$\mathcal{O}(\text{scan}(M))$, $\mathcal{O}(1/B)$ I/Os amortized per element. The moving of elements from v's children to v may leave v's children underfull, so that they have to be filled recursively. However, the I/Os required to do this can be charged to the elements that are moved to v's children. We observe that every level an element travels up the tree costs $\mathcal{O}(1/B)$ I/Os amortized.

Since the tournament tree is initially empty, elements that move up the tree first have to be moved down the tree by means of signals. For every level an element travels up the tree, we can hence charge the signal that moved the element in the opposite direction. This increases the amortized cost per signal by only a constant factor and hence changes the amortized cost per priority queue operation by only a constant factor. Thus, we obtain the following theorem.

Theorem 7.1. *Using an I/O-efficient tournament tree, a sequence of K* Insert, Delete, DeleteMin, *and* DecreaseKey *operations can be processed in $\mathcal{O}(\frac{K}{B}\log_2\frac{N}{B})$ I/Os.*

7.2 An I/O-Efficient Version of Dijkstra's Algorithm

Dijkstra's algorithm [14] can be made I/O-efficient using the tournament tree as the priority queue that stores the vertices of graph $G = (V, E)$ sorted according to their tentative distances from the source s. However, replacing the internal memory priority queue of choice with the tournament tree is not sufficient to immediately obtain an I/O-efficient shortest path algorithm. The problem is that Dijkstra's algorithm tests every neighbor w of the current vertex v whether it has already been finished[5] before trying to update its tentative distance using a DecreaseKey operation. If there is no way to avoid these tests, the algorithm spends one I/O per edge of G, $\mathcal{O}(|E|)$ I/Os in total.

To avoid performing these tests, the shortest path algorithm of [22] performs an Update operation for *all* neighbors of v, excluding its parent in the shortest path tree, irrespective of whether or not they are finished. While this avoids the expensive test for finished vertices, it creates the following problem: Let u be a neighbor of v that has already been finished, and let $\{u, v\}$ be the edge connecting u and v in G. Then the algorithm re-inserts u into priority queue Q with priority $\text{dist}(s, v) + \omega(\{u, v\})$, where $\omega(e)$ denotes the weight of edge e. This will ultimately cause u to be visited for a second time, which is incorrect. We call such a re-insertion of u a *spurious update*. Next we discuss a method that guarantees that the copy of u inserted by a spurious update is deleted from Q using a Delete operation before it can cause a second visit to vertex u.

The method to achieve this is based on the observation that a neighbor u of v that is finished before v performs an update of v before v is finished. By recording this update attempt of u on v in a second priority queue Q', this information can later be used to prevent the spurious update of v on u from doing any harm. In particular, when vertex u attempts to update v's tentative

[5] A vertex v is *finished* when the algorithm has determined the final distance of v from s and has inserted v's neighbors into the priority queue.

distance, vertex u is inserted into Q' with priority $\text{dist}(s,u) + \omega(\{u,v\})$. The next vertex to be visited by the algorithm is now determined from the outcome of two DeleteMin operations, one on Q and one on Q'.

Let (v, p_v) be the entry retrieved from Q, and let (w, p_w) be the entry retrieved from Q'. If $p_w < p_v$, entry (v, p_v) is re-inserted into Q, vertex w is deleted from Q by applying a Delete(w) operation to Q, and then the whole procedure is iterated. If $p_v \leq p_w$, entry (w, p_w) is re-inserted into Q', and vertex v is visited as normal. Let us show that this method achieves the desired goal.

Lemma 7.2. *A spurious update is deleted before the targeted entry can be retrieved using a* DeleteMin *operation.*

Proof. Consider a vertex v and a neighbor u of v that is finished before v, so that v performs a spurious update on u. Denote the spurious update as event A, the deletion of the re-inserted copy of u as event B, and the extraction of the re-inserted copy of u using a DeleteMin operation as event C. We have to show that event B happens after event A, but before event C can occur.

