The Traveler's Problem

D.S. Hirschberg and *L.L. Larmore*
University of California, Irvine
California State University, Dominguez Hills

Abstract

The Traveler's Problem (TP) entails determining the maximum distance that can be traversed along a road, given the locations and room rates of inns along that road, and given the constraints of maximum distance per day and limited budget available for overnight stays at inns. An $O(n^{\frac{5}{3}}(\log n)^{\frac{1}{3}})$ time algorithm is presented for the TP. An $O(n^{\frac{5}{3}}(\log n)^{\frac{4}{3}})$-time algorithm for computing a minimum height B-tree for a dictionary of length n is given, by reducing the problem to $O(\log n)$ instances of the TP.

1. Introduction

In this paper, we describe and solve the *Traveler's Problem*. Suppose a traveler must journey along a road on which there are located various inns, each charging different room rates. The traveler must stop at an inn each night, and pay the cost of that inn out of his limited budget. (He need not pay for an inn at his start or destination.) There is a further restriction on how far the traveler can travel in one day. We assume that the inns are spaced closely enough that the traveler can always travel from one inn to the next inn in one day. The problem is, for each starting point, what is the farthest point down the road the traveler can reach on his limited budget? The duration (*i.e.*, number of intermediate stops) of the journey is not constrained. The version of the problem we solve allows arbitrary starting points.

Besides the intrinsic interest of the Traveler's Problem, the techniques developed in this paper may have relevance to other problems whose solution is expressed by a dynamic programming algorithm. In particular, we note that the ideas contained in this paper have immediate application to a problem on B-trees. Following McCreight [M], Diehr and Faaland

---

Authors' address: Department of Information and Computer Science, University of California, Irvine, CA 92717.

[DF] have given an $O(n^3 \log n)$ time algorithm for constructing a minimum height B-tree for a given scroll of words. We show how their minimum height B-tree problem can be reduced to $O(\log n)$ instances of the Traveler's Problem, and thus obtain an $o(n^2)$-time algorithm for finding a minimum height B-tree.

Formal definition of the Traveler's Problem. Let $w_1, \ldots w_n$ be non-negative weights. (Think of w_i as the cost of staying in the i^{th} inn.) We are also given a constant B (the budget constraint) and an array *Farthest*, where *Farthest*[i] is the greatest index j such that the traveler can travel from the i^{th} inn to the j^{th} inn in one day. We assume that *Farthest* is monotone, *i.e.*, *Farthest*[i+1] ≥ *Farthest*[i]. Without loss of generality, *Farthest*(i) > i for all $i < n$. We define a *journey* from i_0 to i_d to be a list $(i_0, \ldots i_d)$ such that $i_k \le Farthest[i_{k-1}]$. The *cost* of that journey is $\sum_{k=1}^{d-1} w_k$, and we say that the journey is *feasible* if its cost does not exceed B. The output for the problem is an array *Max_Journey* such that *Max_Journey*[i] is the greatest index j for which there is a feasible journey from i to j.

Algorithms and their complexity. The Traveler's Problem (TP) is a restricted version of the all-pairs Minimum Weight Path problem for a directed graph, and hence can be solved in $O(n^3)$ time using Floyd's algorithm. The concavity condition of [HL] is satisfied by the TP, and thus it reduces to the all-pairs Least Weight Subsequence problem. The concavity condition is that if $i_0 < i_1 < j_0 < j_1$ are vertices of a directed acyclic graph, indexed in topological order,

$$\text{wt}(i_0,j_0) + \text{wt}(i_1,j_1) \le \text{wt}(i_0,j_1) + \text{wt}(i_0,j_1)$$

where wt(i,j) is the cost of an edge from i to j. In the TP, either the two sums are equal or the right one is infinity. Thus, by the methods of [HL], the TP can be solved in $O(n^2 \log n)$ time.

More recently, using [AK], Wilber [W] has developed a linear time algorithm for the single source LWS problem, which implies an $O(n^2)$ time algorithm for the TP. In this paper, we give a subquadratic time algorithm for the TP. It remains an open question as to whether the TP be solved in linear time.

2. The Single Start Algorithm

In this section, we give a method of determining the maximum journey from any given starting point in linear time.

We describe an algorithm which computes, for any given i, the value of *Max_Journey*[i], using the methods introduced in [HL]. The time for the SSA is linear — not just $O(n)$, but actually $O(Max\_Journey[i] - i)$. This distinction will become important, as the SSA is used as a subroutine for the more sophisticated algorithms introduced later.

