Faster Optimal Single–Row Placement with Fixed Ordering

U. Brenner J. Vygen
Research Institute for Discrete Mathematics, University of Bonn
Lennéstr. 2, 53113 Bonn, Germany

Abstract

We consider the problem of placing a set of cells in a single row with a given horizontal ordering, minimizing the (weighted) bounding box netlength. We analyze the running time of an algorithm of Kahng, Tucker and Zelikovsky which solves this problem optimally. By using different data structures we are able to improve the worst–case running time in the unweighted case as well as in the presence of netweights.

1. Introduction

The one–dimensional placement problem consists of placing a set of cells within a single row. The objective is to minimize the bounding box netlength estimation. Even this restricted problem is *NP*–hard [1]. In this paper we consider an even more restricted problem, which is easier to solve: we assume a fixed ordering of the cells to be placed. More precisely, we consider the following problem:

Given cells $C_1, \dots, C_n$ with widths $w : \{C_1, \dots, C_n\} \to \mathbb{R}_+$ and a total row width $W \geq \sum_{i=1}^{n} w(C_i)$, a feasible placement is a function $p : \{C_1, \dots, C_n\} \to \mathbb{R}_+$ such that $p(C_{i+1}) \geq p(C_i) + w(C_i)$ for $i = 1, \dots, n-1$ and $W \geq p(C_n) + w(C_n)$. Moreover, we are given a netlist, possibly with net weights, and we ask for a feasible placement with minimum (weighted) bounding box netlength. We call this problem the Ordered Single Row Problem (OSRP).

This problem occurs as the final task of some detailed placement tools for standard cells (see e.g. [3]), considering each row (or each maximal part of a row not blocked by any macros) separately. Even if other methods for detailed placements are used, the OSRP can be applied afterwards to improve the design (as shown in [2]).

The OSRP can be formulated as a linear program which is the dual of an uncapacitated minimum cost flow problem [3]; indeed this also holds when considering several (or all) rows simultaneously. This leads to an $O(m^2 \log^2 m)$–algorithm, where m is the number of nets with at least one movable pin.

Kahng, Tucker and Zelikovsky [2] improved this considerably. They developped a so–called Clumping Algorithm which solves the OSRP optimally. However, their algorithm is slower than stated in [2]: we show that it needs time $\Theta(m \log^2 m)$ in the unweighted case and $\Theta(m^2)$ in the weighted case. We use different data structures to implement the Clumping Algorithm with a running time of $O(m \log m \log\log m)$ in the unweighted case and $O(m \log^2 m)$ in the weighted case. Besides being of theoretical interest, these improvements may also be relevant in practice since current designs already have rows with several thousand cells.

2. Notation

As [2] we define for each net N:

- $L(N)$ ($R(N)$) is the leftmost (rightmost) movable cell which has a pin of N (we consider only nets with at least one movable pin);
- $f_l(N)$ ($f_r(N)$) is the position of the leftmost (rightmost) fixed pin of N.

We assume that every net has at least one fixed pin. This causes no loss of generality: A net N without fixed pins can be replaced by two new nets; the first one connects $L(N)$ with a fixed pin at position W, the second one connects $R(N)$ with a fixed pin at position 0. It is easy to see that the resulting instance is equivalent.

If all pins are on the left edge of their cells (i.e. without pin offsets), the horizontal part of the bounding box netlength of a feasible placement p is

$$\sum_N (\max\{f_r(N), p(R(N))\} - \min\{f_l(N), p(L(N))\}),$$

or equivalently,

$$\sum_N (f_r(N) - f_l(N)) + \sum_{i=1}^{n} cost_{C_i}(p(C_i)),$$

where the cost function $cost_C$ of cell C is defined on the interval $[0, W - w(C)]$ by

$$cost_C(x) := \sum_{N:L(N)=C} \max\{f_l(N) - x, 0\} + \sum_{N:R(N)=C} \max\{x - f_r(N), 0\}.$$

The vertical part of the bounding box netlength is fixed in the OSRP. Of course, (horizontal) pin offsets can easily be taken into account by shifting fixed pins.

