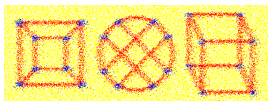
Journal of Graph Algorithms and Applications
http://jgaa.info/ vol. 15, no. 4, pp. 503–532 (2011)

I/O-Efficient Algorithms on Near-Planar Graphs

Herman Haverkort[1] *Laura Toma*[2]

[1]Dept. of Computer Science and Mathematics,
Eindhoven University of Technology, the Netherlands
[2]Dept. of Computer Science,
Bowdoin College, Maine, USA

Abstract

Obtaining I/O-efficient algorithms for basic graph problems on sparse directed graphs has been a long-standing open problem. The best known algorithms for most basic problems on such graphs still require $\Omega(V)$ I/Os in the worst case, where V is the number of vertices in the graph. Nevertheless optimal $O(sort(V))$ I/O algorithms are known for special classes of sparse graphs, like planar graphs and grid graphs. It is hard to accept that a problem becomes difficult as soon as the graph contains a few deviations from planarity. In this paper we extend the class of graphs on which basic graph problems can be solved I/O-efficiently. We discuss several ways to transform graphs that are almost planar into planar graphs (given a suitable drawing), and based on those transformations we obtain the first I/O-efficient algorithms for directed graphs that are almost planar.

Let G be a directed graph that is given as a planar subgraph (V, E) and a set of additional edges E_C. Our main result is a single-source-shortest-paths algorithm that runs in $O(E_C + sort(V + E_C))$ I/Os. When E_C is small our algorithm is a significant improvement over the best previously known algorithms, which required $\Omega(V)$ I/Os. Alternatively, when G is given with a drawing with T crossings, we can compute single-source shortest paths in $O(sort(V + T))$ I/Os. We obtain similar bounds for computing (strongly) connected components, breadth-first and depth-first traversals and topological ordering.

Submitted:	Reviewed:	Revised:	Reviewed:
May 2007	April 2008	June 2008	July 2009
Revised:	Accepted:	Final:	Published:
December 2010	July 2011	August 2011	September 2011
Article type:		Communicated by:	
Regular paper		L. Arge	

Part of this work was done while Herman Haverkort was at Karlsruhe University, supported by the European Commission, FET open project DELIS (IST-001907), and at Aarhus University, supported by the Danish National Science Research Council. Part of this work was done while Laura Toma was supported by National Science Foundation award No. 0728780.

E-mail addresses: cs.herman@haverkort.net (Herman Haverkort) ltoma@bowdoin.edu (Laura Toma)

1 Introduction

When working with massive graphs, only a fraction of the data can be held in the main memory of a computer. Thus, the transfer of blocks of data between main memory and disk, rather than the actual computation, is often the bottleneck. Therefore the running time can be improved considerably by developing *external-memory* or *I/O-efficient algorithms*—algorithms that specifically optimize the number of block transfers between main memory and disk.

As graph problems are widely encountered in practice, I/O-efficient algorithms for graph problems have been an active area of research. Even though significant progress has been made, there is still a significant gap between the lower and the upper bounds for all basic problems. Consider a directed graph (digraph) with non-negative real edge weights. A shortest path from vertex u to vertex v in G is a minimum-length path from u to v in G, where the length of a path is the sum of the weights of the edges on the path. The length of a shortest path is called the *distance* $\delta_G(u, v)$ from u to v in G. The *single-source-shortest-paths (SSSP)* problem is to find shortest paths from a source vertex s to all vertices in G. For *planar* digraphs (graphs that can be embedded in the plane such that no two edges intersect), there exist SSSP-algorithms with upper bounds on the number of block transfers that match proven lower bounds up to a constant factor. However, for general digraphs, the SSSP problem is still open, as are other basic problems such as the computation of connected components (CC), strongly-connected components (SCC), and depth- and breadth-first traversals (DFS, BFS). We note that these problems are also open on undirected graphs, although the best known upper bounds on undirected graphs have seen significant progress in the recent years.

Both from a theoretical and from a practical point of view, it is hard to accept that SSSP should become extremely difficult as soon as a graph contains a few deviations from planarity (like the one in Fig. 1). In practice, networks (e.g. transportation networks) may not be planar. However, when edges are expensive and junctions are cheap, such networks still have a strong tendency to planarity: there will be only relatively few links (e.g., motorways) that cross other edges without connecting to them. Other examples are networks in which each vertex is connected to a few nearby vertices. In such networks, there may be quite a number of crossings, but they are all very 'local'. In this paper we give a characterization of near-planarity covering a wide range of near-planar graphs, and develop the first I/O-efficient algorithms for such graphs. An extended abstract of this work appeared in [19].

I/O-Model and related work: We develop I/O-efficient algorithms using the standard two-level I/O-model of Aggarwal and Vitter [1]. The model defines two parameters: M is the number of vertices/edges that fit into internal memory, and B the number of vertices/edges that fit into a disk block, where $B \leqslant M/2$. An *Input/Output* (or *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 basic bounds in the I/O-model are

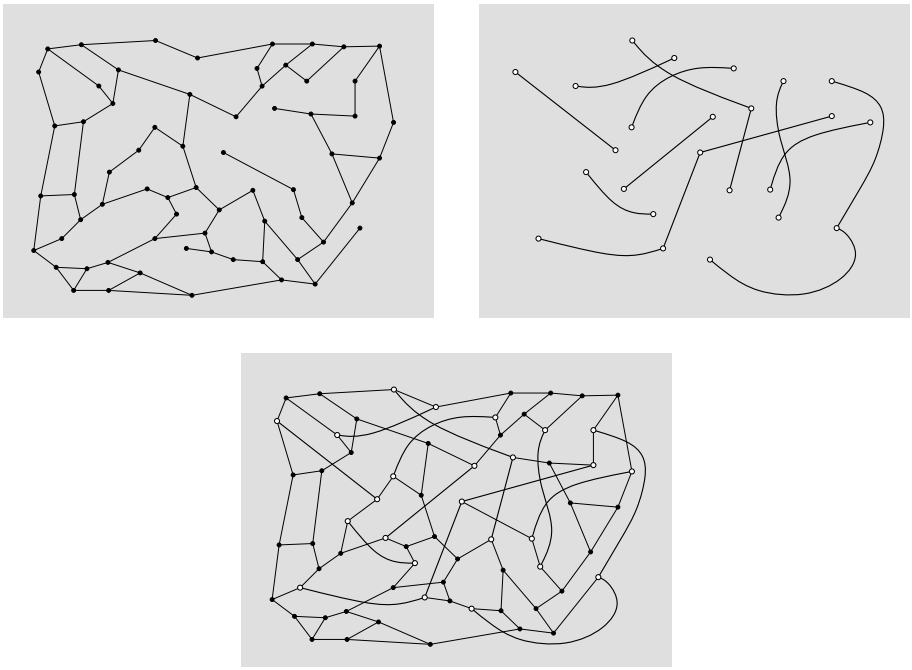

Figure 1: A "near-planar" graph (bottom) consisting of a planar graph (top left) and a set of additional edges (top right).

those for scanning and sorting. The *scanning bound*, $scan(N) = \Theta(\frac{N}{B})$ is the number of I/Os necessary to read N contiguous items from disk. The *sorting bound*, $sort(N) = \Theta(\frac{N}{B} \log_{M/B} \frac{N}{B})$ represents the number of I/Os required to sort N contiguous items on disk [1]. For all realistic values of N, B, and M, $scan(N) < sort(N) \ll N$.

I/O-efficient graph algorithms have been considered by a number of authors; for a recent review see Meyer et al. [27]. On general digraphs $G = (V, E)$ with $V > M$ the best known algorithm for SSSP, as well as for the BFS and DFS traversal problems, use $\Omega(V)$ I/Os in the worst case[1]; their complexity is $O(\min\{(V + \frac{E}{B}) \cdot \log V + sort(E), V + \frac{V}{M} \frac{E}{B}\})$ as shown by Buchsbaum et al. [11], Chiang et al. [12], and Kumar and Schwabe [22]. On sparse graphs, which have $E = O(V)$, the best known bounds are thus $O(V)$ I/Os or worse, which is no better than just running the internal-memory algorithms with an I/O-efficient priority queue in external memory. This is far from the currently best lower bound of $\Omega(\min\{V, sort(V)\} + E/B)$ I/Os [12, 28], which on sparse graphs is practically $\Omega(sort(V))$.

The search for BFS, DFS and SSSP algorithms using $O(sort(E))$ I/Os on general (sparse) graphs has led to a number of improved results for special classes of sparse graphs by Arge and Toma [6], Arge and Zeh [9], and Arge et

[1] We denote the size of a set by its name; the meaning will be clear from the context.

al. [4, 5, 7]. All these algorithms are based on the existence of small separators. For planar graphs, they exploit R-partitions, as introduced by Frederickson [16]. Given a parameter R, for any planar graph $\mathcal{K} = (V, E)$ we can find a subset V_S of $O(V/\sqrt{R})$ separator vertices, such that the removal of V_S partitions $\mathcal{K}$ into $O(V/R)$ subgraphs of size $O(R)$. Moreover, the separator vertices can be "evenly" distributed among the subgraphs, so that each subgraph is adjacent to $O(\sqrt{R})$ separator vertices, called the boundary of the subgraph.

Maheshwari and Zeh [25] showed that such a partition of a planar graph can be computed I/O-efficiently with $O(\mathit{sort}(V))$ I/Os provided that $M \geqslant \min(cB^2, cR\log^2 B)$, for a sufficiently big constant c. All I/O-efficient planar graph algorithms first compute a partition of the graph with $R = \Theta(B^2)$ and use it to reduce the original problem to a smaller problem, defined on the $\Theta(V/B)$ separator vertices. Assuming that $M = \Omega(B^2)$, each subgraph has size $O(B^2) = O(M)$ and thus fits in memory. The reductions rely crucially on the fact that there are a factor of B fewer separator vertices, and they are distributed among the subgraphs, so that each subgraph has a small boundary. Using these ideas, on planar digraphs, SSSP and BFS can be solved in $O(\mathit{sort}(V))$ I/Os as described by Arge et al. [7], and DFS in $O(\mathit{sort}(V)\log\frac{V}{M})$ I/Os as described by Arge and Zeh [9].

Now consider a digraph $G = (V, E \cup E_C)$ that consists of a planar graph $\mathcal{K} = (V, E)$ and a given set of additional edges E_C; with a slight abuse of terminology, we call $G_C = G - \mathcal{K} = (V_C, E_C)$ the *non-planar part* of G, and we call V_C and E_C the *non-planar vertices* and the *non-planar edges*, respectively (refer to Fig. 1). If E_C is empty, G is planar and SSSP can be solved with only $O(\mathit{sort}(V))$ I/Os. On the other hand if G is not planar, running any of the general I/O-efficient SSSP algorithms would result in $\Omega(E_C + V)$ I/Os. Moreover, running the planar SSSP algorithm using a separator of $\mathcal{K}$ would also result in $\Omega(E_C + V)$ I/Os (details in Sec. 2.2). If E_C is small compared to V, we show that we can do much better by refining the separator of $\mathcal{K}$ and extending the ideas behind the planar SSSP algorithm to handle the non-planar part of G.