Let $f(i,j)$ be the minimum cost of any journey from i to j. The values of f can be computed using the following recurrence:

$f(i,j) = 0$, if $j \le Farthest(i)$

$= \min \{ f(i,k) + w_k \mid j \le Farthest(k) \}$, for $j > Farthest(i)$

Using the above recurrence, there is a simple quadratic-time dynamic programming algorithm which, for fixed i, computes the values of $f(i,j)$ in order of increasing j, starting with $j = i$. Then *Max_Journey*(i) can be defined to be the largest j for which $f(i,j) \le B$. (Since f is monotone, by Lemma 1 below, the computation may halt when $f(i,j) > B$ for any j.) Such an algorithm ignores the special properties of the problem which allow speed-up by an entire order of magnitude. Using a stack with a pointer, reminiscent of the input-restricted deque of [HL], we can construct a linear time algorithm for the SSA.

We will assume that $Farthest[i] > i$, since otherwise there are no possible non-trivial journeys. We will also assume existence of a fictitious stop at n+1.

Lemma 1. $f(i,j)$ is doubly monotone. That is, $f(i,j) \le f(i,j+1)$, and $f(i,j) \le f(i-1,j)$.

Proof. Any journey from i to j+1 gives rise to a journey from i to j of the same cost; on the last day, the traveler simply stops at j instead of j+1. More formally, if $(i_0,i_1, \ldots i_{d-1},i_d)$ is a journey, where $i_0 = i$ and $i_d = j+1$, then $(i_0,i_1, \ldots i_{d-1},i_d-1)$ is a journey of the same cost from i to j. Thus, $f(i,j) \le f(i,j+1)$. It can similarly be shown that $f(i,j) \le f(i-1,j)$.

The function g. We now introduce the notation $g(i,j) = f(i,j) + w_j$, except for the special

case $j = i$, where we let $g(i,i) = 0$. The SSA algorithm successively computes $g(i,j)$ for j in the range $i \le j \le Farthest(i)$, using the recurrence

$$g(i,i) = 0$$

$$g(i,j) = w_j + \min\{g(i,k) \mid j \le Farthest(k)\}$$

The values of $f(i,j)$ will be computed also in the process. Those values of g which could be of use in computing future values of g will be saved on a stack, which will be organized in such a way as to allow computation of each $g(i,j)$ in $O(1)$ amortized time.

Lemma 2. Any value $g(i,k)$ such that $g(i,k) > g(i,k')$ for some $k' > k$ plays no role in the solution to the SSA.

Proof. Suppose $f(i,j) = g(i,k)$ for some $j \le Farthest[k]$. By Lemma 1, $j \le Farthest[k']$. Thus $f(i,j) \le g(i,k')$, a contradiction.

We now describe the linear time algorithm for the SSA. We let S be a stack (actually, an output-restricted deque) of indices. We use *Top* and *Bottom* to refer to the extremities of S. We use the operator *Push* (*Pop*) to insert (delete) the element at the top end of S and *Drop* to delete the bottom element of S. At any time, S will contain all indices k for which $g(i,k)$ has been computed and which could possibly be used for computing $f(i,j)$ and $g(i,j)$ for any future j. That index k for which $g(i,k)$ is minimized will always be at the bottom of the stack, and hence will be the correct choice of k in the recurrence.

The invariant of the main loop of the algorithm is that $k \le j \le Farthest[k]$ for all $k \in S$, that $g(i,k) \le g(i,k')$ for any $k,k' \in S$ such that $k < k'$, and that S is maximal subject to those conditions. It follows from Lemma 1 that the choice of k such that $f(i,j) = g(i,k)$ is always $k = Bottom$.

The stack is updated in three ways. When any index (always the bottom one) is out of range, it is dropped from the stack. When a new value of g is computed, the monotonicity of the $g(i,k)$ for $k \in S$ must be preserved when j is pushed onto S. Thus, all k for which $g(i,j) < g(i,k)$ must be removed from S. By monotonicity, these will always be the topmost zero or more items. The final kind of update is pushing j after $g(i,j)$ has been computed.

