/Os. The total I/O complexity for this algorithm is thus $O(\sqrt{(n \cdot m \cdot \log_2 k)/(k \cdot B)} + k \cdot \text{scan}(n+m) + \text{sort}(n+m) + ST(n,m))$ I/Os.

We believe that our implementation can be extended to approximate the graph diameter using the above algorithm.

### I/O-efficient heuristics for approximating graph diameters

Brudaru [42] implemented a heuristic to I/O-efficiently approximate the number of BFS levels from a given source node $s$ in a large undirected unweighted graph.

This heuristic first computes an arbitrary spanning tree $T$ of the undirected graph, roots it at the source node by computing $d_T(s,v)$ for all $v \in V$ and then iteratively computes a new tree $T'$ such that $d_{T'}(s,v) \leq d_T(s,v)\ \forall v \in V$. These iterations come in the following variants:

- "Offline variant": For each $v \in V$ and $\{u,v\} \in E$, we compute $\min_u\{d_T(s,u)+1\}$ and if it is less than $d_T(s,v)$, we mark the edge $\{v,P(v)\}$ ($P(v)$ being the parent of $v$ in rooted $T$) for deletion and the edge leading to the shortest distance for insertion in the next iteration.

- "Online variant": In this variant, as the new tree is being computed and $d_T(s,v)$ reduces, this information is communicated to the neighbors that will be processed ahead (without waiting for the round to finish). We use time-forward processing (cf. Section 2.5.6) to do this communication. We first process the nodes in increasing order of $d_T(s,v)$ and then in decreasing order of $d_T(s,v)$.

Both of these variants are shown to converge fast to a BFS tree. While the "offline variant" requires less time per iteration, it may need a higher number of iterations. Empirical evidence [42] suggests that just one round of iterations is enough to determine whether the number of BFS levels is $O(\log n)$, $O(\sqrt{n})$ or $O(n)$.

This is particularly useful for our BFS software. If we can quickly determine the number of BFS levels to be $O(\sqrt{\frac{n}{\text{scan}(n+m)}})$, we can either use MR_BFS or MM_BFS_D with $\mu := \frac{n}{diam_{app}(G)\cdot\text{scan}(n+m)}$, where $diam_{app}(G)$ is the approximated diameter of the graph. Recall that for each level, MR_BFS incurs $O(\text{sort}(m))$ I/Os and therefore, MR_BFS requires $O(diam(G)\cdot\text{sort}(m))$ I/Os. Similarly for MM_BFS_D, the I/O complexity (cf. Section 3.3) is $O(n/\mu + \mu\cdot\text{scan}(n+m) + \text{sort}(n+m))$. The term $\mu\cdot\text{scan}(n+m)$ comes from the fact that each edge may be scanned $O(\mu)$ times. However, no edge can be scanned more often than the total number of BFS levels ($O(diam(G))$). Thus, the total complexity becomes $O(n/\mu + diam(G)\cdot\text{scan}(n+m) + \text{sort}(n+m))$. Substituting $\mu = \frac{n}{diam_{app}(G)\cdot\text{scan}(n+m)}$ and assuming $diam(G) = O(diam_{app}(G))$, we get a complexity of $O(diam(G)\cdot\text{scan}(n+m) + \text{sort}(n+m))$ I/Os. On the other hand, if the diameter is $\Omega\left(\sqrt{\frac{n}{\text{scan}(n+m)}}\right)$, we can use MM_BFS_D with the worst-case value of $\mu := \max\{1, \sqrt{\frac{n}{\text{scan}(n+m)}}\}$ for a total I/O complexity of $O(\sqrt{n\cdot\text{scan}(n+m)} + \text{sort}(n+m))$.

## 3.13 Discussion

| Problems | Best known upper bounds |
|---|---|
| MST/CC (on undirected graphs) | $O(\text{sort}(m) \cdot \log\log(n \cdot B/m))$ |
| MST/CC (randomized on undirected graphs) | $O(\text{sort}(m))$ |
| List ranking | $O(\text{sort}(m))$ |
| Euler Tour | $O(\text{sort}(m))$ |
| BFS (on undirected graphs) | $O(\sqrt{n \cdot \text{scan}(m)} + \text{sort}(m) + MST(n,m))$ |
| BFS, DFS and Topological ordering (on directed graphs) | $O(\min\{n + \lceil n/M \rceil \cdot \text{scan}(m), (n + \text{scan}(m)) \cdot \log_2 n,\ m\})$ |
| SSSP (on undirected graphs with integer weights) | $O(\sqrt{n \cdot \text{scan}(m) \cdot \log W} + MST(n,m))$ |
| SSSP (on undirected graphs with unbounded weights) | $O(\sqrt{n \cdot \text{scan}(m)} \cdot \log n + MST(n,m))$ |
| APSP (on unweighted undirected graphs) | $O(n \cdot \text{sort}(m))$ |
| APSP (on undirected graphs with non-negative weights) | $O(n \cdot (\sqrt{n \cdot \text{scan}(m)} + \text{scan}(m) \cdot \log(m/B)))$ |

Table 3.16: I/O complexity of state-of-the-art algorithms (assuming $m \geq n$) for graph traversal problems.

We implemented external memory BFS algorithms and showed their comparative analysis. Together with pipelining, disk parallelism, and our heuristic for maintaining the pool, our implementations provide viable BFS traversal on different classes of massive sparse graphs. In particular, we reduced the running time of a few *months* for many graph classes required by IM_BFS to a few *hours* using EM_BFS. We believe that our results can be further improved by fast analysis of graph structure (such as I/O-efficient approximation of graph diameter) and using it to tune parameters for external memory BFS algorithms.

Empirical evidence suggests that MR_BFS performs better on small diameter random graphs. However, the better asymptotic worst-case I/O complexity of MM_BFS_D helps it to outperform MR_BFS for moderate to large diameter sparse graphs with non-simple disk layout, where MR_BFS incurs close to its worst case of $\Omega(n)$ I/Os.

Extending our work to external-memory single-source shortest-paths may benefit

many real world applications. Our code can also be useful for engineering I/O-efficient dynamic BFS algorithms.

In general, the design and analysis of external memory graph traversal algorithms has greatly improved the worst case upper bounds for the I/O complexity of many graph traversal problems. Table 3.16 summarizes the state-of-the-art in external memory graph traversal algorithms on general graphs.

Engineering some of these algorithms has extended the limits of the graph size for which a traversal can be computed in "acceptable time". This in turn means that optimization problems of larger and larger sizes are becoming viable with advances in external memory graph traversal algorithms. We plan to eventually integrate these implementations into an external memory library for graph algorithms.

Many of these algorithms are still far from optimal. Similarly, while implementations of these algorithms provide good results on simple (low or high diameter) graph classes, it is still far from satisfactory for the difficult graph classes. More work is required both in designing and engineering these algorithms, particularly for directed graphs, to make traversal on even larger graphs viable.

# Chapter 4

# Characterizing the performance of Flash memory storage devices

*As knowledge advances, we are able to invent better and better models, which reproduce more and more features of the real world, more and more accurately. Nobody knows whether there is some natural end to this process, or whether it will go on indefinitely. In trying to understand common sense, we shall take a similar course...*

– Edwin Thompson Jaynes (Probability Theory: The Logic of Science, 1993)

Flash memory is a form of non-volatile computer memory that can be electrically erased and reprogrammed. Flash memory devices are lighter, more shock resistant, consume less power and hence are particularly suited for mobile computing. Initially used in digital audio players, digital cameras, mobile phones, and USB memory sticks, flash memory may become the dominant form of end-user storage in mobile computing: Some producers of notebook computers have already launched models (Apple MacBook Air, Sony Vaio UX90, Samsung Q1-SSD and Q30-SSD) that completely abandon traditional hard disks in favor of flash memory (also called solid state disks). Market research company In-Stat predicted [83] in July 2006 that 50% of all mobile computers would use flash (instead of hard disks) by 2013.

Frequently, the storage devices (be it hard disks or flash) are not only used to store data but also to actually compute on it if the problem at hand does not completely fit into main memory (RAM); this happens on both very small devices (like PDAs used for online route planning) and high-performance compute servers (for example when dealing with huge graphs like the web). Thus, it is important to understand the characteristics of the underlying storage devices in order to predict the real running time of algorithms, even if these devices are used as an external memory. In case of hard disks, the access cost depends on the current position of the disk-head and the location that needs to be read/written. This has been well researched; and there are good computation models (cf. Section 2.4) such as the external memory model [3] and the cache-oblivious model [73] that can help in realistic analysis of algorithms that run on hard disks. We would like to have a similar understanding of various access patterns on disks based on flash memory and to come up with computation models capturing the performance of algorithms on these disks. In this chapter, we show our attempt to characterize the performance (read/writes; sequential/random) of flash memory devices by analyzing the effects of random writes, misalignment, aging, past I/O patterns etc. on the access cost. We also discuss the implications of flash memory characteristics on the real running time of basic algorithms.

### State of the art for flash memories.

Recently, there has been growing interest in using flash memories to improve the performance of computer systems [29, 96, 117]. This trend includes the experimental use of flash memories in database systems [96, 117], in Windows Vista's use of USB flash memories as a cache (a feature called ReadyBoost), in the use of flash memory caches in hard disks (e.g., Seagate's Momentus 5400 PSD hybrid drives, which include 256 MB on the drive's controller), and in proposals to integrate flash memories into motherboards or I/O busses (e.g., Intel's Turbo Memory technology).

Most previous algorithmic work on flash memory concerns *operating system* algorithms and data structures that were designed to efficiently deal with flash memory cells wearing out, e.g., block-mapping techniques and flash-specific file systems. A comprehensive overview on these topics was recently published by Gal and Toledo [74]. The development of application algorithms tuned to flash memory is in its absolute infancy. We are only aware of very few published results beyond file systems and wear leveling:

Wu et al. [154, 155] proposed flash-aware implementations of $B$-trees and $R$-trees

without file system support by explicitly handling block-mapping within the application data structures.

Goldberg and Werneck [76] considered point-to-point shortest-path computations on pocket PCs where preprocessed input graphs (road networks) are stored on flash-memory; due to space-efficient internal-memory data-structures and locality in the inputs, data manipulation remains restricted to internal memory, thus avoiding difficulties with unstructured flash memory write accesses. Recently, Sanders et al. [141] also consider this problem. Their algorithm also consists of a preprocessing where "contraction hierarchies" of the road network are computed. The preprocessed external memory graph representation is then stored on the flash disks. However since querying for point-to-point shortest paths involves only read I/Os, they are also able to avoid unstructured writes on the flash memory.

### Goals.

Our first goal is to see how standard algorithms and data structures for basic algorithms like scanning, sorting and searching designed in the RAM model or the external memory model perform on flash storage devices. An important question here is whether these algorithms can effectively use the advantages of the flash devices (such as faster random read accesses) or there is a need for a fundamentally different model for realizing the full potential of these devices.

Our next goal is to investigate why these algorithms behave the way they behave by characterizing the performance of more than 20 different low-end and high-end flash devices under typical access patterns presented by basic algorithms. Such a characterization can also be looked upon as a first step towards obtaining a model for designing and analyzing algorithms and data structures that can best exploit flash memory. Previous attempts [96, 117] at characterizing the performance of these devices reported measurements on a small number of devices (1 and 2, respectively), so it is not yet clear whether the observed behavior reflects the flash devices, in general. Also, these papers didn't study if these devices exhibit any second-order effects that may be relevant.

Our next goal is to produce a benchmarking tool that would allow its users to measure and compare the relative performance of flash devices. Such a tool should not only allow users to estimate the performance of a device under a given workload in order to find a device with an appropriate cost-effectiveness for a particular application, but also allow quick measurements of relevant parameters of a device that can affect the performance of algorithms running on it.

These goals may seem easy to achieve, but they are not. These devices employ complex logical-to-physical mapping algorithms and complex mechanisms to decide which blocks to erase. The complexity of these mechanisms and the fact that they are proprietary mean that it is nearly impossible to tell exactly what factors affect the performance of a device. A flash device can be used by an algorithm designer like a hard disk (under the external memory or the cache-oblivious model), but its performance may be far more complex.

It is also possible that flash memory becomes an additional secondary storage device, rather than replacing the hard disk. Our last, but not least, goal is to find out how one can exploit the comparative advantages of both in the design of application algorithms, when they are used together.

### Outline.

The rest of this chapter is organized as follows. In Section 4.1, we develop a basic understanding of the architecture of flash disks. In Section 4.2, we show how the fundamental algorithms like merge-sort and binary search perform on flash memory devices and how appropriate are the standard computation models in predicting these performances. In Section 4.3, we present our experimental methodology, and our benchmarking program, which we use to measure and characterize the performance of many different flash devices. We also show the effect of random writes, misalignment, controllers and aging on the performance of these devices. In Section 4.4, we provide an algorithm design framework for the case when flash devices are used together with a hard disk. We also show the results of engineering the external memory BFS algorithms for this setting. We conclude with a preliminary computation model for predicting the performance of algorithms on flash memory devices in Section 4.5.

## 4.1 Basics of flash memory disks

Large-capacity flash memory devices use NAND flash chips. All NAND flash chips have common characteristics, although different chips differ in performance and in some minor details. The memory space of the chip is partitioned into blocks called *erase blocks*. The only way to change a bit from 0 to 1 is to erase the entire unit containing the bit. Each block is further partitioned into *pages*, which usually store 2048 bytes of data and 64 bytes of meta-data (smaller chips have pages containing only 512+16 bytes). Erase blocks typically contain 32 or 64 pages.

Bits are changed from 1 (the erased state) to 0 by *programming* (writing) data onto a page. An erased page can be programmed only a small number of times (one to three) before it must be erased again. Reading data takes tens of microseconds for the first access to a page, plus tens of nanoseconds per byte. Writing a page takes hundreds of microseconds, plus tens of nanoseconds per byte. Erasing a block takes several milliseconds. Finally, erased blocks wear out; each block can sustain only a limited number of erasures. The guaranteed numbers of erasures range from 10,000 to 1,000,000. To extend the life of the chip as much as possible, erasures should therefore be spread out roughly evenly over the entire chip; this is called *wear leveling*.

Because of the inability to overwrite data in a page without first erasing the entire block containing the page, and because erasures should be spread out over the chip, flash memory subsystems map *logical block addresses* (LBA) to physical addresses in complex ways [74]. This allows them to accept new data for a given logical address without necessarily erasing an entire block, and it allows them to avoid early wear even if some logical addresses are written to more often than others. This mapping is usually a non-trivial algorithm that uses complex data structures, some of which are stored in RAM (usually inside the memory device) and some on the flash itself.

The use of a mapping algorithm within LBA flash devices means that their performance characteristics can be worse and more complex than the performance of the raw flash chips. In particular, the state of the on-flash mapping and the volatile state of the mapping algorithm can influence the performance of reads and writes. Also, the small amount of RAM can cause the mapping mechanism to perform more physical I/O operations than would be necessary with more RAM.

## 4.2 Implications of flash devices for algorithm design

In this section, we look at how the RAM model and external memory model algorithms behave when running on flash memory devices. In the process, we try to ascertain whether the analysis of algorithms in either of the two models also carry over to the performance of these algorithms obtained on flash devices.

In order to compare flash memory with DRAM memory (used as main memory), we ran a basic RAM model list ranking algorithm on two architectures – one with 8 GB RAM memory and the other with 2 GB RAM, but 32 GB flash memory.

Recall from Section 2.5 that in the list ranking problem, we are given a list with individual elements randomly stored on disk and our goal is to find the distance of each element from the head of the list. The sequential RAM model algorithm consists of just hoping from one element to its successor, and thereby computing the distances of nodes from the head of the list. Here, we do not consider the cost of writing the distance labels of each node.

We stored a $2^{30}$-element list of long integers (8 Bytes) in random order, i.e. the elements were kept in the order of a random permutation generated beforehand. While ranking such a list took minutes in RAM, it took days with flash. This is because even though the random reads are faster on flash disks than the hard disk, they are still much slower than RAM. Furthermore, similar to the case of BFS on hard disk (cf. Figure 1.1), the performance of the RAM model algorithm significantly deviates from its predicted linear time behavior, when the size of the input list approaches and exceeds the available internal memory. Thus, we conclude that the RAM model is not useful for predicting the performance (or even relative performance) of algorithms running on flash memory devices and that some standard RAM model algorithms leave a lot to be desired if they are to be used on external flash devices.

| Algorithm | Hard Disk | Flash |
|---|---|---|
| Generating a random double and writing it | 0.2 $\mu$s | 0.37 $\mu$s |
| Scanning (per double) | 0.3 $\mu$s | 0.28 $\mu$s |
| External memory Merge-Sort (per double) | 1.06 $\mu$s | 1.5 $\mu$s |
| Random read | 11.3 ms | 0.56 ms |
| Binary Search | 25.5 ms | 3.36 ms |

Table 4.1: Runtime of basic algorithms when running on Seagate Barracuda 7200.11 hard disk as compared to 32 GB Hama Solid State Disk.

As Table 4.1 shows, the performance of basic algorithms when running on hard disks and when running on flash disks can be quite different, particularly when it comes to algorithms involving random read I/Os such as binary search on a sorted array. While such algorithms are extremely slow on hard disks necessitating B-trees and other I/O-efficient data structures, they are much faster on flash devices. On the other hand, algorithms involving write I/Os such as merge sort (with two read and write passes over the entire data) run much faster on hard disk than on flash.

It seems that the algorithms that run on flash have to achieve a different tradeoff between reads and writes and between sequential and random accesses than hard disks. Since the cost of accesses does not drop or rise proportionally over the

entire spectrum, the algorithms running on flash devices need to be qualitatively different from the one on hard disk. In particular, they should be able to tradeoff write I/Os at the cost of extra read I/Os. Standard external memory algorithms that assume same cost for reading and writing fail to take advantage of fast random reads offered by flash devices. Thus, there is a need for a fundamentally different model for realistically predicting the performance of algorithms running on flash devices.

## 4.3 Characterization of flash memory devices

In order to see why the standard algorithms behave as mentioned before, we characterize more than 20 flash storage devices. This characterization can also be looked at as a first step towards a model for designing and analyzing algorithms and data structures running on flash memory. We start this section by describing our hardware and software resources that were designed for this characterization.

### 4.3.1 Configuration

Our tests were performed on many different machines:

- A 1.5GHz Celeron-M with 512 MB RAM
- A 3.0GHz Pentium 4 with 2 GB RAM
- A 2.0Ghz Intel dual core T7200 with 2 GB RAM
- A 2 $\times$ Dual-core 2.6 GHz AMD Opteron with 2.5 GB RAM

All of these machines were running a 2.6 Linux kernel.

The devices included USB sticks, compact-flash and SD memory cards and solid state disks (of capacities 16 GB and 32 GB). They include both high-end and low-end devices. The USB sticks were connected via a USB 2.0 interface, memory cards were connected through a USB 2.0 card reader (made by Hama) or PCMCIA interface, and solid state disks with IDE interface were installed in the machines using a 2.5 inch to 3.5 inch IDE adapter and a PATA serial bus.

### Our benchmarking tool and methodology.

Standard disk benchmarking tools like `zcav` [108, 156] fail to measure characteristics that are important in flash devices (e.g., write speeds, since they are similar to read speeds on hard disks, or sequential-after-random writes); and commercial benchmarks tend to focus on end-to-end file-system performance, which does not characterize the performance of the flash device in a way that is useful to algorithm designers. Therefore, we decided to implement our own benchmarking program that is specialized (designed mainly for LBA flash devices), but highly flexible and can easily measure the performance of a variety of access patterns, including random and sequential reads and writes, with given block sizes and alignments, and with operation counts or time limits.

Our benchmarking software (running under linux) performs a series of experiments on a given block devices according to instructions in an input file. Each line in the input file describes one experiment, which usually consists of many reads or writes. Each experiment can consist of sequential or random reads or writes with a given block size. The accesses can be aligned to a multiple of the block size or misaligned by a given offset. Sequential accesses start at a random multiple of the block size. Random accesses generate and use a permutation of the possible starting addresses (so addresses are not repeated unless the entire address space is written). The line in the input file describes the number of accesses or a time limit. An input line can instruct the program to perform a self scaling experiment [47], in which the block size is repeatedly doubled until the throughput increases by less than 2.5%.

The buffers that are written to flash include either the approximate age of the device (in number of writes) or the values `0x00` to `0xff`, cyclically.

The block device is opened with the `O_DIRECT` flag, to disable kernel caching. We did not use raw I/O access, which eliminates main memory buffer copying by the kernel, because it exhibited significant overheads with small buffers. We assume that these overheads were caused by pinning user-space pages to physical addresses. In any case, buffer copying by the kernel probably does not have a large influence at the throughput of flash memories (we never measured more than 30 MB/s).

We used this program to run a standard series of tests on each device. The first tests measure the performance of aligned reads and writes, both random and sequential, at buffer sizes that start at 512 and double to 8 MB or to the self-scaling limit, whichever comes last. For each buffer size, the experiment starts by sequen-

tially writing the entire device using a 1 MB buffer, followed by sequential reads at the given buffer size, then random reads, then sequential writes, and finally random writes. Each pattern (read/write, sequential/random) is performed 3 times, with a time limit of 30 seconds each (90 seconds total for each pattern).

We also measure the performance of sequential writes following bursts of random writes of varying lengths (5, 30, and 60 seconds). As in the basic test, each such burst-sequential experiment follows a phase of sequentially writing the entire device. We measure and record the performance of the sequential writes at a higher resolution in this test, using 30 phases of 4 seconds each, to assess the speed at which the device recovers from the random writes. We tested random bursts of both 2 KB writes and of random writes at the same buffer size as the subsequent sequential writes.

Finally, we also measure the performance of misaligned random writes. These experiments consisted of 3 phases of 30 seconds for each buffer size and for each misalignment offset.

Entire-device sequential writes which separate different experiments are meant to bring the device to roughly the same state at the beginning of each test. We cannot guarantee that this always returns the logical-to-physical mapping to the same state (it probably does not), but it allows the device some chance to return to a relatively simple mapping.

We also used the program to run endurance tests on a few devices. In these experiments, we alternate between 1000 sequential writes of the entire logical address space and detailed performance tests. In the detailed phases we read and write on the device sequentially and randomly, in all relevant buffer sizes 3 times 30 seconds for each combination. The phases consisting of 1000 writes to the entire address space wear out the device at close to the fastest rate possible, and the detailed experiments record its performance as it wears out.

It is possible that there are other factors that influence performance of some LBA flash devices. However, since many modifications to the benchmarking methodology can be implemented simply by editing a text file, the benchmarking program should remain useful even if more behaviors need to be tested in the future. Of course, some modifications may also require changes to the program itself (e.g., the alignment parameter was added relatively late to the program).

### 4.3.2 Result and Analysis

**Performance of steady, aligned access patterns.**

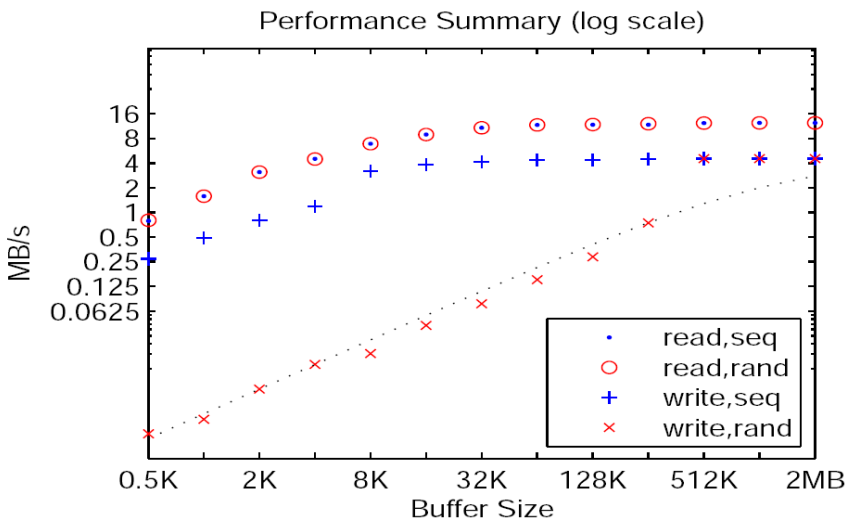



Figure 4.1: Performance (in logarithmic scale) of the 1 GB Toshiba TransMemory USB flash drive.

Figures 4.1 and 4.2 show the performance of two typical devices under the aligned access patterns. The other devices that we tested varied greatly in the absolute performance that they achieved, but not in the general patterns; all followed the patterns shown in Figures 4.1 and 4.2.

In all the devices that we tested, random writes using small block sizes were slower than all the other access patterns. The difference between random writes and other access patterns is particularly large at small buffer sizes, but it is usually still evident even on fairly large block sizes (e.g., 256 KB in Figure 4.1 and 128 KB in Figure 4.2). In most devices, small-buffer random writes were at least 10 times slower than sequential writes with the same buffer size, and at least 100 times slower than sequential writes with large buffers. Table 4.2 shows the read/write access time with two different block sizes (512 Bytes and 2 MB) for sequential and random accesses on some of the devices that we tested.