Our results: In this paper we extend the class of directed graphs that admit I/O-efficient algorithms. We introduce a class of near-planar graphs and show how to find small separators for planar subgraphs of such graphs, that gracefully depend on the non-planarities. Using these separators, we develop the first I/O-efficient SSSP, BFS, DFS, topological sorting and (strongly) connected components algorithms for near-planar digraphs.

The main ingredient of our results is a partitioning theorem for a non-planar digraph $G = (V, E \cup E_C)$ consisting of a planar graph $\mathcal{K} = (V, E)$ and a given set of additional edges E_C; let $G_C = G - \mathcal{K} = (V_C, E_C)$ be the non-planar part of G. Starting with an R-partition of the planar part $\mathcal{K}$ of G, we show how to refine it to restrict the number of non-planar vertices $v \in V_c$ per subgraph and ensure that the number of subgraphs and separator vertices is not too large. More precisely, we show that an R-partition can be refined so that no subgraph contains more than $O(\sqrt{R})$ vertices of V_C, while adding no more

than $O(\sqrt{VV_C}/R^{1/4})$ vertices to the separator and increasing the number of subgraphs in the partition by no more than $O(V_C/\sqrt{R})$.

Using this refined R-partition and extending the ideas behind the planar SSSP algorithm, we show how to compute SSSP on G in $O(E_C + \mathit{sort}(V + E_C))$.

We generalize our result to digraphs $G = (V, E \cup E_C)$ such that $\mathcal{K} = (V, E)$ can be drawn in the plane with T crossings. If we know for each edge (u, v) of $\mathcal{K}$ which edges it crosses, and in which order these crossings occur when traversing the edge from u to v, we can compute SSSP on such a graph G in $O(E_C + \mathit{sort}(V + T + E_C))$ I/Os.

When a graph is *near-planar* in the sense that $T = O(V)$ and $E_C = O(V/B)$, these bounds reduce to $O(\mathit{sort}(V))$, whereas the best known SSSP-algorithm for general graphs require $O((V + \frac{E}{B}) \cdot \log \frac{V}{B} + \mathit{sort}(E)) \supset O(V)$ I/Os. If information about a suitable drawing (that is, the location of its vertices) of a graph is given, our results allow the computation of SSSP in $O(\mathit{sort}(E))$ I/Os on graphs with crossing number $O(E)$, on graphs that are k-embeddable in the plane for constant k, on graphs with skewness $O(E/B)$ and on graphs with splitting number $O(E/B)$. We obtain similar results for BFS, DFS, topological ordering and SCC.

Outline: The paper is organized as follows. Sec. 2 presents background on planar partitions and describes how to extend the results to graphs that are not planar. Sec. 3 describes how to use a refined, non-planar partition to compute SSSP efficiently. Sec. 4 extends our approach to other basic graph problems—BFS, DFS, topological sort, and (strongly) connected components. In Sec. 5 we explain how our technique could be used for problems on several types of graphs that are near-planar according to measures of planarity proposed in the literature. We conclude in Sec. 6 and give directions for further research.

2 Partitioning a Near-Planar Graph

In this section we give an overview of partitions of planar graphs as described by Frederickson [16] and discuss how to extend his result to obtain a partition with similar good properties on graphs that are not planar.

We assume that we work with a directed graph $G = (V, E \cup E_C)$ that consists of a planar subgraph $\mathcal{K} = (V, E)$ of constant degree and a set of edges E_C; Let $G_C = (V_C, E_C) = G - \mathcal{K}$ denote the non-planar part of G, where a vertex $v \in V_C$ if it is an endpoint of an edge in E_C. We call the edges of G_C *cross-link edges*, the vertices of G_C *cross-link vertices* and G_C the *cross-link graph*; for an example see Fig. 1. The graph G is directed, but in this section, when computing a partition of G, we ignore the direction of the edges. For this section and the next two sections we assume that the vertices and edges in G_C are known and labeled as such. We assume that $\mathcal{K}$ has degree at most three; in Sec. 5 we will discuss how to transform any planar graph $\mathcal{K} = (V, E)$ of higher degree into a planar graph $\mathcal{K}'$ with $O(V)$ vertices, $O(E)$ edges, and degree at most three, so

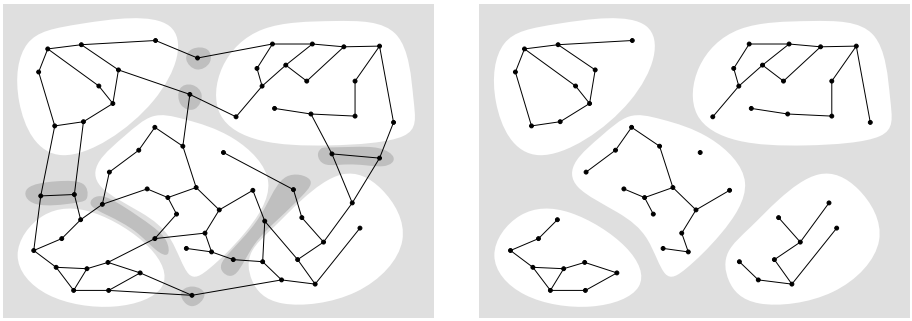

Figure 2: *Left:* Partition of a planar graph into subgraphs (white background) and separator vertices (dark background). Each dark region constitutes a boundary set. *Right:* Removing the separator vertices and their incident edges would make the graph fall apart into its subgraphs.

that our algorithms can be run on $\mathcal{K}' + E_C$ and the results can easily be mapped back to G.

2.1 Planar partition

By applying the separator theorem by Lipton and Tarjan [24] recursively, Frederickson [16] showed that any planar graph can be partitioned into subgraphs of arbitrarily small size with a small number of separator vertices. More precisely he showed the following:

Theorem 1 *(Frederickson [16]) For any planar graph $\mathcal{K} = (V, E)$, given a parameter $R \leq V$, we can find a subset $V_S \subset V$ of $O(V/\sqrt{R})$ vertices, such that the removal of V_S partitions $\mathcal{K}$ into subgraphs $\mathcal{K}_i$ such that:*

1. *there are $O(V/R)$ subgraphs (clusters);*
2. *each subgraph has size $O(R)$, and*
3. *(the vertices in) each $\mathcal{K}_i$ is (are) adjacent to $O(\sqrt{R})$ vertices of V_S.*

We call such a partition an *R-partition*—refer to Fig. 2. We use the following notation: the vertices in V_S are *separator vertices*, and each of the subgraphs is a *cluster*; the set of vertices in $\mathcal{K} - \mathcal{K}_i$ adjacent to $\mathcal{K}_i$ are the *boundary vertices* $\partial\mathcal{K}_i$ (or simply the *boundary*) of $\mathcal{K}_i$. We use $\overline{\mathcal{K}_i}$ to denote the graph consisting of $\mathcal{K}_i$, $\partial\mathcal{K}_i$ and the subset of edges of E connecting vertices in $\mathcal{K}_i \cup \partial\mathcal{K}_i$.

The set of separator vertices can be partitioned into maximal subsets so that the vertices in each subset are adjacent to precisely the same set of subgraphs (Fig. 2). These sets are the *boundary sets* of the partition. Assuming the graph has constant degree (we will discuss how to ensure this in Sec. 5), there exists an R-partition with only $O(V/R)$ boundary sets [16].

2.2 Refining a planar partition

Given a non-planar graph G, we start by computing an R-partition for its planar subgraph $\mathcal{K} = (V, E)$. Let G_i denote the subgraph induced by $\mathcal{K}_i$ in G. The separator V_S is a separator for $\mathcal{K}$ but not necessarily for G, because any subgraph in $\mathcal{K}$ may contain up to $O(R)$ cross-link vertices that are connected by cross-link edges to cross-link vertices in other subgraphs, bypassing the separator.

A straightforward way to get a separator for G would be to add all cross-link vertices V_C to V_S; however, the planar SSSP algorithm of Arge et al. [7], run on the basis of such a separator, would use $\Omega(E_C + V)$ I/Os (see Remark 3.1). This is essentially the same as runnning any of the general SSSP algorithms, and no better than just running Dijkstra's algorithm in external memory with an external priority queue.

Therefore, we need a more sophisticated method to get a separator for G. In the remainder of this section we show how to refine the partition of $\mathcal{K}$ to incorporate the cross-link vertices of G while ensuring that each subgraph contains $O(\sqrt{R})$ cross-link vertices and that the total number of separator vertices and subgraphs is not too large. The main conclusion of this section is the following.

Lemma 1 *Given a subgraph $G = (V, E)$ of a planar graph with $|\partial G| = O(\sqrt{V})$, and a weight function $w : V \to \mathbb{R}$ such that $\sum_{v \in V} w(v) = W$, we can find a subset $S \subset V$ of size $O(\sqrt{VW})$ which separates G into a set of $O(W)$ subgraphs G' with the following properties:*

- *each subgraph $G' = (V', E')$ has a total weight $\sum_{v \in V'} w(v)$ of at most 1.*
- *each subgraph $G' = (V', E')$ has a boundary $\partial G'$ with $O(\sqrt{V})$ vertices.*

Proof: The proof follows the proof of Lemmas 1 and 2 from Frederickson [16], which is based on recursive application of the separator theorem by Lipton and Tarjan [24] in two phases: first with uniform weights on the vertices, and then with weights on the separator vertices only. However, we use a non-uniform weight function in the first phase. Note that we are not interested in low-weight separators: it is the weight of the subgraphs that counts.

The first phase of the recursive procedure is as follows. When G has weight $w(G)$ at most 1, we are done. Otherwise, by applying Lipton and Tarjan's separator theorem, we find a subset S of at most $2\sqrt{2}\sqrt{V}$ vertices of V such that S separates $G - S$ into two subgraphs A and B that each have weight at most $\frac{2}{3}w(G)$. We partition the subgraphs A and B recursively. This procedure results in a number of subgraphs. By construction each subgraph $G' = (V', E')$ has weight at most 1, and it can be seen that the number of subgraphs is $O(W)$. However, the boundary $\partial G'$ of a subgraph G' may still have more than $O(\sqrt{V})$ vertices—in the second phase of the algorithm we will subdivide each subgraph further into subgraphs with boundary size $O(\sqrt{V})$. But first we show that so far, the total number of vertices in the subsets S that were selected throughout the recursive process is $O(\sqrt{VW})$. Let $s(V, W)$ be the maximum number of separator vertices that may be selected while recursively partitioning a planar

graph induced by a set of V vertices with weight W. Note that each of its subgraphs A and B has total weight at most $\frac{2}{3}W$, and at least one of them has at most $V/2$ vertices. Therefore $s(V,W)$ is bounded by the following recursive expression:

$$s(V,W) \leqslant \max_{0<\alpha\leqslant 1/2, 1/3\leqslant\beta\leqslant 2/3} c\sqrt{V} + s(\alpha V, \beta W) + s((1-\alpha)V, (1-\beta)W)$$

where $s(V,W) = 0$ if $W \leqslant 1$, and $c = 2\sqrt{2}$. Subgraphs without vertices have zero weight, thus $s(0,0) = 0$. Since no graph with 0 vertices and weight $W > 0$ exists, no separator vertices can ever be selected while recursively partitioning such a graph, so $s(0,W) = 0$ by definition.