The function $cost_C$ describes the contribution of cell C to the bounding box netlength. So the task is to find a feasible placement p minimizing

$$\sum_{i=1}^{n} cost_{C_i}(p(C_i)).$$

In the second part of this paper we shall also consider weights $\tau(N) > 0$ for all nets N; in this case $cost_C(x) = \sum_{N:L(N)=C} \tau(N) \max\{f_l(N) - x, 0\} + \sum_{N:R(N)=C} \tau(N) \max\{x - f_r(N), 0\}$. The cost function $cost_C$ is the sum of convex piecewise linear functions and hence is convex and piecewise linear itself.

Without loss of generality $n \leq 2m$ since a cell C, for which no net N with $R(N) = C$ or $L(N) = C$ exists, can be placed arbitrarily; we can delete C and increase the width of its predecessor by $w(C)$.

For each cell C we denote by $[l(C), r(C)]$ the interval within $[0, W - w(C)]$ where $cost_C$ assumes its minimum. Of course, $l(C) = r(C)$ is possible.

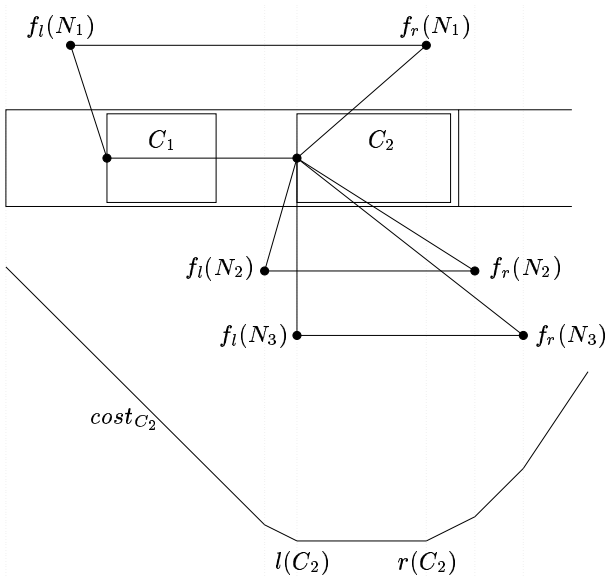

Figure 1.

Example: Figure 1 shows an instance of OSRP with two cells and three nets N_1, N_2 and N_3. Here $C_1 = L(N_1)$, and $C_2 = R(N_1) = L(N_2) = R(N_2) = L(N_3) = R(N_3)$. The cost function of C_2 has six different slopes. C_2 is placed on position $l(C_2)$.

It is convenient to represent the cost function $cost_C$ by a set $M(C)$ of slope changes:

$$M(C) := \{(f_l(N), -\tau(N)) : C = L(N)\} \cup \{(f_r(N), \tau(N)) : C = R(N)\}.$$

The slope of the cost function at an arbitrary position y (with $y \neq x$ for all $(x, \sigma) \in M(C)$) can be computed by adding the positive weights of the elements to the left of y and the negative weights of the elements to the right of y, i.e. $\sum_{(x,\sigma)\in M(C):x<y} \max\{0, \sigma\} + \sum_{(x,\sigma)\in M(C):x>y} \min\{0, \sigma\}$. Note that for all $(x, \sigma) \in M(C)$ we have $x \leq l(C)$ or $x \geq r(C)$. Moreover, $\sum_{i=1}^{n} |M(C_i)| = 2m$.

3. The Clumping Algorithm

The following algorithm of Kahng, Tucker and Zelikovsky [2] solves the OSRP optimally:

CLUMPING ALGORITHM

Input: An ordered list $\mathcal{L}$ of cells $C_1, \ldots, C_n$ and their widths $w : \{C_1, \ldots, C_n\} \to \mathbb{R}_+$; nets $N_1, \ldots, N_m$ and their weights $\tau : \{N_1, \ldots, N_m\} \to \mathbb{R}_+$.

