External Memory Algorithms for Diameter and All-Pairs Shortest-Paths on Sparse Graphs

Lars Arge[1,⋆], Ulrich Meyer[2,⋆⋆], and Laura Toma[3,⋆⋆⋆]

[1] Department of Computer Science, Duke University, Durham, NC 27708, USA.
[2] Max-Planck-Institut für Informatik, 66123 Saarbrücken, Germany.
[3] Department of Computer Science, Bowdoin College, Brunswick, ME 04011, USA.

Abstract. We provide I/O-efficient algorithms for diameter and all-pairs shortest-paths (APSP) on undirected graphs $G(V,E)$. For general non-negative edge weights and $E/V = o(B/\log V)$ our approaches are the first to achieve $o(V^2)$ I/Os. We also show that unweighted APSP can be solved with just $O(V \cdot \mathrm{sort}(E))$ I/Os; these approaches require $O(V^2)$ space. For diameter computations we additionally provide I/O-space tradeoffs. Finally, we provide improved results for diameter and APSP directed planar graphs.

1 Introduction

Computing shortest paths and the diameter of a graph are two of the most fundamental problems in algorithmic graph theory. For example, research in web modeling uses shortest path and diameter computations as primitive routines for investigating the structure of the web. The diameter of a graph is also used in many algorithms as a lower or upper bound. Further applications come from Geographic Information Systems (GIS). In the recent years an increasing number of these applications need to run on massive graphs.

When working with massive graphs, only a fraction of the data can be held in the main memory of a state-of-the-art computer. Thus, the block transfer of data between main memory and secondary, disk-based memory, and not the internal memory computation, is often the bottleneck. Therefore, efficient external memory (or I/O-efficient) algorithms with optimized data access patterns can lead to considerable runtime improvements. Unfortunately, current shortest paths algorithms are only I/O-efficient on dense graphs whereas most real-world graphs are sparse. Therefore we aim to develop I/O-efficient shortest-path algorithms for sparse graphs of arbitrary structure. As a side effect of our research we also obtained a significantly improved algorithms for the special case of planar directed graphs (digraphs).

⋆ Supported in part by the National Science Foundation through ESS grant EIA–9870734, RI grant EIA-9972879 and CAREER grant EIA-9984099.
⋆⋆ Support by DFG grant SA 933/1-1. Part of this work was done while visiting Duke.
⋆⋆⋆ Part of this work was done while being a PhD-student at Duke University.

Problem Definition. Let $G = (V, E)$[4] be an undirected graph and let s and t be two vertices in G. The shortest path $\delta(s,t)$ from s to t is the path of minimum length among all paths from s to t in G, where the length of a path is the sum of the weights of its edges. The length of the shortest path $\delta(s,t)$ is called the *distance* from s to t in G. The *single-source shortest-path* (SSSP) problem finds the shortest paths from a source vertex to all other vertices in G. The *all-pairs shortest-paths* (APSP) problem finds the shortest path between every pair of vertices in G. Often, one only needs the shortest-path distances and not in the paths themselves. The *diameter* of G is the maximum distance between any two vertices of G. For unweighted graphs, SSSP and APSP are also referred to as *breadth-first search* (BFS) and *all-pairs breadth-first search* (AP-BFS). We will call a graph sparse iff $E = O(V)$.

I/O-Model. Our results assume the standard two-level I/O-model with one (logical) disk [1]. The model defines two parameters: M is the number of vertices/edges that fit into internal memory, and B is the number of vertices/edges that fit into a disk block, where $M < c \cdot V$ for some positive constant $c < 1$ (fully-external model) and $1 \leq B \leq M/2$. It is common practice to treat B and M as parameters even though for a fixed machine they may be constant (for example $B \simeq 10^6$ and $M \simeq 10^9$). An *Input/Output* (or simply *I/O*) is the operation of transferring a block of data between main memory and disk. The *I/O-complexity* of an algorithm is the number of I/Os it performs. The *scanning bound*, $\text{scan}(N) = \frac{N}{B}$ is the number of I/Os necessary to read N contiguous items from disk. The *sorting bound*, $\text{sort}(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ represents the number of I/Os required to sort N contiguous items on disk [1]. For all realistic values of V, B, and M, $\text{scan}(V) < \text{sort}(V) \ll V$ and $\log V \ll B$. An external-memory algorithm for a graph problem with internal-memory complexity $C(V, E)$ is usually called I/O-efficient if it performs $O(\text{sort}(C(V, E)))$ I/Os.

1.1 Previous Work

There exists a vast number of results for APSP; for a survey see [16]. The classical method to solve APSP requires $\tilde{O}(V \cdot E)$ time[5] by subsequently running an SSSP algorithm for each vertex in the graph. Even though there are improved APSP algorithms for *dense* graphs (the currently best solution based on matrix multiplication requires $O(V^{2.575})$ time [17]), the classical method still constitutes by far the fastest way to solve APSP on general *sparse* graphs.

I/O-efficient graph algorithms have been considered by a number of authors; for a recent review see [12]. A direct conversion of the classical APSP approach to external memory requires an SSSP algorithm that is I/O-efficient on sparse graphs. However, the currently best algorithm for unweighted SSSP (i.e., BFS)

[4] For convenience we use the name of a set to denote both the set and its cardinality. Furthermore, we will assume $E = \Omega(V)$ in order to simplify notation.