The SSA

```
S ← empty stack
Push(i)
g(i,i) ← 0
j ← i
loop
        j ← j+1
        while j > Farthest[Bottom] loop
                Drop
        end while
        if S is empty or g(i,Bottom) > B  then Exit
        g(i,j) ← g(i,Bottom) + w_j
        while g(i,j) < g(i,Top) loop
                Pop
        end while
        Push(j)
end loop
Max_Journey[i] ← j–1
```

Time analysis. The number of values of j which will be pushed onto S during execution of the algorithm is $Max\_Journey[i] - i + 1$. Each item is also removed from the stack at most once, so the total time for the algorithm is linear in that quantity.

Best journeys. There can be multiple journeys of maximum length. Ties are resolved as follows among feasible journeys starting between the same two indices. First, a journey of lower cost is better. Among two journeys of the same cost, the one which is first in the lexical ordering of lists is better. (For example, (1,4,9) is better than (1,5,9), if they have the same cost.) If an array of backpointers is retained, the SSA not only computes $Max\_Journey[i]$ but also computes the best journey of maximum length from i.

The TP can be solved in $O(n^2)$ time, by executing the SSA once for each i. This method fails to make use of the overlapping work which is done by the SSA each time it is executed. In subsequent sections, we show how to do this.

3. The Bottleneck Algorithm

In this section, we describe how to compute the array *Max_Journey* if the road contains sufficiently many bottlenecks, *i.e.*, long sections of road with very few inns. (The traveler must spend at least one night somewhere in a bottleneck, and has very few choices.)

Bottlenecks. We define a *bottleneck* to be any interval $\beta = [b,c]$ such that every journey that starts before b and ends after c must contain a stop in β. We can think of a bottleneck as a stretch of road that is so long that the traveler must make at least one overnight stop while traversing that stretch. For example, $[i,Farthest(i)]$ is a bottleneck.

If β is a bottleneck, we say that i is *bottlenecked* by β if every maximum length journey from i must start, end, or contain a stop in β. It is easily seen that i is bottlenecked by $\beta = [b,c]$ if and only if $[i,Max\_Journey(i)] \cap [b,c]$ is non-empty, *i.e.*, $i \le c$ and $b \le Max\_Journey(i)$.

Back_Journey. We define $Back\_Journey(j)$ to be the smallest i for which $j \le Max\_Journey(i)$. The problem of computing *Back_Journey* is equivalent to and also symmetric to the problem of computing *Max_Journey*, hence $Back\_Journey(j)$ can be computed for a specific j by running the SSA backwards from j.

Theorem 3. If $\beta = [b,c]$ is a bottleneck of length l, and if $Back\_Journey(b) \le r \le b$ and $s = Max\_Journey(c)$, then $Max\_Journey(i)$ can be computed for all $i \in [r,c]$ in $O((s-r)l)$ time.

Proof. Use the SSA to compute $f(k,j)$ for all $k \in \beta$, taking $O((s-b)l)$ time. Then use the SSA (backwards from k) to compute $f(i,k)$ for all $i \in [r,c]$, taking $O((c-r)l)$ time.

For $k \in \beta$, define $Max\_Journey\_Stop_k(i)$ to be the largest j for which there is a feasible journey from i to j which has a stop at k. Thus,

$Max\_Journey\_Stop_k(i) = k$, if $f(i,k) \le B < f(i,k) + w_k$

= the maximum j such that $f(i,k) + w_k + f(k,j) \le B$, if $f(i,k) + w_k \le B$

= undefined, otherwise

Since f is doubly monotone (Lemma 1), $Max\_Journey\_Stop_k$ can be computed for all $i \in [Back\_Journey(k),k]$ by the following loop.

$j \leftarrow Max\_Journey(k)$ (Use the SSA algorithm)
$Max\_Journey\_Stop_k(k) \leftarrow j$
$i \leftarrow k–1$
while $i \geq r$ and $f(i,k) \leq B$ **loop**
    while $j > k$ and $f(i,k)+w_k+f(k,j) > B$ **loop**
        $j \leftarrow j – 1$
    end loop
    $Max\_Journey\_Stop_k(i) \leftarrow j$
end loop

It takes $O(s–r)$ time to execute that loop, for each k, hence $O((s–r)l)$ time to compute $Max\_Journey\_Stop_k(i)$ for all k and all i. Finally, $Max\_Journey(i)$ is the maximum of $Max\_Journey\_Stop_k(i)$ over all k for which it is defined. Those maxima can all be computed in $O((s–r)l)$ time. This concludes the proof of Theorem 3.