Output: A feasible placement $p : \{C_1, \ldots, C_n\} \to \mathbb{R}_+$.

① Add an auxiliary element C_0 at the front of $\mathcal{L}$ and set $p(C_0) := 0$ and $w(C_0) := 0$.
② **For** $i = 0, \ldots, n$:
  Initialize $M(C_i)$, $l(C_i)$ and $r(C_i)$.
③ **For** $i = 1, \ldots, n$: PLACE$(C_i, \mathcal{L})$;
④ Use $p : \mathcal{L} \to \mathbb{R}_+$ to compute the positions of the original cells $p : \{C_1, \ldots, C_n\} \to \mathbb{R}_+$.

PLACE$(C, \mathcal{L})$

① Let C' be the predecessor of C in $\mathcal{L}$.
② **If** $p(C') + w(C') \leq r(C)$
then $p(C) := \max\{p(C') + w(C'), l(C)\}$
else COLLAPSE$(C', C, \mathcal{L})$ and PLACE$(C', \mathcal{L})$.

COLLAPSE$(C', C, \mathcal{L})$

① Shift the coordinates of all entries of $M(C)$ by $w(C')$ units to the left.
② $M(C') := M(C') \cup M(C)$.
③ Find in $M(C')$ the new positions of $l(C')$ and $r(C')$ (with respect to the cost function of the merged cell made of C' and C).
④ $w(C') := w(C') + w(C)$.
⑤ Remove C from $\mathcal{L}$.

The function PLACE first tries to place cell C at the leftmost position which is to the right of its (already placed) predecessor C' and within $[l(C), r(C)]$. If this is impossible because $r(C) < p(C') + w(C')$, we apply COLLAPSE to merge C' and C to a single cell. In

particular, C' and C are merged if there is not enough space for C to the right of C'.

The coordinates of $M(C)$ must be shifted by $w(C')$ units to the left in ① of COLLAPSE in order to take the new pin offsets into account.

After the procedure PLACE has been called for all cells in ③, it is easy to derive the positions of the original cells from the placement of the merged cells in ④.

Theorem 3.1 *The* CLUMPING ALGORITHM *finds an optimum placement.*

For the simple proof we refer to [2].

4. Analysis of the running time

Evidently the running time of the CLUMPING ALGORITHM mainly depends on the data structures used to represent the sets $M(C_i)$. Kahng, Tucker and Zelikovsky [2] recommend red-black trees, since they allow inserting an element and determining the successor or predecessor in logarithmic time. Steps ② and ③ of COLLAPSE can then be implemented (assuming $|M(C)| \leq |M(C')|$) by scanning $M(C)$ from left to right, inserting the elements of $M(C)$ into $M(C')$ and shifting $l(C')$ and $r(C')$ by one position to the left or to the right if necessary after an insertion. (If the net weights are not constant, it may be necessary to shift $l(C')$ and $r(C')$ by more than one position.)

If $|M(C)| > |M(C')|$, then we interchange the roles of C and C'. We introduce a global offset for the new cell, $-w(C')$, and move the coordinates of all entries of $M(C')$ by $w(C')$ to the right.

For unit weights $\tau \equiv 1$ the running time is $O(m \log^2 m)$: Merging two cells takes $O(l \log m)$ time, where $l = \min\{|M(C)|, |M(C')|\}$. Since the cardinality of the united set $M(C)$ is at least $2l$, each element of $\bigcup_{i=1}^{n} M(C_i)$ can be only $\log m$ times element of the smaller set in COLLAPSE. Since all other parts of the algorithm can easily be implemented in $O(m \log m)$ time, we obtain an overall running time of $O(m \log^2 m)$.

