3. Partitioning a Path

In this section we present an optimal algorithm to perform parametric search on a path of n vertices. The first phase gathers information with which subsequent feasibility tests can be performed in $o(n)$ time. The second phase then completes the parametric search using this faster feasibility testing. Our discussion focuses on the max-min problem; at the end of the section we identify the changes necessary for the min-max problem. Throughout this section we assume that the vertices of any path or subpath are indexed in increasing order from the start to the end of the path.

We first consider the running time of *PATH0*, to determine how the approach might be accelerated. All activities except for feasibility testing use a total of $O(n)$ time. Feasibility testing uses $\Theta(n \log n)$ time in the worst case, since there will be $\Theta(\log n)$ values to be tested in the worst case, and the test takes $\Theta(n)$ time. It seems unlikely that one can reduce the number of tests that need to be made, so that to design a linear-time algorithm, it seems necessary to design a feasibility test that takes $o(n)$ time. We show how to realize such an approach.

We shall represent the path by a partition into subpaths, each of which can be searched in time proportional to the logarithm of its length. Each such subpath will possess a property that makes feasibility testing easier. Either the subpath will be singular or stable. A subpath P' is *singular* if it consists of one vertex, and it is *stable* if no value in $M(P')$ falls in the interval (λ_1, λ_2). If a subpath is stable, then the position of any one cut determines the positions of all other cuts in the subpath, irrespective of what value of λ is chosen from within (λ_1, λ_2). If suitable data structures are set up when the stable subpath is added to the partition of P, then a linear scan of the subpath is not needed.

To make the representation simple, the subpaths in the partition are restricted to have lengths that are powers of 2. Each subpath will consist of vertices whose

indices are $(j-1)2^i+1,\ldots,j\,2^i$ for integers $j>0$ and $i\geq 0$. Initially the partition will consist of n singular subpaths. Each nonsingular subpath will have $i>0$. When introduced into the partition, it will replace its two *constituent subpaths*, the first with indices $(2j-2)2^{i-1}+1,\ldots,(2j-1)2^{i-1}$, and the second with indices $(2j-1)2^{i-1}+1,\ldots,(2j)2^{i-1}$.

The partition of path P into the subpaths can be represented using three arrays $last[1..n]$, $ncut[1..n]$ and $next[1..n]$. Consider any subpath in the partition, with first vertex v_f and last vertex v_t. Let v_l be an arbitrary vertex in the subpath. The array *last* identifies the end of a subpath, given the first vertex of a subpath. Thus $last(l)=t$ if $l=f$ and is arbitrary otherwise. Given a cut on the subpath, the array *next* identifies, in constant time, a cut further on in that subpath. The array *ncut* identifies the number of cuts skipped in moving to that further cut. Let $w(l,t)$ be the sum of the weights of vertices v_l through v_t. If $w(l,t)<\lambda_2$, then $next(l)=0$ and $ncut(l)=0$. Otherwise, $next(l)\geq l$ is the index of the last vertex before a cut, given that $l=1$ or v_l is the first vertex after a cut. Then $ncut(l)$ is the number of cuts after v_l up to and including the one following $v_{next(l)}$. We assume that the last cut on a subpath will leave a (possibly empty) subset of vertices of total weight less than λ_2. (Note that the last cut on the path as a whole must then be ignored.)