The Bottleneck Algorithm. We define an *l-bottleneck* to be a bottleneck of length at most l. We say that an index i is *l-bottlenecked* if there exists some l-bottleneck β such that i is bottlenecked by β.

Theorem 4. $Max\_Journey(i)$ can be computed for all indices which are l-bottlenecked in $O(nl)$ time.

We first construct a list $\beta_1, \ldots \beta_M$ of *selected l-bottlenecks* such that every l-bottlenecked index is bottlenecked by some selected l-bottleneck. That is, the list must satisfy the following condition: If j is an l-bottlenecked index then there is a selected bottleneck β such that $\beta \cap [j, Max\_Journey(j)]$ is non-empty.

The set of all l-bottlenecks can be easily identified by scanning the array *Farthest*. In particular, b is the left endpoint of an l-bottleneck if and only if $b = 1$ or $Farthest[b–1] \leq b+l–1$. Let B be the set of all left endpoints of l-bottlenecks. The selected l-bottlenecks are chosen by the following loop:

$b_1 \leftarrow 1$
$m \leftarrow 1$
loop

$c_m \leftarrow \min\{n, b_m + l - 1\}$
$\beta_m \leftarrow [b_m, c_m]$
if $c_m = n$ **then** Exit
$m \leftarrow m+1$
$j \leftarrow Max\_Journey(c_{m-1})$ (Use the SSA)
if $\mathrm{B} \cap [b_{m-1}+1, j]$ not empty **then**
    $b_m \leftarrow$ maximum element of $\mathrm{B} \cap [b_{m-1}+1, j]$
else
    $b_m \leftarrow$ minimum element of $\mathrm{B} \cap [j+1, n]$
end if
end loop
$M \leftarrow m$

Lemma 5. Every index i which is l-bottlenecked is bottlenecked by some β_m.

Proof. If $i \in \beta_m$, then it is bottlenecked by β_m. Assume that $c_{m-1} < i < b_m$. If $Max\_Journey(i) \geq b_m$, then i is bottlenecked by β_m. Otherwise, by monotonicity of *Max_Journey*, $Max\_Journey(c_{m-1}) \leq Max\_Journey(i) < b_m$. By the definition of b_m, this implies that $\mathrm{B} \cap [i, Max\_Journey(i)] = \varnothing$, *i.e.*, i is not l-bottlenecked.

For any m, define $r_m = \max\{Back\_Journey(b_m), c_{m-1}\}$ and $s_m = Max\_Journey(c_m)$.

Lemma 6. $\sum_{m=1}^{M} (s_m - r_m) \leq 3n$.

Proof. For any $1 < m < M$, $b_{m+1} > s_{m-1} = Max\_Journey(c_{m-1})$, since otherwise it would contradict the definition of b_m as the maximum element in $\mathrm{B} \cap [c_{m-1}, s_{m-1}]$. Thus the sums of the even and odd terms of $\sum_{k=1}^{M} (s_m - c_m)$ are both at most n. The sum $\sum_{m=1}^{M} (c_m - r_m)$ clearly does not exceed n. The result follows.

We now return to the proof of Theorem 4. We first need to show that it takes linear time to select all β_m. All parts of the loop which does this task are clearly linear, except for the part which computes j. But the SSA can be executed in linear time, and thus all executions of this step take $O(n)$ time by Lemma 6. By Theorem 3, $Max\_Journey(i)$ can be computed for all indices in the range $[r_m, c_m]$ in $O(s_m - r_m)$ time. By Lemma 5, every l-bottlenecked index lies in $[r_m, c_m]$ for some m and so, by Lemma 6, $Max\_Journey(i)$ can be computed for all i which are l-

bottlenecked in $O(nl)$ time.

4. The Limited Duration Algorithm

In this section, we show how *Max_Journey* can be computed faster if it is known in advance that the maximum length journey takes at most d stops for some fixed d.

We introduce the LDA algorithm, which has one parameter d. LDA(d) computes *Max_Journey*(i) for all indices i, under the assumption that there is a maximum length journey from i of *duration* no greater than d. (We define the *duration* of a journey to be the number of steps, this is actually the usual definition of *length* of a path.)