We claim that this recurrence solves to $s(V,W) \leqslant 10c\sqrt{VW} - 6c\sqrt{V}$ for $W > 1$. We proof this by induction on W, starting with the base case $1 < W \leqslant 3$. It is easy to see that a subgraph with weight at most 3 is subdivided further at most 4 times, so that for $1 < W \leqslant 3$, we have $s(V,W) \leqslant 4c\sqrt{V} \leqslant 10c\sqrt{VW} - 6c\sqrt{V}$. It remains to prove that for $W > 3$, the following inequality holds for all $0 < \alpha \leqslant 1/2$ and $1/3 \leqslant \beta \leqslant 2/3$ (dividing out all factors $c\sqrt{V}$):

$$10\sqrt{W} - 6 \geqslant 1 + 10\left(\sqrt{\alpha\beta} + \sqrt{(1-\alpha)(1-\beta)}\right)\sqrt{W} - 6\left(\sqrt{\alpha} + \sqrt{1-\alpha}\right) \quad (1)$$

To prove this, we distinguish three cases:

- If $\alpha \leqslant 1/32$ and $\beta \leqslant 1/2$, Equation (1) follows from $\sqrt{\alpha}+\sqrt{1-\alpha} \geqslant 1$ and $\sqrt{\alpha\beta}+\sqrt{(1-\alpha)(1-\beta)}+1/(10\sqrt{W}) < \sqrt{1/32}\sqrt{1/2}+\sqrt{2/3}+1/(10\sqrt{3}) < 1$.
- If $\alpha \leqslant 1/32$ and $\beta > 1/2$, Equation (1) follows from $\sqrt{\alpha} + \sqrt{1-\alpha} \geqslant 1$ and $\sqrt{\alpha\beta} + \sqrt{(1-\alpha)(1-\beta)} + 1/(10\sqrt{W}) < \sqrt{\alpha(1-\beta)} + \sqrt{(1-\alpha)\beta} + 1/(10\sqrt{W}) < \sqrt{1/32}\sqrt{1/2} + \sqrt{2/3} + 1/(10\sqrt{3}) < 1$.
- If $\alpha > 1/32$, Equation (1) follows from $\sqrt{\alpha} + \sqrt{1-\alpha} - 1 > 1/6$ and $\sqrt{\alpha\beta} + \sqrt{(1-\alpha)(1-\beta)} \leq 1$.

Thus, it follows that the total number of vertices selected as separator vertices is at most $s(V,W) \leq 10c\sqrt{VW} - 6c\sqrt{V} = O(\sqrt{VW})$.

The second phase of our algorithm subdivides each subgraph that results from the first phase recursively until each subgraph has boundary size $O(\sqrt{V})$. As with Frederickson, the number of subdivisions required is $O(S/\sqrt{V})$ and the number of separator vertices added in each step is $O(\sqrt{V})$, thus only $O(S)$ separator vertices are added. The number of subgraphs in the second phase increases by $O(S/\sqrt{V}) = O(\sqrt{VW/V}) = O(\sqrt{W})$ and therefore stays $O(W)$. □

We note that the first phase of the above proof effects to computing a weighted version of Frederickson's partition, and could be obtained using the results of Aleksandrov et al. [2]. However, we believe that the proof above is simpler.

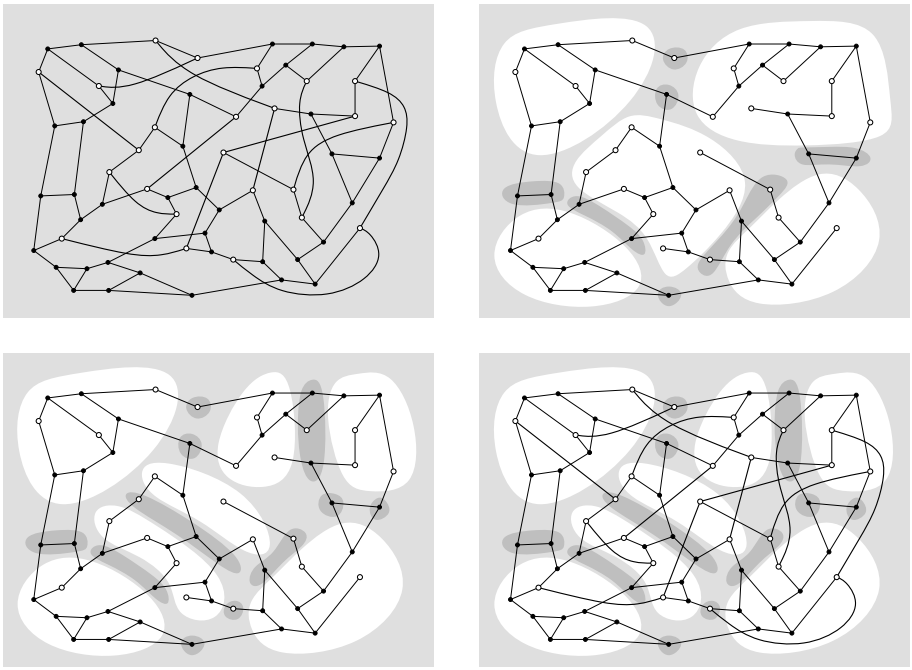

Figure 3: *Top left:* A near-planar graph that consists of a planar subgraph and a set of cross-links. Circles represent cross-link vertices. *Top right:* Partition of the planar subgraph into subgraphs (white background) and boundary sets (dark background), before refinement. If $c\sqrt{R} = 6$, the central subgraph and the top right subgraph have too many cross-link vertices. *Bottom left:* Partition after refinement. *Bottom right:* Partition after refinement with cross-links.

Our algorithm for refining the R-partition of $\mathcal{K}$ proceeds by applying Lemma 1 to each subgraph G_i that has more than $c\sqrt{R}$ cross-link vertices, for some fixed constant c. For each such subgraph we assign weight $1/(c\sqrt{R})$ to every cross-link vertex in G_i and weight 0 to every other vertex. We get that each resulting subgraph has $O(\sqrt{R})$ cross-link vertices and $O(\sqrt{R})$ vertices on its boundary—see Fig. 3. We use Lemma 1 to bound the total number of separator vertices and number of subgraphs resulting from the refinement. We have the following.

Lemma 2 *After refining the R-partition of $\mathcal{K}$, the total number of vertices in V_S is $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4})$.*

Proof: Let G_i be a subgraph in the original partition of G (before refinement) that had more than $c\sqrt{R}$ cross-link vertices. Subgraph $\overline{G_i}$ has total weight:

$$W_i = \sum_{v \in \overline{G_i}} w(v) = \frac{|\overline{G_i} \cap V_C|}{c \cdot \sqrt{R}}.$$

From Lemma 1 we get that the number of separator vertices obtained by refining a subgraph G_i is:

$$O\left(\sqrt{|\overline{G_i}| \cdot W_i}\right) = O\left(\sqrt{R \cdot \frac{|\overline{G_i} \cap V_C|}{\sqrt{R}}}\right) = O\left(R^{1/4}\sqrt{|\overline{G_i} \cap V_C|}\right).$$

Summed over all subgraphs G_i this adds $O(R^{1/4}\sum_{G_i}(|\overline{G_i} \cap V_C|)^{1/2})$ separator vertices in total. Since $2\sqrt{(a+b)/2} \geqslant \sqrt{a}+\sqrt{b}$, the worst case occurs if the cross-link vertices V_C are evenly distributed over the $O(V/R)$ subgraphs G_i, and we get:

$$R^{1/4}\sum_{G_i}\sqrt{|\overline{G_i} \cap V_C|} \leqslant R^{1/4} \cdot O(V/R) \cdot O\left(\sqrt{\frac{V_C R}{V}}\right) = O\left(\frac{V}{R^{3/4}} + \frac{\sqrt{VV_C}}{R^{1/4}}\right).$$

Adding this to the $O(V/\sqrt{R})$ vertices that were already in V_S before we started refining the partition, we get a total of $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4})$. □

Recall that a boundary set of a planar partition is a maximal set of separator vertices adjacent to the same subgraphs. In our case G is not planar and we compute a partition of $\mathcal{K} = G - E_C$; thus, a boundary set in the refined partition is a maximal set of separator vertices that are adjacent (ignoring the cross-link edges) to precisely the same set of subgraphs. For simplicity, we can think of all the cross-link vertices in a subgraph G_i which are not in V_S as an additional "boundary" set of that subgraph.

Lemma 3 *After refining the R-partition of $\mathcal{K}$, the total number of subgraphs and the number of boundary sets in the partition is $O(V/R + V_C/\sqrt{R})$.*

Proof: Let G_i be a subgraph in the original partition (before refinement) that had more than $c\sqrt{R}$ cross-link vertices. By Lemma 1, the number of subgraphs obtained by refining $\overline{G_i}$ is $O(W_i) = O(|\overline{G_i} \cap V_C|/(c \cdot \sqrt{R}))$. The total number of subgraphs that we obtain from refinement, summed over all subgraphs in the original partition, is thus $\sum_{G_i} O(|G_i \cap V_C|/\sqrt{R}) = O(V/R + V_C/\sqrt{R})$. Adding $O(V/R)$ for the subgraphs that already had at most $c\sqrt{R}$ cross-link vertices and did not need to be subdivided further, we get a total of $O(V/R + V_C/\sqrt{R})$ subgraphs in the refined partition.

To bound the number of boundary sets, we model the refined partition as a *region graph*: each subgraph represents a vertex, and two vertices are connected by an edge if the corresponding subgraphs share a boundary vertex. By the analysis of [16], from the fact that the vertices in the input graph have degree at most three, it follows that the region graph is planar and that the worst-case number of boundary sets is asymptotically the same as the number of subgraphs G_i. □

Lemma 4 *If $R \leq M/c$, for a sufficiently big constant c, then the R-partition of $\mathcal{K}$ can be refined with $O\left(sort(V + E_C)\right)$ I/Os.*

Proof: Given that we know, before refining the partition, for every non-separator vertex the subgraph that contains it, we can sort the partition into an edge-list representation. This representation consists of a a list of edges, with the out-edges of each vertex appearing consecutively in the list; thus, the out-edges of vertices in each subgraph G_i are stored consecutively. This representation can be obtained in $O(sort(V))$ I/Os. Furthermore we label the cross-link vertices in each subgraph as such, in $O(sort(V + E_C))$ I/Os.