We believe that the high cost for random writes of small blocks is because of the LBA mapping algorithm in these devices. These devices partition the virtual and

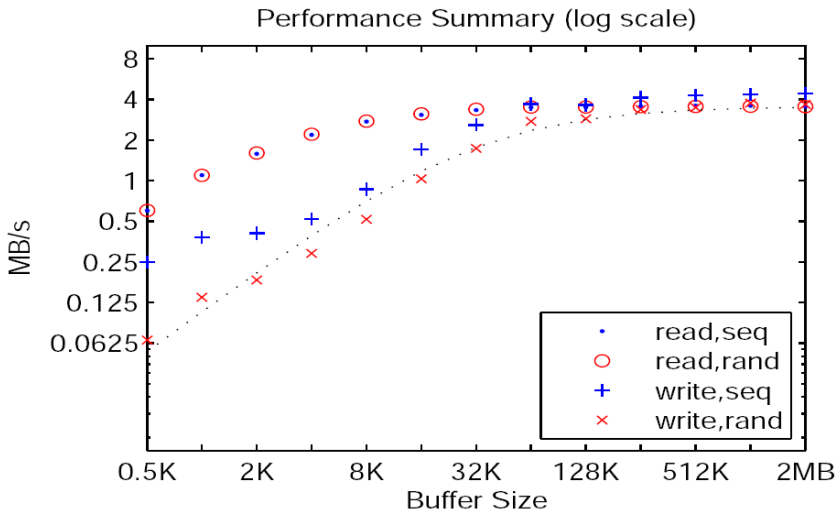



Figure 4.2: Performance (in logarithmic scale) of the 1 GB Kingston compact-flash card.

physical address spaces into chunks larger than an erase block; in many cases 512 KB. The LBA mapping maps areas of 512 KB logical addresses to physical ranges of the same size. On encountering a write request, the system writes the new data into a new physical chunk and keeps on writing contiguously in this physical chunk till it switches to another logical chunk. The logical chunk is now mapped twice. Afterwards, when the writing switches to another logical chunk, the system copies over all the remaining pages in the old chunk and erases it. This way every chunk is mapped once, except for the active chunk, which is mapped twice. On devices that behave like this, the best random-write performance (in seconds) is on blocks of 512 KB (or whatever is the chunk size). At that size, the new chunk is written without even reading the old chunk. At smaller sizes, the system still ends up writing 512 KB, but it also needs to read stuff from the old location of this chunk, so it is slower. We even found that on some devices, writing randomly 256 or 128 KB is slower than writing 512 KB, in absolute time.

In most devices, reads were faster than writes in all block sizes. This typical behavior is shown in Figure 4.1. But as Figure 4.2 shows, this is not a universal behavior of LBA flash devices. In the device whose performance is shown in Figure 4.2, large sequential writes are faster than large sequential reads. This shows that designers of such devices can trade off read performance and write perfor-

| DEVICE | | Buffer size 512 Bytes | | | | Buffer size 2 MB | | | |
|---|---|---|---|---|---|---|---|---|---|
| NAME | SIZE | SR | RR | SW | RW | SR | RR | SW | RW |
| KINGSTON DT SECURE | 512 MB | 0.97 | 0.97 | 0.64 | 0.012 | 33.14 | 33.12 | 14.72 | 9.85 |
| MEMOREX MINI TRAVELDRIVE | 512 MB | 0.79 | 0.79 | 0.37 | 0.002 | 13.15 | 13.15 | 5.0 | 5.0 |
| TOSHIBA TRANSMEMORY | 512 MB | 0.78 | 0.78 | 0.075 | 0.003 | 12.69 | 12.69 | 4.19 | 4.14 |
| SANDISK U3 CRUZER MICRO | 512 MB | 0.55 | 0.45 | 0.32 | 0.013 | 12.8 | 12.8 | 5.2 | 4.8 |
| M-SYSTEMS MDRIVE | 1 GB | 0.8 | 0.8 | 0.24 | 0.005 | 26.4 | 26.4 | 15.97 | 15.97 |
| M-SYSTEMS MDRIVE 100 | 1 GB | 0.78 | 0.78 | 0.075 | 0.002 | 12.4 | 12.4 | 3.7 | 3.7 |
| TOSHIBA TRANSMEMORY | 1 GB | 0.8 | 0.8 | 0.27 | 0.002 | 12.38 | 12.38 | 4.54 | 4.54 |
| SMI FLASH DEVICE | 1 GB | 0.97 | 0.54 | 0.65 | 0.01 | 13.34 | 13.28 | 9.18 | 7.82 |
| KINGSTON CF CARD | 1 GB | 0.60 | 0.60 | 0.25 | 0.066 | 3.55 | 3.55 | 4.42 | 3.67 |
| KINGSTON DT ELITE HS 2.0 | 2 GB | 0.8 | 0.8 | 0.22 | 0.004 | 24.9 | 24.8 | 12.79 | 6.2 |
| KINGSTON DT ELITE HS 2.0 | 4 GB | 0.8 | 0.8 | 0.22 | 0.003 | 25.14 | 25.14 | 12.79 | 6.2 |
| MEMOREX TD CLASSIC 003C | 4 GB | 0.79 | 0.17 | 0.12 | 0.002 | 12.32 | 12.15 | 5.15 | 5.15 |
| 120 × CF CARD | 8 GB | 0.68 | 0.44 | 0.96 | 0.004 | 19.7 | 19.5 | 18.16 | 16.15 |
| SUPERTALENT SOLID STATE FLASH DRIVE | 16 GB | 1.4 | 0.45 | 0.82 | 0.028 | 12.65 | 12.60 | 9.84 | 9.61 |
| HAMA SOLID STATE DISK 2.5" IDE | 32 GB | 2.9 | 2.18 | 4.89 | 0.012 | 28.03 | 28.02 | 24.5 | 12.6 |
| IBM DESKSTAR HARD DRIVE | 60 GB | 5.9 | 0.03 | 4.1 | 0.03 | 29.2 | 22.0 | 24.2 | 16.2 |
| SEAGATE BARRACUDA 7200.11 HARD DISK | 500 GB | 6.2 | 0.063 | 5.1 | 0.12 | 87.5 | 69.6 | 88.1 | 71.7 |

Table 4.2: The tested devices and their performance (in MBps) under sequential and random reads and writes with block size of 512 Bytes and 2 MB. The notations SR, RR, SW and RW stand for sequential reads, random reads, sequential writes and random writes, respectively.

mance. Optimizing for write performance can make sense for some applications, such as digital photography where write performance can determine the rate at which pictures can be taken. To professional photographers, this is more important than the rate at which pictures can be viewed on camera or downloaded to a computer.

Poor random-write performance is not a sign of poor design, but part of a tradeoff. All the devices that achieve sequential-write performance of over 15 MB/s (on large buffers) took more than 100 ms for small random writes. The two devices with sub-10ms random writes achieved write bandwidths of only 6.9 and 4.4 MB/s. The reason for this behavior appears to be as follows. To achieve high write bandwidths, the device must avoid inefficient erasures (ones that require copying many still-valid pages to a new erase block). The easiest way to ensure that sequential writes are fast is to always map contiguous logical pages to contiguous physical pages within an erase block. That is, if erase blocks contain, say 128 KB, then each contiguous logical 128 KB block is mapped to the pages of one erase block. Under aligned sequential writes, this leads to optimal write throughput. But when the host writes small random blocks, the device performs a read-modify-write of an entire erase block for each write request, to maintain the invariant of the address mapping.

On the other hand, the device can optimize the random-write performance by writing data to any available erased page, enforcing no structure at all on the address mapping. The performance of this scheme depends mostly on the state of the mapping relative to the current access pattern, and on the amount of surplus physical pages. If there are plenty of surplus pages, erasures can be guaranteed to be effective even under a worst-case mapping. Suppose that a device with $n$ physical pages exports only $n/2$ logical pages. When it must erase a block to perform the next write, it contains $n/2$ obsolete pages, so on at least one erase block half the pages are obsolete. This guarantees a 50% erasure effectiveness. If there are only few surplus pages, erasures may free only a single page. But if the current state of the mapping is mostly contiguous within each erase block and the access pattern is also mostly contiguous, erasures are effective and do not require much copying.

This tradeoff spans a factor of 10 or more in random-write performance and a factor of about 4 or 5 in sequential-write performance. System designers selecting an LBA flash device should be aware of this tradeoff, decide what tradeoff their system requires, and choose a device based on benchmark results.

Another nearly-universal characteristic of the flash devices is the fact that sequential reads are not faster than random reads. The read performance does depend on block size, but usually not on whether the access pattern is random or sequential. On a few exceptional devices where the sequential reads are faster than random reads, the difference between the two access patterns (for 2 MB block size) is very small.

The performance in each access pattern usually increases monotonically with the

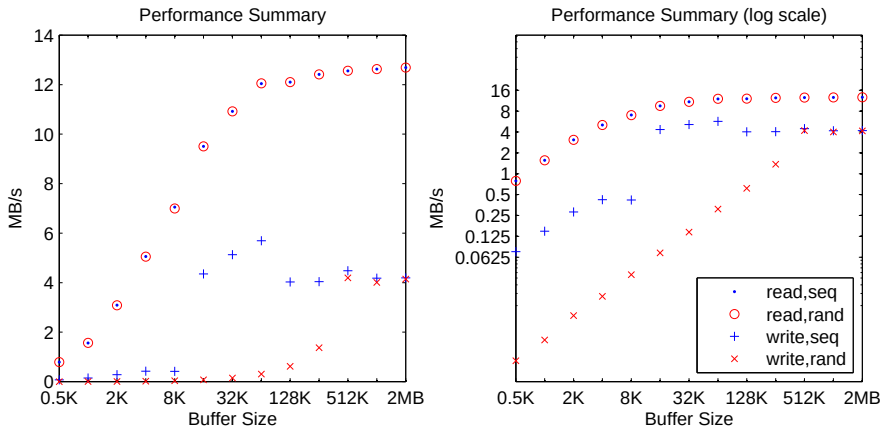



Figure 4.3: Speeds of the 512 MB Toshiba TransMemory USB flash device. This device achieves its maximum write speed at a 64 KB buffer size.

block size, up to a certain saturation point. Reading and writing small blocks is always much slower than the same operation on large blocks. But Figure 4.3 shows an exception. The best sequential-write performance of this occurs with blocks of 64 KB; on larger blocks, performance drops (by more than 20%).

**Comparison to hard disks.** Quantitatively, the only operation in which LBA flash devices are faster than hard disks is random reads of small buffers. Many of these devices can read a random page in less than a millisecond, sometimes less than 0.5ms. This is at least 10 times faster than current high-end hard disks, whose random-access time is 5-15ms. Even though the random-read performance of LBA flash devices varies, all the devices that we tested exhibited better random-read times than those of hard disks.

In all other aspects, most of the flash devices tested by us are inferior to hard disks. The random-write performance of LBA flash devices is particularly bad and particularly variable. A few devices performed random writes about as fast as hard disks, e.g., 6.2ms and 9.1ms. But many devices were more than 10 times slower, taking more than 100ms per random write, and some took more than 300ms.

Even under ideal access patterns, flash devices we have tested provide smaller I/O bandwidths than hard disks. One flash device reached read throughput approaching 30 MB/s and write throughput approaching 25 MB/s. Hard disks can achieve well over 100 MB/s for both reads and writes. Even disks designed for laptops

can achieve throughput approaching 60 MB/s. Flash devices would need to improve significantly before they outperform hard disks in this metric. The possible exception to this conclusion is large-capacity flash devices utilizing multiple flash chips, which should be able to achieve high throughput by writing in parallel to multiple chips.

### Performance of large number of random writes.

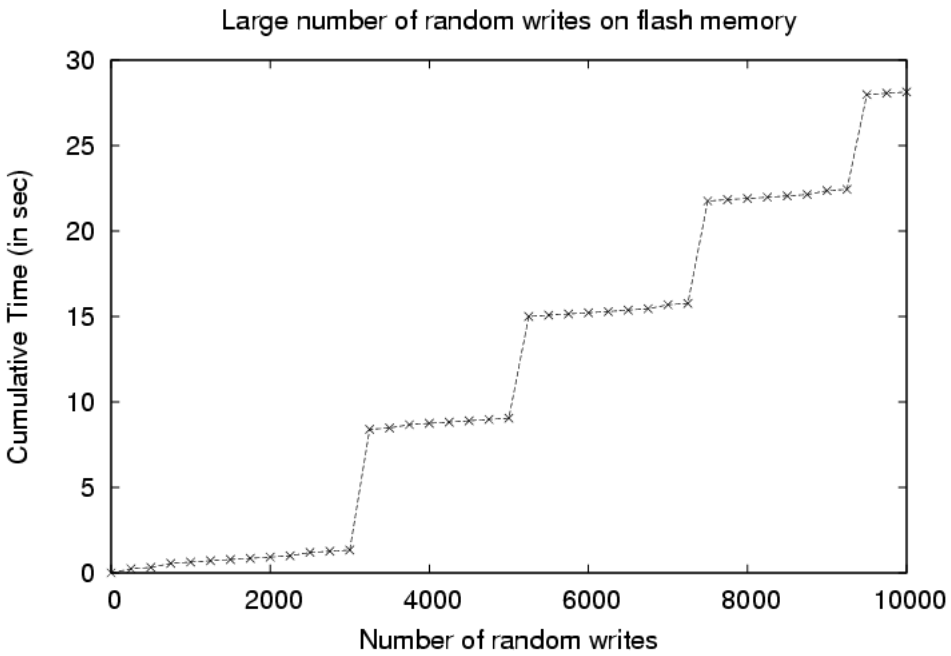



Figure 4.4: Total time taken by large number of random writes on a 32 GB Hama Solid state disk.

We observed an interesting phenomenon (Figure 4.4) while performing large number of random writes on a 32 GB Hama (2.5" IDE) solid state disk. After the first 3000 random writes (where one random write is writing a 8-byte real number at a random location in a 8 GB file on flash), we see some spikes in the total running time. Afterwards, these spikes are repeated regularly after about every 2000 random writes. This behavior is not restricted to the Hama solid state disk but is observed in many other flash devices too.

We believe that it is because the random writes cause many updates in the page table. After a while, the controller rearranges the pages in the blocks to simplify the LBA mapping. This process takes 5-8 seconds while really writing the data on the disk takes less than 0.8 seconds for 2000 random writes, causing the spikes in the total time.

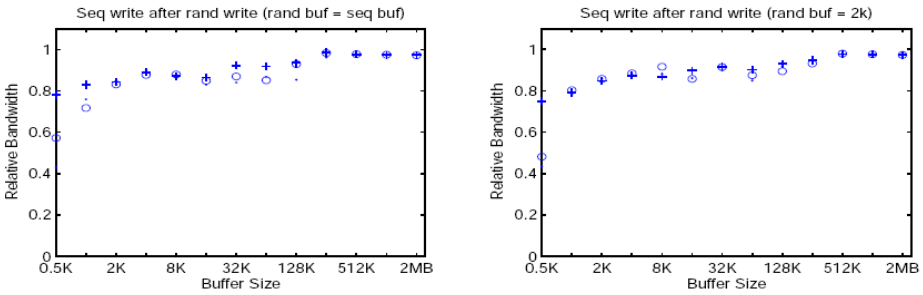



Figure 4.5: Graphs showing the effect of random writes on subsequent sequential writes on Toshiba 1 GB TransMemory USB flash drive.

### Effect of random writes on subsequent operations.

On some devices, a burst of random writes slows down subsequent sequential writes. The effect can last a minute or more, and in rare cases hours (of sustained writing). No such effect was observed on subsequent reads.

Figure 4.5 presents the performance of one such device. In these experiments, we performed $t$ seconds of random writing, for $t = 5, 30$ and 60. We then measured the performance of sequential writes during each 4 second period for the next 120 seconds. The two graphs in Figure 4.5 show the median performance in these 30 4-second periods relative to the steady-state performance of the same pattern (read or write and with the same block size). As we can see, for very small blocks the median performance in the two minutes that follow the random writes can drop by more than a factor of two. Even on larger blocks, performance drops by more than 10%. Figure 4.6 presents the performance of a device in which random writes slow down subsequent sequential operations. In these experiments, we performed $t$ seconds of random writing, for $t = 5, 30$ and 60. We then measured the performance of sequential writes during each 4 second period for the next 120 seconds. The two graphs in the middle show the median performance in these 30 4-second periods relative to the steady-state performance of the same pattern (read or write and with the same block size). As we can see, for very small blocks the median performance in the two minutes that follow the random writes can drop by more than a factor of two. Even on larger blocks, performance drops by more than 10%.

The two graphs in the middle row of Figure 4.6 differ in the block size during the $t$ seconds of random writes. In the middle-left graph, the random writes were of the same size as the subsequent operation, whereas in the middle-right graph the random writes were always of 2 KB buffers. The behavior of this particular device

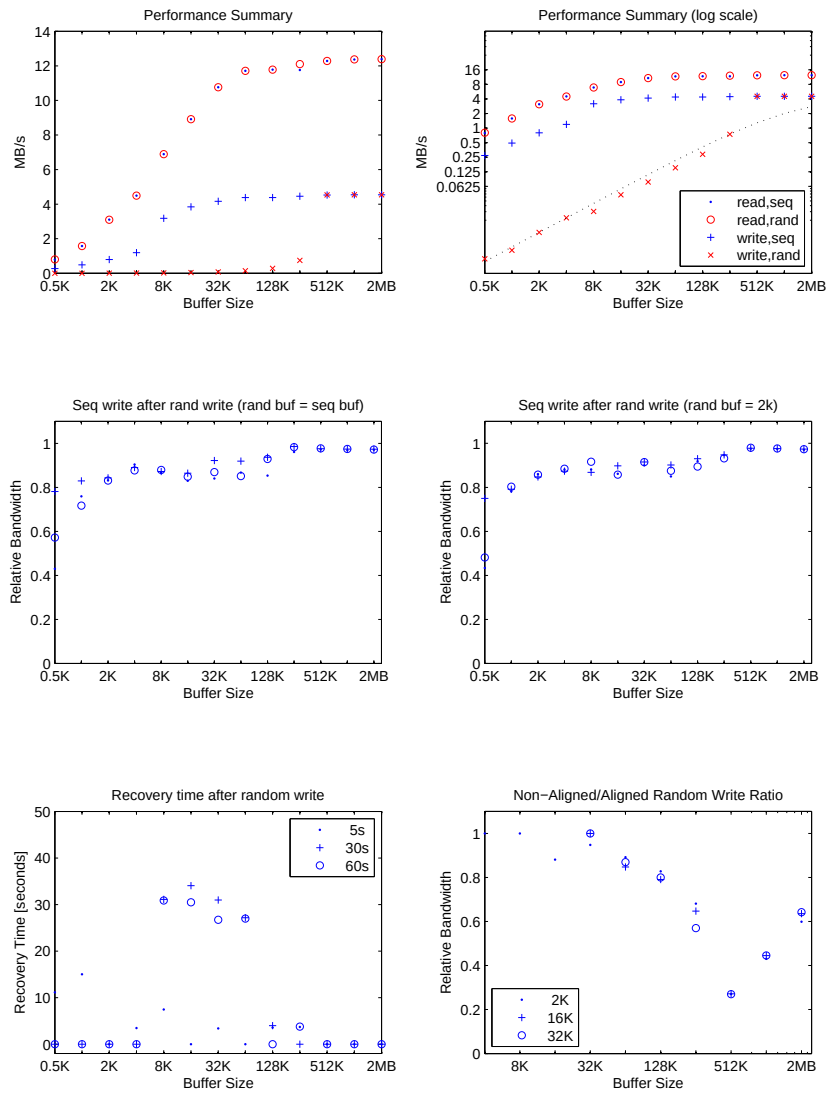



Figure 4.6: Toshiba TransMemory USB flash drive results. The top two graphs show the speeds. The two graphs in the middle show how the device is affected by random writes. The bottom left graph shows the time it takes to return back to 60% of the median speed. The bottom right graph shows the effect of misaligned calls on random writes.

in the two cases is similar, but on other devices later the two cases differ. When the two cases differ, random writes of 2 KB usually slow down subsequent writes more than random writes of larger blocks. This is typified by the results shown in Figure 4.7.

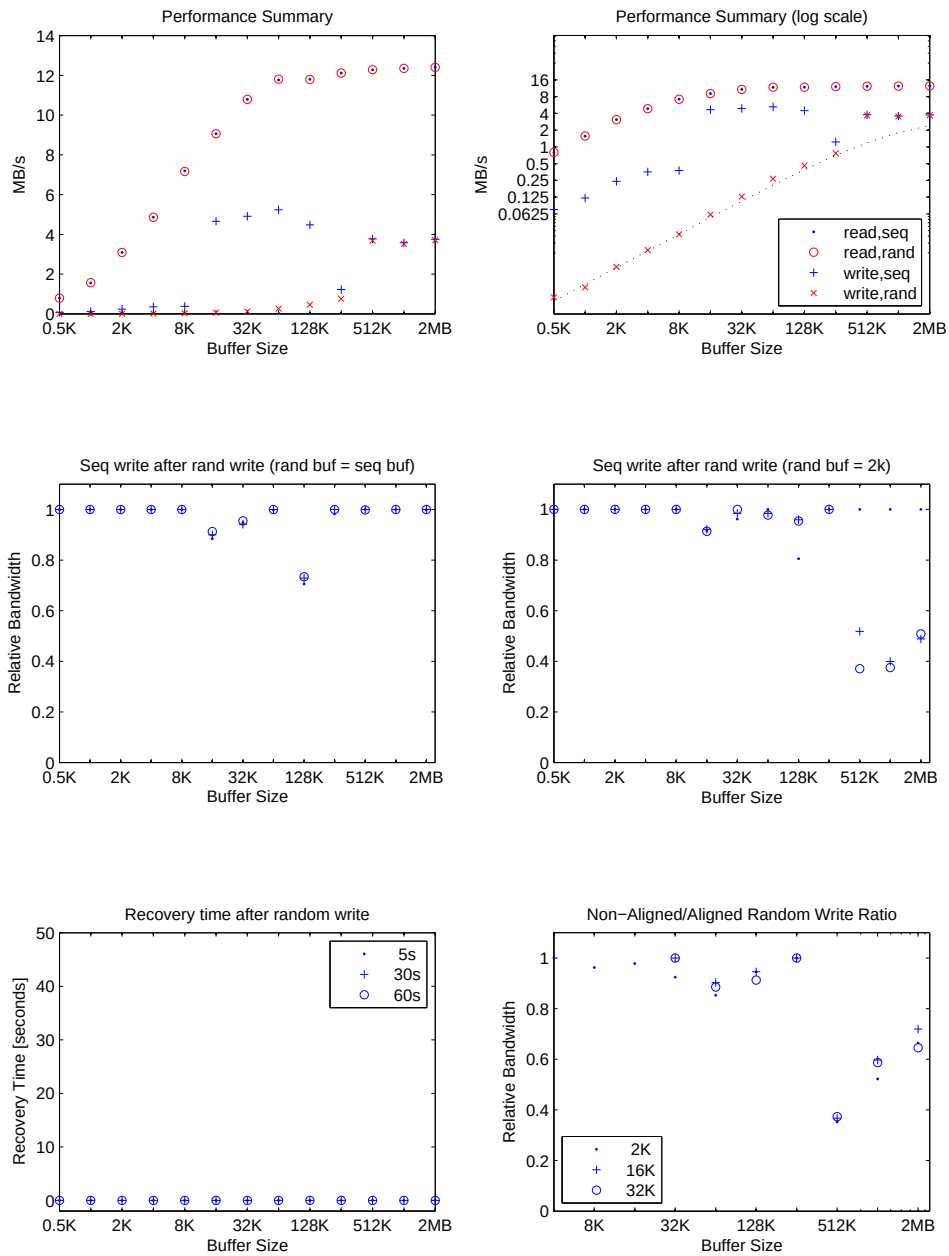



Figure 4.7: Results of the M-Systems mDrive 100 USB device, showing a constant decrease in the sequential write speed, with no recovery time.

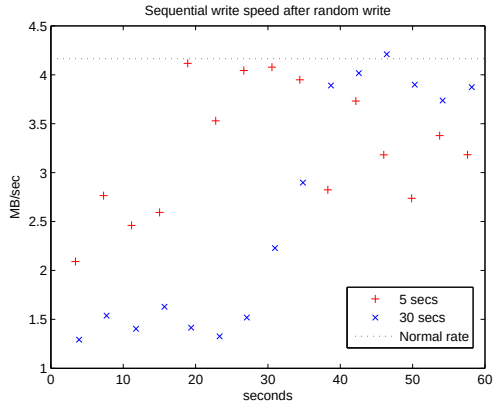



Figure 4.8: A time line showing the sequential write performance with 32 KB blocks of the device in Figure 4.6. The time line starts at the end of 5 or 30 seconds of random writes (again with a 32 KB buffer size). The markers show the write bandwidth in each 4-second period following the random writes.