We define $Max\_Journey_d(i)$ to be the maximum value of j for which there is a feasible journey from i to j of length *exactly* d. We also define $f_d(i,j)$ to be the minimum cost of any journey from i to j of duration d. If no such journey exists, $f_d(i,j) = \infty$. Finally, we define $g_0(i,i) = 0$ and, for $i < j$, $g_d(i,j) = f_d(i,j) + w_j$.

The values of f_d and g_d may be jointly defined by the following recurrence:

$g_0(i,i) = 0$

$g_0(i,j) = \infty$, for $j \neq i$

$f_d(i,j) = \min\{\ g_{d-1}(i,k) \mid j \leq Farthest(k)\ \}$, for $d > 0$

$g_d(i,j) = f_d(i,j) + w_k$, for $d > 0$

It is now possible to compute all $f_d(i,j)$ in $O(n^2 d)$ time, and hence to compute $Max\_Journey_d(i)$ for all i. Unfortunately, that time exceeds that of the simple quadratic time algorithm for *Max_Journey* obtained by applying the SSA for each index. We will show how to use the techniques of [HL] and the previous sections to compute the entire array $Max\_Journey_d$ in $O(nd\log n)$ time.

Definitions. Fix d and i. We define $J_{i,d}$ to be the best feasible journey of duration d, starting from i. For $e \in [0,d]$, define $x_{i,d,e}$ to be the e^{th} vertex of $J_{i,d}$.

Lemma 7. If it is known that $a_e \le x_{i,d,e} \le b_e$ for all $e \in [1,d]$ then $J_{i,d}$ can be computed in $O(\sum_{e=1}^{d}(b_e - a_e + 1))$ time.

Proof. We reduce the problem to an instance of the TP. Let X be the rectangle $[0,d]\times[0,n]$, which we consider to be linearly ordered using row-major order. We define a new instance of the TP as follows:

1. The points are the elements of X in row-major order.
2. $w(\langle r,i\rangle) = w_i + D$, where $D > B$ is constant.
3. The budget is $B + (d-1)D$.
4. $Farthest(\langle r,i\rangle) = \langle r+1, Farthest(i)\rangle$

We define a journey in this new problem to be *regular* if it starts in the 0^{th} row, has duration d, and advances by one row each step. There is a one-to-one correspondence between feasible journeys of duration d in the original problem and feasible regular journeys in the new problem: the journey $(i_0, \ldots i)$ corresponds to $(\langle 0,i_0\rangle, \ldots \langle d,i_d\rangle)$. Furthermore, any maximal length journey starting from the 0^{th} row must be regular. Thus finding the optimal journey from $\langle 0,i\rangle$ in the new problem is equivalent to finding the optimal journey of duration d from i in the original problem. Using the SSA, this can be done in $O(nd)$ time, which is not as good as what the theorem claims. However, by the hypothesis of the lemma, we need only consider a certain subset of X, namely

$$Y = \{\langle r,j\rangle \mid r \in [0,d],\ j \in [a_r,b_r]\}$$

Applying the SSA to this subset of X yields the claimed bound.

Lemma 8. For any fixed d, and for $i < j$, $x_{i,d,e} \le x_{j,d,e}$ for all $e \in [0,d]$.

Proof. Let J' be the journey of length d whose e^{th} stop, for all e, is $\min\{ x_{i,d,e}, x_{j,d,e} \}$ and let J'' be the journey whose e^{th} stop is $\max\{ x_{i,d,e}, x_{j,d,e} \}$. Now J' is a journey from i to $Max\_Journey_d(i)$ and J'' is a journey from j to $Max\_Journey_d(j)$. The combined cost of those two journeys equals the combined cost of $J_{i,d}$ and $J_{j,d}$, which are both of minimum cost. Either J' is better than $J_{i,d}$ or J'' is better than $J_{j,d}$ or $J' = J_{i,d}$ and $J'' = J_{j,d}$. The first two of those three choices are impossible, since $J_{i,d}$ and $J_{j,d}$ are defined to be best. The statement of the lemma follows immediately.

Lemma 9. Suppose that we are given a subinterval $I \subseteq [0,n]$, and intervals $I_r = [a_r,b_r] \subseteq [0,n]$ for all $r \in [1,d]$ such that, for each $i \in I$, $x_{i,d,r} \in I_r$. Then $J_{i,d}$ can be computed for all $i \in I$ in $O((1+\log|I|) \sum_{r=1}^{d} |I_r|)$ time.