[5] We use $\tilde{O}(f(V, E))$ as a shorthand for $O(f(V, E) \cdot (\log V)^{O(1)})$.

needs $\Omega(V/\sqrt{B})$ I/Os on sparse graphs [11]; in the case of general[6] non-negative edge weights, even $\Omega(V)$ I/Os are needed [8] (resulting in $\Omega(V^2)$ I/Os for the respective APSP conversion). If the graph is directed then already the currently best BFS algorithms take $\Omega(V)$ I/Os [4, 5]. This is far from the currently best lower bound for BFS (and SSSP) of $\Omega(\min\{V, \text{sort}(V)\} + E/B)$ I/Os.

The long and so far unsuccessful search for a BFS/SSSP algorithm using $\tilde{O}(\text{sort}(E))$ I/Os on general graphs has led to a number of improved results for special graph classes (e.g., BFS and SSSP can be solved using $O(\text{sort}(V))$ I/Os on planar graphs [2, 10]). Seemingly, it has also discouraged researches to explore the potentials of I/O-efficient APSP/diameter algorithms for general sparse graphs. We show that the latter problems can be tackled even if the respective I/O-efficient BFS/SSSP algorithms may not exist.

1.2 New Results

We provide the first I/O-efficient algorithm for the diameter and APSP on sparse undirected graphs with general non-negative edge weights (Sect. 2). Under the realistic condition $E/V \leq B/\log V$ our method needs $O(V \cdot (\sqrt{(V \cdot E \cdot \log V)/B} + \text{sort}(E)))$ I/Os, which is $O(V^2 \cdot \sqrt{(\log V)/B})$ I/Os on sparse graphs. Compared to the best previous approach (V times SSSP [8]) this is an improvement by a factor of up to $\Theta(\sqrt{B/\log V})$. We also show that AP-BFS can be solved with just $O(V \cdot \text{sort}(E))$ I/Os (Sect. 3.1). For sparse graphs this means an improvement by a factor of nearly $\sqrt{B}$.

The solutions above require $O(V^2)$ external space, thus matching the size of the output for APSP. For diameter computations (output size $O(1)$) it is desirable to have working space $\tilde{O}(E)$. We therefore provide I/O-space tradeoffs (Sect. 3.2): in particular, we show how to solve the unweighted diameter problem on sparse graphs using $O(\text{sort}(k \cdot V^2 \cdot B^{1/k}))$ I/Os with $O(k \cdot V)$ space for any integer k, $3 \leq k \leq \log B$. Note that for the extreme case $k = \lfloor \log B \rfloor$ we require $O(\text{sort}(V^2 \cdot \log B))$ I/Os and $O(V \cdot \log B)$ space for sparse unweighted graphs.

Finally, in Sect. 4 we consider the class of planar digraphs. We show that on this class of graphs APSP can be computed optimally in $O(\text{scan}(V^2))$I/Os and the diameter in only $O(\text{sort}(V^2)/B)$ I/Os and $O(V)$ space. This means a factor of B less I/Os than the previously best approach.

2 I/O-Efficient APSP on General Sparse Graphs

In this section we give an APSP algorithm for undirected sparse graphs with non-negative edge weights and show that it needs $O(V \cdot (\sqrt{(V \cdot E \cdot \log V)/B} + \text{sort}(E)))$ I/Os assuming $E/V \leq B/\log V$. Before we dive into details we first review some basic concepts for external-memory SSSP.

[6] It has been shown, however, that there is an algorithm for SSSP with bounded edge weights that requires $O(\sqrt{(V \cdot E/B) \cdot \log(W/w)} + \text{sort}(E))$ I/Os, where W and w are the largest and smallest weights in G, respectively [13].

2.1 Preliminaries

Dijkstra's algorithm [6] is the classical internal-memory SSSP approach. It uses a priority queue Q to store all vertices of G that have not been settled yet; the priority of a vertex v in Q is the length of the currently shortest known path from s to v. The next vertex v to be settled is retrieved from Q using a *deletemin* operation. Then the algorithm relaxes the edges between v and all its non-settled neighbors, that is, performs a $\textit{decrease\_key}(w, \text{dist}(s, v) + \omega(v, w))$ operation for each such neighbor w whose priority is greater than $\text{dist}(s, v) + \omega(v, w)$.