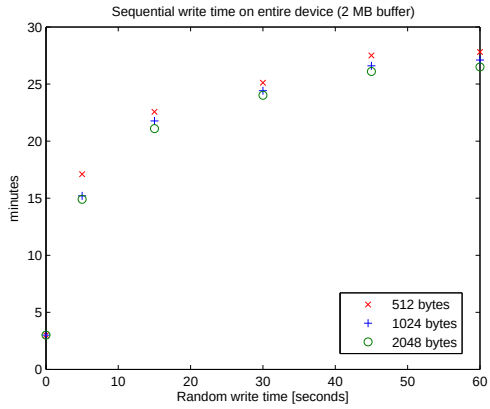



Figure 4.9: An example of extreme recovery times, as observed in the 2 GB Kingston DT Elite 2.0. The graph shows the time (measured in minutes) it takes to write the entire device sequentially with a 2 MB buffer size after random writes of 5 to 60 seconds. Random writes were performed using buffer sizes of at most 2 KB.

In experiments not reported here we explored the effects of random writes on subsequent read operations and on subsequent random writes. We did not discover any effect on these subsequent operations, so we do not describe the detailed results of these experiments.

The graph on the lower-left corners of Figures 4.6 and 4.7 show how long it took

the device to recover back to 60% of the median performance in the two minutes following the random writes. The device in Figure 4.6 usually recovers immediately to this performance level, but in some buffer sizes, it can take it 20-30 seconds to recover. Note that recovery here means a return to a 0.6 fraction of the median post-random performance, not to the base performance in the particular access pattern.

Figure 4.8 presents the recovery time in a different way, on a time line. After a 30 seconds random write time, the speed of the sequential write slows down to about 30% of the normal speed. After 30 seconds of a sequential write, the speed climbs back towards the normal speed. We have seen similar behaviors in other devices that we tested.

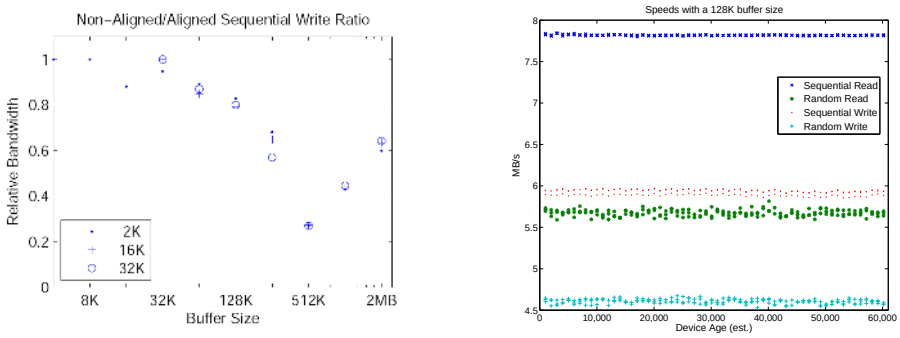
On the high-end 2 GB Kingston DT Elite 2 device, random writes with buffer sizes of 2 KB or less cause a drop in the the performance of subsequent sequential writes to less than 5% of the normal (with the same buffer size). The device did not recover to its normal performance until it was entirely rewritten sequentially. Normally, it takes 3 minutes to write the entire device sequentially with a buffer size of 2 MB, but after random small-buffer writes, it can take more than 25 minutes, a factor of 8 slower (Figure 4.9). We observed the same behavior in the 4 GB version of this device.

We have also observed many devices whose performance was not affected at all by random writes.

### Effects of misalignment.

On many devices, misaligned random writes achieve much lower performance than aligned writes. In this setting, alignment means that the starting address of the write is a multiple of the block size. We have not observed similar issues with sequential access and with random reads.

Figure 4.10a shows the ratio between misaligned and aligned random writes on 1 GB TRANSMEMORY USB flash device. The misalignment is by 2 KB, 16 KB and 32 KB. All of these sizes are at most as large as a single flash page. Many of the devices that we have tested showed some performance drop on misaligned addresses, but the precise effect varied from device to device. For example, the 128 MB SuperTalent USB device is affected by misalignment by 2 KB but not by misalignments of 16 KB or 32 KB.




(a) (b)

Figure 4.10: Effect of misalignment and aging on the performance of flash devices.

### Effects of Aging.

We were not able to detect a significant performance degradation as devices get older (in terms of the number of writes and erasures). Figure 4.10b shows the performance of 512 MB SANDISK CRUZER MICRO USB device as a function of the number of sequential writes on the entire device. The performance of each access pattern remains essentially constant, even after 60,000 writes. On 512 MB KINGSTON DATATRAVELER II+ USB device, we ran a similar experiment writing more than 320,000 times, exceeding its rated endurance by at least a factor of 3 and did not observe any slowing down with age.

### Effect of different controller interfaces.

We connected a compact-flash card via a USB 2.0 interface, PCMCIA interface and an IDE interface (using a card reader) and found that the connecting interface does not affect the relative access patterns (sequential vs. random, read vs. write and the effect of different block sizes) of the flash devices. However, the maximum throughputs that we could obtain from USB 2.0, PCMCIA and IDE interface are 19.8 MBps, 0.95 MBps, and 2.16 MBps for read and 18.2 MBps, 0.95 MBps, and 4.38 MBps for write, respectively.

## 4.4 Designing algorithms to exploit flash when used together with a hard disk

Till now, we discussed the characteristics of the flash memory devices and the performance of algorithms running on architectures where the flash disks replace the hard disks. Another likely scenario is that rather than replacing hard disks, flash disks may become an additional secondary storage, used together with hard disks. From the algorithm design point of view, it leads to many interesting questions. A fundamental question here is how can we best exploit the comparative advantages of the two devices while running an application algorithm.

The simple idea of directly using external memory algorithms with input and intermediate data randomly striped on the two disks treats both the disks as equal. Since the sequential throughput and the latency for random I/Os of the two devices is likely to be very different, the I/Os of the slower disk can easily become a bottleneck, even with asynchronous I/Os.

The key idea in designing efficient algorithms in such a setting is to restrict the random accesses to a static data-structure. This static data-structure is then kept on the flash disk, thereby exploiting the fast random reads of these devices and avoiding unnecessary writing. The sequential read and write I/Os are all limited to the hard disk.

We illustrate this basic framework with the help of external memory BFS algorithm of Mehlhorn and Meyer [106] (MM_BFS). Recall from Section 3.3 that MM_BFS involves a preprocessing phase that groups the nodes of the input graph into disjoint clusters of small diameter and stores the adjacency lists of the nodes in a cluster contiguously on the disk. After each BFS level, some clusters are merged into an efficiently accessible data structure (hot pool). This hot pool is then scanned for the adjacency lists of the nodes in the current level and these adjacency lists are then removed from the hot pool.

This algorithm is well suited for our framework as random I/Os are mostly restricted to the data structure keeping the graph clustering, while the hot pool accesses are mostly sequential. Also, the graph clustering is only stored once whereas the hot pool is modified (read and written) in every iteration. Thus, we keep the graph clustering data structure on the flash disk and the hot pool on the hard disk.

We ran our implementation (cf. Section 3.6) of this algorithm on the graph class shown in Figure 4.11. This graph class is a tree with $\sqrt{B}+1$ BFS levels. Level

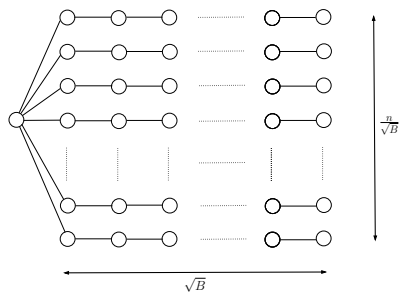



Figure 4.11: A graph class that forces the Mehlhorn/Meyer BFS algorithm to incur its worst case I/O complexity.

| Operation | Random striping | | Our strategy | |
|---|---|---|---|---|
| | 1 Flash + 1 Hard disk | 2 Hard disks | Same cluster size | Smaller cluster size |
| I/O wait time | 10.5 | 6.3 | 7.1 | 5.8 |
| Total time | 11.7 | 7.5 | 8.1 | 6.3 |

Table 4.3: Timing (in hours) for the second phase of Mehlhorn/Meyer's BFS algorithm on $2^{28}$-node graph.

0 contains only the source node which has an edge to all nodes in level 1. Levels $1 \ldots \sqrt{B}$ have $\frac{n}{\sqrt{B}}$ nodes each and the $i^{th}$ node in $j^{th}$ level ($1 < j < \sqrt{B}$) has an edge to the $i^{th}$ node in levels $j-1$ and $j+1$.This graph class has large diameter ($\sqrt{B}+1$) and the hot pool size is greater than the available internal memory for the most part of the execution. As such, it is one of the difficult graph classes for all our EM BFS implementations.

As compared to striping the graph as well as pool randomly between the hard disk and the flash disk, the strategy of keeping the graph clustering data structure in flash disk and hot pool in hard disk performs around 25% better. Table 4.3 shows the running time for the second phase of the algorithm for a $2^{28}$-node graph. Although the number of I/Os in the two cases are nearly the same, the time spent waiting for I/Os is much better for our disk allocation strategy, leading to better overall runtime.

The cluster size in the BFS algorithm was chosen in a way so as to balance the random reads and sequential I/Os on the hard disks, but now in this new setting, we can reduce the cluster size as the random I/Os are being done much faster by the flash memory. Our experiments suggest that this leads to even further improvements in the runtime of the BFS algorithm.

## 4.5 Conclusion

We have characterized the performance of flash storage devices by benchmarking more than 20 different such devices. We conclude that the read/write/erase behavior of flash is radically different than that of other external block devices like hard disks. Though flash devices have faster random access than the hard disk, they can neither provide the read/write throughput of the disks[1], nor provide faster random writes than hard disks. We found out that access costs on flash devices also depend on the past history (particularly, the number of random writes done before) and misalignment, but not on the aging of devices.

We also showed that the existing RAM model and external memory algorithms can not realize the full potential of the flash devices. Many interesting open problems arise in this context such as how best can one sort (or even search) on a block based device where the read and write costs are significantly different.

Furthermore, we observe that in the setting where the flash becomes an additional level of secondary storage and used together with hard disk rather than replacing it, one can exploit the comparative advantages of both by restricting the random read I/Os to a static data structure stored on the flash and using the hard disk for all other I/Os.

Our results indicate that there is a need for more experimental analysis to find out how the existing external memory and cache-oblivious data structures like priority queues and search trees perform, when running on flash devices. Such experimental studies should eventually lead to a model for predicting realistic performance of algorithms and data structures running on flash devices, as well as on combinations of hard disks and flash devices. Coming up with a model that can capture the essence of flash devices and yet is simple enough to design and analyze algorithms and data structures, remains an important challenge.

As a first model, we may consider a natural extension of the standard external-memory model that will distinguish between block accesses for reading and writing. The cost measure for an algorithm incurring $x$ read I/Os and $y$ write I/Os could be $x + c_W \cdot y$, where the parameter $c_W > 1$ is a penalty factor for write accesses.

An alternative approach might be to assume different block transfer sizes, $B_R$ for reading and $B_W$ for writing, where $B_R < B_W$ and $c_R \cdot x + c_W \cdot y$ (with $c_R, c_W > 1$) would be the modified cost measure.

[1] As of late 2007, the ones that could provide were far more expensive than the same capacity hard disk

# Chapter 5

# Dynamic topological ordering

*What we imagine is order is merely the prevailing form of chaos.*

— Kerry Thornley

There has been a growing interest in dynamic graph algorithms over the last two decades due to their applications in a variety of contexts including operating systems, information systems, network management, assembly planning, VLSI design and graphical applications. Typical dynamic graph algorithms maintain a certain property (e. g., connectivity information) of a graph that changes (a new edge inserted or an existing edge deleted) dynamically over time. An algorithm or a problem is called *fully dynamic* if both edge insertions and deletions are allowed, and it is called *partially dynamic* if only one (either only insertion or only deletion) is allowed. If only insertions are allowed, the partially dynamic algorithm is called incremental; if only deletions are allowed, it is called decremental. While a number of fully dynamic algorithms have been obtained for various properties on undirected graphs (see [65] and references therein), the design and analysis of fully dynamic algorithms for directed graphs has turned out to be much harder (e. g., [72, 134, 136, 137]). Much of the research on directed graphs is therefore concentrated on the design of partially dynamic algorithms instead (e. g., [24, 50, 91]). In this chapter, we focus on the analysis of algorithms for maintaining a topological ordering of directed graphs in an incremental setting.

A topological order $T$ of a directed graph $G = (V, E)$ (with $n := |V|$ and $m := |E|$) is a linear ordering of its nodes such that for all directed paths from $x \in V$ to $y \in V$ ($x \neq y$), it holds that $T(x) < T(y)$. A directed graph has a topological ordering

if and only if it is acyclic. There are well-known algorithms for computing the topological ordering of a directed acyclic graph (DAG) in $O(m+n)$ time in an offline setting (see e. g. [51]). In a fully dynamic setting, each time an edge is added or deleted from the DAG, we are required to update the bijective mapping $T$. In the online/incremental variant of this problem, the edges of the DAG are not known in advance but are inserted one at a time (no deletions allowed). As the topological order remains valid when removing edges, most algorithms for online topological ordering can also handle the fully dynamic setting. However, there are no good bounds known for the fully dynamic case. Most algorithms are only analyzed in the online setting.

Given an arbitrary sequence of edges, the online cycle detection problem is to discover the first edge which introduces a cycle. Till now, the best known algorithm for this problem involves maintaining an online topological order and returning the edge after which no valid topological order exists. Hence, results for online topological ordering also translate into results for the online cycle detection problem. Online topological ordering is required for incremental evaluation of computational circuits [15] and in incremental compilation [104, 120] where a dependency graph between modules is maintained to reduce the amount of recompilation performed when an update occurs. An application for online cycle detection is pointer analysis [126].

For inserting $m$ edges, the naïve way of computing an online topological order each time from scratch with the offline algorithm takes $O(m^2+mn)$ time. Marchetti-Spaccamela, Nanni, and Rohnert [105] gave an algorithm (MNR) that can insert $m$ edges in $O(mn)$ time. Alpern, Hoover, Rosen, Sweeney, and Zadeck proposed an algorithm [15] (AHRSZ) which runs in $O(|\rangle\hat{K}\langle| \log(|\rangle\hat{K}\langle|))$ time per edge insertion with $|\rangle\hat{K}\langle|$ being a local measure of the insertion complexity. However, there is no analysis of AHRSZ for a sequence of edge insertions. Katriel and Bodlaender (KB) [91] analyzed a variant of the AHRSZ algorithm and obtained an upper bound of $O(\min\{m^{\frac{3}{2}}\log n, m^{\frac{3}{2}}+n^2\log n\})$ for inserting an arbitrary sequence of $m$ edges. In addition, they show that their algorithm runs in time $O(m\cdot k\cdot \log^2 n)$ for a DAG for which the underlying undirected graph has a treewidth $k$. Also, they give an $O(n\log n)$ algorithm for DAGs whose underlying undirected graph is a tree. The algorithm by Pearce and Kelly (PK) [124] empirically outperforms the other algorithms for random edge insertions leading to sparse random DAGs, although its worst-case runtime is inferior to KB.

In this chapter, we propose a simple algorithm that works in $O(n^{2.75}\sqrt{\log n})$ time and $O(n^2)$ space, thereby improving upon the results of Katriel and Bodlaender for dense DAGs. With some simple modifications in our data structure, we can

get $O(n^{2.75})$ time with $O(n^{2.25})$ space or $O(n^{2.75})$ *expected* time with $O(n^2)$ space. Our algorithm can also be used for online cycle detection in graphs. Moreover, it permits an arbitrary starting point, which makes a hybrid approach possible, i. e., using the PK or KB algorithm for sparse graphs and ours when the graphs become dense.

We conjecture that our analysis can be improved. We reduce the problem of tighter analysis of our algorithm to a combinatorial graph problem.

We also show how we can externalize our algorithm and get a better amortized bound than the $O(sort(m))$ I/Os per edge bound based on time-forward processing.

We also present the first average-case analysis of online topological ordering algorithms. We prove an expected runtime of $O(n^2 \operatorname{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for AHRSZ, KB and PK.

The rest of this chapter is organized as follows. In Section 5.1, we review the previous algorithms for dynamic topological ordering. In Section 5.2, we describe our algorithm and the data structures involved. In Section 5.3, we give the correctness argument for our algorithm, followed by an analysis of its runtime in Sections 5.4 and 5.5. The details of our implementation and an empirical comparison with other algorithms follow in Section 5.6. Section 5.7 shows the reduction of tighter analysis of our algorithm to a combinatorial problem. Section 5.8 describes the externalization of our algorithm. Section 5.9 shows our average-case analysis for AHRSZ, KB and PK. Section 5.10 discusses recent advances on improving the upper bounds for this problem. Section 5.11 concludes with some open problems related to dynamic topological ordering.

## 5.1 Related work

This section first introduces some notations and then reviews the previous algorithms MNR, AHRSZ, KB, and PK. We keep the current topological order as a bijective function $T\colon V \to [1..n]$. In this and the subsequent sections, we will use the following notations: $d(u,v)$ denotes $|T(u) - T(v)|$, $u < v$ is a short form of $T(u) < T(v)$, $u \to v$ denotes an edge from $u$ to $v$, and $u \rightsquigarrow v$ expresses that $v$ is reachable from $u$. Note that $u \rightsquigarrow u$, but *not* $u \to u$. The *degree* of a node is the sum of its in- and out-degree. We will also refer to $T(v)$ as the priority of the node $v$.

Consider the $i$-th edge insertion $u \to v$. We say that an edge insertion is *invalidat-*

*ing* if $u > v$ before the insertion of this edge. We define $R_B^{(i)} := \{x \in V \mid v \leq x \wedge x \rightsquigarrow u\}$, $R_F^{(i)} := \{y \in V \mid y \leq u \wedge v \rightsquigarrow y\}$ and $\delta^{(i)} = R_F^{(i)} \cup R_B^{(i)}$. Let $|\delta^{(i)}|$ denote the number of nodes in $\delta^{(i)}$ and let $\|\delta^{(i)}\|$ denote the number of edges incident to nodes of $\delta^{(i)}$. Note that $\delta^{(i)}$ as defined above is different from the adaptive parameter $\delta$ of the bounded incremental computation model. If an edge is non-invalidating, then $|R_B^{(i)}| = |R_F^{(i)}| = |\delta^{(i)}| = 0$. Note that for an invalidating edge, $R_F^{(i)} \cap R_B^{(i)} = \emptyset$ as otherwise the algorithms will just report a cycle and terminate.

We now describe the insertion of the $i$-th edge $u \rightarrow v$ for all the algorithms. Assume for the remainder of this section that $u \rightarrow v$ is an invalidating edge, as otherwise none of the algorithms do anything for that edge. Let $AR^{(i)}$ be the set of all nodes $x$ such that $v \leq x \leq u$. We define an algorithm to be *local* if it only changes the ordering of nodes in $AR^{(i)}$ to compute the new topological order $T'$ of $G \cup \{(u,v)\}$. All of these algorithms are local and they work in two phases – a "discovery phase" and a "relabelling phase".

**MNR** is probably the simplest of these algorithms. A depth-first search starting from $v$ and limited to nodes in $AR^{(i)}$ marks all nodes in $R_F^{(i)}$ as visited. Thereafter, all marked nodes are shifted up in the topological ordering immediately after $u$. For this, all nodes in $\{AR^{(i)} \setminus R_F^{(i)}\}$ are moved down appropriately in the topological order. The relative order of the nodes in $R_F^{(i)}$ remains intact.

In the discovery phase of **PK**, the set $\delta^{(i)}$ is identified using a forward depth-first search from $v$ (giving a set $R_F^{(i)}$) and a backward depth- first search from $u$ (giving a set $R_B^{(i)}$). The relabelling phase is also very simple. It sorts both sets $R_F^{(i)}$ and $R_B^{(i)}$ separately in increasing topological order and then allocates new priorities according to the relative position in the sequence $R_B^{(i)}$ followed by $R_F^{(i)}$. It does not alter the priority of any node not in $\delta^{(i)}$, thereby greatly simplifying the relabeling phase. The runtime of PK for a single edge insertion is $\Theta(\|\delta^{(i)}\| + |\delta^{(i)}| \log |\delta^{(i)}|)$.

Alpern et al. [15] used the bounded incremental computation model [134] and introduced the measure $|\rangle \hat{K} \langle|$. For an invalidated topological order $T$, the set $K \subseteq V$ is a *cover* if for all $x, y \in V$: $(x \rightsquigarrow y \ \wedge \ y < x \ \Rightarrow \ x \in K \vee y \in K)$. This states that for any connected $x$ and $y$ which are incorrectly ordered, a cover $K$ must include $x$ or $y$ or both. $|K|$ and $\|K\|$ denote the number of nodes and edges touching nodes in $K$, respectively. We define $|\rangle K \langle| := |K| + \|K\|$ and a cover $\hat{K}$ to be *minimal* if $|\rangle \hat{K} \langle| \leq |\rangle K \langle|$ for any other cover $K$. Thus, $|\rangle \hat{K} \langle|$ captures the minimal amount of work required to calculate the new topological order $T'$ of $G \cup \{(u,v)\}$ assuming that the algorithm is local and that the adjacent edges must be traversed.

**AHRSZ**s *discovery phase* marks the nodes of a cover $K$ by marking some of the unmarked nodes $x, y \in \delta^{(i)}$ with $x \rightsquigarrow y$ and $y < x$. This is done recursively by moving two frontiers starting from $v$ and $u$ towards each other. Here, the crucial decision is which frontier to move next. AHRSZ tries to minimize $\|K\|$ by balancing the number of edges seen on both sides of the frontier. The recursion stops when forward and backward frontier meet. Note that we do not necessarily visit all nodes in $R_F^{(i)}$ ($R_B^{(i)}$) while extending the forward frontier (backward frontier). It can be proven [15] that the marked nodes indeed form a cover $K$ and that $|\rangle K \langle| \leq 3\,|\rangle \hat{K} \langle|$.

The *relabeling phase* employs the dynamic priority space data structure due to Dietz and Sleator [60]. This permits new priorities to be created between existing ones in $O(1)$ amortized time. This is done in two passes over the nodes in $K$. During the first pass, it visits the nodes of $K$ in reverse topological order and computes a strict upper bound on the new priorities to be assigned to each node. In the second phase, it visits the nodes in $K$ in topological order and computes a strict lower bound on the new priorities. Both together allow to assign new priorities to each node in $K$. Thereafter they minimize the number of different labels used to speed up the operations on the priority space data structure in practice. It can be proven that the discovery phase with $|\rangle \hat{K} \langle|$ priority queue operations dominates the time complexity, giving an overall bound of $O(|\rangle \hat{K} \langle| \log |\rangle \hat{K} \langle|)$.

**KB** is a slight modification of AHRSZ. In the discovery phase AHRSZ counts the total number of edges incident on a node. KB counts instead only the in-degree of the backward frontier nodes and only the out-degree of the forward frontier nodes. In addition, KB also simplified the relabeling phase. The nodes visited during the extension of the forward (backward) frontier are deleted from the dynamic priority space data-structure and are reinserted, in the same relative order among themselves, after (before) all nodes in $R_B^{(i)}$ ($R_F^{(i)}$) not visited during the backward (forward) frontier extension. The algorithm thus computes a cover $K \subseteq \delta^{(i)}$ and its complexity per edge insertion is $O(|\rangle K \langle| \log |\rangle K \langle|)$. The worst case running time of KB for a sequence of $m$ edge insertions is $O(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$.

## 5.2 Algorithm

We keep the current topological order as a bijective function $T\colon V \to [1\,..\,n]$. If we start with an empty graph, we can initialize $T$ with an arbitrary permutation, otherwise $T$ is the topological order of the initial graph, computed offline. In this and the subsequent sections, we will use the following notations: $d(u,v)$ denotes

$|T(u) - T(v)|$, $u < v$ is a short form of $T(u) < T(v)$, $u \to v$ denotes an edge from $u$ to $v$, and $u \rightsquigarrow v$ expresses that $v$ is reachable from $u$. Note that $u \rightsquigarrow u$, but *not* $u \to u$.