Proof. By induction on $|I|$. If $|I| = 1$, the result follows directly from Lemma 7. Suppose then that $|I| > 1$. Write $|I|$ as the disjoint union of equal (or nearly so) subintervals $|I'|$ and $|I''|$, and let j be the endpoint of I' that divides I' from I''. By Lemma 7, we can compute $J_{j,d}$ in $O(\sum_{r=1}^{d} |I_r|)$ time. Let $c_r = x_{j,d,r}$, let $I'_r = [a_r,c_r]$ and $I''_r = [c_r,b_r]$. By Lemma 8, $x_{i,d,r} \in J_r$ for all $i \in J$ and $x_{i,d,r} \in J'_r$ for all $i \in J'$. By the inductive hypothesis, $J_{i,d}$ can be computed for all $i \in I'$ in $O((1+\log|I'|) \sum_{r=1}^{d} |I_r|)$ time and for all for all $i \in I''$ in $O((1+\log|I''|) \sum_{r=1}^{d} |I''_r|)$ time. The claimed bound follows.

Corollary 10. The values of $Max\_Journey_d(i)$ can be computed for all i in $O(nd\log n)$ time.

Corollary 11. The values of $Max\_Journey_e(i)$ can be computed for all i and all $e \le d$ in $O(nd^2\log n)$ time.

5. The Hybrid Algorithm

In this section, we use the methods of the previous two sections in combination to solve the TP in subquadratic time in the worst case. The basic idea is this: the Bottleneck and Limited Duration algorithms are both subquadratic under certain favorable conditions. But these conditions are in some sense "opposite:" if there are no bottlenecks, then we know the traveler can make the longest journey in few stops. We do not need the hypotheses of the BA or the LDA to hold over the entire road. The *Hybrid Algorithm* instroduced in this section basically uses the BA to compute *Max_Journey* over the portion of the range which is affected by bottlenecks, and the LDA for the remaining portion. The net result is a subquadratic general algorithm.

Consider integers l and d such that $l(d+1) \geq 2n$.

Theorem 12. Any maximum length journey of minimum cost for its length is either l-bottlenecked or has duration no greater than d.

Proof. Let $J = i_0, \ldots i_T$ be a maximum length journey of minimum cost for its length, and assume that i_0 is not l-bottlenecked. Consider three consecutive stops of J, namely i_{t-1}, i_t, and i_{t+1}. If $i_{t+1} - i_{t-1} < l$, then i_t can be eliminated from the journey, because i_0 is not l-bottlenecked. This contradicts the hypothesis that J has minimum cost. Thus, the length of J must be at least $l \lfloor \frac{T}{2} \rfloor$, but which of course cannot exceed n. It follows that $T \leq d$.

Theorem 13. The TP can be solved in $O(n^{\frac{5}{3}}(\log n)^{\frac{1}{3}})$ time.

Proof. Pick $l = n^{\frac{2}{3}}(\log n)^{\frac{1}{3}}$ and pick $d = \lfloor \frac{2n}{l} \rfloor$. Then $l(d+1) \geq 2n$. Compute *Max_Journey*(i) for every i which is l-bottlenecked in $O(nl)$ time, by Theorem 4. By Corollary 11, the limited duration algorithm computes $Max\_Journey_e(i)$ for all i and for all $e \leq d$ in $O(nd^2 \log n)$ time. If *Max_Journey*(i) was not computed by the bottleneck algorithm, its value will be the least of the $Max\_Journey_e(i)$, by Theorem 12. Thus, *Max_Journey*(i) is computable for all i in $O(n^{\frac{5}{3}}(\log n)^{\frac{1}{3}})$ time.

6. B-Trees

Suppose we wish to construct a dictionary for a set of n words of varying length in the form of a B-tree. We define a B-tree to be an ordered tree where all leaves have the same depth. Each node has a fixed capacity, enabling it to store varying numbers of words, depending on the lengths of those words, and the internal nodes contain enough information to guide the search for a desired word. One design (which we call the “guide prefix" case) stores all words in the leaves. The internal nodes, in this case, contain only pointers to other nodes and prefixes of words from the dictionary, only long enough to guide the search. For example, if the contents of an internal node consisted of

$$(p_0, \text{‘b’}, p_1, \text{‘br’}, p_2)$$

then that node would have three children, accessed through the pointers p_0, p_1, p_2. The words in the subtree accessed by p_0 would all be less than 'b', the words in the subtree accessed by p_1 would be at least 'b' but less than 'br', and the words in the subtree accessed by p_2 would be at least 'br'. An alternative design is for actual words to be used as guides in the internal nodes, these words then need not appear in the leaves. We call this the "guide word" design. In either case, the problem is to build a B-tree of minimum height, subject to a given node size. (The internal nodes must allow space for pointers.) We will assume that the words are not so large as to make construction of a B-tree impossible.