The recursive subdivision algorithm works on one subgraph of the R-partition at a time, each of which can be processed in main memory if $R \leq M/c$. Thus, the total number of I/Os needed are the I/Os needed to load the subgraphs G_i and their boundaries, one at a time, into memory, and output the results. With the above representation each subgraph G_i and all its adjacent edges can be loaded into memory in $O(R/B)$ I/Os, or $O(V/B)$ I/Os in total. The total time to refine the partition is thus $O(sort(V + E_C))$ I/Os. □

Overall we have the following:

Theorem 2 *Let G be a graph that consists of a planar subgraph $\mathcal{K}$ of constant degree and a set of edges E_C, and let V_C be the set of vertices of V that are endpoints of edges in E_C. Given a parameter $1 \leq R < V$, there exists a set of vertices $V_S \subset V$ whose removal separates $\mathcal{K}$ into a set of subgraphs G_i with the following properties:*

1. *the total number of vertices in V_S is $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4})$;*
2. *there are $O(V/R + V_C/\sqrt{R})$ subgraphs G_i in $\mathcal{K} - V_S$;*
3. *each subgraph contains $O(R)$ vertices, is adjacent to $O(\sqrt{R})$ separator vertices and contains $O(\sqrt{R})$ cross-link vertices;*
4. *the number of boundary sets is asymptotically the same as the number of subgraphs.*

Furthermore, if $M \geqslant min(cB^2, cR\log^2 B)$ for a sufficiently big constant c, then the above set V_S can be computed with $O\left(sort(V + E_C)\right)$ I/Os.

Proof: The size of the partition follows from Lemmas 2 and 3. Provided that $M \geqslant \min(cB^2, cR\log^2 B)$ the R-partition can be computed in $O(sort(V))$ I/Os using the algorithm of Maheshwari and Zeh [25]. With one pass through the partition and E_C one can label the cross-link vertices. Thus, the partition can be refined with $O(sort(V + E_C))$ I/Os by Lemma 4. □

Representation of the refined partition. We refer to the partition of Theorem 2 as a *refined partition* of G. For the rest of the paper we assume that the refined partition of G is given in edge-list representation, as follows. Let V_σ be the list of vertices of G in the following order: all vertices in $V - (V_S \cup V_C)$ are at the front of V_σ grouped by the subgraphs G_i, and, within the same subgraph, by vertex ID; then follow all the separator and cross-link vertices $v \in V_S \cup V_C$ grouped by boundary set, and, within the same boundary set in order of their

vertex ID (remember that all cross-link vertices in a subgraph which are not in V_S are considered to be another boundary set of that subgraph). Given that we know for each vertex $v \in V_S \cup V_C$ the boundary set which contains it and for every other vertex the subgraph which contains it, we can produce V_σ in $O(sort(V))$ I/Os. Moreover, also in $O(sort(V))$ I/Os, we can associate to each vertex v its position $\sigma(v)$ in V_σ.

From the ordering σ we produce an edge-list of G by sorting the edges (u, v) by $(\sigma(u), \sigma(v))$. In this list, all the edges contained in or outgoing from a subgraph G_i are consecutive and can be accessed sequentially; similarly, all out-edges of a boundary set are consecutive. Given the refined partition and the ordering V_σ, this edge-list representation of the partition can be obtained in $O(sort(V + E_C))$ I/Os.

3 Non-planar SSSP using a Refined Partition

In this section we show how to use a refined partition of a non-planar digraph $G = \mathcal{K} \cup G_C$ to compute single-source shortest paths I/O-efficiently.

The approach follows the one used by the I/O-efficient algorithm for planar digraphs by Arge et al. [7], which is as follows. First, compute an R-partition of the planar graph $\mathcal{K}$, while ignoring the directions of edges. Given the partition, compute a substitute digraph $\mathcal{K}^R$ defined on the separator vertices. The graph $\mathcal{K}^R$ is a *reduced* version of $\mathcal{K}$ (it has fewer vertices), and it is constructed such that for any pair of vertices in $\mathcal{K}^R$, the length of the shortest path between them in $\mathcal{K}^R$ is the same as in $\mathcal{K}$. The substitute graph $\mathcal{K}^R$ is obtained by replacing each subgraph with a complete graph on its boundary vertices; the weight of each edge (u, v) between two boundary vertices u, v of a subgraph $\mathcal{K}_i$ is the distance from u to v in that subgraph. In addition, $\mathcal{K}^R$ contains the source vertex s and edges to the boundary of its subgraph, with weights defined in a similar way. The substitute graph $\mathcal{K}^R$ as computed by Arge et al. [7] has $O(V/\sqrt{R})$ vertices and $O(V)$ edges. Using $\mathcal{K}^R$ the SSSP computation can now be accomplished in two steps: (1) Compute SSSP in $\mathcal{K}^R$; by construction, we thus get the lengths of the shortest paths to separator vertices in $\mathcal{K}$; (2) Compute the shortest paths to non-separator vertices (vertices inside the subgraphs $\mathcal{K}_i$).

To extend this approach to a non-planar graph G we have to incorporate the non-planar part of G. A straightforward way to do this would be to construct an R-partition for $\mathcal{K}$ and add all cross-link vertices V_C to V_S. However, as we will explain below in Remark 3.1, the algorithm of Arge et al. [7], run on the basis of such a separator, would use $\Omega(V + E_C)$ I/Os in the worst case. Below we show how to exploit Theorem 2 to define the substitute graph G^R of a refined R-partition and get a better result. We will prove the following.

Theorem 3 *If $M = \Omega(B^2)$ the distances from a given source vertex s to all other vertices in a directed graph $G = \mathcal{K} \cup G_C$ can be computed in $O\left(E_C + sort(V + E_C)\right)$ I/Os.*

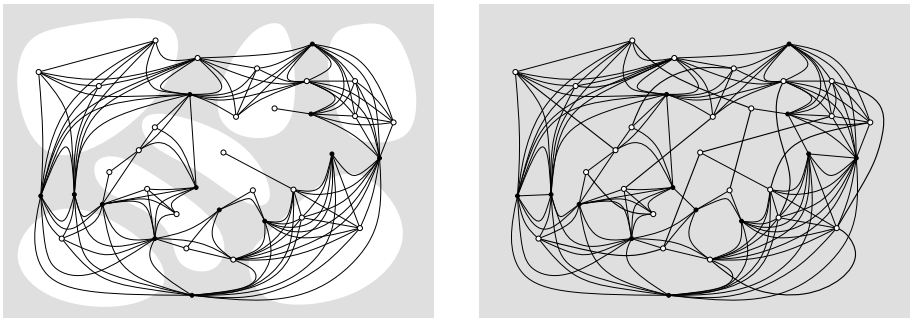

Figure 4: Construction of the substitute graph. *Left:* Within each subgraph, we draw all edges between separator vertices and cross-link vertices that have a path in $\mathcal{K}$ between them that does not go through any other boundary vertices. *Right:* The complete substitute graph consisting of edges inside subgraphs, edges between separator vertices, and cross-links, before adding the source vertex s.

3.1 The substitute graph

We construct the substitute graph G^R on the basis of a refined R-partition that divides G into subgraphs G_i, as explained in Sec. 2. Note that a shortest path between two arbitrary vertices in G enters and exits a subgraph $\overline{G_i}$ either through a boundary vertex or through a cross-link vertex. Therefore the substitute graph G^R will be defined on both the separator *and* the cross-link vertices.

- First, G^R includes the edges between the separator vertices in V_S, and the edges between the cross-link vertices V_C, i.e., the cross-link graph G_C.
- Second, it includes the union of all graphs G_i^R obtained by replacing each subgraph $\overline{G_i}$ as follows: the vertices of G_i^R are the boundary vertices ∂G_i of G_i and the cross-link vertices $V_C \cap G_i$ of G_i, and there is an edge from u to v in G_i^R if there is a path from u to v in $\overline{G_i}$ that does not pass through any vertices of ∂G_i other than u and v. The edge (u, v) has weight equal to the length of a shortest path from u to v in $\overline{G_i}$. Note that G_i^R contains edges between boundary vertices, between cross-link vertices and boundary vertices, and between cross-link vertices, see Fig. 4.
- Third, if the SSSP source vertex s is not a separator or a cross-link vertex, we add it to G^R and add edges from s to all the boundary vertices and all cross-link vertices of the subgraph G_i containing s (for which there exists a path from s in G_i); as above, the weight of an edge (s, v) is the length of a shortest path from s to v in $\overline{G_i}$.

Lemma 5 *The substitute graph G^R has $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4} + V_C)$ vertices and $O(V + V_C\sqrt{R} + E_C)$ edges.*

Proof: The number of vertices in the substitute graph is $V_S + V_C + 1$, which, by Theorem 2, is $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4} + V_C)$.

By Theorem 2, there are $O(V/R + V_C/\sqrt{R})$ subgraphs in total, each of which has $O(\sqrt{R})$ boundary vertices, $O(\sqrt{R})$ cross-link vertices, and possibly a source vertex; thus each complete graph G_i^R has $O(R)$ edges in total. In total $\cup G_i^R$ has $O(V/R + V_C/\sqrt{R}) \cdot O(R) = O(V + V_C\sqrt{R})$ edges. Add the $O(V_S+E_C) = O(V/\sqrt{R}+\sqrt{VV_C}/R^{1/4}+E_C)$ cross-link edges and edges between separator vertices in the partition, and the claimed bound follows. □

Recall that $\delta_G(u, v)$ denotes the length of the shortest path from u to v in G.

Lemma 6 *For any pair of vertices $u, v \in V_S \cup V_C \cup \{s\}$, we have $\delta_{G^R}(u, v) = \delta_G(u, v)$, that is, G^R maintains shortest paths between its vertices.*

Proof: We will first prove $\delta_{G^R}(u, v) \leqslant \delta_G(u, v)$, and then prove $\delta_{G^R}(u, v) \geqslant \delta_G(u, v)$, from which the lemma follows.

Let p be a shortest path in G from u to v. Let $u' \in V_S \cup V_C \cup \{s\}$ be a vertex on p and let v' be the next vertex from $V_S \cup V_C \cup \{s\}$ on p. Thus the part $p_{u'v'}$ of p between u' and v' is either a single edge (which is also included in G^R), or it only visits vertices within a single subgraph $\overline{G_i}$. In the latter case, $p_{u'v'}$ must be a shortest path from u' to v' in G_i (otherwise we could replace $p_{u'v'}$ with a shortest path from u' to v' in G_i and get a shorter path p, which is impossible by the definition of p). By the construction of G^R, there must be an edge (u', v') in G^R which corresponds to $p_{u'v'}$. This shows that $\delta_{G^R}(u, v) \leqslant \delta_G(u, v)$.

To prove that $\delta_{G^R}(u, v) \geqslant \delta_G(u, v)$, let p be a shortest path in G^R from u to v. Consider an edge (u', v') of p. The edge is either an edge in G with weight at least $\delta_G(u', v')$; or it is an edge in some graph G_i^R, in which case its weight is equal to $\delta_{G_i}(u', v') \geqslant \delta_G(u', v')$. Therefore the total length of p is at least the total length of some path from u to v in G. This means that $\delta_{G^R}(u, v) \geqslant \delta_G(u, v)$ and this concludes the proof. □

Lemma 7 *Given a refined partition as in Theorem 2, an edge-list representation of the substitute graph G^R can be computed in*
$O\left((V/R + V_C/\sqrt{R}) \cdot \lceil\sqrt{R}/B\rceil + \mathit{sort}(|G^R|)\right)$ *I/Os,*
provided that $R \leq M/c$, for a sufficiently large constant c.