Kahng, Tucker und Zelikowski [2] even state a running time of $O(m \log m)$. However, we shall now prove that no better bound than $O(m \log^2 m)$ can be shown for their implementation:

Theorem 4.1 *Let c be a sufficiently small positive constant, such that* COLLAPSE$(C, C', \mathcal{L})$ *takes at least $c\, l \log l$ elementary steps, where $l = \min\{|M(C)|, |M(C')|\}$. Moreover we assume that the* CLUMPING ALGORITHM *needs at least $\frac{c}{8}$ steps for instances with just one cell.*

Then for each $k \in \mathbb{N}$ there is an instance of the OSRP *with 2^k cells and 2^k nets of unit weight, such that the* CLUMPING ALGORITHM *takes at least $c(k-1)^2 2^{k-3}$ steps.*

Proof (Sketch): By induction on k we show the existence of an instance $I(k, a, s)$ of OSRP with the following properties, for each $a > 0$ and each $s \in \{0, -1\}$.

- The instance has 2^k cells of unit width; each cell has precisely one pin, located at the left edge of the cell.
- The instance has 2^k nets of unit weight; each net has one movable pin and one or two fixed pins. If a net has two fixed pins, then one of them has x-coordinate 0.
- The CLUMPING ALGORITHM merges all cells to one; the cost function of the final merged cell has slope s in the interval $[0, a]$ and assumes its minimum in a.
- The CLUMPING ALGORITHM takes at least $c(k-1)^2 2^{k-3}$ steps.

For $k = 0$ and $s = 0$ consider an instance whose only cell has one pin which is connected by a three–terminal net to fixed pins at coordinates 0 and a. For $k = 0$ and $s = -1$, the cell's pin is connected by a two–terminal net to a fixed pin at coordinate a.

For the induction step we want to show the existence of an instance $I(k_0, a_0, s_0)$ with the properties above by using the induction hypothesis that claims the existence of instances $I(k, a, s)$ for $k < k_0$, $a > 0$ and $s \in \{0, 1\}$. We apply the induction hypothesis to $k = k_0 - 1$, $a = a_0 + 2^{k_0} - 1$, $s = -1$; the result are the first $2^{k_0 - 1}$ cells of our instance. Next we apply the induction hypothesis to $k = k_0 - 2$, $a = a_0 + 2^{k_0} + 2^{k_0 - 1} - 1$ and $s = 0$, then to $k = k_0 - 3$, $a = a_0 + 2^{k_0} + 2^{k_0 - 1} + 2^{k_0 - 2} - 1$ and $s = 0$, and so on. Up to now we have an instance with $2^{k_0} - 1$ cells and nets. If we apply the CLUMPING ALGORITHM to this instance, we have at the end k_0 merged cells, the first of which has width $2^{k_0 - 1}$, the second one has width $2^{k_0 - 2}$ and the last one has width 1. These are placed directly next to each other. The CLUMPING ALGORITHM needs at least $\sum_{i=0}^{k_0 - 1} c(i-1)^2 2^{i-3}$ steps for this instance.

To complete our instance we create one more cell (the rightmost one), whose pin is connected to a fixed pin at coordinate 0 (for $s = 0$) resp. to two fixed pins at coordinates 0 and $a_0 + 2^{k_0}$ (for $s = -1$). If we apply the CLUMPING ALGORITHM to the completed instance it first behaves as if run on the incomplete instance without the rightmost cell. At the end this cell is considered and merged successively with its predecessors, because initially we have $l(C_{2^{k_0}}) = 0$ and either $r(C_{2^{k_0}}) = a_0 + 2^{k_0}$ or $r(C_{2^{k+1}}) = 0$. Since each predecessor C of $C_{2^{k_0}}$ has $l(C) \leq a_0 + 2^{k_0} - 1$, $C_{2^{k_0}}$ is indeed merged with all its predecessors into a single

cell. One easily checks that the instance has the other asserted properties.