An I/O-efficient version of Dijkstra's algorithm has to (a) avoid accessing adjacency lists at random, (b) deal with the lack of efficient *decrease_key* operations in current external-memory priority queues, and (c) efficiently remember settled vertices. The SSSP algorithm of [8], **KS-SSSP** for short, ignores (a) and spends $\Omega(1)$ I/Os on retrieving the adjacency list of each settled vertex. As for (b), **KS-SSSP** uses an I/O-efficient "tournament tree" priority-queue, I/O-TT for short, which emulates *insert* and *decrease_key* operations using a weaker *update* operation, described below. As for (c), **KS-SSSP** performs an *update* operation for *every* neighbor of a settled vertex, which eliminates the need to remember previously settled vertices, but may re-insert settled vertices into the priority queue Q ("spurious updates"). Using a second priority queue Q^*, these re-inserted vertices are removed from Q before they can be settled for a second time: **KS-SSSP** proceeds in $O(E)$ rounds each of which examines the top-priority elements of both Q and Q^* and removes at least one of them (and eliminates a spurious element of Q if required)[7]

The I/O-TT of [8] supports three operations: (i) *deletemin* (extract the element $\langle x, k\rangle$ with smallest key k and replace it by the new entry $\langle x, \infty\rangle$); (ii) *delete(x)* (replace $\langle x, \textit{oldkey}\rangle$ by $\langle x, \infty\rangle$); (iii) *update(x,newkey)* (replace $\langle x, \textit{oldkey}\rangle$ by $\langle x, \textit{newkey}\rangle$ if $\textit{newkey} < \textit{oldkey}$). Note that (ii) and (iii) do not require the old key to be known.

The I/O-TT is based on a complete binary tree, where some rightmost leaves may be missing. Let $M' = c \cdot M$ for some positive constant $c < 1$; the I/O-TT for V elements has $\lceil V/M' \rceil$ leaves and hence $O(\log(V/M))$ levels. Elements with indices in the range $\{i \cdot M', \ldots, (i+1) \cdot M' - 1\}$ are mapped to the i-th leaf. The index range of internal nodes of the I/O-TT is given by the union of the index ranges of their children. Internal nodes of the I/O-TT keep a list of at least $M'/2$ and at most M' elements each (sorted according to their priorities). If the list of a tree node v contains z elements, then they are the smallest z out of all those elements in the tree being mapped to the leaves that are descendants of v. Furthermore, each internal node is equipped with a signal buffer of size M'. Initially, the I/O-TT stores the elements $\langle 0, +\infty\rangle, \langle 1, +\infty\rangle, \ldots, \langle V-1, +\infty\rangle$, out of which the lists of internal nodes keep at least $M'/2$ elements each.

[7] The original version of **KS-SSSP** [8] does not properly handle adjacent vertices with the same SSSP-distance. This can be fixed by processing all such vertices in one round [15]. Essentially the same idea also works with our new APSP approach. Details will be given in the full version of this paper.

The operations (i)–(iii) generate *signals* which serve to propagate information down the tree; signals are inserted into the root node, which is always kept in internal memory. When a signal arrives in a node it may create, delete or modify an element kept in this node; the signal itself may be discarded, altered or remain unchanged. Non-discarded signals are stored until the capacity of the node's buffer is exceeded; then they are sent down the tree towards the unique leaf node its associated element is mapped to. The following amortized bound has been shown:

Lemma 1 ([8]). *On an I/O-efficient tournament tree with V elements, any sequence of z delete/deletemin/update operations requires at most $O(z/B \cdot \log(V/M)) = O(z/B \cdot \log(V/B))$ I/Os.*

2.2 Multi-Tournament-Trees and Concurrent SSSP Computations

We first consider how to bundle I/O-TTs in order to support I/O-efficient APSP. Then we present our new approach, **Fast-APSP**. Note that the latter bound of Lemma 1 is obtained by choosing $M' = \Theta(B)$. Setting $M' \ll B$ will result in a worse bound of $O(z/M' \cdot \log(V/M'))$ I/Os, which is usually not desired. However, such a choice allows us to bundle the corresponding nodes of a number of I/O-TTs in one disk block. Concretely speaking, we consider *I/O-efficient multi-tournament-trees*, I/O-MTTs for short. Let $L \geq 2$ a parameter to be fixed later. An I/O-MTT consists of L independent I/O-TTs $T_0, \ldots, T_{L-1}$ with parameter $M' = \Theta(B/L)$. In particular, this means that the root nodes of all L bundled I/O-TTs can be kept in one block in internal memory while performing operations on the different I/O-TTs (unless refilling occurs which, however, is already accounted for in the amortized I/O-bound of $O(\log(V/M')/M') = O(L \log V/B)$ I/Os for single operations on each of the bundled I/O-TTs).

Our new approach, **Fast-APSP**, takes **KS-SSSP** from Sect. 2.1 as a building block. However, it solves APSP by working on all V underlying SSSP problems concurrently. This requires V priority-queue pairs (Q_i, Q_i^*), $0 \leq i < V$, where the entries of (Q_i, Q_i^*) belong to the i-th SSSP problem. The set of priority queues is implemented using I/O-MTTs. This requires $O(V^2)$ space.

Fast-APSP proceeds in $O(V)$ rounds. In each round it loads the roots of all its I/O-MTTs and extracts a settled graph node with smallest distance from each of the $L/2$ priority queue pairs (Q_i, Q_i^*) of the current I/O-MTT. Note that for each pair (Q_i, Q_i^*) this may require some initial *deletemin* operations on Q_i^* (coupled with *delete* operations on Q_i) before a settled graph node can be extracted from Q_i. Instead of accessing the required adjacency lists of settled nodes for each SSSP problem separately, **Fast-APSP** creates a node sequence $\mathcal{K}$ of the extracted settled vertices from this round, sorted according to these nodes' indices. Then it applies a parallel scan of the graph representation in order to retrieve the adjacency lists of all nodes in $\mathcal{K}$. Finally, another sorting and parallel scanning is applied to move these adjacency lists back to the priority-queue pairs of the SSSP problems they originated from in order to perform the necessary priority queue update operations there. This requires another cycling through all

I/O-MTTs, which can be overlapped with the beginning of the next round. The total size of data scanned and sorted is bounded by $O(V^2)$ during each round and by $O(V \cdot E)$ summed over all rounds. All computed distance values can be trivially appended to an output list of size $O(V^2)$ which is eventually sorted in order to produce the final distance matrix.

