Fortgeschrittene Datenstrukturen — Vorlesung: Distance Oracles (ctd.), String B-Trees

Schriftführer: Patrick Flick *patrick.flick@gmail.com*

2.2.2012

1 Chapter 6: Distance Oracles (ctd.)

1.1 Short Note: Dijkstra's Algorithm

In a graph $G = (V, E)$ with non-negative edge weights $\omega(e), e \in E$, Dijkstra's *single source shortest path (SSSP)* algorithm finds for a given source vertex $s \in V$ the shortests path between that vertex s and every other vertex. We denote the length of the shortest path from v to w by $\delta(v, w)$, i.e. Dijktra's algorithm computes $\delta(s, v)$ for all $v \in V$. Algorithm 1 shows the full pseudo-code of Dijkstra's algorithm. In short, Dijkstra's algorithm declares all nodes unscanned and initializes an array d with ∞ which holds an upper bound on the shortest path: $\delta(s, v) \leq d[v]$. First all edges out of s are relaxed. While there are unscanned nodes with $d[v] < \infty$, take such a node v with minimal $d[v]$ and relax all its outgoing edges and declare v scanned.

The relax procedure for node v takes all edges (v, w) and sets $d[w]$ to $d[v] + \omega(v, w)$ if this new distance is smaller then the current estimate $d[w]$.

In a graph with non-negative edge weights $\omega(v, w)$, Dijkstra's algorithm ensures that for each scanned node $d[v] = \delta(s, v)$ and thus at termination for all reachable nodes $v : d[v] = \delta(s, v)$.

The time complexity of Dijkstra's algorithm depends upon the time complexity of the `insert`, `deleteMin` and `decreaseKey` operations of the used priority queue as follows:

$$T_{Dijkstra} = O(m \cdot T_{decreaseKey}(n) + n \cdot (T_{deleteMin}(n) + T_{insert}(n))) \tag{1}$$

For a Fibonacci Heap this is:

$$T_{Dijkstra} = O(m + n \lg n) \tag{2}$$

1.2 Computing $\delta(A_i, v)$ and $p_i(v)$

For computing $\delta(A_i, v)$ and $p_i(v)$ for all $1 \leq i \leq k - 1$, Dijkstra's algorithm is used. For this a new node s is added to the graph with an edge (s, v) to any $v \in A_i$ with weight $\omega(s, v) = 0$ and thus $\delta(s, v) = 0$. Dijkstra's algorithm is started from node s, which returns the distances $\delta(A_i, v)$ in $d[v]$ for every node v in the graph. The $p_i(v)$'s can be extracted from the *parent* array also returned by Dijkstra's algorithm.

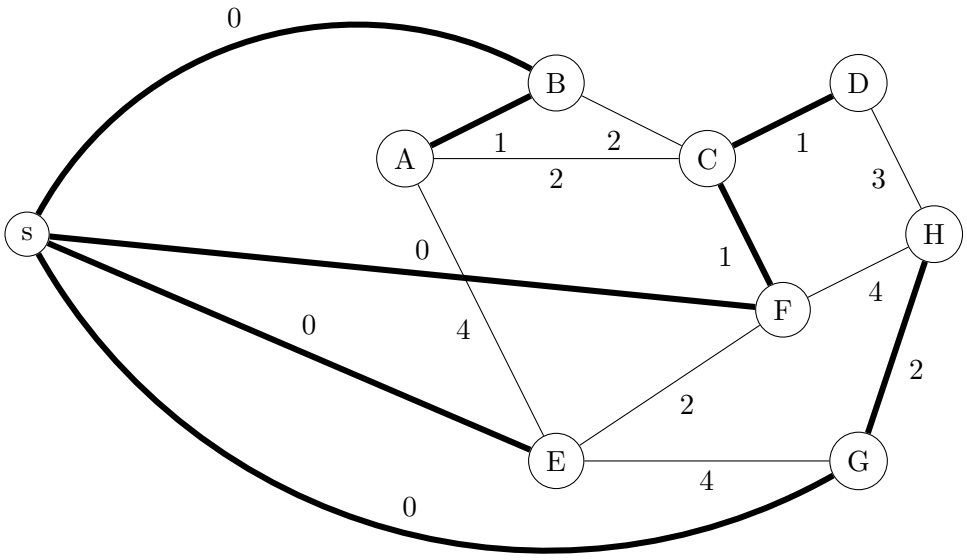
The running time is $O(n \lg n + m)$ per level and thus $O(k(n \lg n + m))$ in total.