Figure 5.1 gives the pseudo code of our algorithm. Throughout the process of inserting new edges, we maintain some data structures which are dependent on the current topological order. Inserting a new edge $(u,v)$ is done by calling INSERT$(u,v)$. If $v > u$, we do not change anything in the current topological order and simply insert the edge into the graph data structure. Otherwise, we call REORDER to update the topological order as well as the data structures dependent on it. As we will prove in Theorem 4, detecting $v = u$ in a call of REORDER$(u,v)$ indicates a cycle. If $v < u$, we first collect the sorted sets $A$ and $B$. $A$ is the set of out-neighbors of $v$ whose topological order is not greater than $T(u)$. Analogously, $B$ is the set of in-neighbors of $u$ whose topological order is not less than $T(v)$. If both $A$ and $B$ are empty, we swap the topological order of the two nodes and update the data structures. Otherwise, we recursively call REORDER until everything inside is topologically ordered. To make these recursive calls efficient, we first merge the sorted sets $\{v\} \cup A$ and $B \cup \{u\}$ and (using this merged list) compute the set $\{u' \colon (u' \in B \cup \{u\}) \wedge (u' \geq v')\}$ for each node $v' \in \{v\} \cup A$. The collection of sets $A$ and $B$ and the update operations are described in more detail after the data structures have been introduced.

## Data structure

We store the current topological order as a set of two arrays by maintaining the bijective mapping $T$ and its inverse $T^{-1}$. This ensures that finding $T(u)$ and $T^{-1}(i)$ are constant time operations.

The graph itself is stored as an array of vertices. For each vertex we maintain two adjacency lists, which keep the incoming and outgoing edges separately. Each adjacency list is stored as an array of buckets of vertices. Each bucket contains at most $t$ nodes for a fixed $t$. Depending on the concrete implementation of the buckets, the parameter $t$ is later chosen to be approximately $n^{0.75}$ so as to balance the number of inserts and deletes from the buckets and the extra edges touched by the algorithm. The $i$-th bucket ($i \geq 0$) of a node $x$ contains all adjacent nodes $y$ with $i \cdot t < d(x,y) \leq (i+1) \cdot t$. The nodes of a bucket are stored with node index (and not topological order) as their key. This has the advantage that there is no change necessary if two nodes that lie in the same bucket are swapped. The bucket can be kept as a balanced binary tree, as an array of $n$-bits, or as a hash-table of a universal hashing function. The only requirement for the bucket data structure is

```
INSERT(u, v)
    ▷ Insert edge (u, v) and calculate new topological order
1   if v ≤ u then REORDER(u,v)
2   insert edge (u, v) in graph
REORDER(u, v)
    ▷ Reorder nodes between u and v if v ≤ u
1   if u = v then report detected cycle and quit
2   A := {w: v → w and w ≤ u}
3   B := {w: w → u and v ≤ w}
4   if A = ∅ and B = ∅
      then ▷ Correct the topological order
5            swap T(u) and T(v)
6            update the data structure
      else ▷ Reorder node pairs between v and u
7            for v′ ∈ {v} ∪ A in decreasing topological order
8              for u′ ∈ B ∪ {u} ∧ v′ ≤ u′ in increasing topological order
9                  REORDER(u′,v′)
```

Figure 5.1: Our algorithm

that it should provide efficient support for the following three operations:

1. *Insert*: Insert an element in a given bucket.
2. *Delete*: Given an element and a bucket, find out if that element exists in that bucket. If yes, delete the element from there and return 1. Else, return 0.
3. *Collect-all*: Copy all the elements from the bucket to some vector.

Depending on how we choose to implement the buckets, we get different runtimes. This will be discussed in Section 5.5. We will now discuss how we do the insertion of an edge, computation of $A$ and $B$, and updating the data structure under swapping of nodes in terms of the above three basic operations.

Inserting an edge $(u,v)$ means inserting node $v$ in the forward adjacency list of $u$ and $u$ in the backward adjacency list of $v$. This requires $O(1)$ bucket inserts.

For given $u$ and $v$, the set $A := \{w\colon v \to w \text{ and } w < u\}$ sorted according to the current topological order can be computed from the adjacency list of $v$ by sorting all nodes of the first $\lceil d(u,v)/t \rceil$ outgoing buckets and choosing all $w$ with $w < u$. This can be done by $O\big(d(u,v)/t\big)$ collect-all operations on buckets. This means traversing all elements of $A$ as well as all elements of the $\lceil d(u,v)/t \rceil$-th outgoing bucket. Overall $O(|A|+t)$ elements are visited. These elements are integers in the range $\{1\,..\,n\}$ and can be sorted in $O(|A|+t)$ time using a two-pass radix sort algorithm since $t$ is chosen such that $t \geq n^{0.75}$. The set $B$ is computed likewise from the incoming edges.

When we swap two nodes $u$ and $v$, we need to update the adjacency lists of $u$ and $v$ as well as that of all nodes $w$ that are adjacent to $u$ and/or $v$. First, we show how to update the adjacency lists of $u$ and $v$. If $d(u,v) > t$, we build their adjacency lists from scratch. Otherwise, the new bucket boundaries will differ from the old boundaries by $d(u,v)$ and at most $d(u,v)$ nodes will need to be transferred between any pair of consecutive buckets. The total number of transfers are therefore bounded by $d(u,v)\lceil n/t \rceil$. Determining whether a node should be transferred can be done in $O(1)$ using the inverse mapping $T^{-1}$ and as noted above, a transfer can be done in $O(1)$ bucket inserts and deletes. Hence, updating the adjacency lists of $u$ and $v$ needs at most $\min\{n, d(u,v)\lceil n/t \rceil\}$ bucket inserts and deletes.

Let $w$ be a node which is adjacent to $u$ or $v$. Its adjacency list needs to be updated only if $u$ and $v$ are in different buckets. This corresponds to $w$ being in different buckets of the adjacency lists of $u$ and $v$. Therefore, the number of nodes to be transferred between different buckets for maintaining the adjacency lists of all $w$'s is the same as the number of nodes that need to be transferred for maintaining the

adjacency lists of $u$ and $v$, i. e., $\min\{n, d(u,v)\lceil n/t \rceil\}$.

Updating the mappings $T$ and $T^{-1}$ after such a swap is trivial and can be done in constant time. Thus, we conclude that swapping nodes $u$ and $v$ can be done by $O(d(u,v)\lceil n/t \rceil)$ bucket inserts and deletes.

## 5.3 Correctness

In this section we will show the following theorem.

**Theorem 1** *The above algorithm returns a valid topological order after each edge insertion.*

*Proof.* For a graph with no edges, any ordering is a correct topological order, and therefore, the theorem is trivially correct. Assuming that we have a valid topological order of a graph $G$, we show that when inserting a new edge $(u,v)$ using INSERT$(u,v)$, our algorithm maintains the correct topological order of $G' := G \cup \{(u,v)\}$. If $u < v$, this is trivial.

We need to prove that $x < y$ for all nodes $x$, $y$ of $G'$ with $x \rightsquigarrow y$. If there was a path $x \rightsquigarrow y$ in $G$, Lemma 2 gives $x < y$. Otherwise (if there is no $x \rightsquigarrow y$ in G), the path $x \rightsquigarrow y$ must have been introduced to $G'$ by the new edge $(u,v)$. Hence $x < y$ in $G'$ by Lemma 3 since there is $x \rightsquigarrow u \rightarrow v \rightsquigarrow y$ in $G'$. □

**Lemma 2** *Given a DAG G and a valid topological order, if $u \rightsquigarrow v$ and $u < v$, then all subsequent calls to* REORDER *will maintain $u < v$.*

*Proof.* Let us assume the contrary. Consider the first call of REORDER which for a node pair $u, v$ with $u \rightsquigarrow v$ and $u < v$ leads to $u > v$. Either this call led to swapping $u$ and $w$ with $v \leq w$ or it caused swapping $w$ and $v$ with $w \leq u$. Note that in our algorithm, a call of REORDER$(u,v)$ leads to a swapping only if $A = \emptyset$ and $B = \emptyset$. Assuming that it was the first case (swapping $u$ and $w$) caused by the call to REORDER$(u,w)$, $A = \emptyset$. However, since $u, v$ is the first such pair to get violated, $x \in A$ for an $x$ with $u \rightarrow x \rightsquigarrow v$, leading to a contradiction. The other case is proved analogously. □

**Lemma 3** *Given a DAG G with $v \rightsquigarrow y$ and $x \rightsquigarrow u$, a call of* REORDER$(u,v)$ *will ensure that $x < y$.*

*Proof.* Consider the recursion tree of a call to REORDER, in which the recursive

calls emanating in lines 7 and 8 are its children. The proof follows by induction on the recursion tree height of REORDER$(u,v)$. For leaf nodes (calls of REORDER with zero recursion tree height) of the recursion tree, $A = B = \emptyset$. If $x < y$ before this call, Lemma 2 ensures that $x < y$ will still hold. Otherwise, $y := v$ and $x := u$. The swapping of $u$ and $v$ in line 5 gives $x < y$.

We assume this lemma to be true for calls of REORDER up to a certain recursion tree height and consider a call with a higher recursion tree. If $A \neq \emptyset$, then there is a $\tilde{v}$ such that $v \to \tilde{v} \rightsquigarrow y$, otherwise $\tilde{v} := v = y$. If $B \neq \emptyset$, then there is a $\tilde{u}$ such that $x \rightsquigarrow \tilde{u} \to u$, otherwise $\tilde{u} := u = x$. Hence $\tilde{v} \rightsquigarrow y < x \rightsquigarrow \tilde{u}$. The **for**-loops of lines 7 and 8 will call REORDER$(\tilde{u},\tilde{v})$. By the inductive hypothesis, this will ensure $x < y$. According to Lemma 2, further calls to REORDER will maintain $x < y$. □

**Theorem 4** *The algorithm detects a cycle if and only if there is a cycle in the given edge sequence.*

*Proof.* "$\Rightarrow$": First, we show that within a call to INSERT$(u,v)$, there are paths $v \rightsquigarrow v'$ and $u' \rightsquigarrow u$ for each recursive call to REORDER$(u',v')$. This is trivial for the first call to REORDER and follows immediately by the definition of $A$ and $B$ for all subsequent recursive calls to REORDER. This implies that if the algorithm indicates a cycle in line 1 of REORDER, there is indeed a cycle $u \to v \rightsquigarrow v' = u' \rightsquigarrow u$. In fact, the cycle itself can be computed using the recursion stack of the current call to REORDER.

"$\Leftarrow$": Consider the edge $(u,v)$ of the cycle $v \rightsquigarrow u \to v$ inserted last. Since $v \rightsquigarrow u$ before the insertion of this edge, the topological order computed will satisfy $v < u$ (Theorem 1) and therefore, REORDER$(u,v)$ would be called. In fact, all edges in the path $v \rightsquigarrow u$ will obey the current topological ordering and by Lemma 2, it will remain so for all subsequent calls of REORDER. We prove by induction on the number of nodes in the path $v \rightsquigarrow u$ (including $u$ and $v$) that whenever $v \rightsquigarrow u$ and REORDER$(u,v)$ is called, it detects the cycle. A call of REORDER$(u',v')$ with $u' = v'$ or REORDER$(u',v')$ with $v' \to u'$ clearly reports a cycle. Consider a path $v \to x \rightsquigarrow y \to u$ of length $k > 2$ and the call of REORDER$(u,v)$. As noted before, $v < x \leq y < u$ before the call to REORDER$(u,v)$. Hence $x \in A$ and $y \in B$ and a call to REORDER$(y,x)$ will be made in the for loop of lines 7 and 8. As $y \rightsquigarrow x$ has $k-2$ nodes in the path, the call to REORDER$(y,x)$ (by our inductive hypothesis) will detect the cycle. □

## 5.4 Runtime

The following theorem is the main result of this section.

**Theorem 5** *Incremental topological ordering can be maintained while processing any sequence of edge insertions using $O(n^{3.5}/t)$ bucket inserts and deletes, $O(n^3/t)$ bucket collect-all operations collecting $O(n^2t)$ elements, and $O(n^{2.5} + n^2t)$ operations.*

*Proof.* Consider the pseudo code in Figure 5.1. Since there can be a maximum of $n(n-1)/2$ edges inserted in a DAG, there are $O(n^2)$ calls of INSERT. Inserting an edge in the graph involves $O(1)$ bucket operations and therefore, the total cost of Line 2 of INSERT is $O(n^2)$.

Lemma 8 shows that REORDER is called $O(n^2)$ times. Line 1 of REORDER requires $O(1)$ operations per call of REORDER, except the one time it does encounter a cycle (when it requires $O(n)$ time). Lemma 10 shows that the calculation of the sets $A$ and $B$ over all calls of REORDER can be done by $O(n^3/t)$ bucket collect-all operations touching $O(n^2t)$ edges, and $O(n^{2.5} + n^2t)$ operations. Lines 4 and 5 require $O(1)$ operations per call of REORDER. In Lemma 12, we prove that all the updates can be done by $O(n^{3.5}/t)$ bucket inserts and deletes.

For lines 7 and 8 of the pseudo-code, we first merge the two sorted sets $A$ and $B$. This takes $O(|A| + |B|)$ operations. For a particular node $v' \in \{v\} \cup A$, we can compute the set $V' = \{u' \colon (u' \in B \cup \{u\}) \wedge (u' \geq v')\}$ (as required by line 8) using this merged set in complexity $O(1 + |V'|)$, which is also the number of calls of REORDER emanating for this particular node. Summing over the entire *for* loop of line 7, the total complexity of lines 7 and 8 is $O(|A| + |B| +$ number of calls of REORDER emanating from here). Since by Lemma 9, the summation of $|A| + |B|$ over all calls of REORDER is $O(n^2)$ and by Lemma 8, the total number of calls to REORDER is also $O(n^2)$, we get a total of $O(n^2)$ operations for lines 7 and 8. The theorem follows by simply adding the complexity of each line. □

**Lemma 6** REORDER *is local, i. e., a call to* REORDER$(u,v)$ *does not affect the topological ordering of nodes $w$ such that either $w < v$ or $w > u$ just before the call was made.*

*Proof.* This lemma can be proven by induction on the level of the recursion tree of a call to REORDER$(u,v)$. For the leaf node of the recursion tree, $|A| = |B| = 0$ and the topological order of $u$ and $v$ is swapped, not affecting the topological ordering

of any other node.

We assume this lemma to be true up to a certain tree level. To see that it is also valid for one level higher, note that the arrays $A$ and $B$ contain elements $w$ such that $v < w < u$. Since each call of REORDER in the **for**-loop of line 7 and 8 is from an element of $A$ to an element of $B$ and all of these calls are themselves local by our induction hypothesis, this call of REORDER is also local. □

**Lemma 7** *If two nodes are swapped in a call of* REORDER*, their relative order will remain unchanged in the future.*

*Proof.* Let us assume, two nodes $u'$ and $v'$ are swapped within one of the recursive calls of REORDER invoked by INSERT$(u,v)$. After the insertion of edge $(u,v)$, there is a path $u' \rightsquigarrow u \rightarrow v \rightsquigarrow v'$. Therefore, by Lemma 2 the relative order of $u'$ and $v'$ will not be changed in any subsequent call of INSERT.

It remains to prove that also within the recursion tree of REORDER$(u,v)$, the relative order of $u'$ and $v'$ will not be changed after they have been swapped. This is ensured by the order in which the two **for**-loops in lines 7 and 8 iterate since there can be no calls to REORDER$(u',w)$ with $w > v'$ or REORDER$(w,v')$ with $u < u'$ after the call of REORDER$(u',v')$. □

**Lemma 8** REORDER *is called $O(n^2)$ times.*

*Proof.* As we have proven that the algorithm is correct in section 5.3, we now know that for each pair $(u,v)$ the following holds: If REORDER$(u,v)$ is called, then $v \leq u$ holds before and $u \leq v$ holds afterwards. As by Lemma 7 this implies that REORDER$(u,v)$ can only be called once for each pair $(u,v)$, the number of calls to REORDER can be upper bounded by $n^2$. □

**Lemma 9** *The summation of $|A| + |B|$ over all calls of* REORDER *is $O(n^2)$.*

*Proof.* Consider arbitrary nodes $u$ and $v'$. We prove that for all $v \in V$, $v' \in A$ happens only once over all calls of REORDER$(u,v)$. This proves that $\sum |A| \leq n$, for all such calls of REORDER$(u,v)$. Therefore, summing up for all $u \in V$, $\sum |A| \leq n^2$ over all calls of REORDER.

In order to see that for all $v \in V$, $v' \in A$ happens only once over all calls of REORDER$(u,v)$, consider the first such call. Since $v' \in A$, $v' < u$ and $v \rightarrow v'$ before the call was made. By Lemma 3, $u < v'$ after this call and hence, $v' \notin A$ for any call of REORDER afterwards. As for calls within the recursive substructure of the first call, the order in which these calls are made ensures that there will be no calls

of REORDER$(u,w)$ for any $w < v'$ before REORDER$(u,v')$ and since $u < v'$ after REORDER$(u,v')$, $v' \notin A$ for REORDER$(u,w)$.

Analogously, it can be proven that for arbitrary nodes $v$ and $v'$ and for all $u \in V$, $v' \in B$ happens only once over all calls of REORDER$(u,v)$. The proof for $\sum |B| \leq n^2$ follows similarly and it completes the proof of this lemma. □

**Lemma 10** *Calculating the sorted sets A and B over all calls of* REORDER *can be done by $O(n^3/t)$ bucket collect-all operations touching a total of $O(n^2 t)$ elements and $O(n^{2.5} + n^2 t)$ operations for sorting these elements.*

*Proof.* Consider the calculation of set $A$ in a call of REORDER$(u,v)$. As discussed before in section 5.2, we look at the out adjacency list of $u$, stored in the form of buckets. In particular, we will need $O(d(u,v)/t)$ bucket collect-all operations touching $O(|A|+t)$ elements to calculate $A$. The additional worst-case factor of $t$ stems from the last bucket visited. Summing up over all calls of REORDER, we get $O\big(\sum d(u,v)/t\big)$ collect-all's touching $\sum(|A|+|B|+t)$ elements. Since $d(u,v) \leq n$ for every call of REORDER$(u,v)$ and there are $O(n^2)$ calls of REORDER (Lemma 8), there are $O(n^3/t)$ bucket collect-all operations. Also, since $\sum(|A|+|B|) = O(n^2)$ by Lemma 9, the total number of elements touched is $O(n^2 + \sum t) = O(n^2 t)$. Since the keys are in the range $\{1..n\}$, we can use a two-pass radix sort to sort the elements collected from the buckets. The total sorting time over all calls of REORDER is $\sum(2(|A|+t)+\sqrt{n}) + \sum(2(|B|+t)+\sqrt{n}) = O(n^{2.5} + n^2 t)$. □

**Lemma 11** $\sum d(u,v) = O(n^{5/2})$ *where the summation is taken over all calls of* REORDER*(u,v) in which u and v are swapped.*

*Proof.* Let $T^*$ denote the final topological ordering and

$$X(T^*(u), T^*(v)) := \begin{cases} d(u,v) & \text{if REORDER}(u,v) \text{ leads to a swapping} \\ 0 & \text{otherwise} \end{cases}$$

As Lemma 7 implies that each node pair is swapped at most once, the variable $X(i,j)$ is clearly defined. Next, we model a few linear constraints on $X(i,j)$, formulate it as a linear program and use this LP to prove that $\max\{\sum_{i,j} X(i,j)\} = O(n^{5/2})$. By definition of $d(u,v)$ and $X(i,j)$,

$$0 \leq X(i,j) \leq n \text{ for all } i,j \in [1..n].$$

For $j \leq i$, the corresponding edges $(T^{*\,-1}(i), T^{*\,-1}(j))$ go backwards and thus are never inserted at all. Consequently,

$$X(i,j) = 0 \text{ for all } j \leq i.$$

Now consider an arbitrary node $u$, which is finally at position $i$, i. e., $T^*(u) = i$. Over the insertion of all edges, this node has been moved left and right via swapping with several other nodes. Strictly speaking, it has been swapped right with nodes at final positions $j > i$ and has been swapped left with nodes at final positions $j < i$. Hence, the overall movement to the right is $\sum_{j>i} X(i,j)$ and to left is $\sum_{j<i} X(j,i)$. Since the net movement (difference between the final and the initial position) must be less than $n$,

$$\sum_{j>i} X(i,j) - \sum_{j<i} X(j,i) \leq n \text{ for all } 1 \leq i \leq n.$$

Putting all the constraints together, we aim to solve the following linear program.

$$\max \sum_{\substack{1\leq i\leq n\\ 1\leq j\leq n}} X(i,j) \text{ such that}$$

(i) $X(i,j) = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$,
(ii) $0 \leq X(i,j) \leq n$ for all $1 \leq i \leq n$ and $i < j \leq n$,
(iii) $\sum_{j>i} X(i,j) - \sum_{j<i} X(j,i) \leq n$ for all $1 \leq i \leq n$.

Note that these are necessary constraints, but not sufficient. But this is enough for our purpose as an upper bound to the solution of this LP will give an upper bound for the $\sum X(i,j)$ in our algorithm. In order to prove the upper bound on the solutions of this LP, we consider the dual problem

$$\min \left[ n \sum_{\substack{0\leq i<n\\ i<j<n}} Y(i\cdot n+j) \;+\; n \sum_{0\leq i<n} Y(n^2+i) \right] \text{ such that}$$

(i) $Y(i\cdot n+j) \geq 1$ for all $0 \leq i < n$ and $j \leq i$,
(ii) $Y(i\cdot n+j) + Y(n^2+i) - Y(n^2+j) \geq 1$ for all $0 \leq i < n$ and $j > i$,
(iii) $Y(i) \geq 0$ for all $0 \leq i < n^2+n$.

and the following feasible solution for the dual:

$$\begin{aligned} Y(i\cdot n+j) &= 1 && \text{for all } 0 \leq i < n \text{ and } 0 \leq j \leq i,\\ Y(i\cdot n+j) &= 1 && \text{for all } 0 \leq i < n \text{ and } i < j \leq i+1+2\sqrt{n},\\ Y(i\cdot n+j) &= 0 && \text{for all } 0 \leq i < n \text{ and } j > i+1+2\sqrt{n},\\ Y(n^2+i) &= \sqrt{n-i} && \text{for all } 0 \leq i < n. \end{aligned}$$

This solution has a value of $n^2 + 2n^{5/2} + n\sum_{i=1}^{n}\sqrt{i} = O(n^{5/2})$, which by the primal-dual theorem is a bound on the solution of the original LP.

In fact, it can be shown that there is a solution to primal LP whose value is $O(n^{5/2})$, namely

$$X(i,j) = 0 \qquad \text{for all } 0 \le i < n \text{ and } 0 \le j \le i,$$

$$X(i,j) = n \qquad \text{for all } 0 \le i < n \text{ and } i < j \le i + \lceil \tfrac{\sqrt{1+8i}-1}{2} \rceil,$$

$$X(i,j) = 0 \qquad \text{for all } 0 \le i < n \text{ and } j > i + \lceil \tfrac{\sqrt{1+8i}-1}{2} \rceil. \qquad \square$$

**Lemma 12** *Updating the data structure over all calls of* REORDER *requires* $O(n^{3.5}/t)$ *bucket inserts and deletes.*

*Proof.* Our data structure requires $O(d(u,v)\,n/t)$ bucket inserts and deletes to swap two nodes $u$ and $v$. Lemma 7 shows that each node pair is swapped at most once. Hence, summing up over all calls of REORDER$(u,v)$ where $u$ and $v$ are swapped, we need $O(\sum d(u,v)\,n/t) = O(n^{3.5}/t)$ bucket inserts and deletes using Lemma 11. $\square$

## 5.5 Bucket data structure

We get different runtimes and space requirements of our algorithm depending on the data structures of the buckets used:

(a) Balanced binary trees (see e. g. [77]): Balanced binary trees give us $O(1+\log\tau)$ time insert and delete and $O(1+\tau)$ time collect-all operation, where $\tau$ is the number of elements in the bucket. Therefore, by Theorem 5, the total time required will be $O(n^2 t + n^{3.5}\log n/t)$. Substituting $t = n^{0.75}\sqrt{\log n}$, we get a total time of $O(n^{2.75}\sqrt{\log n})$. The total space requirement will be $O(n^2)$ as a balanced binary tree needs $O(t)$ nodes for storing at most $t$ elements.

(b) $n$-bit array: A bucket that stores at most $t$ elements can be kept as an $n$-bit array, where each bit is 0 or 1 depending on whether or not the element is present in the bucket. Also, we can keep a list of all elements in the bucket. To insert, we just flip the appropriate bit and insert at the end of the list. To delete, we just flip the appropriate bit. To collect all, we go through the list and for each element in the list, we check if the corresponding bit is 1 or 0. If it is 0, we also remove it from the list. This gives us constant-time insert

and delete and the time for collect-all operation will be the total output size plus the total number of delete. Each delete is counted once in collect-all as we remove the corresponding element from the list after the first collect-all. By Theorem 5, the total time required will be $O(n^2t + n^{3.5}/t)$, giving us $O(n^{2.75})$ for $t = n^{0.75}$. The total space requirement will be $O(n)$ for each bucket, leading to a total of $O(n^{2.25})$ for $O(n^2/t)$ buckets.