Proof: To compute G^R we need to load, one at a time, each subgraph G_i into memory together with its boundary and cross-link vertices; compute all pairs' shortest paths (APSP) between its boundary and cross-link vertices; and write the edges and their weights to disk. Assume the representation of the partition at the end of Sec. 2. Loading the subgraphs $\overline{G_i}$ into memory takes $O(\mathit{scan}(|\overline{G_i}|))$ I/Os, plus $\lceil\sqrt{R}/B\rceil$ I/Os for each boundary set of G_i. The total number of boundary sets is $O(V/R + V_C/\sqrt{R})$ by Theorem 2, and each is loaded $O(1)$ times (because of the assumption that the degree of each vertex is at most three). Thus the total number of I/Os required to load the subgraphs $\overline{G_i}$ into memory is $O((V/R + V_C/\sqrt{R}) \cdot \lceil\sqrt{R}/B\rceil + \mathit{scan}(|G|))$. If $R \leq M/c$, each subgraph $\overline{G_i}$ fits in memory and the APSP computation on a subgraph can be done without

further I/O. Writing all the edges of G^R to disk and sorting them in the end by vertex index to obtain the edge-list of G^R requires $O(sort(|G^R|))$ I/Os. Thus, in total, the computation of G^R uses $O((V/R+V_C/\sqrt{R})\cdot\lceil\sqrt{R}/B\rceil+scan(|G|)+sort(|G^R|)) = O((V/R+V_C/\sqrt{R})\cdot\lceil\sqrt{R}/B\rceil+sort(|G^R|))$ I/Os. □

3.2 Computing $\delta_G(s,v)$ for all vertices $v \in G^R$

The distances $\delta_G(s,v) = \delta_{G^R}(s,v)$ from s to all vertices v in the substitute graph G^R can be computed by adapting Dijkstra's algorithm as discussed by Arge et al. [7]. One of the problems with SSSP in external memory is figuring out, when relaxing an edge (u,v), the current tentative distance of vertex v. This distance is necessary in order to be able to delete the vertex from the priority queue—known external-memory priority queues support N insertions, extractions and deletions in $O(sort(N))$ I/Os (which is what we can afford) only if the deletion operations are given the element to be deleted together with its current priority. To this end, in addition to using a priority queue, we maintain a list L that stores the tentative distances from s to all the vertices in G^R, that is, in $V_S \cup V_C \cup \{s\}$. When extracting a vertex from the priority queue, we retrieve the tentative distances of its out-neighbors from L. For each out-neighbor w of v, we check whether its tentative distance as stored in L is greater than $d(v)$ plus the weight of the edge $\overrightarrow{vw}$; if it is, we update the distance of w in L, delete the old entry of w from the priority queue and insert a new entry for w with the updated distance into the queue.

To analyze the I/O-complexity of the computation, we bound the number of accesses to the priority queue and to the list L. On the priority queue we perform in total $O(V(G^R)) = O(V_S + V_C)$ `ExtractMin`s, and $O(E(G^R)) = O(V+V_C\sqrt{R}+E_C)$ `Delete`s and `Insert`s; in total there are $O(|G^R|)) = O(V+V_C\sqrt{R}+E_C)$ operations. These operations can be performed in $O(sort(|G^R|)) = O(sort(V+V_C\sqrt{R}+E_C))$ I/Os using an I/O-efficient priority queue, e.g. the queue from Arge [3].

The list L is accessed $O(E(G^R)) = O(V+V_C\sqrt{R}+E_C)$ times; this is because every vertex in L is accessed once by each incoming edge in G^R. Of course, we cannot afford one I/O per edge. In order to perform the accesses to L efficiently, we store L in the following order: all vertices in V_S are at the front of L, grouped by boundary set, followed by the vertices in $V_C - V_S$, grouped by the subgraph G_i that contains them. With this order the vertices in the same boundary set, as well as cross-link vertices in the same subgraph, are consecutive in L.

Lemma 8 *The accesses to the list L can be performed in $O\left(V_S + E_C + (V/\sqrt{R}+V_C)\cdot\lceil\sqrt{R}/B\rceil\right)$ I/Os.*

Proof: The accesses to the list L are of three types: (1) $O(E_C)$ accesses through the cross-link edges of G^R; (2) $O(V_S)$ accesses through edges between separator vertices; and (3) $O(V + V_C\sqrt{R})$ accesses through the edges in the substitute graphs G_i^R. The first two types of accesses clearly take $O(V_S + E_C)$ I/Os. We now analyze the third type of accesses to L by counting the number of

accesses per boundary set (while ignoring the cross-link edges, which are counted separately in (1)).

Recall that a boundary set is a maximal set of separator vertices which are adjacent in $\mathcal{K}$ (that is, ignoring cross-link edges) to precisely the same subgraphs G_i. Every vertex $v \in V_S \cup V_C \cup \{s\}$ in G^R that is processed needs to access the tentative distances of its out-neighbors in L: that is, every separator vertex $v \in V_S$ needs to access all the boundary vertices and cross-link vertices of all subgraphs G_i adjacent to v; every vertex $v \in \{s\} \cup V_C \setminus V_S$ needs to access all the boundary vertices and all cross-link vertices in the subgraph G_i containing v. Every time a vertex in a boundary set needs to be accessed, the other vertices in the boundary set need to be accessed as well, since the vertices of a boundary set are adjacent to the same subgraphs. For simplicity, we think of all the cross-link vertices in a subgraph G_i as an additional "boundary" set of that subgraph. Overall, each boundary set of G^R is accessed once by each of the vertices on the boundaries of the subgraphs adjacent to the boundary set, and by each of the cross-link vertices in these subgraphs. By Theorem 2, each subgraph G_i has $O(\sqrt{R})$ boundary and $O(\sqrt{R})$ cross-link vertices. Thus, each boundary set is accessed $O(\sqrt{R})$ times for each adjacent subgraph.

By Theorem 2, the number of boundary sets is asymptotically the same as the number of subgraphs G_i, and each boundary set is adjacent to $O(1)$ subgraphs. We get that the total number of accesses to boundary sets is $O(\sqrt{R}) \cdot O(V/R + V_C/\sqrt{R}) = O(V/\sqrt{R} + V_C)$. Since boundary sets are stored consecutively in L (including the "boundary" set consisting of the $O(\sqrt{R})$ cross-link vertices of a subgraph), each boundary set can be accessed in $O(\lceil \sqrt{R}/B \rceil)$ I/Os.

Thus, the accesses to boundary sets use in total $O(V/\sqrt{R}+V_C)\cdot\lceil \sqrt{R}/B \rceil$ I/Os. Adding the $O(V_S)$ accesses between separator vertices and the $O(E_C)$ I/Os to L caused by the cross-link edges (type (1) and (2)), we get a total of $O(V_S + E_C + (V/\sqrt{R} + V_C) \cdot \lceil \sqrt{R}/B \rceil)$ I/Os. □

Putting the operations on the priority queue and the accesses to the list L (Lemma 8) together, we get:

Lemma 9 *The distances from s to all other vertices in G^R can be computed in $O\left(V_S + E_C + (V/\sqrt{R} + V_C) \cdot \lceil \sqrt{R}/B \rceil + sort(|G^R|)\right)$ I/Os, provided that $R \leq M/c$ for a sufficiently large constant c.*

Remark 3.1 *As mentioned above, it is straightforward to get a separator for G by simply adding all cross-link vertices V_C to a separator for $\mathcal{K}$, instead of refining the separator according to Theorem 2. However, the algorithm of Arge et al. [7], run on the basis of such a straightforward separator, would use $\Omega(E_C+V)$ I/Os. To see why this is true, consider a graph G with $\Theta(V/B)$ cross-link vertices and compute a B^2-partition for it. Imagine that the $\Theta(V/B)$ cross-link vertices are distributed in the subgraphs of the partition as follows: of the $\Theta(V/B^2)$ subgraphs, $\Theta(V/B^3)$ of them contain $\Theta(B^2)$ cross-link vertices each, and each such subgraph has $\Theta(B)$ boundary sets on its boundary. If we run the algorithm of [7] on this partition, each cross-link vertex's extraction from the priority queue*

will cause each boundary set in its subgraph to be accessed, which may cause $\Theta(B)$ *I/Os per cross-link vertex; in addition, the relaxation of each cross-link edge will cause an access. This adds up to* $\Theta(V/B \cdot B + E_C) = \Theta(V + E_C)$ *I/Os in total. This is essentially the same as runnning any of the general SSSP algorithms, and no better than just running Dijkstra's algorithm in external memory with an external priority queue.*

3.3 Computing $\delta_G(s, v)$ for all vertices $v \in V - (V_S \cup V_C)$

Since shortest paths in G^R are the same as in G (by Lemma 6), after computing SSSP on G^R we know $\delta(s, v)$ for all $v \in V_S \cup V_C$. The final step in the SSSP algorithm computes the lengths of the shortest paths to all vertices in $V - (V_S \cup V_C)$.

Consider a vertex $v \in G_i - V_C$. The shortest path $\delta(s, v)$ to v must cross into G_i either through a boundary vertex or a cross-link vertex of G_i. It is easy to see that $\delta(s, v) = \min_{u \in \partial G_i \cup (V_C \cap G_i)}\{\delta(s, u) + \delta_{G_i}(u, v)\}$. Thus, $\delta(s, v)$ can be computed locally in each subgraph G_i.

For every subgraph G_i in turn, we load G_i and its boundary and cross-link vertices—now marked with shortest path lengths—into memory, and use an internal-memory algorithm to compute $\delta(s, v)$ for every $v \in G_i - V_C$ using the above formula.

The total number of I/Os in this step is the number of I/Os required to load the subgraphs $\overline{G_i}$ into memory and to retrieve $\delta(s, u)$ for all boundary and cross-link vertices in each G_i. Assume that the shortest paths to vertices in $V_S \cup V_C$ are stored in the list L as above. Using similar arguments as in the proof of Lemma 7 and 8 we get that each boundary set in L is accessed $O(1)$ times. In total we get:

Lemma 10 *Given the distances from* s *to all vertices* G^R*, the distances from* s *to all vertices in* G *can be computed in*
$O\left(scan(|G|) + (V/R + V_C/\sqrt{R}) \cdot \lceil \sqrt{R}/B \rceil\right)$ *I/Os,*
provided that $R \leq M/c$ *for a sufficiently large constant* c*.*

3.4 Putting everything together

Putting Theorem 2, Lemma 7, Lemma 9, and Lemma 10 together, we get that the total number of I/Os needed to compute $\delta_G(s, v)$ for all vertices $v \in G$ is:

$$O\left(sort(V + E_C) + sort(|G^R|) + V_S + E_C + (V/\sqrt{R} + V_C)\lceil \sqrt{R}/B \rceil\right),$$

provided $M \geqslant \min(cB^2, cR \log^2 B)$, for a sufficiently large constant c. Substituting $|G^R| = O(V + V_C\sqrt{R} + E_C)$ (Lemma 5) and $V_S = O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4})$ (Theorem 2), we get that the total number of I/Os is:

$$O\left(sort(V + V_C\sqrt{R} + E_C) + \frac{V}{\sqrt{R}} + \frac{\sqrt{VV_C}}{R^{1/4}} + E_C + \left(\frac{V}{\sqrt{R}} + V_C\right)\left\lceil \frac{\sqrt{R}}{B} \right\rceil\right).$$

We can balance the terms in the expression above by choosing the subgraph size R appropriately. We assume that $M = \Omega(B^2)$. We have two cases:

- If $V_C < V/B$, we choose $R = B^2$. Then $\sqrt{VV_C}/R^{1/4} = O(V/B)$ and $V_C\sqrt{R} = O(V)$, and the above bound becomes $O(E_C + \mathit{sort}(V + E_C))$.
- If $V_C > V/B$, we choose $R = (V/V_C)^2 = O(B^2) = O(M)$. Then $V/\sqrt{R} = V_C = O(E_C)$, $\sqrt{VV_C}/R^{1/4} = V_C$ and $V_C\sqrt{R} = V$, and the above bound becomes $O(E_C + \mathit{sort}(V + E_C))$.