If $\delta(v, A_i) = \delta(v, A_{i+1})$, we ensure that $p_{i+1}(v) = p_i(v)$ by settings $p_i(v)$ appropriately. This property will be needed when outputting entire paths of length at most $\texttt{dist}_k(u, v)$ instead of only the distances $\texttt{dist}_k(u, v)$.

Algorithm 1 Dijkstra's algorithm

```
function DIJKSTRA(s : NodeId) : NodeArray×NodeArray
    d = ⟨∞, ..., ∞⟩ : NodeArray of ℝ ∪ {∞}
    parent = ⟨⊥, ..., ⊥⟩ : NodeArray of NodeId
    parent[s] = s;    d[s] = 0
    Q : NodePQ;    Q.insert(s)
    while Q ≠ ∅ do
        u = Q.deleteMin
        for edge e = (u, v) ∈ E do
            // Relax
            if d[u] + ω(e) < d[v] then
                d[v] = d[u] + ω(e)
                parent[v] = u
                if v ∈ Q then Q.decreaseKey(v)
                else Q.insert(v)
    return (d,parent)
```

1.2.1 Example



For $A_i = \{B, E, F, G\}$

Dijkstra's algorithm gives:

$$\begin{aligned}
\delta(A, A_i) &= 1, \quad p_i(A) = B \\
\delta(B, A_i) &= 0, \quad p_i(B) = B \\
\delta(C, A_i) &= 1, \quad p_i(C) = F \\
\delta(D, A_i) &= 2, \quad p_i(D) = F \\
&\ldots, \ldots
\end{aligned}$$

1.3 Bunches and Clusters

To compute the *bunches*, we take a detour via *clusters*. Clusters can be thought of being the inverses of bunches, and they are formally defined as follows: :

$$\text{For all } w \in A_i \setminus A_{i+1}: \quad C(w) = \{v \in V : \delta(w, v) < \delta(A_{i+1}, v)\} \tag{3}$$

In words, the cluster of w consists of all vertices closer to w than to any element of A_{i+1}. From the definition follows that:

$$v \in C(w) \Leftrightarrow w \in B(v) \tag{4}$$

and thus

$$\sum_{w \in V} |C(w)| = \sum_{v \in V} |B(v)| = kn^{1+1/k} \tag{5}$$

1.4 Computing Clusters

Each cluster $C(w)$ is computed using a modified version of Dijkstra's *single source shortest path (SSSP)* algorithm starting from w. The modification is that an edge (u, v) is relaxed only if $d[u] + \omega(u, v) < \delta(A_{i+1}, v)$, where $d[u]$ is the upper bound on $\delta(w, u)$ maintained by the algorithm and u is the unscanned node with the smallest $d[u]$ value.

The complexity of the modified Dijkstra algorithm using Fibonacci Heaps is

$$O(|E(C(w))| + |C(w)| \lg n) \tag{6}$$

where $E(C(w))$ is the set of edges touching vertices of $C(w)$.