(c) Uniform Hashing [121]: A data structure based on uniform hashing coupled with a list of elements in the bucket operated in the same way as the $n$-bit array will give an expected constant-time insert and delete and the same bound for collect-all as for the $n$-bit array. This gives an expected total time of $O(n^2t + n^{3.5}/t)$. With $t = n^{0.75}$ this yields an expected time of $O(n^{2.75})$. Since the hashing based data structure as described in [121] takes only linear space, the total space requirement is $O(n^2)$.

## 5.6 Empirical comparison

We conducted our experiments on a 2.4 GHz Opteron machine with 8GB of main memory running Debian GNU/Linux. For PK, MNR, and AHRSZ we used the C++/Boost based implementation of David J. Pearce (see [124]). For our algorithm (AFM), we implemented variant (b) of section 5.5 using C++/STL. Additionally, we also implemented a local (cf. Lemma 6) variant of KB using an ordered bi-directional list data structure [60]. The code of AFM and KB is available upon request. All codes were compiled using gcc 3.3 in 32-bit mode and optimization level `-O3`. The timings were measured using the `gettimeofday` function of `<sys/time.h>` and all the results are averaged over 10 runs each.

We examined all five algorithms on two classes of DAGs. First, we considered random edge insertion sequences leading to a complete DAG. This random DAG model by [26] is similar to the well-known $G(n,m)$ random graph model of [66]. On a random edge sequence, all the algorithms are quite fast and none of them encounters its worst-case behavior. Therefore, we also considered a particular sequence of edges which we believe is a hard instance of the problem. This edge sequence is similar to the worst-case sequence given by [91] for their algorithm. On this sequence, KB, PK, MNR, and AHRSZ (the variant choosing the smallest permitted priority) face their worst-case of $\Omega(n^3)$ operations, while our algorithm takes $\Omega(n^{2.5})$ time complexity. This sequence of edges is depicted in Fig. 5.2. Let us briefly describe its structure. For a graph with $n$ nodes, we divide the set of nodes into four blocks of different sizes: block 1 consists of nodes $[0..n/3)$,

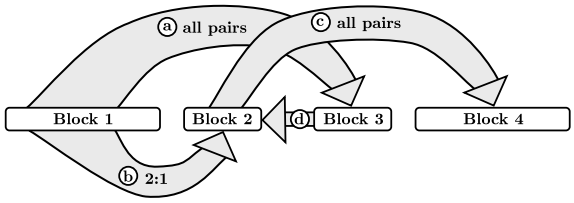



Figure 5.2: Our hard-case graph

block 2 of nodes $[n/3\,..\,n/2)$, block 3 of nodes $[n/2\,..\,2n/3)$, and block 4 of nodes $[2n/3\,..\,n)$. First, we insert $n-4$ edges such that within each block, the vertices form a directed path from left to right. Then we insert the following edges,

(a) $\overrightarrow{\forall}\, j \in [0..n/3)\ \overleftarrow{\forall}\, k \in [0..n/6)$ : add edge$(j, k+n/2)$,

(b) $\overrightarrow{\forall}\, j \in [0..n/6)$ : add edge$(2j, j+n/3)$ and edge$(2j+1, j+n/3)$,

(c) $\overrightarrow{\forall}\, j \in [0..n/6)\ \overleftarrow{\forall}\, k \in [0..n/3)$ : add edge$(j+n/3, k+2n/3)$,

(d) $\overrightarrow{\forall}\, j \in [0..n/6)\ \overleftarrow{\forall}\, k \in [0..n/6)$ : add edge$(j+n/2, k+n/3)$,

where $\overrightarrow{\forall}$ denotes going from left to right in the **for**-loop and $\overleftarrow{\forall}$ the other way around.

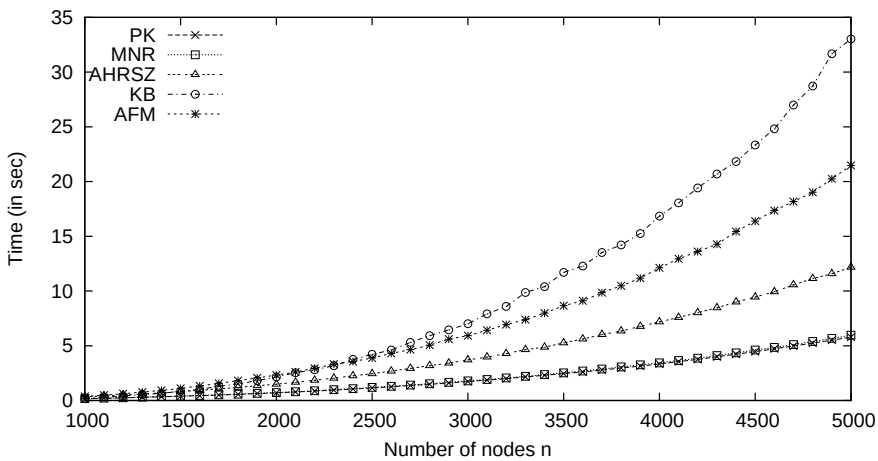



Figure 5.3: Experimental data on full random graphs with varying $n$.

Fig. 5.3 shows the runtimes of the five algorithms in consideration for random

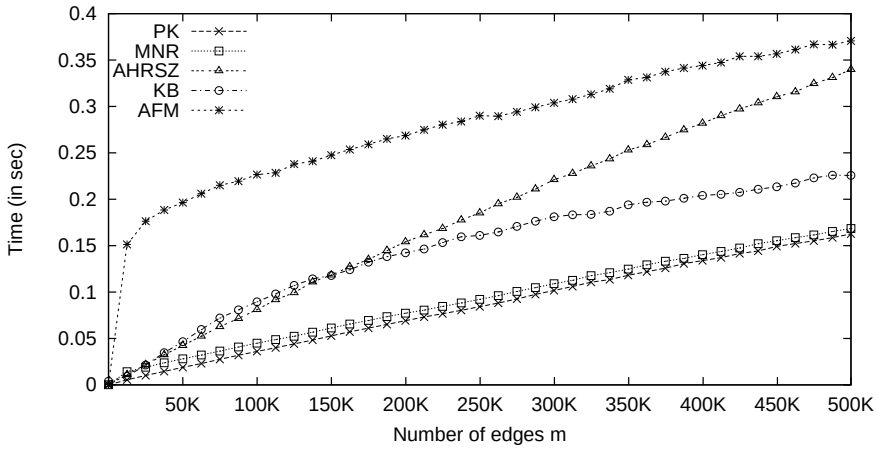



Figure 5.4: Experimental data on random graphs with $n = 1000$ and varying $m$.

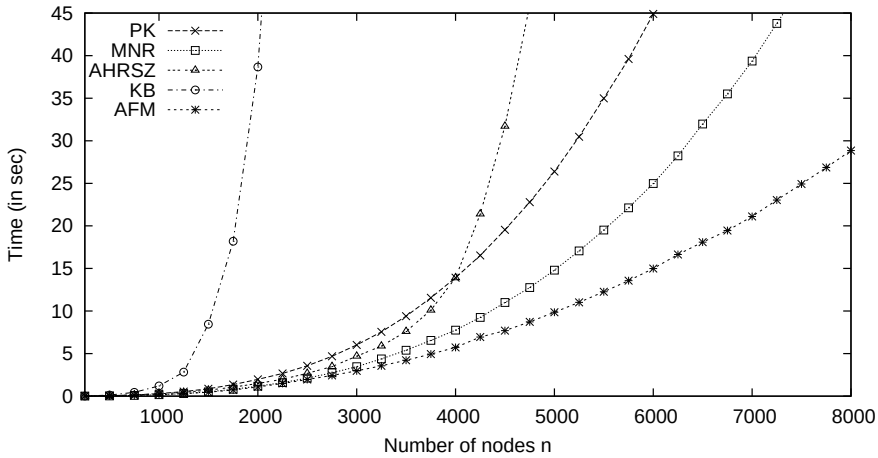



Figure 5.5: Experimental data on a class of hard instances with varying $n$.

edge sequences leading to complete DAGs with varying number $n$ of vertices (and with $m = \binom{n}{2}$). We see that AFM is approximately 30% faster than KB and a constant factor of 2-4 away from AHRSZ, MNR, and PK.

Fig. 5.4 shows the average runtimes for random graphs with $n = 1000$ and a varying number of edges. AFM looses a lot during the insertion of the first $O(n \log n)$ edges because in this phase, updating the data structures after every swapping proves very costly. But after that, the curves between AFM and PK/MNR/KB are almost parallel, while the slope for AHRSZ is around 2 times that of AFM. For practical purposes, we believe therefore that a hybrid approach would perform best. That is, one inserts the first $O(n \log n)$ edges with either PK or KB and then

inserts the remaining edges with our algorithm.

Fig. 5.5 shows the runtimes of the five algorithms in consideration on the class of hard edge sequences described before. The difference in asymptotic behaviour as discussed before is clear from the graph.

## 5.7 Towards a tighter analysis of our algorithm

It is not clear if the analysis of our algorithm as shown in section 5.4 and section 5.5 is tight. We conjecture that the analysis of our algorithm can be improved. In this section we describe an approach that can potentially improve the analysis.

Consider the following problem: We are given two sets $A$ and $B$ of nodes and we construct a graph based on the following rules:

- We start with an empty graph
- In order to add an edge in the graph, we select a node $u \in B$ and $v \in A$, swap them (i.e., after the swap, $u \in A$ and $v \in B$), and insert a directed edge from $u$ to $v$.
- At no point of this construction, there should be an edge from any node in $B$ to any node in $A$.

Figure 5.6 shows an example with valid and invalid moves for constructing such graphs.

Our combinatorial problem is to bound the maximum number of edges $E(|A|,|B|)$ that can be inserted in this way.

Here are a few properties that we can conclude about the resulting graph:

**Theorem 13** *The resulting graph will be a directed acyclic graph.*

*Proof.* We will prove this by contradiction. Assume that there is a directed cycle in the resulting graph and consider the last edge $e = (u,v)$ of this cycle being inserted. In other words, before the insertion of this edge, there is a path from $v$ to $u$ and this edge completes the cycle. After inserting this edge, $u \in A$ and $v \in B$. Since, there is a path from $v$ to $u$, there will be some edge in the path that goes from some node in $B$ to some node in $A$ (as the path starts from $B$ and eventually

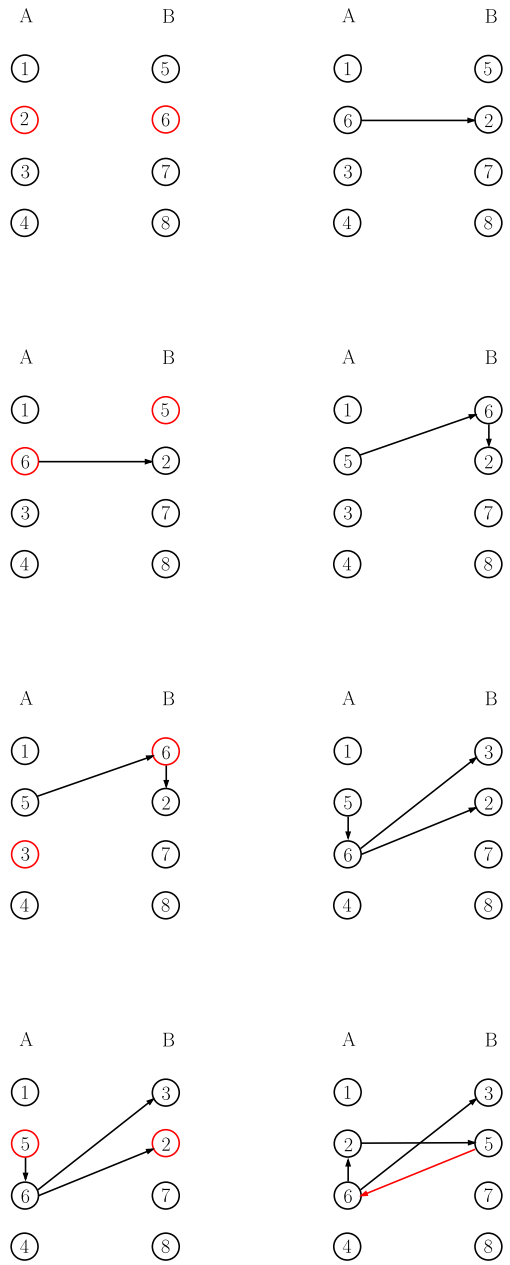



Figure 5.6: An example of inserting edges in the combinatorial graph. The node pairs marked in red in the images on the left are being considered for putting the next edge and the right side shows the resulting ordering of nodes. The first three edge insertions are legal while the last edge is not allowed.

reaches $A$). This edge clearly violates our constraint and thus the edge $e = (u,v)$ will not be inserted in the first place. This leads to a contradiction and proves the fact that there can't be directed cycles in this graph. □

**Theorem 14** $E(n/2, n/2) = \Omega(n \log n)$.

*Proof.* In order to prove this, we need to show an example where a graph with $\Omega(n \log n)$ edges can be constructed in this way. If $|A| = |B| = 1$, then we simply swap the two nodes and insert the corresponding edge. Otherwise, we first recursively build two graphs with $n/4$ nodes in each set. Let's call the sets $A$ and $B$ of the first graphs as $A_1$, $B_1$ and that of the second graph as $A_2$, $B_2$. Then we sort the nodes in both sets of both graphs topologically and then insert the $n/4$ edges in the following sequence: We start from the topologically smallest node in set $B_1$ and insert an edge to topologically biggest node in set $A_2$. Thereafter we put the edge from the next smallest (topologically) node of $B_1$ to second largest (topologically) node of set $A_2$ of the second graph and so on. It is easy to check that this sequence of edge insertions never leads to any edge from $B := B_1 \cup B_2$ to $A := A_1 \cup A_2$. Constructing the graph in such a way, we find that $E(n/2, n/2) \geq 2E(n/4, n/4) + n/4$. Since $E(1,1) = 1$, $E(n/2, n/2) = \Omega(n \log n)$. □

**Theorem 15** $E(i, n-i) = O(n^{3/2})$ *for all* $1 \leq i \leq n-1$.

*Proof.* The resulting graph will have the following properties:

- It is a directed acyclic graph (cf. Theorem 13)
- The difference between in-degree and out-degree of any node is at most one. This follows from the fact that a node goes from a set $B$ to a set $A$ iff its out-degree increases by one and a node goes from a set $A$ to a set $B$ iff its in-degree increases by one. Since all nodes start from either $A$ or $B$ and end up in $A$ or $B$, the difference between the in-degree and out-degree of any node can be atmost one.

Next, we show that a DAG in which each node has $|\text{out-degree} - \text{in-degree}| \leq 1$ has $O(n^{3/2})$ edges. This is shown by an LP based proof. Let $T^*$ denote the final topological ordering and $X(T^*(u), T^*(v)) := 1$ iff there is an edge from $u$ to $v$. Thus, the maximum number of edges in such a DAG is equal to

$$\max \sum_{\substack{1 \leq i \leq n \\ 1 \leq j \leq n}} X(i,j) \text{ such that}$$

(i) $X(i,j) = 0$ for all $1 \leq i \leq n$ and $1 \leq j \leq i$,
(ii) $0 \leq X(i,j) \leq 1$ for all $1 \leq i \leq n$ and $i < j \leq n$,
(iii) $\sum_{j>i} X(i,j) - \sum_{j<i} X(j,i) \leq 1$ for all $1 \leq i \leq n$.

Similar to the proof of Lemma 11, it can be shown that the solution of this LP and hence, the maximum number of edges in such a DAG is $O(n^{3/2})$. □

The following is our main theorem that links the maximum number of edges in this graph to the analysis of online topological ordering algorithms.

**Theorem 16** $\sum_{u,v} d(u,v) \leq \sum_{i=1}^{n-1} E(i,n-i)$

*Proof.* Consider a particular position $(i,n-i)$ in the topological ordering, i.e., $i$ nodes are to the left and $n-i$ nodes are to the right of this position. We say that a node-pair $(u,v)$ crosses the position $(i,n-i)$ if in the topological ordering before swapping the nodes $u$ and $v$, $T(u) > i$ and $T(v) \leq i$ and after the swapping $T(u) \leq i$ and $T(v) > i$.

Throughout the execution of the online topological ordering algorithm, the number of node-pairs that cross this position can be at most $E(i,n-i)$. This is because the nodes to the left and right can be thought of as belonging to two different sets and we never allow edges from the right of this position to the left. Whenever we want to insert an edge, the algorithm first swaps their location and always puts the edge from the left to the right.

Consider the set $L := \{((u,v),(i,n-i)) | \text{node-pair } (u,v) \text{ crosses the position } (i,n-i)\}$. Clearly, $|L| = \sum_{u,v} d(u,v)$ as each node-pair $(u,v)$ crosses $d(u,v)$ positions. For every position $(i,n-i)$, the number of node-pairs crossing this position is at most $E(i,n-i)$ as shown before. Since, $L = \cup_{1 \leq i \leq n-1} |\{(u,v) | (u,v) \text{ crosses } (i,n-i)\}$, $|L| \leq \sum_{i=1}^{n-1} E(i,n-i)$. Putting together, $\sum_{u,v} d(u,v) = |L| \leq \sum_{i=1}^{n-1} E(i,n-i)$. □

This implies that if one can prove that for all $1 \leq i \leq n-1$, $E(i,n-i) = o(n^{3/2})$, than the analysis of all the topological ordering algorithm relying on $\sum_{u,v} d(u,v)$ for their analysis such as ours (cf. Section 5.10 for another algorithm that relies on Lemma 11) will get improved.

## 5.8 Dynamic topological ordering in external memory

Many information retrieval applications rely on being able to query ontology (e.g., Gene Ontology, SUMO, Cyc, YAGO, DBpedia etc.) graphs for connectivity, reachability, BFS, shortest paths, steiner trees etc. [89] to learn relations between different semantic entities. Natural relations (e.g., $x$ is located in $y$, $w$ is a sub-class of $z$) between these entities are often acyclic and transitive and can thus be modeled as directed acyclic graphs [148]. These ontology DAGs can be quite large. For instance, DBpedia 3.1 has more than 100 million edges [52].

In external memory, efficient computation of topological ordering is particularly important as many different traversal problems such as reachability, BFS, SSSP etc. can be reduced to computing topological ordering in $O(sort(m))$ I/Os. This is done using the technique of time-forward processing (cf. Section 2.5.6) as follows: Given the topological ordering of the DAG $G(V,E)$, we sort the adjacency lists according to the topological ordering of their tail nodes and we process the nodes in this order. We ignore all nodes until we reach the source node. We mark the source node as reachable or visited with BFS level zero or distance zero from the source. This information is then propagated to its out-neighbors who will be processed in future using an external memory priority queues. The information is entered into the priority queue with the topological number of the head node as the key. When we process any node $v$ after having processed the source node, we first extract all the information from the priority queue kept for this node (with $v$'s topological number as its key) by its in-neighbors. The reachability, BFS level or shortest path distance for this node is then computed based on this information. This is then propagated forward to its out-neighbors using the external memory priority queue. Since all the priority queue operations can be performed in $O(sort(m))$ I/Os and sorting the adjacency lists also requires $O(sort(m))$ I/Os, reachability, BFS and shortest paths can all be computed on large DAGs using $O(sort(m) + TO(n,m))$ I/Os, where $TO(n,m)$ is the number of I/Os required to compute the topological ordering of a DAG with $n$ nodes and $m$ edges.

The best-known algorithm for computing topological ordering in external memory is based on directed DFS [43] and requires $O((n+m/B)\log_2 \frac{n}{B} + sort(m))$ I/Os. The naïve way of recomputing from scratch whenever a new edge is inserted requires the same number of I/Os and is thus, very inefficient.

Fortunately, we can improve upon this by using time-forward processing. We know the topological ordering $T_{old}$ of the DAG before the new edge is inserted

and we process the nodes in that order. As in all the dynamic topological ordering algorithms seen so far, we do not do anything if the new edge $(u,v)$ is not invalidating. Otherwise, for all nodes $w$ such that $T_{old}(w) < T_{old}(v)$, we assign $T_{new}(w) = T_{old}(w)$ as they are not affected by the new edge. We start processing the nodes by assigning $T_{new}(v) := T_{old}(u)+1$. This information is then propagated forward using an external memory priority queue by inserting $T_{new}(v)+1$ with priority $T_{old}(v')$, for each out-neighbor $v'$ of $v$. If a node $x$ being processed has not received any information from its in-neighbors, $T_{new}(x) := T_{old}(x)$. Otherwise, $x$ updates its topological number as the maximum of all entries extracted from the priority queue with the priority $T_{old}(x)$, and $T_{old}(x)$. This is then communicated forward by inserting $T_{new}(x)+1$ with priority $T_{old}(x')$ for each out-neighbor $x'$ of $x$.

In case we want to get $T_{new}\colon V \to [1\,..\,n]$, we can easily do so by sorting the nodes according to $T_{new}$ and assigning them numbers one to $n$. The whole process of computing a new topological ordering thus only requires $O(sort(m))$ I/Os.

Our algorithm can be externalized to give an $O\left(n^{2.75} \cdot \sqrt{\frac{\log_{M/B} n \cdot \log_B n}{B}}\right)$ I/Os for maintaining the topological ordering under the insertion of $m$ edges. For inserting $m'$ edges into a DAG with $m$ edges, this is an improvement over the $O(sort(m))$ I/O algorithm if $m' = \omega\left(\frac{n^{2.75}}{sort(m)} \cdot \sqrt{\frac{\log_{M/B} n \cdot \log_B n}{B}}\right)$.

Since our algorithm requires to keep $O(n^2/t)$ $(= O(n^{1.25}))$ buckets simultaneously, it is not possible to even keep one element per bucket in the internal memory if $n^{1.25} > M$. We therefore, keep all the buckets completely in the external memory. These buckets are implemented as dynamic B-trees. Inserting an element requires $O(1)$ I/Os, non-lazy deletion (which includes searching) requires $O(\log_B n)$ I/Os and collect-all operation requires $O(1+k/B)$ I/Os for collecting $k$ elements. Recall from Theorem 5 that our algorithm requires $O(n^{3.5}/t)$ bucket inserts and deletes, and $O(n^3/t)$ bucket collect-all operations collecting $O(n^2 \cdot t)$ elements for processing any sequence of edge insertions. These operations require $O\left(\frac{n^{3.5} \cdot \log_B n}{t} + \frac{n^2 \cdot t}{B}\right)$ I/Os in total.

Sorting all elements collected from the buckets to compute sets $A$ and $B$ can be done using external memory sorting algorithms (cf. Section 2.5.2). In the worst case, there may be $\Omega(n^2)$ calls (one for each call of REORDER) sorting $O(n^2 \cdot t)$ elements in total. Summing over all calls, this requires $O(n^2 + n \cdot t \cdot sort(n))$ I/Os.

All other operations including accesses to $T$ and $T^{-1}$ require $O(n^{3.5}/t)$ I/Os.

Thus, the externalized version of our algorithm requires $O\left(\frac{n^{3.5}\cdot\log_B n}{t} + n\cdot t\cdot sort(n)\right)$ I/Os. Substituting $t := n^{0.75}\cdot\sqrt{\frac{B\cdot\log_B n}{\log_{M/B} n}}$, we get that our external dynamic topological ordering algorithm requires $O\left(n^{2.75}\cdot\sqrt{\frac{\log_{M/B} n\cdot\log_B n}{B}}\right)$ I/Os.

## 5.9 Average-case analysis of online topological ordering algorithms

The algorithm by Pearce and Kelly (PK) [124] empirically outperforms the other algorithms for random edge insertions, although its worst-case runtime is inferior to KB. This difference in the behavior of online topological ordering algorithms between random edge insertion sequences (REIS) and worst-case sequences lead us to the theoretical study of online topological ordering algorithms on REIS.

In this section, we show an expected runtime of $O(n^2\log^2 n)$ for inserting all edges of a complete DAG in a random order with PK. Also, we show an expected runtime of $O(n^2\log^3 n)$ for complete random edge insertion sequences for AHRSZ and KB.

Recall from Section 2.3 that by directing the edges of an undirected random graph from lower to higher indexed vertices, we obtain the random DAG model of Barak and Erdős [26]. Depending on the underlying random graphs, we get two random DAG models - $DAG(n,m)$ and $DAG(n,p)$. In this section, we will prove our main results on the $DAG(n,m)$ model since it is better suited to describe incremental addition of edges. However, since the independence of edges in the $DAG(n,p)$ model makes the analysis easier, we will prove our results first on $DAG(n,p)$ and then use Theorem 2 to get the corresponding $DAG(n,m)$ results.