This concludes the proof of Theorem 3.

In the above algorithm we only discussed how to compute the length of the shortest paths. If we are interested in finding the actual paths, we can easily augment the algorithm to output the edges in the shortest path tree. Given a tree, Hutchinson et al. [20] showed how to store it such that for any vertex t, the shortest path between the source (root) s and t can be returned in k/B I/Os, where k is the number of vertices on the path. This data structure can be constructed from the computed distances in $O(\mathit{sort}(V))$ I/Os.

Corollary 4 *Let $G = (\mathcal{K} \cup G_C)$ be a directed non-planar graph. A data structure can be constructed in $O\left(E_C + \mathit{sort}(V + E_C)\right)$ I/Os such that the shortest path from a fixed source vertex s to a given vertex t can be found in $O(k/B)$ I/Os, where k is the number of vertices on the path.*

4 Other Non-planar Graph Problems using a Refined Partition

The refined R-partition can be exploited for other basic graph problems on non-planar graphs. The computation of a breadth-first search order is simply a special case of SSSP. Below we mention results for topological order, (strongly) connected components (SCC, CC) and depth-first search (DFS). All algorithms use the refined R-partition of G, which, according to Theorem 2, can be computed in $O(\mathit{sort}(V + E_C))$ I/Os if $M \geqslant \min(cB^2, cR\log^2 B)$.

4.1 Topological order

Let $G = \mathcal{K} \cup G_C$ be a directed acyclic non-planar graph. A refined partition of G can be used to compute a topological ordering on G by extending the algorithm for planar graphs of Arge and Toma [6]. The basic idea is that an ordering of the vertices in order of the lengths of their longest paths from a source vertex gives a valid topological order. The algorithm is similar to SSSP, except that the substitute graph is defined to encode reachability and each vertex is labeled with the length (total weight) of its longest path from a source. The size of G^R is the same as the size of the substitute graph in Sec. 3. The algorithm proceeds by sorting the vertices of G^R in topological order and then processing them in this order to compute the lengths of their longest paths in G^R. Finally each

subgraph is loaded into memory to compute the lengths of the longest paths to internal vertices. Sorting all vertices in order of these lengths will put them in topological order.

Analysis: If $R \leq M/c$, for a sufficiently large constant c, then each subgraph fits in memory. Computing the initial separator, computing G^R, and the last step can be done as in Sec. 3 with $O(sort(V + E_C) + sort(|G^R|) + (V/R + V_C/\sqrt{R})\lceil\sqrt{R}/B\rceil)$ I/Os, provided $M \geqslant \min(cB^2, cR\log^2 B)$. Computing a topological order and longest paths on G^R will cause $O(1)$ I/Os per vertex, cross-link edge and boundary set, making a total of $O(V_S + E_C + (V/\sqrt{R} + V_C) \cdot \lceil\sqrt{R}/B\rceil)$ I/Os (the proof is the same as for Lemma 8).

From Theorem 2 we know that the size of V_S is $O(V/\sqrt{R} + \sqrt{VV_C}/R^{1/4})$, and by Lemma 5, the size of G^R is $O(V + V_C\sqrt{R} + E_C)$. Substituting this in the above bounds and adding them up we get the following.

Theorem 5 *Let $G = \mathcal{K} \cup G_C$ be a non-planar directed acyclic graph. If $M = \Omega(B^2)$, a topological ordering of G can be computed with $O\left(E_C + sort(V + E_C)\right)$ I/Os.*

4.2 Strongly-Connected Components (SCC)

The same ideas can be used to compute the strongly-connected components of G based on the algorithm by Arge and Zeh [9]. The algorithm is similar to SSSP and topological order, in that it computes a partition, defines a substitute graph G^R, computes SCC on G^R and then computes SCC on the entire graph loading each subgraph into memory, one by one. The substitute graph is defined to encode reachability between vertices in the same way as it is defined above for topological order, except that it is not weighted. To compute SCC on G^R the standard algorithm explores the edges in depth-first search manner and finds out, when exploring an edge (v, w), whether w has been explored before and whether it causes any of the current components to merge. The challenge is to find out the status of w with $o(1)$ I/Os per edge. Arge and Zeh [9] showed how this can be done by keeping the active vertices in 3 stacks: one to store one vertex per each active component, one to store all vertices in order of discovery, and one to store the adjacency lists of these vertices. The key for I/O-efficiency is to store the out-edges of a vertex on the stack in order of the boundary set of the target vertex. This defines the so-called "stack segments". The stack segment on top of the (adjacency list) stack is kept up-to-date with the SCC labels of the target nodes. It is shown in [9] that maintaining this invariant causes $O(1)$ I/Os per vertex, and $O(B)$ I/Os per boundary set. For the full proof we refer the reader to [9].

The analysis can be extended to use the refined partition. The size and time to compute G^R are the same. The only difference with computing topological order is computing SCC on G^R. Using the same arguments as in [9], every vertex in G^R will cause $O(1)$ accesses to a stack segment; since a stack segment is stored consecutively, each access takes $O(\lceil\sqrt{R}/B/\rceil)$ I/Os. Every stack segment will be

accessed once by each of the vertices on the boundary of the subgraphs adjacent to the boundary set, and by each cross-link vertex. In addition, there are $O(E_C)$ acceses to stack segments caused by the cross-link edges. Skipping the details which are the same as in the proof of Lemma 9 we get the following.

Theorem 6 *Let $G = \mathcal{K} \cup G_C$ be a non-planar directed acyclic graph. If $M = \Omega(B^2)$, the strongly-connected components of G can be computed with $O\left(E_C + sort(V + E_C)\right)$ I/Os.*

If the graph G is undirected, the SCC algorithm gives an upper bound for computing the connected components of G.

Theorem 7 *Let $G = \mathcal{K} \cup G_C$ be a non-planar undirected graph. If $M = \Omega(B^2)$, the connected components of G can be computed with $O\left(E_C + sort(E)\right)$ I/Os.*

We note that it is possible to obtain this CC bound directly: either by using graph contraction; or by constructing a *sparse* substitute graph G^R that encodes all connectivity information between its boundary and cross-link vertices, and using it to compute CC on G^R and further on G.

4.3 Depth-First Search (DFS)

To compute a depth-first search order, we extend the ideas from Meyer [26], who computes DFS on a planar graph (V, E) in $O(V/\sqrt{B} + sort(V))$ I/Os. Assume we are given a refined R-partition for G, with $R \leq M/c$, for a sufficiently large constant c. We run the standard internal-memory DFS algorithm on the partition. Whenever DFS reaches a vertex inside a subgraph G_i, we load the entire subgraph together with its visited-node information into memory. While DFS stays inside G_i it will not require any further I/O. When it moves outside G_i we update the visited-node information on disk with the progress on G_i.

Analysis: As before, if $M \geqslant \min(cB^2, cR\log^2 B)$, then computing the partition and G^R takes $O(sort(V+E_C)+sort(|G^R|)+(V/R+V_C/\sqrt{R})\lceil\sqrt{R}/B\rceil)$ I/Os. As in Lemma 5, the size of G^R is $O(V + V_C\sqrt{R} + E_C)$, so the above bound becomes $O(sort(V + V_C\sqrt{R} + E_C) + V/R + V_C/\sqrt{R})$.

Every subgraph can be entered through one of its boundary vertices or cross-link vertices. Loading a subgraph takes $O(R/B)$ I/Os. Each subgraph is loaded $|\partial G_i|+|V_C \cap G_i| = O(\sqrt{R}+|V_C \cap G_i|)$ times. By Theorem 2, there are $O(V/R+V_C/\sqrt{R})$ subgraphs, so over all subgraphs the above adds up to:

$$O\left(\left(\sqrt{R}\cdot\left(\frac{V}{R}+\frac{V_C}{\sqrt{R}}\right)+\sum_i |V_C\cap G_i|\right)\left\lceil\frac{R}{B}\right\rceil\right) =$$
$$O\left(\frac{V}{\sqrt{R}}+V_C+\frac{V\sqrt{R}}{B}+\frac{V_C R}{B}\right)$$

In addition, the depth-first search accesses to the separator and cross-link vertices take $O(V_S+E_C) = O(V/\sqrt{R}+\sqrt{VV_C}/R^{1/4}+E_C)$ I/Os. Overall we get:

$$O\left(sort(V + V_C\sqrt{R} + E_C) + \frac{V}{\sqrt{R}} + \frac{V\sqrt{R}}{B} + \frac{V_C R}{B} + \frac{\sqrt{VV_C}}{R^{1/4}} + E_C\right).$$

If $V_C \leq V/\sqrt{B}$, then we choose $R = B$; in this case $V_C\sqrt{R} \leq V$ and $\sqrt{VV_C}/R^{1/4} \leq V/\sqrt{B}$ and the overall bound becomes $O(sort(V+E_C)+V/\sqrt{B}+E_C)$ I/Os. If $V_C > V/\sqrt{B}$, then we choose $R = (V/V_C)^2$; in this case $V_C\sqrt{R} = V$, $V/\sqrt{R} = V_C$, $V\sqrt{R}/B = V_C R/B < V/\sqrt{B}$ and $\sqrt{VV_C}/R^{1/4} = V_C$ and again the overall bound becomes $O(sort(V + E_C) + V/\sqrt{B} + E_C)$ I/Os.

Theorem 8 *Let $G = \mathcal{K} \cup G_C$ be a non-planar directed graph. If $M = \Omega(B^2)$, a depth-first search ordering of G can be computed with $O(E_C + V/\sqrt{B} + sort(V + E_C))$ I/Os.*

5 Planarizing graphs

As before, we denote by $\mathcal{K} = (V, E)$ the planar part of G, and by $G_C = (V_C, E_C) = G - \mathcal{K}$ the non-planar parts of G. In the previous sections of this paper we made two assumptions: firstly, that the planar part $\mathcal{K}$ and the non-planar part G_C are known, and secondly, that all vertices have degree at most three in $\mathcal{K}$. At first sight these assumptions may seem somewhat limiting, and on top of that, if $E_C = \omega(V/B)$, the performance of our algorithms suffers. In this section we discuss approaches to deal with these problems. We will discuss heuristics to find a good set E_C such that $G - E_C$ is crossing-free, we will discuss transformations that allow us to deal with $O(V)$ crossings while keeping the number of I/Os needed by our algorithms within $O(sort(V))$, and we discuss transformations that allow us to deal with vertices of degree more than three.