1.4.1 Computing Bunches from Clusters

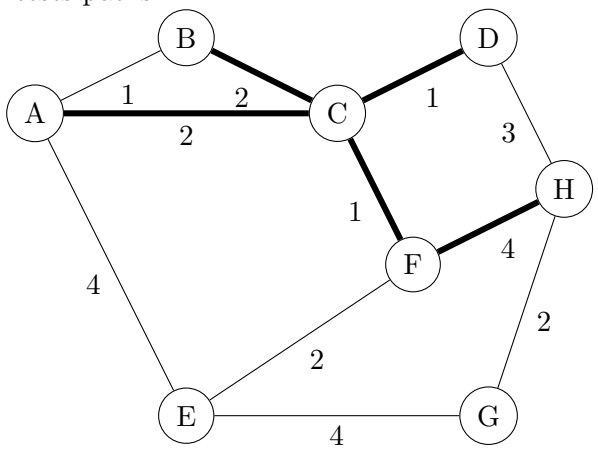
From the clusters, we can easily generate the bunches. Recall that $w \in B(v) \Leftrightarrow v \in C(w)$. This conversion can be done in

$$O(\sum |C(w)|) = O(\sum |B(v)|) \tag{7}$$

time. As before, we store the witnesses $p_i(v)$, the distances $\delta(A_i, v)$, and the hash tables for $B(v)$. Additionally, we also store the shortest path trees that span the clusters $C(w)$, for every $w \in V$. The total size is therefore the same as before.

1.4.2 Example

Given the same graph as in the previous example and $A_0 = V$, $A_1 = \{B, E, F, G\}$, $A_2 = \{E, F\}$, $A_3 = \{E\}$ and $A_4 = \emptyset$. Calculating the cluster $C(F)$ with the modified Dijkstra gives the following shortests paths:



And thus $C(F) = \{A, B, C, D, F, H\}$. The nodes E and G are never visted by the algorithm, because of the modified condition for relaxation.

The other clusters are:

$$
\begin{aligned}
i = 3: \quad & C(E) = \{A, B, C, D, E, F, G, H\} \\
i = 2: \quad & C(F) = \{A, B, C, D, F, H\} \\
i = 1: \quad & C(B) = \{A, B\} \\
& C(G) = \{G, H\} \\
i = 0: \quad & C(A) = \{A\} \\
& C(C) = \{C, D\} \\
& C(D) = \{D\} \\
& C(H) = \{H\}
\end{aligned}
$$

The bunch $B(A)$ is now easily gotten as $B(A) = \{A, B, E, F\}$ because A appears in $C(A)$, $C(B)$, $C(E)$ and $C(F)$ using the identity $v \in C(w) \Leftrightarrow w \in B(v)$. The other bunches are obtained

similarly:

$$
\begin{aligned}
B(A) &= \{A, B, E, F\} \\
B(B) &= \{A, B, E, F\} \\
B(C) &= \{C, E, F\} \\
B(D) &= \{C, D, E, F\} \\
B(E) &= \{E\} \\
B(F) &= \{E, F\} \\
B(G) &= \{E, G\} \\
B(H) &= \{E, F, G, H\}
\end{aligned}
$$

1.4.3 Preprocessing Time

The total preprocessing time is dominated by the computation of the clusters. Let $E(v)$ denote the set of edges touching the vertex v. Then the total preprocessing time is

$$\sum_{w \in V} (|E(C(w))| \cdot T_{decreaseKey} + |C(w)| \cdot (T_{deleteMin} + T_{insert})) \tag{8}$$

$$= \sum_{w \in V} (|E(C(w))| + |C(w)| \lg n) \qquad \text{(using a Fibonacci Heap)} \tag{9}$$

$$= \sum_{w \in V} |E(C(w))| + \lg n \sum_{v \in V} |B(v)|. \tag{10}$$

For the first term:

$$\sum_{w \in V} |E(C(w))| \tag{11}$$

$$\leq \sum_{\substack{w \in V \\ v \in C(w)}} |E(v)| \qquad \text{(also count intra-cluster edges)} \tag{12}$$

$$= \sum_{\substack{v \in V \\ w \in B(v)}} |E(v)| \qquad \text{(since } v \in C(w) \Leftrightarrow w \in B(v)\text{)} \tag{13}$$

$$= \sum_{v \in V} |B(v)| \cdot |E(v)| \qquad \text{(since the sum doesn't depend on } w\text{)} \tag{14}$$

$$= O(\sum_{v \in V} kn^{1/k} \cdot |E(v)|) \qquad \text{(since Exp}[|B(v)|] = kn^{1/k}\text{)} \tag{15}$$

$$= O(kmn^{1/k}) \qquad \text{(since } \sum_{v \in V} |E(v)| = 2m\text{)} \tag{16}$$

is the expected running time.