### 5.9.1 Analysis of PK

When inserting the $i$-th edge $u \to v$, PK only regards nodes in $\delta^{(i)} := \{x \in V \mid v \leq x \leq u \wedge (v \rightsquigarrow x \vee x \rightsquigarrow u)\}$ with "$\leq$" defined according to the current topological order. As discussed in Section 5.1, PK performs $O(\|\delta^{(i)}\| + |\delta^{(i)}|\log|\delta^{(i)}|)$ operations for inserting the $i$-th edge. The intuition behind the proofs in this section is that in the early phase of edge-insertions (the first $O(n\log n)$ edges), the graph is sparse and so only a few edges are traversed during the DFS traversals. As the

graph grows, fewer and fewer nodes are visited in DFS traversals ($|\delta^{(i)}|$ is small) and so the total number of edges traversed in DFS traversals (bounded above by $\|\delta^{(i)}\|$) is still small.

Theorems 19 and 25 of this section show for a random edge insertion sequence (REIS) of $N$ edges that $\sum_{i=1}^{N} |\delta^{(i)}| = O(n^2)$ and $\mathbf{E}\left[\sum_{i=1}^{N} \|\delta^{(i)}\|\right] = O(n^2 \log^2 n)$. This proves the following theorem.

**Theorem 17** *For a random edge insertion sequence (REIS) leading to a complete DAG, the expected runtime of PK is $O(n^2 \log^2 n)$.*

A comparable pair (of nodes) are two distinct nodes $x$ and $y$ such that either $x \rightsquigarrow y$ or $y \rightsquigarrow x$. We define a potential function $\Phi_i$ similar to Katriel and Bodlaender [91]. Let $\Phi_i$ be the number of comparable pairs after the insertion of $i$ edges. Clearly,

$$\begin{aligned} \Delta\Phi_i := \Phi_i - \Phi_{i-1} \geq 0 \quad & \text{for all } 1 \leq i \leq m, \\ \Phi_0 = 0, \quad \text{and} \quad & \Phi_M \leq n(n-1)/2. \end{aligned} \tag{5.1}$$

**Theorem 18** *For all edge sequences, (i) $|\delta^{(i)}| \leq \Delta\Phi_i + 1$ and (ii) $|\delta^{(i)}| \leq 2\Delta\Phi_i$.*

*Proof.* Consider the $i$-th edge $(u,v)$. If $u < v$, the theorem is trivial since $|\delta^{(i)}| = 0$. Otherwise, each vertex of $R_F^{(i)}$ and $R_B^{(i)}$ (as defined in Section 5.1) gets newly ordered with respect to $u$ and $v$, respectively. The set $\bigcup_{x \in R_B^{(i)}} (x,v) \cap \bigcup_{x \in R_F^{(i)}} (u,x) = \{(u,v)\}$. This means that overall at least $|R_F^{(i)}| + |R_B^{(i)}| - 1$ node pairs get newly ordered:

$$\Delta\Phi_i \geq |R_F^{(i)}| + |R_B^{(i)}| - 1 = |\delta^{(i)}| - 1.$$

Also, since in this case $\Delta\Phi_i \geq 1$, $|\delta^{(i)}| \leq 2\Delta\Phi_i$. □

**Theorem 19** *For all edge sequences, $\displaystyle\sum_{i=1}^{N} |\delta^{(i)}| \leq n(n-1) = O(n^2)$.*

*Proof.* By Theorem 18 (i), we get $\displaystyle\sum_{i=1}^{N} |\delta^{(i)}| \leq \sum_{i=1}^{N} (\Delta\Phi_i + 1) = \Phi_N + N \leq n(n-1)/2 + n(n-1)/2 = n(n-1)$. □

The remainder of this section provides the necessary tools step by step to finally prove the desired bound on $\sum_{i=1}^{N} \|\delta^{(i)}\|$ in Theorem 25. One can also interpret $\Phi_i$ as a random variable in $DAG(n,m)$ with $m = i$. The corresponding function $\Psi$ for $DAG(n,p)$ is defined as the total number of comparable node pairs in $DAG(n,p)$. Pittel and Tungol [129] showed the following theorem.

**Theorem 20** *For $p := c\log(n)/n$ and $c > 1$, $\mathbf{E}_p[\Psi] = (1+o(1))\frac{n^2}{2}\left(1-\frac{1}{c}\right)^2$.*

Using Theorem 2, this result can be transformed to $\Phi$ as defined above for $DAG(n,m)$ and gives the following bounds for $\mathbf{E}_M[\Phi_k]$.

**Theorem 21** *For $n\log n < k \leq N - 2n\log n$,*

$$\mathbf{E}_M[\Phi_k] = (1+o(1))\frac{n^2}{2}\left(1-\frac{(n-1)\log n}{2(k+n\log n)}\right)^2.$$

*For $N - 2n\log n < k \leq N - 2\log n$,*

$$\mathbf{E}_M[\Phi_k] = (1+o(1))\frac{n^2}{2}\left(1-\frac{(n-1)\log n}{2(k+\sqrt{\log n\,(N-k)})}\right)^2.$$

We skip the rather technical proof of this theorem for the sake of better readability. Readers are referred to [6] for the formal proof of this theorem.

The degree sequence of a random graph is a well-studied problem. The following theorem is shown in [33].

**Theorem 22** *If $pn/\log n \to \infty$, then almost every graph $G$ in the $G(n,p)$ model satisfies $\Delta(G) = (1+o(1))\,pn$, where $\Delta(G)$ is the maximum degree of a node in $G$.*

As noted in Section 2.3, the undirected graph obtained by ignoring the directions of $DAG(n,p)$ is a $G(n,p)$ graph. Therefore, the above result is also true for the maximum degree (in-degree + out-degree) of a node in $DAG(n,p)$. Using Theorem 1, the above result can be transformed to $DAG(n,m)$, as well.

**Theorem 23** *With probability $1-O(\frac{1}{n})$, there is no node with degree higher than $21\frac{m}{n}$ for sufficiently large $n$ and $m > n\log n$ in $DAG(n,m)$.*

The formal and rather technical proof of this theorem can be found in [6]. Here, we only give a high level idea of the proof.

*Rough Sketch.* We examine the following two functions:

- $f_1(g)$: Number of nodes with degree at least $g(n)$
- $f_2(g) := f_1^2(g)$

For $f_1, f_2$ in $G(n,p)$, $g(n) := pn + 2\sqrt{pqn\log n}$, and some constant $c$, Bollobás

[32] showed

$$\begin{aligned}
&\mathbf{E}_p\left[f_1(g)\right] = O\left(\tfrac{1}{n}\right), \\
&\sigma_p^2(f_1(g)) = \mathbf{E}_p\left[f_2(g)\right] - \mathbf{E}_p^2\left[f_1(g)\right] \leq c \cdot \mathbf{E}_p\left[f_1(g)\right].
\end{aligned} \tag{5.2}$$

We transform these mean and variance results to $G(n,m)$ by breaking down the analysis depending on $m$. At first, consider the simpler case of $m > \left(\lfloor \frac{N}{n \log n} \rfloor - 2\right) n \log n$. For sufficiently large $n$, $21 \cdot \frac{m}{n} \geq n-1$ in this case and therefore, no node can have degree higher than it.

Next, we consider $m \in (kn \log n, (k+1)\, n \log n]$ for $1 \leq k < l$, where $l := \lfloor \frac{N}{n \log n} \rfloor - 2$, and we prove the theorem for each interval. Choosing $p_k := (k+2)\frac{n \log n}{N}$, $q_k := 1 - p_k$, and $g_k(n) := p_k n + 2\sqrt{p_k q_k n \log n}$ satisfies the conditions in Theorem 1 and therefore, $\mathbf{E}_M\left[f_i(g_k)\right] = \mathbf{E}_{p_k}\left[f_i(g_k)\right] + o(1)$ for $i = 1,2$ and $1 \leq k < l$. Using Equation (5.2), we get $\mathbf{E}_M\left[f_1(g_k)\right] = O(\mathbf{E}_{p_k}\left[f_1(g_k)\right]) = O\left(\frac{1}{n}\right)$ and

$$\begin{aligned}
\sigma_M^2(f_1(g_k)) &= \mathbf{E}_M\left[f_2(g_k)\right] - \mathbf{E}_M^2\left[f_1(g_k)\right] = O\big(\mathbf{E}_{p_k}\left[f_2(g_k)\right] - \mathbf{E}_{p_k}^2\left[f_1(g_k)\right]\big) \\
&= O(\sigma_{p_k}^2(f_1(g_k))) = O(\mathbf{E}_{p_k}\left[f_1(g_k)\right]) = O\left(\tfrac{1}{n}\right).
\end{aligned}$$

Having transformed the mean and variance of $f_1(g_k)$ to $G(n,m)$ model, we use a variant of Chebyshev's inequality ($\Pr\{|X-\mu| \geq \nu\} \leq \frac{\sigma^2}{\nu^2}$) (cf. Section 2.2) to get

$$\Pr\{|f_1(g_k) - \mu| \geq 1 - \mu\} \leq O\left(\frac{1}{n(1-\mu)^2}\right) = O\left(\tfrac{1}{n}\right).$$

Since $f_1(g_k)$ is a non-negative random variable, $\Pr\{f_1(g_k) \geq 1\} = \Pr\{|f_1(g_k) - \mu| \geq 1 - \mu\} = O\left(\frac{1}{n}\right)$. In other words, with probability $(1 - O(\frac{1}{n}))$, there is no node with a degree higher than $g_k(n) (\leq \frac{21m}{n})$ in any interval.

Since any random $DAG(n,m)$ must have been obtained by taking a random graph $G(n,m)$ and ordering the edges, the degree of a node in $DAG(n,m)$ is the same as the degree of the corresponding node in $G(n,m)$. Therefore, with probability $1 - O(\frac{1}{n})$, there is no node with a degree higher than $21\frac{m}{n}$ in $DAG(n,m)$. □

As the maximum degree of a node in $DAG(n,i)$ is $O(i/n)$, we finally just need to show a bound on $\sum_i (i \cdot |\delta^{(i)}|)$ to prove Theorem 25. This is done in the following theorem.

**Theorem 24** *For DAG$(n,m)$ and $r := N - 2\log n$,*

$$\mathbf{E}\left[\sum_{i=1}^{r}(i\cdot|\delta^{(i)}|)\right] = O(n^3\log^2 n).$$

*Proof.* Let us decompose the analysis in three steps. First, we show a bound on the first $n\log n$ edges. By definition of $\delta^{(i)}$, $|\delta^{(i)}| \le n$. Therefore,

$$\sum_{i=1}^{n\log n} i\cdot\mathbf{E}\left[|\delta^{(i)}|\right] \le \sum_{i=1}^{n\log n} i\cdot n = O\left(n^3\log^2 n\right). \tag{5.3}$$

The second step is to bound $\sum_{i=n\log n}^{t} i\cdot|\delta^{(i)}|$ with $t := N - 2n\log n$. For this, Theorem 18 (ii) shows for all $k$ such that $n\log n < k < t$ that

$$\mathbf{E}\left[\sum_{i=k}^{t}|\delta^{(i)}|\right] \le 2\mathbf{E}\left[\sum_{i=k}^{t}\Delta\Phi_i\right] = 2\mathbf{E}\left[\Phi_t - \Phi_{k-1}\right] = 2\mathbf{E}\left[\Phi_t\right] - 2\mathbf{E}\left[\Phi_{k-1}\right]. \tag{5.4}$$

The function hidden in the $o(1)$ in Theorem 20 is decreasing in $p$ [129]. Hence, also the $o(1)$ in Theorem 21 is decreasing in $k$. Plugging this in Equation (5.4) yields (with $s := n\log n$)

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=k}^{t}|\delta^{(i)}|\right] &\le (1+o(1))\,n^2\left(\left(1-\frac{(n-1)\log n}{2(t+s)}\right)^2 - \left(1-\frac{(n-1)\log n}{2(k-1+s)}\right)^2\right)\\
&= (1+o(1))\,n^2(n-1)\log n\left(\frac{2}{2(k-1+s)} - \frac{2}{2(t+s)} + \right.\\
&\qquad\qquad \left.\frac{(n-1)\log n}{4}\left(\frac{1}{(t+s)^2} - \frac{1}{(k-1+s)^2}\right)\right)\\
&\le (1+o(1))\,n^2(n-1)\log n\left(\frac{1}{k-1+s} - \frac{1}{t+s}\right)\\
&\le (1+o(1))\,n^2(n-1)\log n\frac{1}{k-1}.
\end{aligned} \tag{5.5}$$

By linearity of expectation and Equation (5.5),

$$\begin{aligned}
\mathbf{E}\left[\sum_{i=s+1}^{t} i|\delta^{(i)}|\right] &= \sum_{i=s+1}^{t}\left(i\mathbf{E}\left[|\delta^{(i)}|\right]\right) \leq \sum_{j=1}^{\log(\lceil\frac{t}{s}\rceil)}\left(2^j s \sum_{i=2^{(j-1)}s+1}^{2^j s} \mathbf{E}\left[|\delta^{(i)}|\right]\right) \\
&\leq \sum_{j=1}^{\log(\lceil\frac{t}{s}\rceil)}\left(2^j s \sum_{i=2^{(j-1)}s+1}^{t} \mathbf{E}\left[|\delta^{(i)}|\right]\right) \\
&\leq \sum_{j=1}^{\log(\lceil\frac{t}{s}\rceil)}\left(2^j s(1+o(1))n^2(n-1)\log n\frac{1}{2^{(j-1)}s}\right) \\
&= \sum_{j=1}^{\log(\lceil\frac{t}{s}\rceil)}\left(2(1+o(1))n^2(n-1)\log n\right) \\
&= 2(1+o(1))n^2(n-1)\log^2 n = O(n^3\log^2 n).
\end{aligned}$$

For the last step consider a $k$ such that $t < k < r$. Theorem 18 (ii) gives

$$\mathbf{E}\left[\sum_{i=k}^{r}|\delta^{(i)}|\right] \leq 2\mathbf{E}\left[\sum_{i=k}^{r}\Delta\Phi_i\right] = 2\mathbf{E}\left[\Phi_r - \Phi_{k-1}\right] = 2\mathbf{E}\left[\Phi_r\right] - 2\mathbf{E}\left[\Phi_{k-1}\right].$$

Using Theorem 21 and similar arguments as before, this yields (with $s(k) := \sqrt{\log n\,(N-k)}$)

$$\begin{aligned}
&\mathbf{E}\left[\sum_{i=k}^{r}|\delta^{(i)}|\right] \\
&\quad\leq (1+o(1))n^2\left(\left(1-\frac{(n-1)\log n}{2(r+s(r))}\right)^2 - \left(1-\frac{(n-1)\log n}{2(k-1+s(k-1))}\right)^2\right) \\
&\quad= (1+o(1))n^2(n-1)\log n\left(\frac{2}{2(k-1+s(k-1))} - \frac{2}{2(r+s(r))} + \right. \\
&\qquad\qquad \left.\frac{(n-1)\log n}{4}\left(\frac{1}{(r+s(r))^2} - \frac{1}{(k-1+s(k-1))^2}\right)\right).
\end{aligned}$$

Since $k+s(k)$ is monotonically increasing for $t < k < r$, $\frac{1}{(k+s(k))^2}$ is a monotonically decreasing function in this interval. Therefore, $\frac{1}{(r+s(r))^2} - \frac{1}{(k-1+s(k-1))^2} < 0$,

which proves the following equation.

$$\begin{aligned}\mathbf{E}\left[\sum_{i=k}^{r}|\delta^{(i)}|\right] &\le (1+o(1))\,n^2(n-1)\log n\left(\frac{1}{k-1+s(k-1)}-\frac{1}{r+s(r)}\right)\\ &\le (1+o(1))\,n^2(n-1)\log n\frac{1}{k-1}. \qquad (5.6)\end{aligned}$$

By linearity of expectation and Equation (5.6),

$$\begin{aligned}&\mathbf{E}\left[\sum_{i=N-2n\log n+1}^{r} i\,|\delta^{(i)}|\right]\\ &\quad= \sum_{i=N-2n\log n+1}^{r}\left(i\,\mathbf{E}\left[|\delta^{(i)}|\right]\right)\\ &\quad\le (N-2\log n)\sum_{i=N-2n\log n+1}^{r}\mathbf{E}\left[|\delta^{(i)}|\right]\\ &\quad\le (N-2\log n)\,(1+o(1))\,n^2(n-1)\log n\frac{1}{N-2n\log n-1}\\ &\quad= O(n^3\log n).\end{aligned}$$

□

**Theorem 25** *For DAG$(n,m)$,* $\mathbf{E}\left[\sum_{i=1}^{N}\|\delta^{(i)}\|\right]=O(n^2\log^2 n)$.

*Proof.* By definition of $\|\delta^{(i)}\|$, we know $\|\delta^{(i)}\|\le i$ and hence

$$\sum_{i=1}^{n\log n}\|\delta^{(i)}\|=O(n^2\log^2 n).$$

Again, let $r:=N-2\log n$. Theorem 23 tells us that with probability greater than $\left(1-\frac{c'}{n}\right)$ for some constant $c'$, there is no node with degree $\ge \frac{c\,i}{n}$ (for $c=21$). Since the degree of an arbitrary node in a DAG is bounded by $n$, we get with Theorems 19 and 24,

$$\begin{aligned}\mathbf{E}\left[\sum_{i=n\log n+1}^{r}\|\delta^{(i)}\|\right] &= O\left(\mathbf{E}\left[\sum_{i=n\log n+1}^{r}\frac{c\,i\,|\delta^{(i)}|}{n}\right]+\mathbf{E}\left[\sum_{i=n\log n+1}^{r}\frac{n\,c'\,|\delta^{(i)}|}{n}\right]\right)\\ &= O\left(\frac{1}{n}\mathbf{E}\left[\sum_{i=1}^{r}(i\,|\delta^{(i)}|)\right]+n^2\right)\\ &= O\left(\frac{1}{n}\left(n^3\log^2 n\right)+n^2\right)=O(n^2\log^2 n).\end{aligned}$$

By again using the fact that the degree of an arbitrary node in a DAG is at most $n$, we obtain

$$\mathbf{E}\left[\sum_{i=r+1}^{N} \|\delta^{(i)}\|\right] = O\Big(n \cdot \mathbf{E}\left[\sum_{i=r+1}^{N} |\delta^{(i)}|\right]\Big) = O\Big(n \cdot \sum_{i=r+1}^{N} n\Big) = O(n^2 \log n).$$

Thus,

$$\begin{aligned}\mathbf{E}\left[\sum_{i=1}^{N} \|\delta^{(i)}\|\right] &= \mathbf{E}\left[\sum_{i=1}^{n\log n} \|\delta^{(i)}\|\right] + \mathbf{E}\left[\sum_{i=n\ \log n+1}^{r} \|\delta^{(i)}\|\right] + \mathbf{E}\left[\sum_{i=r+1}^{N} \|\delta^{(i)}\|\right] \\ &= O(n^2 \log^2 n) + O(n^2 \log^2 n) + O(n^2 \log n) = O(n^2\ \log^2 n). \quad \square\end{aligned}$$

### 5.9.2 Other average-case results

Recall from Section 5.1 that for an invalidated topological order $T$, a set $K \subseteq V$ is a *cover* if for all $x, y \in V$: $(x \rightsquigarrow y \ \wedge\ y < x \ \Rightarrow\ x \in K \vee y \in K)$. In order to prove that the expected complexity of AHRSZ on REIS is $O(n^2 \log^3 n)$, observe that $\delta^{(i)}$ is a valid cover. Therefore, by definition of $|\rangle \hat{K}^{(i)} \langle|$ as minimal cover, it follows that $|\rangle \hat{K}^{(i)} \langle| \leq |\rangle \delta^{(i)} \langle| = |\delta^{(i)}| + \|\delta^{(i)}\|$. Note that the complexity of maintaining the topological ordering with AHRSZ while inserting an edge is $O(|\rangle \hat{K} \langle| \log |\rangle \hat{K} \langle|)$ (cf. Section 5.1. The expected complexity of AHRSZ on REIS is thus $\mathbf{E}\left[\sum_{i=1}^{m} |\rangle \hat{K}^{(i)} \langle| \log |\rangle \hat{K} \langle|\right]$. Using Theorems 19 and 25 we get,

$$\mathbf{E}\left[\sum_{i=1}^{m} |\rangle \hat{K}^{(i)} \langle| \log |\rangle \hat{K} \langle|\right] \leq \log n \cdot \sum_{i=1}^{m} |\delta^{(i)}| + \mathbf{E}\left[\sum_{i=1}^{m} \|\delta^{(i)}\|\right] = O(n^2 \log^3 n)$$

KB also computes a cover $K \subseteq \delta^{(i)}$ and its complexity per edge insertion is $O(|\rangle K \langle| \log |\rangle K \langle|)$. Therefore, $|\rangle K \langle| \leq |\delta^{(i)}| + \|\delta^{(i)}\|$ and with a similar argument as above, the expected complexity of KB on REIS is $O(n^2 \log^3 n)$.

An interesting question in all this analysis is how many edges will actually invalidate the topological ordering and force any algorithm to do something about them. Here, we show a non-trivial upper bound on the expected value of the number of invalidating edges on REIS. Consider the following random variable: $\textsc{inval}(i) = 1$ if the $i$-th edge inserted is an invalidating edge; $\textsc{inval}(i) = 0$ otherwise.

**Theorem 26** $\mathbf{E}\left[\sum_{i=1}^{m} \textsc{INVAL}(i)\right] = O(\min\{m, n^{\frac{3}{2}} \log^{\frac{1}{2}} n\})$.

*Proof.* If the $i$-th edge is invalidating, $|\delta^{(i)}| \geq 2$; otherwise $\textsc{INVAL}(i) = |\delta^{(i)}| = 0$. In either case, $\textsc{INVAL}(i) \leq |\delta^{(i)}|/2$. Thus, for $s := n^{\frac{3}{2}} \log^{\frac{1}{2}} n$ and $t := \min\{m, N - 2n \log n\}$,

$$\begin{aligned}\mathbf{E}\left[\sum_{i=s+1}^{t} \textsc{INVAL}(i)\right] &\leq \mathbf{E}\left[\sum_{i=s+1}^{t} \frac{|\delta^{(i)}|}{2}\right] \leq (1+o(1)) \frac{n^2(n-1)\log n}{2s} \\ &\leq \frac{(1+o(1))}{2} n^{\frac{3}{2}} \log^{\frac{1}{2}} n.\end{aligned}$$

The second inequality follows by substituting $k := s+1$ in Equation (5.5). Also, since the number of invalidating edges can be at most equal to the total number of edges, $\sum_{i=1}^{s} \textsc{INVAL}(i) \leq s$.

$$\begin{aligned}\mathbf{E}\left[\sum_{i=1}^{m} \textsc{INVAL}(i)\right] &= \mathbf{E}\left[\sum_{i=1}^{s} \textsc{INVAL}(i)\right] + \mathbf{E}\left[\sum_{i=s+1}^{t} \textsc{INVAL}(i)\right] + \mathbf{E}\left[\sum_{i=t}^{m} \textsc{INVAL}(i)\right] \\ &\leq O(s) + O(n^{\frac{3}{2}} \log^{\frac{1}{2}} n) + O(n \log n) = O(n^{\frac{3}{2}} \log^{\frac{1}{2}} n).\end{aligned}$$

The second bound $\mathbf{E}[\sum_{i=1}^{m} \textsc{INVAL}(i)] \leq m$ is obvious by definition of $\textsc{INVAL}(i)$. □

## 5.10 Recent advances in online topological ordering algorithms

Recently, Haeupler et al. [78] gave two new algorithms for online topological ordering. Their algorithm for the sparse case requires $O(m^{3/2})$ time while their algorithm for the dense case requires $O(n^{2.5})$ time, independent of the number of edges inserted. Their algorithm for the dense case crucially relies on our Lemma 11. Independently, Liu and Chao [97] gave an algorithm with $\tilde{O}(n^{2.5})$ bound. Their algorithm is largely based on our algorithm, but uses buckets of exponentially increasing sizes. Very recently, Bender et al. [27] gave an $O(n^2 \log^2 n)$ algorithm for this problem. It can be further improved to $O(n^2 \log n)$ [69].

Regarding lower bounds, Ramalingam and Reps [133] show that an adversary can force any algorithm maintaining explicit labels to need $\Omega(n \log n)$ time complexity

for inserting $n-1$ edges. Katriel [90] gave a class of examples on which any local algorithm that maintains the topological order as an explicit mapping $T: V \rightarrow [1..n]$ must do $\Omega(n^2)$ node relabellings for inserting $n$ edges. Heupler et al. [79] show a class of examples on which any local algorithm must do $\Omega(nm^{1/2})$ node relabellings for inserting $n$ edges.