For placing the rightmost cell the algorithm needs at least $\sum_{i=0}^{k_0-1} 2^i c \log(2^i)$ steps, hence the overall running time is at least $c\sum_{i=0}^{k_0-1} 2^i(\frac{(i-1)^2}{8}+i) > c(k_0-1)^2 2^{k_0-3}$ (the last inequality follows by an easy induction on k_0). □

5. Improved Implementation

We now suggest an improved implementation. To simplify our notation we assume $x \neq x'$ for different elements $(x,\sigma),(x',\sigma') \in M(C)$ for each cell C. We store $M(C)$ as a set of at most $\lceil \log(2m) \rceil$ ordered doubly–linked lists. The elements $(x,\sigma) \in M(C)$ in each list are sorted by their coordinate x.

For each list of $M(C)$ we maintain a pointer to the rightmost list element to the left of $l(C)$. When merging two cells C and C' (with $|M(C)| \leq |M(C')|$), we just add a list representing $M(C)$ to $M(C')$. If after this the number of lists of C' exceeds $\lceil \log(2m) \rceil$, we merge two of them.

We now describe our implementation in detail. When merging two cells C and C' in COLLAPSE, where $|M(C)| \leq |M(C')|$, we proceed as follows (implementing ② and ③ of COLLAPSE):

- Suppose $M(C')$ is stored in t lists $M_1,\ldots,M_t$ (where $t \leq \lceil \log(2m) \rceil$). Merge all lists of $M(C)$ into a single ordered list M_{t+1}.
- Find in $M_1 \cup \ldots \cup M_{t+1}$ the $|M_{t+1}|$ rightmost elements to the left of $l(C')$ and the $|M_{t+1}|$ leftmost elements to the right of $r(C')$. Let M be an ordered list containing $l(C')$, $r(C')$, the first $|M_{t+1}|$ predecessors of $l(C')$, the first $|M_{t+1}|$ successors of $r(C')$, and all elements in between $l(C')$ and $r(C')$.
- The new positions of $l(C')$ and $r(C')$ must be elements of M. Therefore they can be found by scanning M_{t+1} and successively updating $l(C')$ and $r(C')$ in M.
- If $t+1 > \lceil \log(2m) \rceil$, then choose two lists among $\{M_1,\ldots,M_{t+1}\}$ whose lengths differ by at most a factor of 2. Merge these lists.

Theorem 5.1 *With the above implementation the overall running time of* COLLAPSE*, and hence of the* CLUMPING ALGORITHM*, is $O(m \log m \log\log m)$ for unit net weights.*

Proof: The running time of the CLUMPING ALGORITHM is dominated by the time needed for COLLAPSE. With the above implementation there are three relevant contributions:

1. The time for computing M_{t+1}:

We can compute M_{t+1} in time $O(|M_{t+1}| \log\log m)$: Let $M(C)$ be stored in $M'_1,\ldots,M'_{t'}$. Merge the lists M'_{2i-1} and M'_{2i} to one list for $i \in \{1,\ldots,\lfloor \frac{t'}{2} \rfloor\}$. This can be done in time $O(|M_{t+1}|)$ and the result is a set of $\lceil \frac{t'}{2} \rceil$ lists that contain the elements of $M(C)$. By iterating this method one gets a single list in time $O(|M_{t+1}| \log\log m)$ because the number of lists decreases in each iteration by a constant factor.

Finally we compute a pointer to the rightmost element of M_{t+1} to the left of $l(C')$ by simple scanning of the list (in $O(|M(C)|)$ time).

As $|M(C') \cup M(C)| \geq 2|M(C)|$, each element can belong to M_{t+1} at most $O(\log m)$ times. Hence the overall running time for computing the sets M_{t+1} in all calls to COLLAPSE is bounded by $O(m \log m \log\log m)$.