Given the partition into subpaths, we describe feasibility test *FTEST1*. Let λ be the value to be tested, with $\lambda_1<\lambda<\lambda_2$. For each subpath, we use binary search to find the first cut, and then, follow *next* pointers and add *ncut* values to identify the number of cuts on the subpath. When we follow a path of *next* pointers, we will compress this path. This turns out to be a key operation as we consider subpath merging and its effect on feasibility testing. (Note that the path compression makes *FTEST1* a function with side effects.)

```
func FTEST1 (path P, integer k, real λ)
    f ← 1
    numcut ← −1; remainder ← 0
    while f ≤ n do /* search the next subpath: */
        t ← last(f)
        if remainder + w(f, t) < λ
        then remainder ← remainder + w(f, t)
        else
            numcut ← numcut + 1
            Binary search for the smallest r such that w(f, r) + remainder ≥ λ.
            if r < t
            then
                (s, sumcut) ← search_next_path (r, t)
                numcut ← numcut + sumcut
                compress_next_path (r, s, t, sumcut)
            endif
            remainder ← w(s + 1, t)
        endif
        f ← t + 1
    endwhile
    if numcut ≥ k then return("lower") else return("upper") endif
endfunc

search_next_path (vertex_index l, t)
    sumcut ← 0
    while l < t and next(l + 1) ≠ 0
        sumcut ← sumcut + ncut(l + 1)
        l ← next(l + 1)
    endwhile
    return((l, sumcut))

compress_next_path (vertex_index l, s, t, integer sumcut)
    while l < t and next(l + 1) ≠ 0
        sumcut ← sumcut − ncut(l + 1)
        ncut(l + 1) ← ncut(l + 1) + sumcut
        temp ← next(l + 1)
        next(l + 1) ← s
        l ← temp
    endwhile
```

Use of this feasibility test is alone not enough to guarantee a reduction in the time for feasibility testing. This is because there is no assurance that the interval (λ_1, λ_2)

will be narrowed in a manner that allows longer subpaths to quickly replace shorter subpaths in the partition of path P. To achieve this effect, we reorganize the positive values from $M(P)$ into submatrices that correspond in a natural way to subpaths. Furthermore, we associate weights with these submatrices and use these weights in selecting the weighted median for testing. The weights will place a premium on resolving first the values from submatrices corresponding to short subpaths. However, using weights could mean that many values of small weight could be considered repeatedly, causing the total time for selection to exceed $\Theta(n)$. To offset this effect, we also find the unweighted median, and test this value. This approach guarantees that at least half of the submatrices' representatives are discarded on each iteration, so that each submatrix inserted into $\mathcal{M}$ need be charged only a constant for its share of the total work in selecting values to test.

We now proceed to a broad description of *PATH1*. The basic structure follows that of *PAR_SEARCH*, in that there will be the routines *PATH1_init_mat*, *PATH1_test_val*, and *PATH1_update_mat*. However, we will slip the set-up and manipulation of the data structures for the subpaths into the routines *PATH1_init_mat* and *PATH1_update_mat*. Let $large(M)$ be the largest element in submatrix M, and let $small(M)$ be the smallest element in M.

PATH1_init_mat:

Initialize $\mathcal{M}$ to be empty.
Call *mats_for_path*$(P, 1, n)$ to insert submatrices of $M(P)$ into $\mathcal{M}$.
for $l \leftarrow 1$ **to** n **do** $last(l) \leftarrow l$; $next(l) \leftarrow 0$; $ncut(l) \leftarrow 0$ **endfor**

PATH1_test_val:

$R \leftarrow \emptyset$
for each M in $\mathcal{M}$ **do**
    if $large(M) < \lambda_2$ **then** Insert $large(M)$ into R with weight $w(M)/4$. **endif**
    if $small(M) > \lambda_1$ **then** Insert $small(M)$ into R with weight $w(M)/4$. **endif**
endfor
Select the weighted median element λ in R.

if $FTEST1(P, k, \lambda) =$ "$lower$" **then** $\lambda_1 \leftarrow \lambda$ **else** $\lambda_2 \leftarrow \lambda$ **endif**
Remove from R any values no longer in (λ_1, λ_2).
if R is not empty
then
Select the unweighted median element λ' in R.
if $FTEST1(P, k, \lambda') =$ "$lower$" **then** $\lambda_1 \leftarrow \lambda'$ **else** $\lambda_2 \leftarrow \lambda'$ **endif**
endif

PATH1_update_mat:

while there is an M in $\mathcal{M}$ such that $small(M) \geq \lambda_2$ or $large(M) \leq \lambda_1$
or $small(M) \leq \lambda_1 \leq \lambda_2 \leq large(M)$ **do**
if $small(M) \geq \lambda_2$ or $large(M) \leq \lambda_1$
then
Delete M from $\mathcal{M}$.
if this is the last submatrix remaining for a subpath P'
then $glue\_paths(P')$
endif
endif
if $small(M) \leq \lambda_1$ and $large(M) \geq \lambda_2$
then Split M into four square submatrices, each of weight $w(M)/8$.
endif
endwhile

Figure 1: Initial submatrices for $M(P)$ in *PATH1*

Figure 2: TO BE FILLED IN??