Fast-APSP replaces $\Omega(V^2)$ I/Os for separate adjacency-list accesses by $O(\text{sort}(V \cdot E))$ I/Os for joint accesses. The I/O complexity for the priority-queue operations is as follows: over $O(V)$ rounds, the cycling through the roots of the $O(V/L)$ I/O-MTTs takes $O(V^2/L)$ I/Os. On each of the bundled $2 \cdot V$ I/O-TTs we perform $O(E)$ priority queue operations; this accounts for another $O(VE \cdot L \log V/B)$ I/Os. Sorting the output list requires $O(\text{sort}(V^2))$ I/Os. The sum of these terms is optimized choosing $L = 2 \cdot \sqrt{B \cdot V/(E \cdot \log V)} \geq 2$. We have obtained the following:

Theorem 1. *APSP on undirected sparse graphs with non-negative edge weights can be solved using* $O\left(V \cdot \left(\sqrt{(V \cdot E \cdot \log V)/B} + \mathit{sort}(E)\right)\right)$ *I/Os and* $O(V^2)$ *space whenever* $E/V \leq B/\log V$.

3 AP-BFS and Unweighted Diameter on General Graphs

In this section we give improvements for APSP and diameter on general undirected graphs with *unweighted* edges (AP-BFS). We first present an $O(V \cdot \text{sort}(E))$ I/O solution based on the BFS algorithm in [11]. In the worst case, however, this approach requires $\Theta(V^2)$ space even if we omit producing the output matrix (i.e., in the case of diameter computation). Therefore, we propose an alternative approach, which also serves as a basis for our I/O-space tradeoffs. In this section we assume $E = O(V \cdot B)$ since for dense graphs V times BFS [14] will trivially result in $O(V \cdot \text{sort}(E))$ I/Os, too.

3.1 AP-BFS and Unweighted Diameter using $O(V \cdot \text{sort}(E))$ I/Os

Our new AP-BFS algorithm, **Fast-AP-BFS**, builds on the **Fast-BFS** algorithm of [11]. **Fast-BFS** operates in two phases. Let μ be a parameter $\mu \in [0, 1]$ to be fixed later. The first phase partitions G using a spanning tree/ Euler tour method into $O(\mu \cdot V)$ disjoint subgraphs (clusters) S_i such that the distance between any two vertices of S_i is at most $1/\mu$ in G. Each S_i consists of at most $1/\mu$ nodes. For each cluster S_i, it creates a file $\mathcal{F}_i$, which contains the adjacency lists of all vertices in S_i. The second phase of **Fast-BFS** is a modified version of the BFS-approach of [14] in the sense that it keeps a *hot pool* $\mathcal{H}$ of adjacency lists. $\mathcal{H}$ prevents the algorithm from performing V random accesses to the adjacency lists. The hot pool consists of all adjacency lists $\mathcal{F}_i$ such that the current level $L(t)$ of the BFS-tree contains a vertex in S_i. $\mathcal{H}$ is kept sorted by vertex indices. To construct the next level $L(t+1)$ of the BFS-tree it scans $\mathcal{H}$ and $L(t)$ in parallel: if the adjacency list of a vertex $u \in L(t)$ is found in $\mathcal{H}$ then it is extracted from $\mathcal{H}$; otherwise, if it is not in $\mathcal{H}$, the data from $\mathcal{F}_i$ corresponding to the cluster

S_i containing vertex u is merged into the hot pool. After all the adjacency lists of the vertices in $L(t)$ have been obtained, the tentative next level $L(t+1)$ can be generated, duplicates removed and vertices that appear in $L(t-1)$ or $L(t)$ discarded, just like in [14]. The remaining vertices constitute $L(t+1)$.

The main idea of the I/O-analysis is as follows: Scanning and sorting all BFS levels and merging each $\mathcal{F}_i$ into the pool takes $O(\text{sort}(E))$ I/Os in total. Each list $\mathcal{F}_i$ is copied into the pool precisely once. Since there are $O(\mu \cdot V)$ clusters this takes $O(\mu \cdot V + \text{scan}(E))$ I/Os in total. Once $\mathcal{F}_i$ is brought in the hot pool, the adjacency list of a vertex in S_i stays in the pool until the BFS-tree reaches the vertex and deletes its adjacency list from the pool. Because the shortest path between any two vertices of S_i in G is $O(1/\mu)$ the adjacency list of a vertex may stay in the pool for at most $O(1/\mu)$ levels. Thus every adjacency list is scanned $O(1/\mu)$ times. Scanning the adjacency lists of all vertices in the graph throughout the algorithm takes $O(1/\mu \cdot E/B)$ I/Os in total. The total number of I/Os for the two stages is minimized by choosing $\mu = \sqrt{E/(V \cdot B)}$. The total resulting bound for **Fast-BFS** is $O(\sqrt{V \cdot E/B} + \text{sort}(E) \cdot \log\log(V \cdot B/E))$ I/Os [11].