Assume that all vertices have different distances from s.[6] Under this assumption $\text{dist}(s,u) < \text{dist}(s,v)$ because u is finished before v. Moreover, $\text{dist}(s,v) \leq \text{dist}(s,u) + \omega(\{u,v\})$. The latter implies that event B happens after event A because event A happens when vertex v is retrieved from Q with priority $\text{dist}(s,v)$, and event B happens when the copy of u inserted into Q' with priority $\text{dist}(s,u) + \omega(\{u,v\})$ is retrieved from Q'. The former implies that $\text{dist}(s,u) + \omega(\{u,v\}) < \text{dist}(s,v) + \omega(\{u,v\})$, so that event B happens before event C. This proves the lemma. □

Lemma 7.2 shows that the modified version of Dijkstra's algorithm described above is correct. It remains to analyze its I/O-complexity. The algorithm spends $\mathcal{O}(|V| + \text{scan}(|E|))$ I/Os to access all adjacency lists because every adjacency list is touched once, namely when the corresponding vertex is finished. The number of priority queue operations performed by the algorithm is $\mathcal{O}(|E|)$: Every edge of G causes two insertions into priority queue Q' and two updates of priority queue Q, one each per endpoint. All other priority queue operations can be partitioned into sequences of constant length so that each sequence decreases the total number of elements stored in Q and Q' by at least one. Hence, only $\mathcal{O}(|E|)$ such sequences are executed. Using a tournament tree as priority queue Q and a buffer tree [2] as priority queue Q', the total cost of all priority queue operations is hence $\mathcal{O}\left(\frac{|E|}{B} \log_2 \frac{|E|}{B}\right)$, and we obtain the following result.

Theorem 7.3. *The single source shortest path problem on an undirected graph $G = (V, E)$ can be solved in $\mathcal{O}\left(|V| + \frac{|E|}{B} \log_2 \frac{|E|}{B}\right)$ I/Os.*

Remark. In the proof of Lemma 7.2 we assume that no two vertices have the same distance from s. It is not hard to see that the proof remains correct if no

[6] If this is not the case, the algorithm needs to be modified. See the remark at the end of this section.

two vertices with the same distance are adjacent. In order to handle adjacent vertices with the same distance, the algorithm has to be modified. In particular, all vertices with the same distance have to be processed simultaneously, similar to the simultaneous construction of levels in the BFS-algorithm from Section 6.2. The reason for this is that there seems to be no way to guarantee that for two adjacent vertices v and w at the same distance from s, non-spurious updates and deletions of spurious updates are processed in the correct order. By processing all vertices at the same distance from s at the same time, it can be guaranteed that these vertices do not update each other's distances at all. The problem with adjacent vertices that have the same distance from s has been noticed in [22]; but the proposed solution is incorrect.

8 Shortest Paths in Planar Graphs

Given that all algorithms for graph searching problems such as BFS, DFS and SSSP spend considerably more I/Os than the lower bound if the graph is sparse, a number of researchers [4, 5, 21, 24–26, 32] have tried to exploit the structure of special classes of sparse graphs in order to solve these problems I/O-efficiently on graphs in these classes. In the remainder of this course we focus on planar graphs and discuss how to solve the above three problems in $\mathcal{O}(\mathrm{sort}(N))$ I/Os. (From now on we use N to denote the size of the vertex set of the given graph G.) For the sake of simplicity we assume that an embedding of the graph is provided as part of the input. This is not a serious restriction because such an embedding can be obtained in $\mathcal{O}(\mathrm{sort}(N))$ I/Os [26, 32].

First we focus on shortest paths and BFS. More precisely, we discuss a shortest path algorithm by Arge *et al.* [4], which of course can also be used to compute a BFS-tree of a planar graph. We assume that the given graph G has degree three[7] and that a regular B^2-partition of G is given. Such a partition is defined as follows: Given a planar graph $G = (V, E)$, a *regular h-partition* of G is a pair $\mathcal{P} = (S, \{G_1, \ldots, G_k\})$, where S is a subset of the vertices of G and graphs $G_1, \ldots, G_k$ are disjoint subgraphs of $G - S$ with the following properties:

(i) $G_1 \cup \cdots \cup G_k = G - S$.
(ii) For every edge in $G - S$, the two endpoints are in the same graph G_i. (That is, each graph G_i is the union of a number of connected components of $G - S$.)
(iii) $|S| = \mathcal{O}\big(N/\sqrt{h}\big)$.
(iv) $k = \mathcal{O}(N/h)$.
(v) Every graph G_i has at most h vertices.
(vi) Every graph G_i is adjacent to at most $\sqrt{h}$ vertices in S. This subset of S is called the *boundary* ∂G_i of G_i.
(vii) Let $S_1, \ldots, S_t$ be a partition of S into subsets so that the vertices in each subset are adjacent to the same set of subgraphs G_i. Then $t = \mathcal{O}(N/h)$. Sets $S_1, \ldots, S_t$ are called the *boundary sets* of partition $\mathcal{P}$. (See Figure 8.1.)

[7] The degree of a graph is the maximum degree of its vertices.