For the second term:

$$\lg n \sum_{v \in V} |B(v)| = \lg n \sum_{v \in V} kn^{1/k} = O(kn^{1+1/k} \lg n) \tag{17}$$

in expectation and the conversion of clusters to bunches takes $\sum |B(v)| = O(kmn^{1/k})$ time.

Computing the $\delta(A_i, v)$ and $p_i(v)$ takes overall $O(k(m + n \lg n))$ time. Hence the total running time is

$$O(kn^{1/k}(n \lg n + m)) \tag{18}$$

1.5 Answering Queues

Distance queries can be answered as before (see Algorithm 2). When the distance query algorithm terminates with $w \in B(v)$ so $v \in C(w)$.

Lemma 1. *For any $v \in V$ and $0 \leq i \leq k-1$: $p_i(v) \in B(v)$.*

Proof. (By induction on i). For $i = k-1$, $p_{k-1}(v) \in A_{k-1} \subseteq B(v)$, so the claim is true. Suppose therefore that $i < k-1$ and that $p_{i+1}(v) \in B(v)$. If $\delta(v, A_i) = \delta(v, A_{i+1})$ we set $p_i \leftarrow p_{i+1}$, so $p_i(v) \in B(v)$. Otherwise $\delta(v, A_i) < \delta(v, A_{i+1})$, since $\delta(v, p_i(v)) = \delta(v, A_i)$, $p_i(v)$ will be inserted into $B(v)$ by the definition of bunches. □

Now since $u, v \in C(w)$, the shortest path tree of $C(w)$ contains a path from u to v of length at most $\delta(w, u) + \delta(w, v)$. We return this path by moving in parallel from u and v towards w, stopping when an already visited node (the LCA of u and v) is reached.

Algorithm 2 Distance Query as used before

```
function DIST_k(u,v)
    w ← u
    i ← 0
    while (w ∉ B(v)) do
        i ← i + 1
        w ← p_i(v)
        (u, v) ← (v, u)
    return δ(w, u) + δ(w, v)
```

2 Chapter 7: String B-Trees

The purpose of String B-Trees is to index a large collection $D = \{S_1, \ldots, S_k\}$ of strings over Σ of total length $N = \sum_i |S_i|$ such that *substring-queries* of the form

$$\texttt{find(P,D)} : \text{return all occurrences of } P \in \Sigma^* \text{ in } D \tag{19}$$

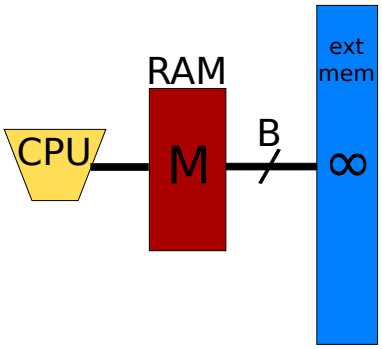
can be answered efficiently. For the RAM model, we already know that suffix trees solve this task optimally (at least for static collections of strings). In this chapter, we shall concentrate on the *external memory* model, which basically measures the performance by the amount of I/O that is generated and not by the time the CPU spends on the instance.

Further reading:

> The String B-Tree: A New Data Structure for String Search in External Memory and its Applications.
>
> *by Paolo Ferragina , Roberto Grossi*
>
> `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.57.5939`

2.1 The EM-Model

The external memory model is like the RAM model of computation, except that the fast internal memory (RAM) is limited to M words, and we have instances of size $N >> M$. Additionally there is an external memory (disks) with unlimited size. An IO-operation transfers a consecutive block of B words from the external to the internal memory, where it can be manipulated by the CPU as usual. The same amount may also be written back from RAM to disk to make room for new data.