Fast-AP-BFS: The straightforward way of performing AP-BFS is to call **FAST-BFS** subsequently for each vertex of the graph using $O(V \cdot \sqrt{V \cdot E/B} + \text{sort}(E) \cdot \log\log(V \cdot B/E))$ I/Os in total. This bound can be improved by running **Fast-BFS** in parallel *from all source vertices in the same cluster* and considering source clusters one after the other. Hence, **Fast-AP-BFS** initially computes (once and for all) the clustering like **Fast-BFS**. Then it iterates through all clusters: given a fixed source cluster S our approach runs **Fast-BFS**(s) for all source nodes $s \in S$ in parallel using a *common* hot pool $\mathcal{H}$. Growing level $L(t)$ in all BFS-trees in turn is called a *round*. Just like in **FAST-BFS**, each file $\mathcal{F}_i$ is brought into the hot pool precisely once for each source cluster. However, once brought in the pool, the adjacency list of every vertex $v \in S_i$ will have to remain in the pool until *all* the BFS-trees of the current source cluster S reach vertex v. This takes at most $O(1/\mu)$ rounds (and simple counting methods are sufficient to remove the respective lists from $\mathcal{H}$ after that time): let the exploration of the node $v \in S_i$ with BFS-level k in the BFS-tree of $s \in S$ have caused merging $\mathcal{F}_i$ into $\mathcal{H}$. Now it is easy to see that the BFS-level of any other node $v' \in S_i$ concerning any other BFS-tree with source $s' \in S$ is at most $k + 2/\mu$. This is a direct consequence of our clustering, which ensures that the shortest-paths $\delta(s, s')$ and $\delta(v, v')$ in G are bounded by $1/\mu$ each.

*I/O-complexity of **Fast-AP-BFS**:* For each of the $O(\mu \cdot V)$ source clusters we have the following contributions: Scanning and sorting all BFS levels and merging all $\mathcal{F}_i$ into the pool still takes $O(\text{sort}(E))$ I/Os per source vertex, or $O((1/\mu) \cdot \text{sort}(E))$ I/Os in total for the source cluster. Also, the number of I/Os to load in the pool all adjacency lists is still $O(\mu \cdot V + \text{scan}(E))$. Finally, each adjacency list stays in $\mathcal{H}$ for at most $2/\mu$ rounds, which accounts for other $O(\text{scan}(E/\mu))$ I/Os. Summing over all source clusters, adding the single preprocessing phase, and recalling $\mu = \sqrt{E/(V \cdot B)}$, we see that **Fast-AP-BFS** requires $O\left(\mu \cdot V \cdot (\mu \cdot V + 1/\mu \cdot \text{sort}(E)) + \text{sort}(E) \cdot \log\log(V \cdot B/E)\right)$ I/Os $=$ $O(V \cdot \text{sort}(E))$ I/Os.

Theorem 2. *Undirected AP-BFS can be computed using $O(V \cdot sort(E))$ I/Os.*

Fast-AP-BFS can be used to compute the (unweighted) diameter in the same I/O-bound and with $O((V + E)/\mu) = O(\sqrt{V \cdot E \cdot B})$ space. For sparse graphs this is $O(V \cdot \sqrt{B})$ space. We sketch how to further reduce the space at the cost of increased I/O.

3.2 I/O-Space Tradeoffs for Unweighted Diameter Computations

In this part we concentrate on an I/O-space tradeoff for the practically most relevant case $E = O(V)$. Our diameter algorithm makes space a top priority: it requires $O(\text{sort}(k \cdot V^2 \cdot B^{1/k}))$ I/Os with $O(k \cdot V)$ space for any integer k, $3 \le k \le \log B$. In the full version of this paper we also propose another approach that prioritizes I/O and results in $O(\text{sort}(k \cdot V^2))$ I/Os using $O(k \cdot V \cdot B^{1/k})$ space.

Space Priority Approach. Let us first give the intuition for $k = 3$. We are still using a clustering but now the clusters have reduced size $O(B^{1/3})$. Instead of running BFS for *all* vertices of a source cluster S *in parallel* we just select *one* arbitrary node $s \in S$ and use **Fast-BFS** to compute BFS(s) in order to find the maximum distance from s to any other node in G. While doing so we append the cluster files $\mathcal{F}_i$ to some sequence $\mathcal{R}$ in the order they are merged into the hot pool $\mathcal{H}$. For each $\mathcal{F}_i$ we also record in which round it was added to $\mathcal{H}$. Using $\mathcal{R}$ we can subsequently execute **Fast-BFS**(s') for each vertex $s' \in S \setminus \{s\}$ separately (and update the global diameter value) without any random I/O: again, we exploit the observation from Sect. 3.1 that the BFS level of any vertex v changes by at most $O(|S|)$ when we consider different source nodes from the same source cluster S. Therefore, by simply scanning $\mathcal{R}$ we can bring all required $\mathcal{F}_i$ to the hot pool in time while not performing unstructured I/O. Hence, for each source cluster S the dominating I/O-costs are: $O(V/B^{1/3} + \text{sort}(V))$ I/Os for BFS(s) with reduced cluster size and $O(B^{1/3} \cdot (\text{scan}(V \cdot B^{1/3}) + \text{sort}(V))$ I/Os for the $O(B^{1/3})$ other BFS computations with sources in S. Summed over all $O(V/B^{1/3})$ source clusters the total I/O is bounded by $O(\text{sort}(V^2 \cdot B^{1/3}))$. As we only execute one BFS at a time the working space is bounded by $O(E)$.