Transforming a non-planar graph G into a planar graph is called *planarizing*, and if we quantify the size of the transformation, it may serve as a measure of how close G is to planarity. The problem of planarizing a graph is much-studied and has obvious applications in, for example, graph drawing and in the manufacturing of VLSI circuits. Several measures of planarity have been defined in the literature, including crossing number, k-embeddability in the plane, skewness, and splitting number. All of these measures can be seen as the size of a transformation that makes the graph planar (replacing crossings by vertices, removing edges, or splitting vertices). For a survey on planarization, see Liebers [23]. The class of near-planar graphs studied in this paper includes graphs which have low crossing number or are k-embeddable for small k (we will handle such graphs with a technique that transforms crossings into vertices), and graphs that have low skewness or low splitting number (we will handle such graphs by viewing them as combinations of a planar graph and a set of cross-links, as in the previous sections).

In all cases we assume that information about a suitable drawing of the graph is given, which will guide our selection of crossings that will be transformed into vertices and edges E_C that are labelled as cross-links, so that the remaining graph $\mathcal{K} = G - E_C$ is planar. Unfortunately this assumption seems difficult to overcome. Finding an optimal set of crossings would imply determining the crossing number of G, and finding an optimal set of cross-links E_C would imply determining a maximum planar subgraph of G—both of these problems are well known to be NP-hard [18, 30]. However, in practice, graphs often come with good drawings. We can use the drawing of the graph to attempt to identify a large planar subgraph of G.

Below we will discuss the measures of planarity mentioned above and discuss how a graph that is near-planar, according to these measures, can be preprocessed so that it can be operated on by the algorithms described in the previous sections of this paper. After that we explain how to deal with vertices of degree more than three.

5.1 Graphs with low skewness

The skewness of a graph $G = (V, E)$ is the minimum size of any set of edges E_C such that $G - E_C$ is planar. When the skewness of a graph is $O(E/B)$ and the set E_C is given, our SSSP algorithm needs only $O(\mathit{sort}(E))$ I/Os, even if the edges and vertices in E_C form a graph that is far from planar (for example a clique which would have $\Theta(E^2/B^2)$ crossings when drawn in the plane). If E_C is not given, it may be difficult to find it. Finding a minimum-size set E_C corresponds to finding a maximum-size planar subgraph of G, which is NP-hard [17].

When a drawing of the graph is given, we can obtain an approximation for a minimum-size set E_C by describing the problem as a vertex cover problem. Let $G' = (V', E')$ be the *crossing graph* in which V' has a node $v(e)$ for every edge e in G, and E' has an arc $(v(e), v(f))$ for every pair of crossing edges e and f in G. A minimum-size set of cross-links E_C that leaves the remaining graph $G - E_C$ planar—that is, without crossings—now corresponds to a minimum-size set of nodes V'_C in G' such that every arc in G' is incident to at least one node in V'_C.

Finding a minimum-size vertex cover is again an NP-complete problem, even for planar graphs [21], but fortunately a factor-two approximation is sufficient for our purposes. We can find such an approximation as follows. We use the algorithm by Zeh [31] to find a maximal matching in the crossing graph, that is, a maximal set of arcs such that no two of them have a node in common. Since the maximal matching leaves no arc uncovered, while any minimum node cover must contain at least one node of every arc in the matching, we have that the nodes in the maximal matching form a factor-two approximation of a minimum-size node cover. The algorithm takes $O(\mathit{sort}(E')) = O(\mathit{sort}(T))$ I/Os, where T is the number of crossings in the input graph. We get the following:

Theorem 9 *Let $G = (V, E)$ be a graph for which we are given a drawing with T crossings. If $M = \Omega(B^2)$, then single-source shortest paths, (strongly) connected components, and a topological order (if G is acyclic) of G can be computed with*

$O\left(E_C + sort(V + T + E_C)\right)$ I/Os, where E_C is the minimum number of edges that needs to be removed from the drawing of G to make it a plane drawing. A depth-first search order can be computed with
$O\left(E_C + V/\sqrt{B} + sort(V + T + E_C)\right)$ I/Os.

5.2 Graphs with small splitting number

Another measure of planarity used in the literature is the splitting number. Splitting a vertex is the process of replacing a vertex u by two vertices u_1, u_2, whereby some of the edges incident to u will be reconnected to u_1, while the remaining edges incident to u are reconnected to u_2. The splitting number of a graph is the minimum number of splittings that is needed to make the graph planar.

Finding the splitting number of a graph is NP-hard [15]. When the splitting number of a graph is $O(E/B)$ and the necessary splittings are given, we can solve the SSSP problem on such a graph in $O(sort(E))$ I/Os, using an approach similar to that for graphs with small skewness. Instead of running the shortest paths algorithm on the original graph, we run it on the planar graph resulting from the splittings, augmented with a zero-weight[2] bidirectional cross-link (u_1, u_2) for every vertex u split into u_1 and u_2. This approach also works for connected components and depth-first search, but not for topological ordering, as it introduces bidirectional edges in the graph.

5.3 Graphs with low crossing number

The crossing number of a graph $G = (V, E)$ is the minimum number of edge crossings needed in any drawing of a given graph in a plane.

Finding the crossing number of a graph is NP-complete [17]. However, when a drawing with T crossings is given, we will show that it can be preprocessed so that our SSSP algorithm described in the previous sections uses $O(sort(V + T))$ I/Os. As before, we assume that all vertices of the graph have degree at most three.

The idea is to represent each crossing x by two *crossing vertices* c and c', which are marked as a *crossing*. Each crossed edge (u, u'), with crossings $x_1, ..., x_n$ in order going from u towards u', is replaced by edges (b_0, c_1), (c_1, c'_1), (c'_1, b_1), (b_1, c_2), $(c_2, c'_2), ..., (c_n, c'_n)$, (c'_n, b_n), where $b_0 = u$ and $b_n = u'$. The edges crossing (u, u') are transformed in the same way; thus, the edge (v, v') that crosses (u, u') in x_i is replaced by a sequence of edges that also includes (c_i, c'_i). For an illustration, see Fig. 5. The vertices b_i are inserted to make it easier to restore the original connectivity of the graph later; we call these vertices *breakers*. The resulting graph is a planar graph with $O(V)$ original vertices, $O(T)$ crossing vertices, and $O(T)$ breakers, and all vertices have degree at most three.

[2]Some SSSP algorithms deal with zero-weight edges by contracting them into single vertices. This is not the case with our SSSP algorithm which can handle zero-weight edges in the same way as positive-weight edges.

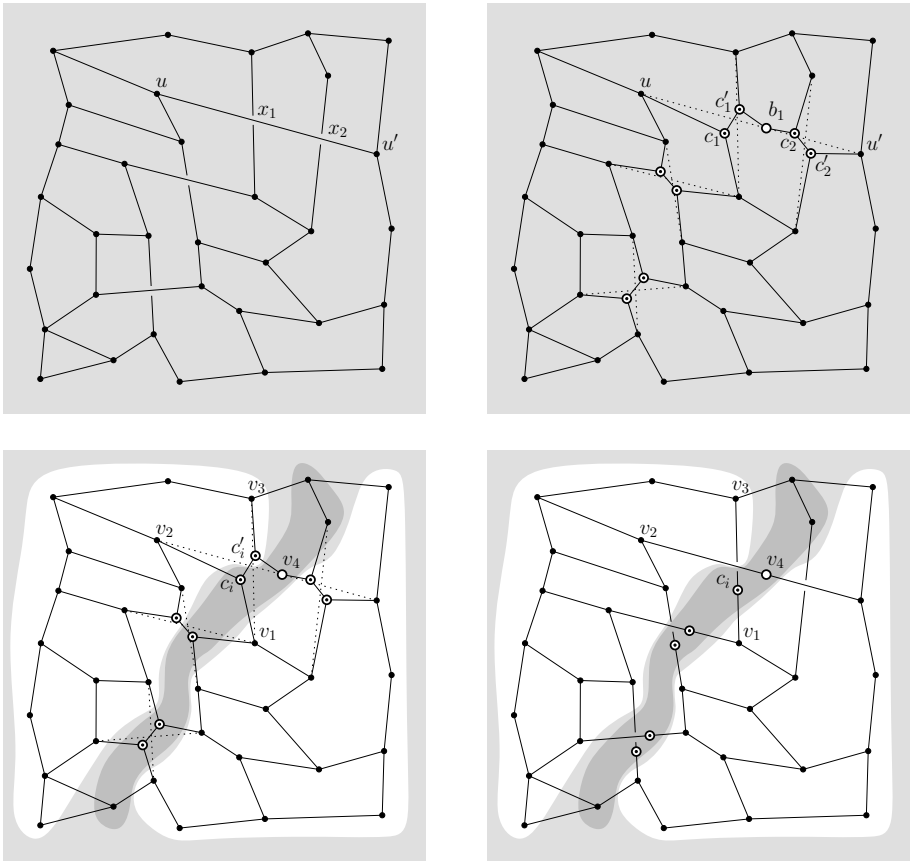

Figure 5: *Top left:* A non-planar graph. *Top right:* Introducing crossing vertices (circles with dots) and breakers (circles without dots). *Bottom left:* A partition into subgraphs (white background), with separator vertices drawn on a dark background. *Bottom right:* Final graph with the original connectivity and a good partition.

We now apply our partitioning scheme from Sec. 2 to the transformed graph. After that, we restore the original connectivity of the graph so that shortest paths, connected components etc. are the same is in the original graph, while we maintain a good partition. We do this as follows. Consider a pair of crossing vertices c_i and c'_i, and let c'_i, v_1, v_2 be the neighbours of c_i in clockwise order, and let c_i, v_3, v_4 be the neighbours of c'_i in clockwise order—see Fig. 5. The vertices $v_1, ..., v_4$ are points on two input edges: more specifically they are original input vertices or breakers. The pair c_i, c'_i models the crossing between the segments (v_1, v_3) and (v_2, v_4) of these input edges. We remove the edges $(c_i, c'_i), (c_i, v_1), (c_i, v_2), (c'_i, v_3)$ and (c'_i, v_4) and the vertices c_i and c'_i from the graph. If v_1 and v_3 lie in the same subgraph $\overline{G_i}$ (that is, a subgraph with its boundary), we put an edge (v_1, v_3) back in. If v_1 and v_3 do not lie in the same subgraph $\overline{G_i}$, we put a vertex c_i and edges (v_1, c_i) and (c_i, v_3) back in, with c_i added to the boundary set between the subgraphs that contain v_1 and v_3. Analogously, we put in an edge (v_2, v_4), or a vertex c'_i and two edges (v_2, c'_i) and (c'_i, v_4).

Note that these operations can only decrease the number of vertices and edges in any subgraph. The number of vertices in the boundary sets may increase by a constant factor, as a pair of vertices c_i, c'_i of which only one vertex was in the boundary set, may be replaced by a pair c_i, c'_i with both vertices in the boundary set.