Below is the procedure *mats_for_path*, which inserts submatrices of appropriate

weight into $\mathcal{M}$. A call with arguments f and t will generate submatrices for all required subpaths of a path containing the vertices with indices $f, \ldots, t$. (In this section, we assume that $f = 1$ and $t = n$, but we state the procedure in this form so that it can be used in the next section too.) The submatrices for the matrix $M(P)$ in Fig. **??** are shown in Fig. **??**. [REDO FOR NEW DEF] Every subpath P' that will appear in some partition of P is initialized with $cleaned(P')$ and $glued(P')$ to **false**, where $cleaned(P')$ indicates whether or not all values in the submatrix associated with P' are outside the interval (λ_1, λ_2) , and $glued(P')$ indicates whether or not all values in $M(P')$ are outside the interval (λ_1, λ_2).

proc *mats_for_path* (**path** P, **integer** f, t)
    $size \leftarrow 1$
    $w \leftarrow 4n^4$
    while $f \leq t$ **do**
        for $i \leftarrow f$ **to** t **by** $size$ **do**
            Insert submatrix $[i \,..\, (i+\lceil size/2 \rceil - 1), (i+\lceil size/2 \rceil) .. \, (i+size-1)]$
                of $M(P)$ into $\mathcal{M}$ with weight w.
            Let P' be the subpath whose vertices have indices $i, \ldots, i + size - 1$.
            $cleaned(P') \leftarrow$ **false**; $glued(P') \leftarrow$ **false**
            endfor
        $size \leftarrow size * 2$
        $w \leftarrow w/2$
        $f \leftarrow size * \lceil (f-1)/size \rceil + 1$
        $t \leftarrow size * \lfloor t/size \rfloor$
    endwhile
endproc

We next describe the procedure *glue_paths*, which checks to see if two constituent subpaths can be combined together. To glue two subpaths P_2 and P_3 together into a subpath P_1, we must have ($glued(P_2)$ **and** $glued(P_3)$ **and** $cleaned(P_1)$). Let P' be any subpath of weight at least λ_2, and let f' and t' be the indices of the first and last vertices in P', resp. Let the λ-*prefix* of P', designated $\lambda pref(P')$, be vertices

$v_{f'}, \cdots, v_l$ in P' where l is the largest index such that $w(f', l) < \lambda_2$. Let the λ-*suffix* of P', designated $\lambda suff(P')$, be vertices $v_l, \cdots, v_{t'}$ in P' where l is the smallest index such that $w(l, t') < \lambda_2$. Procedure *glue_paths* sets the *next* pointers from vertices in $\lambda suff(P_2)$ to vertices in $\lambda pref(P_3)$.

```
proc glue_paths (path P1)
    cleaned(P1) ← true
    if P1 has length 1
    then
        glued(P1) ← true
        Let l be the index of the vertex in P1.
        if w(l,l) ≥ λ2 then next(l) ← l; ncut(l) ← 1 endif
        Reset P1 to be the subpath of which P1 is now a constituent subpath.
    endif
    Let P2 and P3 be the constituent subpaths of P1.
    while glued(P2) and glued(P3) and cleaned(P1) and P1 ≠ P do
        glued(P1) ← true
        Let f2 and t2 be resp. the indices of the first and last vertices in P2.
        Let f3 and t3 be resp. the indices of the first and last vertices in P3.
        last(f2) ← t3
        if w(f2, t3) ≥ λ2
        then
            if w(f2, t2) < λ2
            then f ← f2
            else /* initialize f to the front of λsuff(P2) */
                f ← t2
                while w(f − 1, t2) < λ2 do f ← f − 1 endwhile
            endif
            t ← f3
            while f ≤ t2 and w(f, t3) ≥ λ2 do /* set next pointers for λsuff(P2) */
                while w(f, t) < λ2 do t ← t + 1 endwhile
                ncut(f) ← 1
                next(f) ← t
                f ← f + 1
            endwhile
        endif
        Reset P1 to be the subpath of which P1 is now a constituent subpath.
        Let P2 and P3 be the constituent subpaths of P1.
    endwhile
endproc
```