2. The time for computing M and the new positions of $l(C')$ and $r(C')$:

Recall that for each list we maintain a pointer to the rightmost list element to the left of $l(C')$. We put these $t+1$ elements into a heap and proceed as follows: In each step we remove the rightmost element from the heap (say it was in list i) and insert it into M. Then we insert the preceding element of list i into the heap. The part of M to the right of $r(C')$ is computed similarly, the elements in between $l(C')$ and $r(C')$ can be obtained by scanning M_{t+1}. We get a running time of $O(|M_{t+1}| \log\log m + \log m)$ (inserting an element and removing the minimum element takes logarithmic time). The additional time for computing the new positions of $l(C')$ and $r(C')$ is then $O(|M|) = O(|M_{t+1}|)$.

Since each element can belong to M_{t+1} at most $O(\log m)$ times, and the overall number of calls to COLLAPSE is at most $n = O(m)$, the overall running time for computing M and updating $l(C')$ and $r(C')$ is $O(m \log m \log\log m + m \log m)$.

3. The time for merging two lists (if $t+1 > \lceil \log(2m) \rceil$):

This time is linear in the length of the longer list. Since the longer list grows by at least a factor $\frac{3}{2}$ when subject to merging, each element can belong to a longer list which is merged at most $\log_{\frac{3}{2}}(2m)$ times. Hence the overall running time for this step is $O(m \log m)$. □

6. Net weights

Both implementations discussed above become less efficient in the presence of netweights. In this case the number of successors or predecessors to be considered for updating $l(C')$ and $r(C')$ cannot bounded by the number of elements inserted into $M(C')$. In general one may have to consider all predecessors/successors, and the worst–case running time becomes $\Theta(n^2)$.

However, in the special case when all net weights are within a fixed range, say an interval $[a,b]$, then

one has to consider only $\lceil \frac{b}{a} \rceil$ predecessors and successors of $l(C')$ and $r(C')$ when inserting an element into $M(C')$. Hence the running times of the above implementations increase by a factor $O(\frac{b}{a})$. In many practical situations, this factor can be considered as bounded by a constant. In this case we still have a running time of $O(m \log m \log\log m)$.

In the rest of this paper we consider arbitrary netweights. We show how to implement the CLUMPING ALGORITHM to obtain a running time of $O(m \log^2 m)$ in the general case.

We store $M(C)$ as a list of ordered doubly–linked lists $L_0, \ldots, L_t$ with the following properties:

1. L_0 contains all elements of $M(C)$.
2. L_i $(i > 0)$ is a sublist of L_{i-1} containing the first and the last element of L_{i-1}. For each element e of L_i there is a pointer $\pi(e)$ to the corresponding element of L_{i-1}. For each two consecutive elements e, e' of L_i the number of elements between $\pi(e)$ and $\pi(e')$ in L_{i-1} is 1 or 2.
3. L_t consists of two elements.

So L_i arises from L_{i-1} by iteratively skipping one or two elements. Hence $|L_i| \leq \frac{1}{2}(|L_{i-1}| + 1)$, implying $t = O(\log m)$.

For two consecutive elements $e = (x, \sigma)$ and $e' = (x', \sigma')$ of L_i, $i = 0, \ldots, t$, we store the sum of all absolute values of the weights of the elements of $M(C)$ between e and e' (not counting e and e' themselves): we write $\tau_i(e, e') := \sum_{(y,\tau) \in M(C): x < y < x'} |\tau|$.

We shall use variables $sl(x)$ for the slope of the cost function to the right of x; at any stage we maintain $sl(-\infty) := \sum_{(x,\sigma) \in M(C)} \min\{\sigma, 0\}$ for each cell C.

These data structures can be built initially in $O(m \log m)$ time. We now show how to insert an element e into $M(C)$, maintaining the above data structures. We have lists $L_0, \ldots, L_t$ with properties 1,2,3; we assume $|L_0| = |M(C)| \geq 3$. Moreover we have numbers τ_i and $sl(-\infty)$.