In the following we extend this approach to $k-1 > 2$ levels. In a precomputing step we create a $(k-2)$-level clustering using the spanning tree/ Euler tour method of [11]: level $1 \le i \le k-2$ comprises clusters of (at most) $B^{i/k}$ nodes. Level-$(i-1)$ clusters are derived from level-i clusters by chopping them into (at most) $B^{1/k}$ pieces. Then we iterate over all level-$(k-2)$ source clusters S and for each of them proceed as follows: for an arbitrary vertex $s \in S$ we run **Fast-BFS** using the level-1 clusters and creating the sorted sequence $\mathcal{R}_{k-2}$, which contains the level-1 clusters in sorted order of visiting during **Fast-BFS**(s). Then we call the procedure $A(k-3, S)$ (see below) and after returning from $A()$ remove $\mathcal{R}_{k-2}$.

The procedure $A(\text{int } i, \text{Cluster } C)$ does the following: if $i = 0$ then we compute BFS one-by-one for all nodes v_j in C using $\mathcal{R}_{k-2}$ without random I/O and possibly update the global diameter. Otherwise ($i \geq 1$) we perform the following for each level-i subcluster C' of C: Run **Fast-BFS**(v) from a single arbitrary node v in C' using $\mathcal{R}_{i+1}$ and record from which time on the level-1 clusters are actually needed in the hot pool: this gives a more appropriate sequence $\mathcal{R}_i$ for C' which will be used in the recursive call $A(i-1, C')$. Finally, after returning from $A(i-1, C')$ the sequence $\mathcal{R}_i$ is removed.

Hence, the main difference when going from $k = 3$ to general $k > 3$ is that we apply a stepwise refinement from $\mathcal{R}_{i+1}$ to $\mathcal{R}_i$ whenever we recurse on a subcluster. For each level-$(k-2)$ source cluster we run one **Fast-BFS** computation with random I/O based on files $\mathcal{F}_j$ hosting at most $B^{1/k}$ adjacency lists each; this takes $O(V/B^{(k-2/k} \cdot (V/B^{1/k} + \text{sort}(V \cdot B^{1/k}))$ I/Os. On level-i source clusters we compute BFS without random I/O (using $\mathcal{R}_{i+1}$) but adjacency lists may stay in the hot pool for up to $\Theta(B^{(i-2)/k})$ rounds – this is due to the fact that $\mathcal{R}_{i+1}$ was recorded for a source node that may be at distance $\Theta(B^{(i-2)/k})$ from the current source node. Fortunately, on level i, there are only $O(V/B^{i/k})$ such BFS computations during the whole algorithm, each of which takes $O(\text{sort}(V \cdot B^{(i+1)/k}))$ I/Os. Hence, the whole diameter computation can be accomplished using $O(\text{sort}(k \cdot V^2 \cdot B^{1/k}))$ I/Os. The space bound follows from the observation that at all times the algorithm keeps at most $k-2$ sequences $\mathcal{R}_i$.

Theorem 3. *The diameter of unweighted undirected graphs with $E = O(V)$ edges can be found using $O(sort(k \cdot V^2 \cdot B^{1/k}))$ I/Os and $O(k \cdot V)$ space for any integer k, $3 \leq k \leq \log B$.*

4 APSP for Planar Digraphs

In this section we show how to compute APSP optimally on planar digraphs in $O(\text{scan}(N^2))$ I/Os while improving the straightforward bound of $O(\text{sort}(N^2))$ I/Os. First we give some preliminaries and a review of the planar SSSP algorithm [2, 3] then we continue with the APSP / diameter algorithm.

4.1 Preliminaries

Let $G = (V, E)$ be a planar graph with N vertices. An $f(N)$-*separator* of G is a subset S of the vertices of G of size $f(N)$ such that the removal of S disconnects G into two subgraphs G_1 and G_2, each of size at most $2N/3$. Lipton and Tarjan [9] proved that any planar graph has an $O(\sqrt{N})$-separator. Using this result recursively, Frederickson [7] showed that for any parameter $R \in [1, N]$ a planar graph can be partitioned into $\Theta(N/R)$ subgraphs (clusters) G_i of size $O(R)$ each such that there are $\Theta(N/\sqrt{R})$ separator vertices in total and each cluster is adjacent to $O(\sqrt{R})$ separator vertices (called the *boundary vertices* of G_i or simply the *boundary* ∂G_i of G_i). Denote this partitioning an *R-partition*. Denote $G_i \cup \partial G_i$ as $\overline{G_i}$. The set of separator vertices can be partitioned into maximal subsets so that the vertices in each subset are adjacent to the same set of subgraphs G_i. These sets are called the *boundary sets* of the partition.