## 5.11 Conclusion

In this chapter, we considered the problem of dynamic topological ordering. We have presented the first $o(n^3)$ algorithm for incremental topological ordering. The analysis of our algorithm is however, not tight. A non-trivial lower bound of $O(n^2 \log n)$ for our algorithm can be infered from Theorem 14. However, it is still quite far from the upper bound of $O(n^{2.75})$ for this algorithm. We show some ideas that can potentially lead to tightening the analysis of this algorithm. A better analysis of this algorithm still remains an open problem.

There is still a large gap between the current best lower bounds (cf. Section 5.10) and the upper bound of $O(\min\{m^{1.5}, n^2 \log n\})$. Bridging this gap remains an open problem.

As mentioned at the beginning of this chapter, nothing better is known for online cycle detection so far than to maintain topological ordering in an incremental setting. It is not clear if a faster online cycle detection algorithm can be developed.

The externalization of our algorithm provides interesting new results for dynamic topological ordering in external memory. It would be interesting to see if the faster incremental topological ordering algorithms developed recently also lead to improved external memory results.

We also presented the first average-case analysis of online topological ordering algorithms. We proved an expected runtime of $O(n^2 \operatorname{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for AHRSZ, KB and PK. An interesting question here is whether one can obtain better bounds for the case when there are $m = o(n^2)$ edges inserted into a previously empty DAG or into an arbitrary DAG.

It will also be interesting to find out whether the average-case results can be extended to the fully dynamic case. Note that in the worst case scenario, it is not possible to obtain any interesting results for this case as any algorithm that explicitly maintains the node labellings can be made to do $\Omega(n)$ work for a pair of

insertion and deletion. This can be seen, for example, by maintaining a list DAG, deleting the edge in the middle and inserting a new edge connecting the previous end-point of the list to the previous starting point of the list. However, when the sequence of insertions and deletions is random, such worst case scenarios will happen with low probability and it might be possible to prove some interesting bounds.

For the analysis of these algorithms to make more sense for real applications, we may consider changing our notion of change. Typically, we do not have edges coming one at a time. Rather a few edges get inserted or deleted and we want to use the old topological ordering to compute the new one efficiently. Pearce [123] proposed a modification of online topological ordering, in which a batch of edges are inserted at a time.

# Bibliography

[1] J. Abello, P. Pardalos, and M. Resende. On maximum clique problems in very large graphs. *External Memory Algorithms, AMS-DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, 50:119–130, 1999.

[2] J. Abello, A. Buchsbaum, and J. Westbrook. A functional approach to external graph algorithms. *Algorithmica*, 32:437–458, 2002.

[3] A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31:1116–1127, 1988.

[4] G. Aggarwal, M. Datar, S. Rajagopalan, and M. Ruhl. On the streaming model augmented with a sorting primitive. In *Proceedings of the forty-fifth annual IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 540–549, 2004.

[5] D. Ajwani and T. Friedrich. Average-case analysis of online topological ordering. In *Proceedings of the eighteenth International Symposium on Algorithms and Computation (ISAAC)*, Vol. 4835 of *Lecture Notes in Computer Science (LNCS)*, pp. 464–475. Springer, 2007.

[6] D. Ajwani and T. Friedrich. Average-case analysis of online topological ordering, 2008. `arXiv:0802.1059`.

[7] D. Ajwani, R. Dementiev, and U. Meyer. A computational study of external memory BFS algorithms. In *Proceedings of the seventeenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 601–610, 2006.

[8] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. In *Proceedings of the tenth Scandinavian Workshop on Algorithm Theory (SWAT)*, Vol. 4059 of *Lecture Notes in Computer Science (LNCS)*, pp. 53–64. Springer, 2006.

[9] D. Ajwani, U. Meyer, and V. Osipov. Improved external memory BFS implementation. In *Proceedings of the ninth workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 3–12, 2007.

[10] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design. Technical Report MPI-I-2008-1-001, Max Planck Institut für Informatik, 2008.

[11] D. Ajwani, I. Malinger, U. Meyer, and S. Toledo. Characterizing the performance of flash memory storage devices and its impact on algorithm design.

In *Proceedings of the seventh international Workshop on Experimental Algorithms (WEA)*, pp. 208–219, 2008.

[12] D. Ajwani, T. Friedrich, and U. Meyer. An $O(n^{2.75})$ algorithm for online topological ordering. *ACM Transactions on Algorithms*, 2009. A preliminary version of this paper appeared as [8].

[13] D. Ajwani, U. Meyer, and V. Osipov. Breadth first search on massive graphs. *DIMACS series book devoted to the ninth implementation challenge on shortest paths*, 2009. A preliminary version of this paper appeared as [9].

[14] L. Allulli, P. Lichodzijewski, and N. Zeh. A faster cache-oblivious shortest-path algorithm for undirected graphs with bounded edge lengths. In *Proceedings of the eighteenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 910–919, 2007.

[15] B. Alpern, R. Hoover, B. K. Rosen, P. F. Sweeney, and F. K. Zadeck. Incremental evaluation of computational circuits. In *Proceedings of the first annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 32–42, 1990.

[16] R. Angelova and G. Weikum. Graph-based text classification: Learn from your neighbors. In *Proceedings of the twenty-ninth annual international ACM SIGIR conference on research and development in Information Retrieval*, pp. 485–492, 2006.

[17] L. Arge. The Buffer Tree: A new technique for optimal I/O-algorithms. In *Proceedings of the fourth International workshop on Algorithms and Data Structures (WADS)*, Vol. 955 of *Lecture Notes in Computer Science (LNCS)*, pp. 334–345. Springer, 1995.

[18] L. Arge, G. S. Brodal, and L. Toma. On external-memory MST, SSSP and multi-way planar graph separation. In *Proceedings of the eighth Scandinavian Workshop on Algorithmic Theory (SWAT)*, Vol. 1851 of *Lecture Notes in Computer Science (LNCS)*, pp. 433–447. Springer, 2000.

[19] L. Arge, L. Toma, and J. S. Vitter. I/O-efficient algorithms for problems on grid-based terrains. In *Proceedings of the second Workshop on Algorithm Engeneering and Experiments (ALENEX)*, pp. 217–236, 2000.

[20] L. Arge, M. A. Bender, E. D. Demaine, B. Holland-Minkley, and J. I. Munro. Cache-oblivious priority queue and graph algorithm applications. In *Proceedings of the thirty-fourth annual ACM Symposium on Theory of Computing (STOC)*, pp. 268–276, 2002.

[21] L. Arge, O. Procopiuc, and J. Vitter. Implementing I/O-efficient data structures using TPIE. In *Proceedings of the tenth annual European Symposium on Algorithms (ESA)*, Vol. 2461 of *Lecture Notes in Computer Science (LNCS)*, pp. 88–100. Springer, 2002.

[22] L. Arge, J. S. Chase, P. Halpin, L. Toma, J. S. Vitter, D. Urban, and R. Wickremesinghe. Efficient flow computation on massive grid terrain datasets. *Geoinformatica*, 7:283–313, 2003.

[23] L. Arge, M. T. Goodrich, M. Nelson, and N. Sitchinava. Fundamental parallel algorithms for private-cache chip multiprocessors. In *Proceedings of the twentieth annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 197–206, 2008.

[24] G. Ausiello, G. F. Italiano, A. Marchetti-Spaccamela, and U. Nanni. Incremental algorithms for minimal length paths. *Journal of Algorithms*, 12: 615–638, 1991.

[25] B. Babcock, S. Babu, M. Datar, R. Motwani, and R. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles Of Database Systems (PODS)*, pp. 1–16, 2002.

[26] A. B. Barak and P. Erdős. On the maximal number of strongly independent vertices in a random acyclic directed graph. *SIAM Journal on Algebraic and Discrete Methods*, 5:508–514, 1984.

[27] M. Bender, J. Fineman, and S. Gilbert. Online topological ordering. In *Proceedings of the twentieth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, 2009 (to appear).

[28] G. Bhalotia, A. Hulgeri, C. Nakhe, S. Chakrabarti, and S. Sudarshan. Keyword searching and browsing in databases using BANKS. In *Proceedings of the eighteenth International Conference on Data Engineering (ICDE)*, pp. 431–440, 2002.

[29] A. Birrell, M. Isard, C. Thacker, and T. Wobber. A design for high-performance flash disks. *ACM SIGOPS Operating Systems Review*, 41: 88–93, 2007.

[30] D. K. Blandford, G. E. Blelloch, and I. A. Kash. Compact representations of separable graphs. In *Proceedings of the fourteenth ACM-SIAM annual Symposium On Discrete Algorithms (SODA)*, pp. 679–688, 2003.

[31] D. K. Blandford, G. E. Blelloch, and I. A. Kash. An experimental analysis

of a compact graph representation. In *Proceedings of the sixth Workshop on Algorithm engineering and experiments (ALENEX)*, 2004.

[32] B. Bollobás. Degree sequences of random graphs. *Discrete Maths*, 33: 1–19, 1981.

[33] B. Bollobás. *Random Graphs*. Cambridge, 2001. ISBN 0-521-79722-5.

[34] O. Boruvka. O jistém problému minimálním [About a certain minimal problem]. *Práce, Moravské Prirodovedecké Spolecnosti*, 3:37–58, 1926.

[35] U. Brandes and T. Erlebach (Eds.). *Network Analysis*, Vol. 3418 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2005. ISBN 978-3-540-24979-5.

[36] S. Brin and L. Page. The anatomy of a large-scale hypertextual web search engine. *Computer Networks and ISDN Systems*, 30:107–117, 1998.

[37] G. S. Brodal. Personal communication between Gerth Brodal and Ulrich Meyer.

[38] G. S. Brodal and R. Fagerberg. Funnel heap – a cache oblivious priority queue. In *Proceedings of the thirteenth International Symposium on Algorithms and Computation, (ISAAC)*, pp. 219–228, 2002.

[39] G. S. Brodal and R. Fagerberg. Cache oblivious distribution sweeping. In *Proceedings of twenty-ninth International Colloquium Automata, Languages and Programming (ICALP)*, Vol. 2380 of *Lecture Notes in Computer Science (LNCS)*, pp. 426–438, 2002.

[40] G. S. Brodal, R. Fagerberg, U. Meyer, and N. Zeh. Cache-oblivious data structures and algorithms for undirected breadth-first search and shortest paths. In *Proceedings of the ninth Scandinavian Workshop on Algorithm Theory (SWAT)*, Vol. 3111 of *Lecture Notes in Computer Science (LNCS)*, pp. 480–492. Springer, 2004.

[41] G. S. Brodal, R. Fagerberg, and K. Vinther. Engineering a cache-oblivious sorting algorithm. In *Proceedings of the sixth workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 4–17, 2004.

[42] I. I. Brudaru. Heuristics for average diameter approximation with external memory algorithms. Master's thesis, Max Planck Institut für Informatik, Saarbrücken, Germany, 2007.

[43] A. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. Westbrook. On external memory graph traversal. In *Proceedings of the eleventh an-*

*nual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 859–860. ACM-SIAM, 2000.

[44] T. M. Chan. All-pairs shortest paths with real weights in $O(n^3/\log n)$ time. *Algorithmica*, 50:236–243, 2008.

[45] B. Chazelle. A minimum spanning tree algorithm with inverse-ackermann type complexity. *Journal of the ACM*, 47:1028–1047, 2000.

[46] M. Chen, R. A. Chowdhury, V. Ramachandran, D. L. Roche, and L. Tong. Priority queues and Dijkstra's algorithm. Technical Report TR-07-54, The University of Texas at Austin, Department of Computer Sciences, 2007.

[47] P. M. Chen and D. A. Patterson. A new approach to I/O performance evaluation—self-scaling I/O benchmarks, predicted I/O performance. In *Proceedings of the ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, pp. 1–12, 1993.

[48] Y. J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamasia, D. E. Vengroff, and J. S. Vitter. External memory graph algorithms. In *Proceedings of the sixth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 139–149. ACM-SIAM, 1995.

[49] F. J. Christiani. Cache-oblivious graph algorithms. Master's thesis, University of Southern Denmark, 2005.

[50] S. Cicerone, D. Frigioni, U. Nanni, and F. Pugliese. A uniform approach to semi-dynamic problems on digraphs. *Theoretical Computer Science*, 203: 69–90, 1998.

[51] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, Cambridge, MA, 1989. ISBN 0-262-03141-8.

[52] DBpedia Blog, 2008. `http://blog.dbpedia.org/2008/08/18/dbpedia-31-breaks-100-million-triples-barrier/`.

[53] M. de Kunder. Geschatte grootte van het geïndexeerde world wide web. Master's thesis, Universiteit van Tilburg, 2006. http://www.worldwidewebsize.com/.

[54] R. Dementiev, P. Sanders, D. Schultes, and J. Sibeyn. Engineering an external memory minimum spanning tree algorithm. In *Proceedings of the third International Conference on Theoretical Computer Science (TCS)*, pp. 195–208. Kluwer, 2004.

[55] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template

Library for XXL Data Sets. Technical Report 18, Fakultät für Informatik, University of Karlsruhe, 2005.

[56] R. Dementiev, P. Sanders, and L. Kettner. STXXL: Standard Template library for XXL data sets. In *Proceedings of the thirteenth annual European Symposium on Algorithms (ESA)*, Vol. 3669 of *Lecture Notes in Computer Science (LNCS)*, pp. 640–651, 2005.

[57] R. Dementiev, L. Kettner, and P. Sanders. STXXL: Standard Template library for XXL data sets. *Software: Practice and Experience*, 38:589–637, 2008.

[58] C. Demetrescu, I. Finocchi, and A. Ribichini. Trading off space for passes in graph streaming problems. In *Proceedings of the seventeenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, 2006.

[59] P. K. Desikan, N. Pathak, J. Srivastava, and V. Kumar. Divide and conquer approach for efficient pagerank computation. In *Proceedings of the sixth International Conference on Web Engineering (ICWE)*, pp. 233–240, 2006.

[60] P. F. Dietz and D. D. Sleator. Two algorithms for maintaining order in a list. In *Proceedings of the nineteenth annual ACM Symposium on Theory of Computing (STOC)*, pp. 365–372, 1987.

[61] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerische Mathematik*, 1:269–271, 1959.

[62] DIMACS Implementation Challenge – Shortest Paths. `http://www.dis.uniroma1.it/~challenge9/download.shtml`.

[63] S. Edelkamp and S. Jabbar. Large-scale directed model checking LTL. In *Proceedings of the thirteenth international SPIN workshop on model checking software (SPIN)*, Vol. 3925 of *Lecture Notes in Computer Science (LNCS)*, pp. 1–18, 2006.

[64] S. Edelkamp, S. Jabbar, and S. Schrödl. External A*. In *Proceedings of the twenty-seventh German conference on Artificial Intelligence (KI)*, Vol. 3238 of *Lecture Notes in Artificial Intelligence (LNAI)*, pp. 226–240. Springer, 2004.

[65] D. Eppstein, Z. Galil, and G. F. Italiano. Dynamic graph algorithms. In M. J. Atallah, editor, *Algorithms and Theory of Computation Handbook*, chapter 8. CRC Press, 1999.

[66] P. Erdős and A. Rényi. On random graphs. *Publicationes Mathematicae Debrecen*, 6:290–297, 1959.

[67] P. Erdős and A. Rényi. On the evolution of random graphs. *Matematikai Kutato Intezetenek Kozlemenyei. Magyar Tudomanyos Akademia*, 5:17–61, 1960.

[68] J. Feigenbaum, S. Kannan, A. Mcgregor, and J. Zhang. On graph problems in a semi-streaming model. In *Proceedings of the 31st International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 531–543. Springer-Verlag, 2004.

[69] J. Fineman, 2008. Personal communication.

[70] Flickr Blog. `http://blog.flickr.net/en/2007/11/13/holy-moly/`.

[71] M. L. Fredman and R. E. Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM*, 34:596–615, 1987.

[72] D. Frigioni, A. Marchetti-Spaccamela, and U. Nanni. Fully dynamic shortest paths and negative cycles detection on digraphs with arbitrary arc weights. In *Proceedings of the European Symposium on Algorithms (ESA)*, Vol. 1461 of *Lecture Notes in Computer Science (LNCS)*, pp. 320–331, 1998.

[73] M. Frigo, C. E. Leiserson, H. Prokop, and S. Ramachandran. Cache-oblivious algorithms. In *Proceedings of the fortieth annual Symposium on Foundations of Computer Science (FOCS)*, pp. 285–297. IEEE Computer Society Press, 1999.

[74] E. Gal and S. Toledo. Algorithms and data structures for flash memories. *ACM Computing Surveys*, 37:138–163, 2005.

[75] B. A. Galler and M. J. Fisher. An improved equivalence algorithm. *Communications of the ACM*, 7:301–303, 1964.

[76] A. Goldberg and R. Werneck. Computing point-to-point shortest paths from external memory. In *Proceedings of the seventh workshop on Algorithm Engineering and Experiments (ALENEX)*, pp. 26–40, 2005.

[77] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proceedings of the nineteenth annual Symposium on Foundations of Computer Science (FOCS)*, pp. 8–21. IEEE, 1978.

[78] B. Haeupler, T. Kavitha, R. Mathew, S. Sen, and R. E. Tarjan. Faster algorithms for incremental topological ordering. In *Proceedings of the thirty-fifth International Colloquium on Automata, Languages and Programming (ICALP)*, pp. 421–433, 2008.

[79] B. Haeupler, S. Sen, and R. E. Tarjan. Incremental topological ordering and strong component maintenance, 2008. `arXiv:0803.0792v1`.

[80] Hard Drive Chart, 2007. `http://www23.tomshardware.com/storage.html?modelx=33&model1=117&model2=676&chart=33`.

[81] P. E. Hart, N. J. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics SSC*, 4:100–107, 1968.

[82] M. Henzinger, P. Raghavan, and S. Rajagopalan. Computing on data streams. In *"External Memory algorithms", DIMACS series in Discrete Mathematics and Theoretical Computer Science*, Vol. 50, pp. 107–118, 1999.

[83] In-Stat press release, 2006. `http://www.instat.com/press.asp?ID=1706&sku=IN0603343SI`.

[84] S. Jabbar and S. Edelkamp. I/O efficient directed model checking. In *Proceedings of the sixth international conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, Vol. 3385 of *Lecture Notes in Computer Science (LNCS)*, pp. 313–329, 2005.

[85] S. Jabbar and S. Edelkamp. Parallel external directed model checking with linear I/O. In *Proceedings of the seventh international conference on Verification, Model Checking and Abstract Interpretation (VMCAI)*, Vol. 3855 of *Lecture Notes in Computer Science (LNCS)*, pp. 237–251, 2006.

[86] V. Jarník. O jistém problému minimálním [About a certain minimal problem]. *Práce Moravské Přírodovědecké Společnosti*, 6:57–63, 1930.

[87] K. Kaligosi and P. Sanders. How branch mispredictions affect quicksort. In *Proceedings of the fourteenth annual European Symposium on Algorithms (ESA)*, pp. 780–791, 2006.

[88] D. Karger, P. Klein, and R. Tarjan. A randomized linear time algorithm to find minimum spanning trees. *Journal of the ACM*, 42:321–328, 1995.

[89] G. Kasneci, M. Ramanath, M. Sozio, F. M. Suchanek, and G. Weikum. STAR: Steiner Tree Approximation in Relationship-graphs. In *Proceedings of the twenty-fifth IEEE International Conference on Data Engineering (ICDE)*, 2009 (to appear).

[90] I. Katriel. On algorithms for online topological ordering and sorting. Technical Report MPI-I-2004-1-003, Max Planck Institut für Informatik, Saarbrücken, Germany, 2004.

[91] I. Katriel and H. L. Bodlaender. Online topological ordering. *ACM Transactions on Algorithms*, 2:364–379, 2006. Announced at SODA '05.

[92] J. B. Kruskal. On the shortest spanning subtree of a graph and the traveling salesman problem. *Proceedings of the American Mathematical Society*, 7: 48–50, 1956.

[93] V. Kumar and E. J. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proceedings of the eighth IEEE Symposium on Parallel and Distributed Processing (IPDPS)*, pp. 169–177, 1996.

[94] A. LaMarca and R. E. Ladner. The influence of caching on the performance of sorting. *Journal of Algorithms*, 31:66–104, 1999.

[95] L. Laura, S. Leonardi, S. Millozzi, U. Meyer, and J. F. Sibeyn. Algorithms and experiments for the webgraph. In *Proceedings of the eleventh annual European Symposium on Algorithms (ESA)*, pp. 703–714, 2003.

[96] S.-W. Lee and B. Moon. Design of flash-based DBMS: an in-page logging approach. In *SIGMOD International Conference on Management of Data*, pp. 55–66. ACM, 2007.

[97] H. F. Liu and K. M. Chao. An $\tilde{O}(n^{2.5})$-time algorithm for online topological ordering, 2008. `arXiv:0804.3860v2`.

[98] J. Luxenburger and G. Weikum. Exploiting community behavior for enhanced link analysis and web search. In *Proceedings of the ninth International Workshop on the Web and Databases (WebDB)*, pp. 14–19, 2006.

[99] K. Macherey, F. J. Och, and H. Ney. Natural language understanding using statistical machine translation. In *Proceedings of the seventh European Conference on Speech Communication and Technology (EUROSPEECH)*, pp. 2205–2208, 2001.

[100] A. Maheshwari and N. Zeh. External memory algorithms for outerplanar graphs. In *Proceedings of the tenth International Symposium on Algorithms and Computations (ISAAC)*, Vol. 1741 of *Lecture Notes in Computer Science (LNCS)*, pp. 307–316. Springer, 1999.

[101] A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proceedings of the thirteenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 372–381. ACM-SIAM, 2002.

[102] A. Maheshwari and N. Zeh. I/O-efficient algorithms for graphs of bounded

treewidth. In *Proceedings of the twelfth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 89–90. ACM-SIAM, 2001.

[103] Y. Maon, B. Scheiber, and U. Vishkin. Parallel ear decomposition search (EDS) and st-numbering in graphs. In *Theoretical Computer Science*, Vol. 47, pp. 277–298, 1986.

[104] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. On-line graph algorithms for incremental compilation. In *Proceedings of nineteenth International Workshop on Graph-Theoretic Concepts in Computer Science (WG)*, Vol. 790 of *Lecture Notes in Computer Science (LNCS)*, pp. 70–86, 1993.

[105] A. Marchetti-Spaccamela, U. Nanni, and H. Rohnert. Maintaining a topological order under edge insertions. *Information Processing Letters*, 59: 53–58, 1996.

[106] K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proceedings of the tenth annual European Symposium on Algorithms (ESA)*, Vol. 2461 of *Lecture Notes in Computer Science (LNCS)*, pp. 723–735. Springer, 2002.

[107] K. Mehlhorn and S. Naher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999. ISBN 0521563291.

[108] R. V. Meter. Observing the effects of multi-zone disks. In *USENIX Annual Technical Conference*, pp. 19–30, 1997.

[109] U. Meyer. On trade-offs in external-memory diameter-approximation. In *Proceedings of the eleventh Scandinavian Workshop on Algorithm Theory (SWAT)*, pp. 426–436, 2008.

[110] U. Meyer. On dynamic Breadth-First Search in external-memory. In *Proceedings of the twenty-fifth annual Symposium on Theoretical Aspects of Computer Science (STACS)*, pp. 551–560, 2008.

[111] U. Meyer and V. Osipov. Design and implementation of a practical I/O-efficient shortest paths algorithm, 2008.

[112] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proceedings of the eleventh annual European Symposium on Algorithms (ESA)*, Vol. 2832 of *Lecture notes in Computer Science (LNCS)*, pp. 434–445. Springer, 2003.

[113] U. Meyer and N. Zeh. I/O-efficient undirected shortest paths with unbounded edge lengths. In *Proceedings of the fourteenth annual European*

*Symposium on Algorithms (ESA)*, Vol. 4168 of *Lecture Notes in Computer Science (LNCS)*, pp. 540–551. Springer, 2006.

[114] U. Meyer, P. Sanders, and J. Sibeyn (Eds.). *Algorithms for Memory Hierarchies*, Vol. 2625 of *Lecture Notes in Computer Science (LNCS)*. Springer, 2003. ISBN 3-540-00883-7.

[115] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proceedings of the tenth annual ACM-SIAM Symposium On Discrete Algorithms (SODA)*, pp. 687–694. ACM-SIAM, 1999.

[116] S. Muthukrishnan. *Data Streams: Algorithms and Applications*, Vol. 1 (2) of *Foundations and Trends in Theoretical Computer Science*. NOW, 2005.

[117] D. Myers. On the use of NAND flash memory in high-performance relational databases. Master's thesis, Massachussets Institute of Technology, 2008.

[118] M. Najork and J. Wiener. Breadth-first search crawling yields high-quality pages. In *Proceedings of the tenth International World Wide Web Conference*, pp. 114–118, 2001.

[119] M. E. J. Newman and M. Girvan. Finding and evaluating community structure in networks. *Physical Review E*, 69:026113–1–026113–15, 2004.