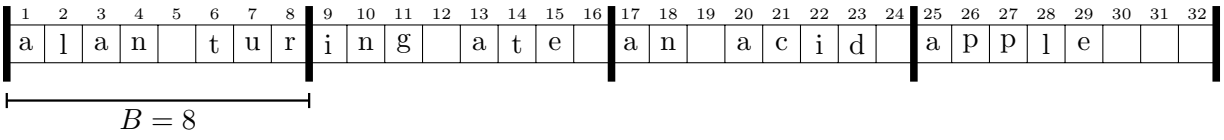
The *performance* of an EM-algorithm is the number of *disk accesses* it makes. As a simple example, consider the trivial task of reading a string of length $N > M$ that is stored consecutively on disk. This takes $O(N/B)$ IO's in the EM-model.

2.2 Basic String B-Tree Layout

Let the strings from D be stored continiously on disk. We identify a string by its starting position.

Example

$D = \{\texttt{alan,turing,ate,an,acid,apple}\}$, $B = 8$



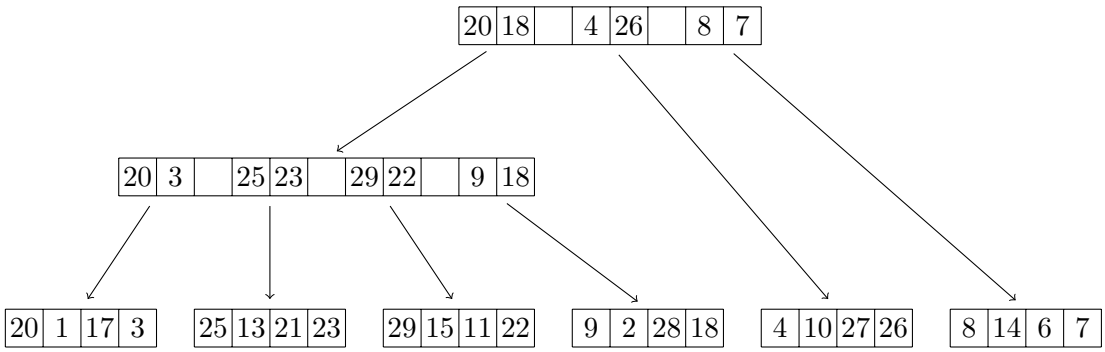
The basic idea of String B-trees is to store the sorted suffixes in a B-tree layout with branching factor $b = \Theta(B)$, where b is chosen such that the information stored at one B-tree node fits into on size B disk block.

An internal node v with children $v_1, \ldots, v_b$ stores separators $L(v_i)$ and $R(v_i)$ for every $1 \leq i \leq b$, where $L(v_i)$ (or $R(v_i)$) is the lexicographically smallest (or largest respectively) suffix stored below v_i.

The tree can be best described bottom-up: Let b lexicographically consecutive suffixes form a *leaf* (the last leaf might contain up to $2b$ strings). Then b' consecutive leaves (from "left to right") form the nodes on level 1 (where $b \leq b' \leq 2b$) and so on, until we have built the root with $\leq 2b$ children.

Example

$D = \{\texttt{alan,turing,ate,an,acid,apple}\}$, $b = 4$



Note that the definition of String B-trees leaves some flexibility. For example, we could also have included the last two leaves directly into the node containing the first four leaves, leading to a tree of depth one less than the one shown in the picture. This flexibility is necessary to allow for fast insertion and deletion of strings to/from D.

Given the basic String B-tree layout, we could now search the tree in a top-down manner, at each node deciding by a search of P within the $L(v_1), R(v_1), \ldots, L(v_k), R(v_k)$ if and where the search should be continued. The problem with this approach is that it creates a high number of IOs: if the tree has height $h = \lg_b N$ and the separators are searched in a binary manner, then this would create order of

$$h \cdot \frac{|P|}{B} \cdot \lg B = \frac{|P|}{B} \lg N \tag{20}$$

IOs, which is worse than optimal by a factor of $\lg N$. Nonetheless, two such searches will identify the interval of all occurrences of P in D, which can be reported in additional $O(\frac{occ}{B})$ IOs.