We now show how the minimum height B-tree problem can be reduced to the Traveler's Problem. Let $x_1, \ldots x_n$ be the (alphabetically ordered) list of words and let $y_1, \ldots y_n$ be the list of potential guides, either words or prefixes. It is helpful in the discussion to include fictitious guides y_0 and y_{n+1}.

Define a boolean function $f(h,i,j)$ for all $h \geq 0$ and $0 \leq i < j \leq n+1$, as follows. $f(h,i,j)$ is *true* if and only if it is possible to construct a B-tree of height h holding the portion of the dictionary consisting of the words found between the guides y_i and y_j, *i.e.*, words lexically at least y_i but less than y_j. The height of the minimum height B-tree for the entire dictionary is thus the minimum h for which $f(h,0,n+1)$ holds. Since f is monotone, *i.e.*, $f(h,i,j)$ $f(h,i,j-1)$, the information represented by f is equivalent to that represented by the two parameter integer function $F(h,i)$ = the largest j for which $f(h,i,j)$.

Lemma 14. All values of $F(0,i)$ can be computed in $O(n)$ time.

Proof. See Diehr and Faaland [DF]. The guide word and guide prefix cases must be treated slightly differently, but only at this level. In the guide word case, $F(0,i)$ is the largest j such that $\sum_{k=i+1}^{j-1} |x_k| \leq$ maximum leaf size. In the guide prefix case, $F(0,i)$ is the largest j such that $\sum_{k=i}^{j-1} |x_k| \leq$ maximum leaf size. In either case, the Diehr-Faaland algorithm solves the problem in linear time.

Lemma 15. If all values of $F(h,i)$ are known, all values of $F(h+1,i)$ can be computed by

solving one instance of the Traveler's problem.

Proof. Let P be the amount of space occupied by one pointer, and let N be the amount of space in one internal node. We reduce the problem to an instance of the TP as follows.

Let $w_i = |y_i| + P$, $B = N - P$, and $Farthest(i) = F(h,i)$. Then $Max\_Journey(i)$ will be the correct value of $F(h+1,i)$ for all i. A journey from i to j corresponds to a B-tree of height $h+1$ for the list of all words between y_i and y_j. The stops of the journey correspond to the guides that appear in the root node. Each guide takes up space consisting of its own length plus P. The root must also contain one extra pointer, thus the budget is decreased by P. A step in the journey from k to l corresponds to existence of a B-tree of height h consisting of those words between y_k and y_l.

Corollary 16. The minimum height B-tree problem can be solved in $O(n^{\frac{5}{3}}(\log n)^{\frac{4}{3}})$ time.

Proof. The height of the B-tree cannot exceed log n.

References

[AK] Aggarwal, A., M. Klawe, S. Moran, P. Shor, R. Wilber. Geometric applications of a matrix searching algorithm. *Algorithmica 2* (1987), 195-208.

[DF] Diehr, G. and B. Faaland. Optimal pagination of B-trees with variable-length items. *Comm. ACM 27*, 3 (March 1984), 241-247.

[HL] Hirschberg, D.S. and L.L. Larmore. The least weight subsequence problem. *SIAM J. Comp. 16*, 4 (Aug. 1987), 628-638.

[LH] Larmore, L.L. and D.S. Hirschberg. Efficient optimal pagination of scrolls. *Comm. ACM 28*, 8 (Aug. 1985), 854-856.

[M] McCreight, E.M. Pagination of B*-trees with variable-length records. *Comm. ACM 20*, 9 (Sept. 1977), 670-674.

[S] Szwarcfiter, J.L. Optimal multiway search trees for variable size keys. *Acta Informatica 21*, (1984), 49-60.

[W] Wilber, R. The concave least weight subsequence revisited. Preprint (1987).

[Y] Yao, F.F. Efficient dynamic programming using quadrangle inequalities. *Proc. 12th Annual ACM Symp. on Theory of Comput.* (April 1980), 429-435.