It would nice if procedure *glue_paths*, after setting pointers in $\lambda suff(P_2)$, would perform pointer jumping, so that *next* pointers for vertices in $\lambda pref(P_1)$ would point to vertices in $\lambda suff(P_1)$. Unfortunately, it appears that requiring *glue_paths* to perform such an activity would result in a total time over all calls of $O(n \log\log n)$ in the worst case. (We do not have a proof of this assertion, but extensive simulation seems to provide strong corroboration for such a conjecture.) We opt for having *glue_paths* do no pointer jumping, and instead we do pointer jumping under the guise of path compression in *FTEST1*. We use an argument based on amortization to show that this works well.

Lemma 3.1. Let P be a path of n vertices. All calls to *glue-paths* will take amortized time of $O(n)$, and *FTEST1* will search each subpath in amortized time proportional to the logarithm of its length.

Proof. Suppose two subpaths P_2 and P_3 are glued together to give P_1, where P_1 has weight at least λ_2. We consider two cases. First suppose either P_2 or P_3 has weight at most λ_1. Let P_m represent this subpath. The time for *glue_paths* is in worst case proportional to the length of P_m. Also, we leave a *glue-credit* on each vertex in the $\lambda pref(P_1)$ and $\lambda suff(P_1)$, and a *jump-credit* on each vertex in the $\lambda pref(P_1)$. The number of credits will be proportional to the length of P_m. We charge this work and the credits to the vertices of P_m, at a constant charge per vertex. It is clear that any vertex in P is charged at most once, so that the total charge to vertices over all calls to *glue_paths* is $O(n)$. The second case is when both P_2 and P_3 have weight at least λ_2. In this case, the time for *glue_paths* is proportional to the sum of the lengths of $\lambda suff(P_2)$ and $\lambda pref(P_3)$, and we use the glue-credits of P_2 and P_3 to pay for this. The jump-credits are used by *FTEST1* rather than *glue-paths*, but we discuss below how they might have been used by *glue-paths*.

Suppose both P_2 and P_3 have weight at least λ_2. Then we might have wanted

to have *glue-paths* jump the *next* pointers so that the *next* pointer for a vertex in $\lambda pref(P_2)$ will be reset to point to $\lambda suff(P_3)$. The time to reset such pointers will be proportional to the length of $\lambda pref(P_2)$. The jump-credits of $\lambda pref(P_3)$ could pay for this, as long as the length of $\lambda pref(P_2)$ is at most some constant (say 2) times the length of the $\lambda pref(P_3)$. When the length of the $\lambda pref(P_2)$ is at most twice the length of the $\lambda pref(P_3)$, we would not want to jump the pointers.

In general we could view a subpath as containing a sequence of λ-*regions*, where each λ-region was once the λ-prefix of some subpath, and the length of a λ-region is less than twice the length of the preceding λ-region. Note that the number of λ-regions in the sequence could be at most the logarithm of the length of the subpath. When gluing two subpaths together, we could concatenate their sequences of λ-regions together, and jump pointers over any λ-region that gets a predecessor whose length is not more than twice its length. Its jump-credits could then be used for this pointer jumping. Searching a subpath in *FTEST1* would then take time at most proportional to its length.