If the graph has bounded degree, which can be ensured for planar graphs using a simple transformation [7], there exists an R-partition that, in addition to the above properties, has only $O(N/R)$ boundary sets [7]. For planar directed graphs R-partitions can be defined and computed in the same way ignoring the direction of the edges.

The main idea of the planar SSSP algorithms [2,3] is to compute a B^2-partition of G and use it to reduce the SSSP problem on G to the SSSP problem on a substitute graph G^R having as vertices the $O(N/B)$ separator vertices and $O(N)$ edges. The substitute graph is constructed using the following observation: let $\delta(s,t)$ be the shortest path from s to t in G, and let α, β be two vertices on the path such that α, β are on the boundary ∂G_i of some cluster G_i and all vertices between α and β on the path are in G_i; then the subpath of $\delta(s,t)$ from α to β is the shortest path from u to v in $\overline{G_i}$. Using this observation the substitute graph is defined so that it contains all separator vertices and the edges between them G; and, for every cluster G_i, for every pair of separator vertices α, β on ∂G_i, it contains an edge (α, β) of weight equal the the weight of the shortest path $\delta_{\overline{G_i}}(\alpha, \beta)$. It can be shown that for any $\alpha, \beta \in G^R$ the shortest path $\delta_{G^R}(\alpha, \beta) = \delta(\alpha, \beta)$. Thus G^R preserves shortest paths in G. Given G^R, SSSP in G^R can be computed exploiting that there are $O(N/B)$ vertices (and thus we can afford $O(1)$ I/Os per vertex) and that the $O(N)$ accesses to the edges can be grouped into $O(N/B)$ accesses to boundary sets.

4.2 Planar I/O-Efficient APSP

To compute APSP we use the same technique as the planar SSSP algorithm, namely computing a substitute graph G^R and computing shortest paths in G^R. The extra ingredient is the idea to compute shortest paths between the vertices of a cluster and the vertices of the graph while the cluster is in memory. The basic steps are the following:

1. Compute a B^2-partition of G into $O(N/B)$ separator vertices and $O(N/B^2)$ clusters of size $O(B^2)$ each and $O(B)$ boundary vertices.
2. Compute the substitute graph G^R defined as in Sect. 4.1.
3. For each cluster G_i
   (a) For every vertex α in ∂G_i compute SSSP(α) in G^R using the planar SSSP algorithm in [2,3].
   (b) For every vertex in $u \in \overline{G_i}$ compute SSSP(u) in G as follows:
   Load $\overline{G_i}$ in memory. For every cluster G_j load in memory $\overline{G_j}$ and the distances from ∂G_i to ∂G_j and, for any vertices $u \in \overline{G_i}$ and $v \in \overline{G_j}$, compute the shortest path from u to v as $d(u,v) = \min_{\alpha \in \partial G_i, \beta \in \partial G_j} \{\delta_{\overline{G_i}}(u,\alpha) + \delta_{G^R}(\alpha,\beta) + \delta_{\overline{G_j}}(\beta, v)\}$. If u, v are in the same cluster then $d(u,v)$ is the smaller of $d(u,v)$ computed as above and $\delta_{G_i}(u,v)$.

It can be shown that $d(u,v)$ is indeed the shortest path from u to v in G. We now discuss in more detail the steps and analyze their I/O-complexity. A B^2-partition can be computed in $O(\text{sort}(N))$ I/Os [10]. The substitute graph G^R

can be computed in $O(\mathrm{scan}(N))$ I/Os as in [2, 3]. Let V_σ be the list of vertices of G in the following order: all separator vertices are at the front of V_σ grouped by boundary set, and within the same boundary set in order of their vertex index; then follow the vertices in the clusters G_i, grouped by cluster and within the same cluster grouped by vertex index. Given that we know for each separator vertex the boundary set which contains it and for every non-separator vertex the cluster which contains it, we can produce V_σ in $O(\mathrm{sort}(N))$ I/Os. Moreover, also in sorting time, we can associate to each vertex v its position $\sigma(v)$ in V_σ.

For each cluster G_i, computing SSSP in G^R from all the vertices on the boundary of G_i uses $O(B \cdot \mathrm{sort}(N))$ I/Os and $O(B \cdot N/B) = O(N)$ space. Let L_i be the resulting list of distances $L_i = \{\delta(\alpha, \beta) | \alpha \in \partial G_i, \beta \in G^R\}$. The next phase of the algorithm will access L_i to retrieve the distances to from ∂G_i to ∂G_j for any cluster G_j. In order to make these accesses efficient we store L_i such that the distances to vertices in the same boundary set are adjacent. We can obtain this representation of L_i by sorting the distances $\delta(\alpha, \beta)$ in L_i primarily by $\sigma(\alpha)$ and secondarily by $\sigma(\beta)$.