INSERT$(e = (x, \sigma))$

① Insert e into L_0 at the appropriate place.

② **For** $i = 1, \ldots, t$:
  If there are neighbours e', e'' in L_i such that L_{i-1} contains three elements between $\pi(e')$ and $\pi(e'')$ in L_{i-1} **then** insert the middle element between e' and e'' into L_i.

③ **If** L_t has three elements **then** let L_{t+1} be a list consisting of the first and the last element of L_t and set $t := t + 1$.

④ Set $sl(-\infty) := sl(-\infty) + \min\{\sigma, 0\}$.

⑤ **For** $i = 1, \ldots, t$:
  Use τ_{i-1} and L_{i-1} to determine τ_i.

⑥ Let L_t be the list $(x_1, \sigma_1), (x_2, \sigma_2)$. Set $l_t := x_1$, $r_t := x_2$ and $sl(l_t) := sl(-\infty) + |\sigma_1|$.

⑦ **For** $i = t - 1, \ldots, 0$:
  Let $e_1 = (z_1, \zeta_1), \ldots, e_k = (z_k, \zeta_k)$ be the sublist of L_i from $\pi(l_{i+1})$ to $\pi(r_{i+1})$.
  Set $j := 1$.
  Repeat
    $j := j + 1$.
    $sl(z_j) := sl(z_{j-1}) + \tau_i(e_{j-1}, e_j) + |\zeta_j|$.
  Until $j = k$ or $sl(z_j) > 0$.
  Set $l_i := e_{j-1}$ and $r_i := e_j$.

⑧ **If** $sl(l_0) > 0$ **then** $r(C) := l_0$ **else** $r(C) := r_0$.
  If $sl(l_0) \geq 0$ **then** $l(C) := l_0$ **else** $l(C) := r_0$.

An easy induction shows that the variables $sl(x)$ in ⑥ and ⑦ contain the slope of the cost function to the right of x. We use the observation that $(x, \sigma) \in M(C)$ means that the slope of $cost_C$ increases at x by $|\sigma|$.

Before termination we always have $sl(l_i) \leq 0$ and $sl(r_i) > 0$. Hence the interval minimizing the cost function must be between l_i and r_i.

To check the running time, we first show how to implement ① in $O(\log m)$ time: Find in each list, beginning with L_t, two successive elements between which e has to be placed. By using the pointers π, this can be done in constant time per list.

Similarly, each iteration in ② takes constant time since we only have to consider the neighbours between e has been inserted. To find these, we can either use reverse pointers π^{-1} in addition to π, or we store the neighbours considered in ①.

So ② takes $O(\log m)$ time. Similarly, ⑤ takes $O(\log m)$ time. ③, ④ and ⑥ obviously take constant time only. Moreover, each iteration of ⑦ can be done in constant time as $k \in \{3, 4\}$.

So the running time of INSERT is indeed $O(\log m)$ as required. When merging C and C' we insert the elements of $M(C')$ into $M(C)$ (or vice versa if $|M(C)| < |M(C')|$); then each element is inserted at most $\log m$ times. This yields an overall running time of $O(m \log^2 m)$ of COLLAPSE and hence of the CLUMPING ALGORITHM, for general net weights.

We thank an anonymous referee of this paper for his useful remarks.

References

[1] M.R. Garey, D.S. Johnson, L. Stockmeyer: Some Simplified *NP*–Complete Graph Problems. Theoretical Computer Science 1 (1976), 237-267

[2] A.B. Kahng, P. Tucker, A. Zelikovsky: Optimization of Linear Placements for Wirelength Minimization with Free Sites. Proc. of the Asia and South Pacific Design Automation Conference, 1999, 241-244

[3] J. Vygen: Algorithms for Detailed Placement of Standard Cells. Proc. of the Conference Design, Automation and Test in Europe (DATE'98), IEEE 1998, 321-324