Of course *glue-paths* does not jump pointers. However, a lazy form of pointer-jumping is found in the path-compression of *FTEST1*, and the same sort of analysis can be seen to apply. Suppose that *FTEST1* follows a pointer to a vertex that was once in the λ-prefix of some subpath. Consider the situation if *glue-paths* had jumped pointers. If that pointer would have been present in the sequence of λ-regions, then charge the operation of following that pointer to *FTEST1*. Otherwise, charge the operation of following that pointer to the jump-credits of the λ-prefix containing the vertex. It follows that the number of pointers followed during a search of a subpath that are not covered by jump-credits is at most the logarithm of the length of the subpath. Also, the binary search to find the position of the first cut on a subpath uses time at most proportional to the logarithm of the length of the subpath. $\Box$

Lemma 3.2. Let P be a path of n vertices. On the i-th iteration of the while-loop of *PATH1*, the amortized time used by feasibility test *FTEST1* will be $O(i(5/6)^{i/5}\, n)$.

Proof. We first consider the weights assigned to submatrices in $\mathcal{M}$. Corresponding to subpaths of lengths $1, 2, 4, \ldots, n$, procedure *mats_for_path* creates n submatrices of size 1×1, $n/2$ submatrices of size 1×1, $n/4$ submatrices of size 2×2, and so on, up through one submatrix of size $n/2 \times n/2$. The total weight of the submatrices corresponding to each path length is $4n^5, n^5, n^5/4, \ldots, 4n^3$, resp. It follows that the total weight for submatrices of all path lengths is less than $(4/3) * 4n^5$.

Let $wgt(M)$ be the weight assigned to submatrix M. Define the *effective weight*, denoted *eff_wgt*(M), of a submatrix M in $\mathcal{M}$ to be $wgt(M)$ if both its smallest value and largest value are contained in the interval (λ_1, λ_2) and $(3/4)wgt(M)$ if only one of its smallest value and largest value is contained in the interval (λ_1, λ_2). Let *eff_wgt*$(\mathcal{M})$ be the total effective weight of all submatrices in $\mathcal{M}$. When a feasibility test renders a value (the smallest or largest) from M no longer in the interval (λ_1, λ_2), we argue that *eff_wgt*(M) is reduced by at least $wgt(M)/4$ because of that value. If both values were contained in the interval (λ_1, λ_2), and one is no longer is, then clearly it is true. If only one value was contained in the interval (λ_1, λ_2), and it no longer is, then M is replaced by four submatrices of effective weights $wgt(M)/8 + wgt(M)/8 + (3/4)wgt(M)/8 + (3/4)wgt(M)/8 < wgt(M)/2$, so that there is a reduction in weight by greater than $wgt(M)/4$. If both values were contained in the interval (λ_1, λ_2), and both are no longer in, then we consider first one and then the other. Thus every element that was in (λ_1, λ_2) but no longer is causes a decrease in *eff_wgt*$(\mathcal{M})$ by an amount at least equal to its weight in R. Values with at least half of the total weight in R find themselves no longer in (λ_1, λ_2). Furthermore, the total weight of R is at least $1/3$ of *eff_wgt*$(\mathcal{M})$. This follows since a submatrix M has either two values in R at a total of $2w(M)/4 = (1/2)w(M)$ or one value at a weight of $w(M)/4 = (1/3)(3w(M)/4)$.

Thus *eff_wgt*($\mathcal{M}$) decreases by a factor of at least $(1/2)(1/3) = 1/6$ per iteration.

Corresponding to the subpaths of length 2^j, there are $n/2^j$ submatrices created by *mats_for_path*. If such a matrix is quartered repeatedly until 1×1 submatrices result, each such submatrix will have weight $n^4/2^{4j-5}$. Thus when as little as $(n/2^j) * (n^4/2^{4j-5})$ weight remains, all subpaths of length 2^j can still be unresolved. This can be as late as iteration i, where i satisfies $(4/3) * 4n^5 * (5/6)^i = n^5/2^{5j-5}$, or $2^{5j} = 6*(6/5)^i$. While all subpaths of length 2^j can still be unresolved on this iteration, at most $(1/2 * 1/2^4)^k = 1/2^{5k}$ of the subpaths of length 2^{j-k} can be unresolved for $k = 1, \ldots, j-1$, and at most $1/2^{5j-2}$ of the subpaths of length 1 can still be unresolved. Also, by Lemma 3.1, each subpath can be searched by *FTEST1* in amortized time proportional to its length. Thus the time to search path P on iteration i is at worst proportional to $(j/2^j)n(1 + 1/2^5 + 1/2^{10} + \cdots)$, which is $O((j/2^j)n)$. From the relationship of i and j, $2^j = 6^{1/5}(6/5)^{i/5}$, and $j = (1/5)\log 6 + (i/5)\log(6/5)$. The lemma then follows. □