For each cluster G_i, loading a cluster G_j in memory takes $O(|G_i|/B) = O(B)$ I/Os; loading its boundary ∂G_j takes $O(1)$ I/Os per boundary set of ∂G_j (since each boundary set has size $O(B)$). Because G has bounded degree, each boundary set can be boundary to $O(1)$ clusters. Thus for each cluster G_i, loading all clusters G_j and their boundaries takes $O(N/B)$ I/Os. Retrieving the distances from a vertex in ∂G_i to all vertices in ∂G_j from the list L_i takes $O(1)$ I/Os per boundary set of ∂G_j since distances to the same boundary set are stored consecutively in L_i. For each vertex in ∂G_i, summed over all clusters G_j, each boundary set is accessed $O(1)$ times, in total $O(N/B^2)$ I/Os. Thus for each cluster G_i, retrieving the distances from ∂G_i to ∂G_j from L_i summed over all clusters G_j takes $O(B \cdot N/B^2) = O(N/B)$ I/Os. Once clusters $\overline{G_i}$, $\overline{G_j}$ and the $|\partial G_i| \cdot |\partial G_j| = O(B^2)$ distances from ∂G_i to ∂G_j are loaded in main memory we can compute the shortest paths $d(u, v) = \min_{\alpha \in \partial G_i, \beta \in \partial G_j} \{\delta_{\overline{G_i}}(u, \alpha) + \delta_{G^R}(\alpha, \beta) + \delta_{\overline{G_j}}(\beta, v)\}$ from vertices $u \in \overline{G_i}$ to vertices $v \in \overline{G_j}$ without further I/Os using a standard internal memory APSP algorithm. There are $|\overline{G_i}| \cdot |\overline{G_j}| = O(B^4)$ distances computed from cluster $\overline{G_i}$ to a cluster $\overline{G_j}$. For each cluster G_i, summed over all clusters G_j, there are $O(B^4 \cdot N/B^2) = O(B^2 \cdot N)$ distances computed in total. Assume that as we compute the output distances from $\overline{G_i}$ to $\overline{G_j}$ we write them to a list L_{ij} with $O(\mathrm{scan}(B^4))$ I/Os; for each cluster G_i this takes $O(\mathrm{scan}(B^2 \cdot N))$ I/Os in total.

Thus, for each cluster G_i, computing the shortest paths from vertices in $\overline{G_i}$ to all vertices in G takes $O(B \cdot \mathrm{sort}(N) + N/B) = O(B \cdot \mathrm{sort}(N))$ I/Os and writing the output distances to lists L_{ij} takes $O(\mathrm{scan}(B^2 \cdot N))$ I/Os. Summed over all clusters G_i the computation of APSP takes $O(N/B^2 \cdot (B \cdot \mathrm{sort}(N) + \mathrm{scan}(B^2 \cdot N))) = O(\mathrm{sort}(N^2)/B + \mathrm{scan}(N^2)) = O(\mathrm{scan}(N^2))$ I/Os. It is interesting to note that *computing* APSP is faster than *outputting* (writing) the shortest paths: $O(\mathrm{sort}(N^2)/B) \ll \mathrm{scan}(N^2)$. We have:

Theorem 4. *APSP in a a planar digraph can be computed in $O(scan(N^2))$ I/Os.*

Computing the Diameter of a Planar Digraph: The planar APSP algorithm can be used to compute the diameter of the graph. The difference is that to compute the diameter it is not necessary to output *all* shortest paths, but only keep track of the maximum distance encountered so far. As a result the diameter algorithm does not incur the $O(\text{scan}(N^2))$ I/O-cost and $O(N^2)$ space-cost of writing all paths.

Theorem 5. *The diameter of a planar digraph can be computed with* $O(sort(N^2)/B)$ *I/Os and* $O(N)$ *space.*

References

1. A. Aggarwal and J. S. Vitter. The Input/Output complexity of sorting and related problems. *Communications of the ACM*, 31(9):1116–1127, 1988.
2. L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. In *Proc. SWAT, LNCS 1851*, pages 433–447, 2000.
3. L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, 2003.
4. A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. SODA*, pages 859–860, 2000.
5. Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. SODA*, pages 139–149, 1995.
6. E. W. Dijkstra. A note on two problems in connection with graphs. *Numerische Mathematik*, 1969.
7. G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.
8. V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. SPDP*, pages 169–177, 1996.
9. R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.*, 36:177–189, 1979.
10. A. Maheshwari and N. Zeh. I/O-optimal algorithms for planar graphs using separators. In *Proc. SODA 2002*, pages 372–381. ACM–SIAM, 2002.
11. K. Mehlhorn and U. Meyer. External-memory breadth-first search with sublinear I/O. In *Proc. ESA 2002*, volume 2461 of *LNCS*, pages 723–735. Springer, 2002.
12. U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.
13. U. Meyer and N. Zeh. I/O-efficient undirected shortest paths. In *Proc. ESA 2003*, volume 2832 of *LNCS*, pages 434–445. Springer, 2003.
14. K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. SODA*, pages 687–694, 1999.
15. N. Zeh. I/O-efficient graph algorithms. In *Proc. EFF summer school on massive data sets*, LNCS. Springer, 2004, to appear.
16. U. Zwick. Exact and approximate distances in graphs - a survey. In *Proc. ESA 2001*, number 2161 in LNCS, pages 33–48. Springer, 2001.
17. U. Zwick. All-pairs shortest paths using bridging sets and rectangular matrix multiplication. *Journal of the ACM*, 49:289–317, 2002.