Our algorithms for shortest paths (Sec. 3), (strongly) connected components, depth-first search, and topological sort will run correctly on the resulting graph. Note that the transformation before and after partitioning can easily be carried out in $O(sort(V + T))$ I/Os. We get the following:

Theorem 10 *Let $G = (V, E)$ be a directed graph for which we are given a drawing with T crossings. If $M = \Omega(B^2)$, then single-source shortest paths, (strongly) connected components, and a topological order (if G is acyclic) of G can be computed with $O\left(sort(V + T)\right)$ I/Os. A depth-first search order can be computed with $O\left((V + T)/\sqrt{B} + sort(V + T)\right)$ I/Os.*

Thus, if a drawing is given where the number of crossings $T = O(E)$ then SSSP, SCC and topological sorting can be solved in $O(sort(E))$ I/Os and DFS in $O(V/\sqrt{B} + sort(E))$ I/Os.

The approach described above can be compared with the approach described in Sec. 5.1 above. Both approaches have the same asymptotic dependency on T; however, the approach described in Sec. 5.1 also depends on E_C, the number of edges that needs to be removed from the drawing of G to make it a plane drawing. The approach described in Sec. 5.1 may nevertheless be advantageous if a large number of crossings is caused by a small number of edges: in that case the approach of Sec. 5.1 will work with a substitute graph with a small number of cross-link vertices, while the approach described here in Sec. 5.3 would work with a substitute graph to which breaker and separator vertices may be added for many crossings.

5.4 Graphs that are k-embeddable in the plane

A graph is k-embeddable in the plane if it can be drawn in the plane so that each edge crosses at most k other edges [29]. Since a k-embeddable graph necessarily has small crossing number, the above approach can be taken.

5.5 Combining crossings and cross-links

Above we mentioned that graphs that have low crossing number (or are k-embeddable in the plane for small k) can be handled efficiently by replacing crossings by special vertices, while graphs with small skewness or small splitting number can be handled efficiently by identifying a small number of cross-link edges. The two approaches can be combined, so that we get the following:

Theorem 11 *Let $G = \mathcal{K} \cup G_C$ be a directed graph and let a drawing for $\mathcal{K}$ with T crossing be given. If $M = \Omega(B^2)$, then single-source shortest paths, (strongly) connected components, and a topological order (if G is acyclic) of G can be computed with $O(E_C + sort(V+T+E_C))$ I/Os. A depth-first search order can can be computed with $O\left(E_C + (V+T)/\sqrt{B} + sort(V+T+E_C)\right)$ I/Os.*

Hence we can find shortest paths in $O(sort(E))$ I/Os on a graph that consists of $O(E/B)$ cross-links and a graph with crossing number $O(E)$, provided the cross-links and the intersections in the remaining graph are given. However, how to find a constant-factor approximation of a minimum-size set of cross-links such that the rest of the graph has crossing number $O(E)$, still remains as an open problem.

5.6 Vertices with degree more than three

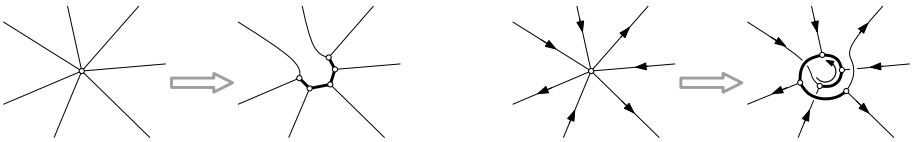

Figure 6: Transforming high-degree vertices into vertices of degree three, in the undirected case (left) and the directed case (right). The fat edges have zero weight.

Graphs with vertices of degree more than three, can be handled by first transforming these vertices into vertices of degree three. If the graph is undirected, a vertex of degree $d > 3$ can easily be transformed into $d-2$ vertices of degree 3 while keeping the graph planar, see Fig. 6 (left).

If the graph is directed, the same transformation can be used when computing shortest paths and strongly-connected components. However, this transformation does not work for topological sorting, since the added zero-weight

edges would have to be undirected (thus introducting cycles). For directed graphs another transformation can be used that does not introduce cycles, see Fig. 6 (right). This transformation may make the graph non-planar: it may introduce up to $d-2$ crossings. However, these can easily be handled with the technique described in Sec. 5.3. This increases the number of I/Os only by a constant factor.

6 Discussion

In this paper we extended the class of graphs for which efficient computations of single-source shortest paths are possible from planar graphs to several classes of near-planar graphs. Our approach yields efficient algorithms for graphs with low crossing number, low splitting number, or low skewness—provided suitable drawings are given. Our techniques can also be applied to compute (strongly) connected components, BFS orderings, DFS orderings, and topological orderings.

In theory, creating suitable drawings is difficult, since identifying a maximum planar subgraph or computing the crossing number, splitting number, or skewness of a graph are NP-complete problems [15, 18, 30]. However, in many practical applications of graph algorithms, graphs are given with a drawing or suitable drawings can be produced by heuristic methods.

Even if a good drawing is given, the method to identify cross-links in a graph of low skewness as described in Sec. 5.1 needs to know all crossings in the drawing. The crossings would need to be given or would need to be computed: in the case of a rectilinear drawing[3] we could do so with the external-memory line segment intersection algorithm by Arge et al. [8] or the randomized algorithm by Crauser et al. [13]. One could hope to find an algorithm that can find an effective set of cross-links without computing all crossings in the drawing first. It would also be interesting to find a constant-factor approximation of a minimum-size set of cross-links such that the rest of the graph has crossing number $O(E)$, so that we may have only very few cross-links and handle the remaining crossings with auxiliary vertices as described in Sec. 5.3.

Furthermore, it would be interesting to look into more measures of planarity that may be exploited, such as thickness or geometric thickness [14].

Acknowledgements

We would like to thank the anonymous reviewer who pointed out that the refined partition can be used for computing strongly-connected components.

[3] A rectilinear drawing in itself already poses limitations. The minimum number of crossings required in a *rectilinear* drawing of the graph may be arbitrarily much greater than the crossing number for drawings with curves [10].

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. Aleksandrov, H. Djidjev, H. Guo, and A. Maheshwari. Partitioning planar graphs with costs and weights. *J. Exp. Algorithmics*, 11:1.5, 2006.

[3] L. Arge. The buffer tree: A technique for designing batched external data structures. *Algorithmica*, 37(1):1–24, 2003.

[4] L. Arge, G. S. Brodal, and L. Toma. On external memory MST, SSSP and multi-way planar graph separation. *Journal of Algorithms*, 53(2):186–206, 2004.

[5] L. Arge, U. Meyer, L. Toma, and N. Zeh. On external-memory planar depth first search. *Journal of Graph Algorithms*, 7(2):105–129, 2003.

[6] L. Arge and L. Toma. Simplified external-memory algorithms for planar DAGs. In *Proc. Scandinavian Workshop on Algorithm Theory*, pages 493–503, 2004.

[7] L. Arge, L. Toma, and N. Zeh. I/O-efficient topological sorting of planar DAGs. In *Proc. ACM Symposium on Parallel Algorithms and Architectures*, pages 85–93, 2003.

[8] L. Arge, D. E. Vengroff, and J. S. Vitter. External-memory algorithms for processing line segments in geographic information systems. In *Proc. Eur. Symp. Algorithms*, volume 979 of *LNCS*, pages 295–310, 1995.

[9] L. Arge and N. Zeh. I/O-efficient strong connectivity and depth-first search for directed planar graphs. In *Proc. IEEE Symp. on Found. of Computer Sc.*, pages 261–270, 2003.

[10] D. Bienstock and N. Dean. Bounds for rectilinear crossing numbers. *Journal of Graph Theory*, 17:333–348, 1993.

[11] A. L. Buchsbaum, M. Goldwasser, S. Venkatasubramanian, and J. R. Westbrook. On external memory graph traversal. In *Proc. Symposium on Discrete Algorithms*, pages 859–860, 2000.

[12] Y.-J. Chiang, M. T. Goodrich, E. F. Grove, R. Tamassia, D. E. Vengroff, and J. S. Vitter. External-memory graph algorithms. In *Proc. Symposium on Discrete Algorithms*, pages 139–149, 1995.

[13] A. Crauser, P. Ferragina, K. Mehlhorn, U. Meyer, and E. Ramos. Randomized external-memory algorithms for some geometric problems. In *Proc. ACM Symposium on Computational Geometry*, pages 259–268, 1998.

[14] M. B. Dillencourt, D. Eppstein, and D. Hirschberg. Geometric thickness of complete graphs. *Journal of Graph Algorithms and Applications*, 4:5–17, 2000.

[15] L. Faria, C. M. H. de Figueiredo, and C. F. X. de Mendonça Neto. Splitting number is NP-complete. *Discrete Applied Mathematics*, 108:65–83, 2001.

[16] G. N. Frederickson. Fast algorithms for shortest paths in planar graphs, with applications. *SIAM Journal on Computing*, 16:1004–1022, 1987.

[17] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W H Freeman & Co, 1979.

[18] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4:312–316, 1983.

[19] H. Haverkort and L. Toma. I/O-efficient algorithms for near-planar graphs. In *Proc.Latin American Theoretical Informatics Symposium*, pages 580–591, 2006.

[20] D. Hutchinson, A. Maheshwari, and N. Zeh. An external memory data structure for shortest path queries. *Discrete Appl. Math.*, 126(1):55–82, 2003.

[21] R. M. Karp. Reducibility among combinatorial problems. In *Proc. Symposium on the Complexity of Computer Computations*, pages 85–103, 1972.

[22] V. Kumar and E. Schwabe. Improved algorithms and data structures for solving graph problems in external memory. In *Proc. IEEE Symposium on Parallel and Distributed Processing*, pages 169–177, 1996.

[23] A. Liebers. Planarizing graphs—a survey and annotated bibliography. *Journal of Graph Algorithms and Applications*, 5(1):1–74, 2001.

[24] R. J. Lipton and R. E. Tarjan. A separator theorem for planar graphs. *SIAM Journal of Applied Math.*, 36:177–189, 1979.

[25] A. Maheshwari and N. Zeh. I/O-efficient planar separators. *SIAM Journal on Computing*, 38(3):767–801, 2008.

[26] U. Meyer. External memory BFS on undirected graphs with bounded degree. In *Proc. Symposium on Discrete Algorithms*, pages 87–88, 2001.

[27] U. Meyer, P. Sanders, and J. F. Sibeyn, editors. *Algorithms for Memory Hierarchies*, volume 2625 of *LNCS*. Springer, 2003.

[28] K. Munagala and A. Ranade. I/O-complexity of graph algorithms. In *Proc. Symposium on Discrete Algorithms*, pages 687–694, 1999.

[29] J. Pach and G. Tóth. Graphs drawn with few crossings per edge. *Combinatorica*, 17:427–439, 1997.

[30] T. Watanabe, T. Ae, and A. Nakamura. On the NP-hardness of edge-deletion and -contraction problems. *Discrete Applied Mathematics*, 6:63–78, 1983.

[31] N. Zeh. *I/O-Efficient Algorithms for Shortest Path Related Problems*. PhD thesis, School of Computer Science, Carleton University, 2002.