Lemma 3.3. The total time for handling $\mathcal{M}$ and performing selection over all iterations of *PATH1* is $O(n)$.

Proof. Using the algorithm of [**?**], the time to perform selection in a given iteration is proportional to the size of R. Define the *effective count*, denoted *eff_cnt*(M), of a submatrix M in $\mathcal{M}$ to be 2 if both its smallest value and largest value are contained in the interval (λ_1, λ_2) and 1 if only one of its smallest value and largest value is contained in the interval (λ_1, λ_2). Let *eff_cnt*($\mathcal{M}$) be the total effective count of all submatrices in $\mathcal{M}$. Then the size of R equals *eff_cnt*($\mathcal{M}$). On any iteration, the result of the feasibility test of λ' is to resolve at least half of the values in R. The time for forming R and performing both selections can be accounted to the values so resolved, at a constant charge for each such value.

It remains to count the number of submatrices inserted into $\mathcal{M}$. Initially $2n - 1$

submatrices are inserted into $\mathcal{M}$. For $j = 1, 2, \ldots, \log n - 1$, consider all submatrices of size $2^j \times 2^j$ that are at some point in $\mathcal{M}$ and that can be quartered. Each of these must have its smallest value at most λ_1 and its largest value at least λ_2. Clearly there will be fewer than $2(n/2^j)$ of them. Thus the number resulting from quartering is less than $8(n/2^j)$. Summing over all j gives $O(n)$ submatrices in $\mathcal{M}$ resulting from quartering. □

We illustrate *PATH1* on path P in Fig. **??**, with $k = 3$. (This is the same example that we discussed near the end of the previous section.) The initial submatrices for $\mathcal{M}$ are shown in Fig. **??**. The submatrix of size 4×4 has weight 2^{11}, the two submatrices of size 2×2 have weight 2^{12}, four submatrices of size 1×1 have weight 2^{13}, and the remaining eight submatrices of size 1×1 have weight 2^{14}. Initially $\lambda_1 = 0$ and $\lambda_2 = \infty$. The *last*, *next* and *ncut* arrays are initialized as stated.

On the first iteration, R contains two copies each of $6, 11, 9, 2, 1, 15, 7, 8$ of weight 2^{12}, two copies each of $17, 11, 16, 15$ of weight 2^{11}, $20, 28, 22, 31$ of weight 2^{10}, and $3, 59$ of weight 2^9. The weighted median of R is 11. For $\lambda = 11$, 3 cuts are required, so λ_1 is reset to 11. The revised version of R will then be $\{15, 15, 17, 17, 16, 16, 15, 15, 20, 28, 22, 31, 59\}$. The median of this is 17. For $\lambda' = 17$, 1 cut is required, so λ_2 is reset to 17. The set $\mathcal{M}$ is changed as follows. All submatrices of size 1×1 are discarded except the one of weight 2^{14} containing 15, and the two of weight 2^{13} containing 15 in one and 16 in the other. All submatrices of size 2×2 are discarded. The submatrix of size 4×4 is quartered, giving four submatrices each of weight 2^8. Three of these submatrices are discarded because their values are all too large. The remaining submatrix (containing $27, 12, 18, 3$) is quartered, giving four submatrices each of weight 2^5. All submatrices but the submatrix of size 1×1 containing 12 are discarded.