[120] S. M. Omohundro, C.-C. Lim, and J. Bilmes. The Sather language compiler/debugger implementation. Technical Report TR-92-017, International Computer Science Institute, Berkeley, 1992.

[121] A. Östlin and R. Pagh. Uniform hashing in constant time and linear space. In *Proceedings of the thirty-fifth Symposium on Theory of Computing (STOC)*, pp. 622–628. ACM, 2003.

[122] D. A. Patterson and J. L. Hennessy. *Computer organization and design. The hardware/software interface*. Morgan Kaufmann Publishers Inc., 3rd edition, 2005.

[123] D. J. Pearce. *Some directed graph algorithms and their application to pointer analysis*. PhD thesis, Imperial College of Science, Technology and Medicine, University of London, 2005.

[124] D. J. Pearce and P. H. J. Kelly. A dynamic topological sort algorithm for directed acyclic graphs. *Journal of Experimental Algorithmics*, 11:1.7.1–24, 2006. Preliminary version appeared as [125].

[125] D. J. Pearce and P. H. J. Kelly. A dynamic algorithm for topologically sorting directed acyclic graphs. In *Proceedings of the third international*

*Workshop on Experimental and Efficient Algorithms (WEA)*, Vol. 3059 of *Lecture Notes in Computer Science*, pp. 383–398, 2004.

[126] D. J. Pearce, P. H. J. Kelly, and C. Hankin. Online cycle detection and difference propagation for pointer analysis. In *Proceedings of the 3rd international IEEE Workshop on Source Code Analysis and Manipulation (SCAM)*, 2003.

[127] S. Pettie. A new approach to all-pairs shortest paths on real-weighted graphs. *Theoretical Computer Science*, 312:47–74, 2004.

[128] S. Pettie and V. Ramachandran. An optimal minimum spanning tree algorithm. *Journal of the ACM*, 49:16–34, 2002.

[129] B. Pittel and R. Tungol. A phase transition phenomenon in a random directed acyclic graph. *Random Structures and Algorithms*, 18:164–184, 2001.

[130] R. C. Prim. Shortest connection networks and some generalisations. *Bell System Technical Journal*, 36:1389–1401, 1957.

[131] N. Rahman and R. Raman. Analysing the cache behaviour of non-uniform distribution sorting algorithm. In *Proceedings of the eighth annual European Symposium on Algorithms (ESA)*, pp. 380–391, 2000.

[132] G. Ramalingam. *Bounded Incremental Computation*, Vol. 1089 of *Lecture Notes in Computer Science (LNCS)*. Springer, 1996. ISBN 978-3-540-61320-6.

[133] G. Ramalingam and T. W. Reps. On competitive on-line algorithms for the dynamic priority-ordering problem. *Information Processing Letters*, 51: 155–161, 1994.

[134] G. Ramalingam and T. W. Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158:233–277, 1996.

[135] J. Reif and P. Spirakis. Expected parallel time and sequential space complexity of graph and digraph problems. *Algorithmica*, 7:597–630, 1992.

[136] L. Roditty and U. Zwick. A fully dynamic reachability algorithm for directed graphs with an almost linear update time. In *Proceedings of the thirty-sixth annual ACM Symposium on Theory of Computing (STOC)*, pp. 184–191, 2004.

[137] L. Roditty and U. Zwick. On dynamic shortest paths problems. In *Proceedings of the twelfth European Symposium on Algorithms (ESA)*, Vol. 3221 of *Lecture Notes in Computer Science*, pp. 580–591. Springer, 2004.

[138] B. Sach and R. Clifford. An empirical study of cache-oblivious priority queues and their application to the shortest path problem, 2008. `arXiv:0802.1026v1`.

[139] P. Sanders. Random permutations on distributed, external and hierarchical memory. *Information Processing Letters*, 67:305–309, 1998.

[140] P. Sanders, S. Egner, and J. H. M. Korst. Fast concurrent access to parallel disks. In *Proceedings of the eleventh ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp. 849–858, 2000.

[141] P. Sanders, D. Schultes, and C. Vetter. Mobile route planning. In *Proceedings of the sixteenth European Symposium on Algorithms (ESA)*, number 5193 in Lecture Notes in Computer Science (LNCS), pp. 732–743. Springer, 2008.

[142] Seagate Technology. `http://www.seagate.com/cda/products/discsales/marketing/detail/0,1081,628,00.html`.

[143] S. Sen and S. Chatterjee. Towards a theory of cache-efficient algorithms. In *Proceedings of the eleventh annual ACM-SIAM Symposium on discrete algorithms (SODA)*, pp. 829–838. SIAM, 2000.

[144] V. Shkapenyuk and T. Suel. Design and implementation of a high-performance distributed web crawler. In *Proceedings of the eighteenth International Conference on Data Engineering (ICDE)*, pp. 357–368. IEEE, 2002.

[145] J. F. Sibeyn. From parallel to external list ranking. Technical report, Max Planck Institut für Informatik, Saarbrücken, Germany, 1997.

[146] J. F. Sibeyn, J. Abello, and U. Meyer. Heuristics for semi-external depth first search on directed graphs. In *Proceedings of the fourteenth annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, pp. 282–292, 2002.

[147] A. Stepanov and M. Lee. *The Standard Template Library*. Hewlett Packard Laboratories, 1995. `http://www.sgi.com/tech/stl/`.

[148] F. M. Suchanek, G. Kasneci, and G. Weikum. Yago – a large ontology from wikipedia and wordnet. *Elsevier Journal of Web Semantics*, 2008. (to appear).

[149] R. E. Tarjan. Efficiency of a good but not linear set union algorithm. *Journal of the ACM*, 22:215–225, 1975.

[150] The Stanford WebBase Project. http://www-diglib.stanford.edu/~testbed/doc2/WebBase/.

[151] TPIE. http://madalgo.au.dk/Trac-tpie/.

[152] J. S. Vitter and E. A. M. Shriver. Algorithms for parallel memory I: Two level memories. *Algorithmica*, 12:110–147, 1994.

[153] E. W. Weisstein. ”Web Graph” from MathWorld – A Wolfram Web Resource. http://mathworld.wolfram.com/WebGraph.html.

[154] C. H. Wu, L. P. Chang, and T. W. Kuo. An efficient R-tree implementation over flash-memory storage systems. In *Proceedings of the eleventh ACM international symposium on Advances in Geographic Information Systems*, pp. 17–24. ACM Press, 2003.

[155] C. H. Wu, L. P. Chang, and T. W. Kuo. An efficient B-tree layer implementation for flash-memory storage systems. *ACM Transactions on Embedded Computing Systems*, 6, 2007.

[156] ZCAV. http://www.coker.com.au/bonnie++/zcav/.

# Summary

The notion of graph traversal is of fundamental importance to solving many computational problems. It has therefore received considerable attention in the computer science literature - many linear or near-linear time algorithms for traversing graphs have been developed. In many modern applications involving graph traversal such as those arising in the domain of social networks, Internet based services, fraud detection in telephone calls etc., the underlying graph is very large and dynamically evolving. For these applications, the simple linear or near-linear time RAM-model static graph traversal algorithms are often inappropriate because of the large number of I/Os they incur. Also, these algorithms can't be easily adapted to the dynamic framework. Furthermore, many application needs are already fulfilled if the total running time is bounded in the average-case and not necessarily in the worst-case. This thesis deals with the design and engineering of graph traversal algorithms for massive and/or dynamic graphs.

We engineer various I/O-efficient Breadth First Search (BFS) algorithms for massive sparse undirected graphs. Our pipelined implementations with low constant factors makes BFS viable on massive graphs. For many graphs with around a billion edges (with 1–3 GB RAM), it reduces the running-time for BFS traversal from a few *months* required by the simple RAM model BFS algorithm to a few *hours*. Our code has now evolved into a software package, that will be eventually integrated into an external memory library.

Our detailed experimental study suggests that a simple external memory BFS algorithm by Munagala and Ranade [115] (MR_BFS) performs quite well on low diameter graphs or when the edges are kept on the disk in the order required for the BFS traversal. The better asymptotic worst-case I/O bound of the BFS algorithm by Mehlhorn and Meyer [106] (MM_BFS) help it to outperform MR_BFS on moderate to large diameter graphs. MM_BFS also benefits from our heuristics that preserve its worst-case guarantees. Exploiting a priori knowledge of the graph structure and disk parallelism further alleviate the I/O bottleneck of MM_BFS. We

also show evidence that the cache-oblivious BFS algorithms are at least a factor of four to five slower than their external memory counterparts, when the input graph resides on the disk.

Flash memory is fast becoming the dominant form of end-user storage in mobile computing. Since storage devices play a crucial role in the performance of (traversal) algorithms when the input (graph) data does not fit in the main memory, it is important to understand the I/O-characteristics of the storage devices to be able to predict the real running times of these algorithms. Such an understanding can also be exploited to design algorithms that are faster in practice. We characterize the performance of NAND flash based storage devices, including many solid state disks. We show that unlike hard disks, these devices have faster random reads than random writes. Interestingly, we found that the cost of random writes on flash devices is non-uniform in time and depends on the I/O-history of the device. We also analyze the effect of misalignments, aging, controller interface, etc. on the performance obtained on these devices. We show that despite the similarities between flash memory and RAM (fast random reads) and between flash disk and hard disk (both are block based devices), the algorithms designed in the RAM model or the external memory model do not realize the full potential of the flash memory devices. Thus, there is a need for a different model that distinguishes between read and write blocks to get the best performance on flash devices.

In the scenario when a solid state disk is used as an additional secondary storage rather than replacing the traditional hard disk, we engineer the I/O-efficient BFS implementation to exploit the comparative advantages of both the disks. We show that on a difficult graph class for external memory BFS, this is at least 25% faster than randomly striping the data on the two disks.

We present a simple algorithm which maintains the topological order of a directed acyclic graph with $n$ nodes under an online edge insertion sequence in $O(n^{2.75})$ time, independent of the number $m$ of edges inserted. For dense DAGs, this is an improvement over the previous best result of $O(\min\{m^{\frac{3}{2}}\log n, m^{\frac{3}{2}} + n^2 \log n\})$ by Katriel and Bodlaender [91]. While our analysis holds only for the incremental setting, our algorithm itself is fully dynamic. The externalization of our algorithm provides interesting new results for dynamic topological ordering in external memory.

We also present the first average-case analysis of online topological ordering algorithms. We prove an expected runtime of $O(n^2 \operatorname{polylog}(n))$ under insertion of the edges of a complete DAG in a random order for various incremental topological ordering algorithms.

# Zusammenfassung

Die Traversierung von Graphen ist von fundamentaler Bedeutung für das Lösen vieler Berechnungsprobleme. Folglich findet sie grosse Beachtung in der Informatik-Literatur; es wurden viele lineare oder fast-lineare Traversierungsalgorithmen vorgeschlagen. Moderne Anwendungen, die auf Graphtraversierung beruhen, findet man unter anderem in sozialen Netzwerken, internetbasierten Dienstleistungen, Betrugserkennung bei Telefonanrufen. In vielen dieser Anwendungen ist der zugrunde liegende Graph sehr gross und ändert sich kontinuierlich. Einfache lineare oder fast-lineare Graphtraversierungs-Algorithmen, die für das RAM-Modell entwickelt wurden, sind in diesen Anwendungen oft nicht adäquat, da sie eine hohe Anzahl von I/O-Zugriffen verursachen. Auch ist es nicht leicht diese Algorithmen für dynamische Szenarien anzupassen. Ferner werden die Anforderungen vieler Anwendungen bereits erfüllt, wenn die Gesamtlaufzeit im Average-Case und nicht unbedingt im Worst-Case begrenzt ist. Diese Arbeit hat den Entwurf und das Entwickeln von Graphtraversierungs-Algorithmen für massive und/oder dynamische Graphen zum Thema.

Wir entwickeln mehrere I/O-effiziente Breitensuch-Algorithmen für massive, dünnbesiedelte, ungerichtete Graphen. Im Zusammenspiel mit Heuristiken zur Einhaltung von Worst-Case-Garantien, ermöglichen unsere pipelinebasierten Implementierungen die Praktikabilität von Breitensuche auf massiven Graphen. Für viele Graphen mit rund eine Milliarde Kanten (mit 1–3 GB RAM) wird die Breitensuch-Laufzeit von wenigen Monaten, die vom einfachen RAM-Modell-Algorithmus zur Breitensuche benötigt werden, auf wenige Stunden reduziert. Unser Code ist als Software-Paket vorhanden, das voraussichtlich in eine Externspeicher-Bibliothek integriert wird.

Unsere detaillierte, experimentelle Untersuchung legt nahe, dass ein einfacher Breitensuchalgorithmus für den externen Speicher, siehe Munagala and Ranade [115] (MR_BFS), gute Leistung erbringt, wenn der Graph einen kleinen Durchmesser hat, oder seine Kanten im Speicher in der Reihenfolge abgelegt sind, die von

der Breitensuche benötigt wird. Die bessere, asymptotische I/O-Grenze für den Worst-Case des Breitensuch-Algorithmus von Mehlhorn und Meyer [106] (MM_BFS) führt zu einer besseren Leistung als bei MR_BFS auf Graphen mit moderatem bis grossem Durchmesser. MM_BFS profitiert auch von unseren Heuristiken, welche die Worst-Case-Garantien bewahren. Das Wissen über die Graphstruktur und den Plattenparallelismus mildern die Wirkung des I/O-Engpasses bei MM_BFS. Wir zeigen auch Indizien dafür auf, dass cache-oblivious Breitensuch-Algorithmen mindestens um Faktor Vier oder Fünf langsamer sind als ihre Pendants für den externen Speicher, wenn der Graph auf der Platte residiert.

Flash-Speicher wird immer mehr zur dominanten Form der Speicherung für Endbenutzer im Mobile Computing. Da Speichermedien eine wichtige Rolle für die Leistung von Traversierungs-Algrithmen spielen, wenn die Daten nicht in den Hauptspeicher passen, ist es notwendig, die I/O-Merkmale von Speichermedien zu verstehen, um reale Laufzeiten für diese Algorithmen vorherzusagen. Dieses Verständnis kann ausgenutzt werden, um Algorithmen zu entwerfen, die in der Praxis schneller sind.

Wir charakterisieren die Leistung von NAND-Flash basierten Speichermedien, einschliesslich vieler solid-state Disks. Wir zeigen, dass diese Medien, im Gegenstz zu Festplatten, einen schnelleren wahlfreien Lese- als Schreibe-Zugriff haben. Interessanterweise haben wir herausgefunden, dass die Kosten des wahlfreien Schreibe-Zugriffs auf Flash-Medien ungleichmässig im Bezug auf die Zeit sind und von der I/O-Historie des Mediums abhängen. Zusätzlich analysieren wir die Wirkung von Ausrichtungsfehlern, Alterung, vorausgehenden I/O-Mustern, usw., auf die Leistung dieser Medien. Wir zeigen, dass trotz der Ähnlichkeiten von Flash-Speicher und RAM (schnelle wahlfreie Lese-Zugriffe) und von Flash-Platten und Festplatten (beide sind blockbasiert) Algorithmen, die für das RAM-Modell oder das Externspeicher-Modell entworfenen wurden, nicht das volle Potential der Flash-Speicher-Medien ausschöpfen. Folglich gibt es also einen Bedarf für ein neues Modell, das zwischen Lese- und Schreibe-Blöcken unterscheidet, um beste Leistung auf Flash-Medien zu gewährleisten.

Wir entwickeln einen I/O-effiziente Breitensuch-Algorithmus für das Szenario, in dem eine solid-state Disk als zusätzlicher Zweitspeicher und nicht als Ersatz für die traditionelle Festplatte benutzt wird, um die komparativen Vorteile beider Disks auszunutzen Wir zeigen, dass dies mindestens 25% schneller ist als ein zufälliges Aufteilen der Daten auf beiden Disks.

Wir stellen einen einfachen Algorithmus vor, der beim Online-Einfügen von Kanten die topologische Ordnung von einem gerichteten, azyklischen Graphen (DAG) mit $n$ Knoten beibehält. Dieser Algorithmus hat eine Laufzeitkomplexität von

$O(n^{2.75})$ unabhängig von der Anzahl $m$ der eingefügten Kanten. Für dichte DAGs ist dies eine Verbesserung des besten, vorherigen Ergebnisses von $O(\min\{m^{\frac{3}{2}} \log n, m^{\frac{3}{2}} + n^2 \log n\})$, siehe Katriel and Bodlaender [91]. Während die Analyse nur im inkrementellen Szenario gütlig ist, ist unser Algorithmus völlständig dynamisch. Die Externalisierung unseres Algorithmus liefert neue interessante Ergebnisse für dynamische, topologische Ordnungen im externen Speicher.

Ferner stellen wir die erste Average-Case-Analyse von Online-Algorithmen zur Unterhaltung einer topologischen Ordnung vor. Für mehrere inkrementelle Algorithmen, welche die Kanten eines kompletten DAGs in zufälliger Reihenfolge einfügen, beweisen wir eine erwartete Laufzeit von $O(n^2 \operatorname{polylog}(n))$.

# Curriculum Vitae

## Personal Data


| | |
|---|---|
| Name | Deepak Ajwani |
| Citizenship | Indian |
| Marital Status | Married |
| Telephone | +45-89425785 |
| Email | ajwani@madalgo.au.dk |
| WWW | http://www.mpi-inf.mpg.de/~ajwani |


## Research Interests

Algorithms for memory hierarchies, Dynamic graph algorithms, Algorithm Engineering

## Education

2005 - **Ph.D. candidate**, Computer Science
*Universität des Saarlandes & Max Planck Institut für Informatik*
Advisor: Prof. Dr. h. c. Kurt Mehlhorn

2003 - 2005 **M.Sc.**, Computer Science
*Universität des Saarlandes & Max Planck Institut für Informatik*
Advisors: Dr.-Ing. Ulrich Meyer and Dr. rer. nat. Peter Sanders

1998 - 2003 **M.Tech. + B.Tech.**, Computer Science and Engineering.
*Indian Institute of Technology, Delhi*
Advisor: Prof. Sandeep Sen

## Work Experience

- Worked as **teaching assistant** for a course on "Algorithms and Data Structures" with Dr.-Ing. Ulrich Meyer, Dr. Ernst Althaus and Dr. Surender Baswana, Universität des Saarlandes.
- Worked as a **software engineer** (July – Oct 2003) with Read-Ink Technologies Pvt. Ltd., a company started by Stanford emeritus Prof. Thomas Binford, to develop a handwriting analysis software for PDAs.
- Worked as **teaching assistant** for a course on "Numeric and Scientific Computing" under Dr. Dheeraj Bhardwaj, Indian Institute of Technology, Delhi.
- Worked as an **intern** with Prof. Ron Shamir, Tel Aviv University during May – July, 2001.

## Awards and Scholarships

- Recipient of **International Max Planck Research School** scholarship.
- Recipient of **Jawahar Gajree Scholarship** from IIT Delhi.
- Secured **99.79 percentile in GATE-2003** (nationwide examination for masters and Ph.D. positions in India).
- Recipient of **Merit-cum-means Scholarship** from IIT Delhi.
- Secured **166th rank (All India) out of a total of 120,000 students in IIT-JEE 1998**.
- Recipient of **National Talent Search Scholarship** by National Council of Education Research and Training (NCERT), India.
- Scholarship offered by Central Board for Secondary Education, India for **excellent performance in X Board** (99 % in Science).

# Publications

## Book chapters

- **Realistic Computer Models**.
  In "Algorithm Engineering", M. Müller-Hannemann and S. Schirra (eds.), Springer, (to be published in 2008).
  Joint work with Henning Meyerhenke.

- **Design and engineering of external memory traversal algorithms for general graphs**.
  Accepted for publication in the Springer LNCS book devoted to the DFG Schwerpunktprogramm 1126 on "Algorithmik grosser und komplexer Netzwerke".
  Joint work with Ulrich Meyer.

## Refereed Journal articles

- **Breadth First Search on Massive Graphs**.
  Accepted for publication in the DIMACS Series book devoted to the 9th Implementation Challenge.
  Joint work with Ulrich Meyer and Vitaly Osipov.

- **An O($n^{2.75}$) Algorithm for Online Topological Ordering**.
  Accepted for publication in the ACM Transactions on Algorithms.
  Joint work with Tobias Friedrich and Ulrich Meyer.

- **Average-case analysis of Online Topological Ordering**.
  (Under submission).
  Joint work with Tobias Friedrich.

## Refereed conference articles

- **Efficient Algorithms for Flash Memories**.
  (Under submission).
  Joint work with Andreas Beckmann, Riko Jacob, Ulrich Meyer and Gabriel Moruz.

- **Characterizing the Performance of Flash Memory Storage Devices and its Impact on Algorithm Design**.
  In Proceedings of the 7th International Workshop on Experimental Algorithms (WEA'08), pp. 208–219, Massachusetts, USA, 2008.
  A detailed version of this paper is available as Max Planck Institut für Informatik Research Report no. MPI-I-2008-1-001
  Joint work with Itay Malinger, Ulrich Meyer and Sivan Toledo.

- **Average-Case Analysis of Online Topological Ordering**.
  In Proceedings of the 18th International Symposium on Algorithms and Computation (ISAAC'07), pp. 464–475, Sendai, Japan, 2007.
  Joint work with Tobias Friedrich.

- **On Computing the Centroid of the Vertices of an Arrangement and Related Problems**.
  In Proceedings of the 10th Workshop on Algorithms and Data Structures (WADS'07), pp. 520–529, Halifax, Canada, 2007.
  Joint work with Saurabh Ray, Raimund Seidel and Hans Raj Tiwary.

- **Conflict-Free Coloring for Rectangle Ranges Using $O(n^{0.382+\varepsilon})$ Colors**.
  In Proceedings of the 19th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA'07), pp. 181–187, San Diego, CA, USA, 2007.
  Joint work with Khaled Elbassioni, Sathish Govindarajan and Saurabh Ray.

- **Improved External Memory BFS Implementations**.
  In Proceedings of the 9th Workshop on Algorithm engineering and experiments (ALENEX'07), pp. 3–12, New Orleans, USA, 2007.
  Also accepted at 9th DIMACS implementation challenge on shortest path, Piscataway, NJ, USA, 2006.
  Joint work with Ulrich Meyer and Vitaly Osipov

- **An $O(n^{2.75})$ Algorithm for Online Topological Ordering**.
  In Proceedings of the 10th Scandinavian workshop on Algorithm Theory (SWAT'06), pp. 53–64, Riga, Latvia, 2006
  A preliminary version of this paper appeared in Electronic Notes in Discrete Mathematics, vol. 25, pp. 7-12, 2006
  Joint work with Tobias Friedrich and Ulrich Meyer

- **A Computational Study of External Memory BFS Algorithms**.
  In Proceedings of the 17th ACM-SIAM Symposium on Discrete Algorithms (SODA'06), pp. 601–610, Miami, USA, 2006
  Joint work with Roman Dementiev and Ulrich Meyer

- **Parallel Algorithm for Real Time Decision System for Financial Markets**.
  Accepted as a poster in the 10th annual International Conference on High Performance Computing (HiPC'03), Hyderabad, India, 2003.
  Joint work with Dheeraj Bhardwaj and Manish Sansi

## Software Projects

- **External Memory BFS**
  This project involved design, implementation and experimentation with external memory Breadth-First Search (BFS) algorithms. The software package consists of pipelined I/O efficient graph generators, BFS decomposition verifiers, Munagala and Ranade's BFS traversal algorithm and the randomized and deterministic versions of Mehlhorn and Meyer's approach. An extensive empirical study analyzing the behaviour of these algorithms for different graph classes in general, and for very large sparse graphs, in particular, has been carried out. On many of these graphs, **this software brings down the runtime of computing BFS level decomposition from months (with standard implementations, e.g. LEDA BFS) to a few hours**, thereby making BFS viable for massive graphs.

- **Cache-efficient FFT**
  The goal of this software project was to study the effects of emulating the cache (as given by S. Sen and S. Chatterjee) on the number of conflict misses and the actual running time of FFT implementation. The project involved a lot of experiments on the actual execution time and cache behaviour of various cache-efficient algorithms for bit-reverse permutations, matrix transposition, general permutations and FFT. **On various architectures, the running time of my implementation in C is significantly better than that of the widely used FFT library FFTW**.

- **Expander**
  During my internship with Prof. Ron Shamir (May – July, 2001) in Tel Aviv University, I designed and implemented a Java software incorporating different visualization tools for clustering algorithms. The software package involved elaborate user documentation and a powerful GUI embedding the clustering algorithms – Hierarchial, K-Means, SOM and CLICK, various normalization routines and different kind of visualization tools like the red-green matrix, similarity data matrix and some designed by me based on

sammon mapping and other heuristics. Particular emphasis was laid to make it reasonably fast, even for large biological data-sets. This software later evolved into EXPANDER (EXpression Analyzer and DisplayER), a tool for the analysis of gene expression data. Currently, it has more than **5,500 downloads and 50 citations**.