As a result of submatrix discarding, a number of subpaths are rendered cleaned and glued. Every subpath of length 1 except v_6 are cleaned and glued, the subpaths

v_1, v_2 and v_3, v_4 are cleaned and glued, the subpath v_1, v_2, v_3, v_4 is cleaned and glued, and the subpath v_5, v_6, v_7, v_8 is cleaned but not glued. When subpath v_6 is glued, $next(6)$ is set to 6, and $ncut(6)$ is set to 1. When subpath v_1, v_2 is glued, $last(1)$ is set to 2, $next(1)$ is set to 2, and $ncut(1)$ is set to 1. When subpath v_3, v_4 is glued, $last(3)$ is set to 4, but no *next* or *ncut* values change. When subpath v_1, v_2, v_3, v_4 is glued, $last(1)$ is set to 4, $next(2)$ is set to 3, and $ncut(2)$ is set to 1. This completes all activity on the first iteration.

On the second iteration, R contains two copies of 15 of weight 2^{12}, two copies each of $16, 15$ of weight 2^{11}, and two copies of 12 of weight 2^3. The weighted median of R is 15. For $\lambda = 15$, 2 cuts are required, so λ_2 is reset to 15. The revised version of R will then be $\{12, 12\}$. The median of this is 12. For $\lambda' = 12$, 3 cuts are required, so λ_1 is reset to 12.

All remaining submatrices are discarded from $\mathcal{M}$. When this happens, all remaining subpaths are cleaned and glued. When subpath v_5, v_6 is glued, $last(5)$ is set to 6, $next(5)$ is set to 6, and $ncut(5)$ is set to 1. When subpath v_7, v_8 is glued, $last(7)$ is set to 8, $next(7)$ is set to 8, and $ncut(7)$ is set to 1. When subpath v_5, v_6, v_7, v_8 is glued, $last(5)$ is set to 8, but no *next* or *ncut* values change. When subpath $v_1, v_2, v_3, v_4, v_5, v_6, v_7, v_8$ is glued, $last(1)$ is set to 8, $next(3)$ and $next(4)$ are set to 6, and $ncut(3)$ and $ncut(4)$ are set to 1. Since $\mathcal{M}$ is empty, *PATH1* will then terminate with $\lambda_1 = 12$, and $\lambda_2 = 15$.

Note that no compression of search paths was performed on the second iteration. Suppose for the sake of example that a subsequent search with $\lambda = 13$ were performed. The initial cut would come after v_2, and $next(3)$ and $next(7)$ would be followed to arrive at v_8. Then $ncut(3)$ would be reset to 2, and $next(3)$ would be reset to 8.

Theorem 3.4. Algorithm *PATH1* solves the max-min k-partitioning problem on a path of n vertices in $O(n)$ time.

Proof. CORRECTNESS!!

The time to initialize $\mathcal{M}$ is clearly $O(n)$. By Lemma 3.3, the total time to select all values to test for feasibility is $O(n)$. By Lemma 3.2, the amortized time to perform feasibility test on iteration i is $O(i(5/6)^{i/5}n)$. Summed over all iterations, this quantity is $O(n)$. By Lemma 3.3, the total time to handle submatrices in $\mathcal{M}$ is $O(n)$. By Lemma 3.1, the time to manipulate data structures for the subpaths is $O(n)$. The time bound then follows. □

We briefly survey the differences needed to solve the min-max problem. Procedure *FTEST1* is similar except that r is the largest index such that $w(f,r)+remainder \leq \lambda$, 1 is subtracted after completion of the for-loop if $remainder = 0$, and $numcut \geq k$ is replaced by $numcut > k$. Procedure *glue_paths* is similar, except that we replace the statement

while $w(f,t) < \lambda_2$ **do** $t \leftarrow t+1$ **endwhile**

by the statement

while $w(f,t+1) < \lambda_2$ **do** $t \leftarrow t+1$ **endwhile**

Note that $w(l,l) \leq \lambda_2$ always holds for the min-max problem.

Theorem 3.5. The min-max k-partitioning problem can be solved on a path of n vertices in $O(n)$ time.

Proof. The above changes will not affect the asymptotic running time of algorithm *PATH1*. □