

Floorplanning representations and algorithms

Laurent Muller (laurent@spiral.dk)
Sathiamoorthy Subbarayan (sathi@itu.dk)
Thomas Antonson (anton@fsr.ku.dk)

19. november 2004

Indhold

1	Introduction	2
2	Floorplanning using sequence pairs	2
2.1	Introduction	2
2.2	Transforming Sequence pairs to placements	2
2.2.1	Graph based transformation	3
2.2.2	Longest common sequence based transformation	6
2.3	A simple algorithm for computing LCS	7
2.4	A faster algorithm for computing LCS	10
2.5	An $O(n \log \log n)$ algorithm for FAST-SP	12
3	Floorplanning using O-trees	12
3.1	L-, B- and LB-compact placements	12
3.2	Constraint graphs	12
3.3	O-trees	13
3.4	O-trees and placement	14
3.5	Number of O-trees	14
3.6	Conversion between O-trees and constraint graphs	16
3.7	Making an O-tree admissible	19
3.8	Floorplanning Algorithm using O-Tree	20
3.9	An $O(n^2)$ deterministic algorithm using an enhanced perturbation	23
4	Comparison	23

1 Introduction

The purpose of this paper is to present two representations for block placements: Sequence pairs and O-trees, and algorithms for evaluating the corresponding placements. Evaluation, and specially fast evaluation of placements are very important in the context of floorplanning, where you want to use local search algorithms to find a good solution to the placement problem.

After having presented the different algorithms associated with the two representations, we will make a small comparison of the two with respect to how fast they are. The paper is strongly based on the two papers [1] and [2].

We now define the PLACEMENT problem. Given a set $B = \{B_1, B_2, \dots, B_n\}$ of rectangular blocks, where every block lies parallel to the coordinate axes and each block B_i is given by a tuple (h_i, w_i) , where h_i and w_i is the height and width of the block B_i respectively. A placement $P = \{(x_i, y_i) | 1 \leq i \leq n\}$ is an assignment of coordinates to the lower left corners of the rectangular blocks such that no two blocks are overlapping. The placement problem consists of finding the placement which minimizes some objective function, which is typically the bounding box area or the approximate wirelength required for interconnections or a weighted combination of the two.

In this paper we only consider the problem of finding the bounding box of minimal area.

2 Floorplanning using sequence pairs

2.1 Introduction

In this section which is strongly based on the article [1] we will discuss a fast evaluation algorithm of placements represented by a sequence pair. The sequence pair representation was introduced by Murata et al. in [3]. We will not describe how to get from a placement to a sequence pair (see [3] for this), but only how to get from a sequence pair to a placement. Simulated annealing, tabu search or some other approach can then be used to find a good solution. The algorithms we present in this section are local search algorithms.

In [3] Murata et al. presented an $O(n^2)$ evaluation algorithm based on constraint graphs for translating sequence pairs into actual placements. We will first give an overview of this algorithm, and then dive more into depth with the two alternate evaluation algorithms described in [1]. In essence these two algorithms are alike, the only difference is that one uses a more sophisticated datastructure that improves the first algorithm's running time from $O(n^2)$ to $O(n \cdot \log n)$. The two algorithms are based on longest common subsequence computations which we will describe shortly.

2.2 Transforming Sequence pairs to placements

Definition 2.1 *Given n elements e_1, \dots, e_n a sequence is an ordering of the n elements ie. a sequence is some permutation of the n elements. We write a sequence as $\langle x_1 x_2 \dots x_n \rangle$ where there is a 1-1 correspondance between x_i 's and e_j 's $i = 1 \dots n, j = 1 \dots n$. A sequence pair is a pair of sequences, which we write (X, Y) where X and Y are two sequences.*

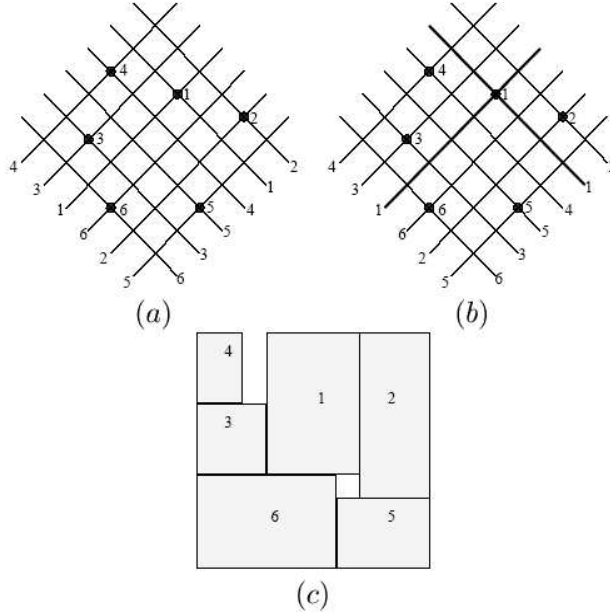


Figure 1: (a) The oblique grid for the sequence pair $\langle 4\ 3\ 1\ 6\ 2\ 5 \rangle, \langle 6\ 3\ 5\ 4\ 1\ 2 \rangle$. (b) The slope lines for the point $(1, 1)$. (c) The corresponding placement.

In the context of placement, each element of course corresponds to a certain module that needs to be placed.

We now describe what constraints a sequence pair (X, Y) implies between the different modules. First define the following sets:

$$\begin{aligned}
 M^{aa}(x) &= \{x'|x' \text{ is after } x \text{ in both } X \text{ and } Y\} \\
 M^{bb}(x) &= \{x'|x' \text{ is before } x \text{ in both } X \text{ and } Y\} \\
 M^{ab}(x) &= \{x'|x' \text{ is after } x \text{ in } X \text{ and } x' \text{ is before } x \text{ in } Y\} \\
 M^{ba}(x) &= \{x'|x' \text{ is before } x \text{ in } X \text{ and } x' \text{ is after } x \text{ in } Y\}
 \end{aligned}$$

Now given a module m with corresponding element x then all modules with corresponding element x' in M^{aa} , M^{bb} , M^{ab} or M^{ba} must be placed *left of* m , *right of* m , *below* m or *above* m respectively.

2.2.1 Graph based transformation

The sequence pair can be translated into an actual placement as follows (this is the algorithm described in [3]). Given an sequence pair (X, Y) draw an 45 degree oblique grid (see figure 1). Along the one side write the sequence X , and along the other write the sequence Y . Now the interesting points on the grid are the crossings (x_i, x_i) where $1 \leq i \leq n$. Call such a point x_i . For each such point the plane is divided by the two crossing slope lines into four cones (see figure 1). Define the four relations “is left to”, “is below”, “is right to” and “is above” as follows.

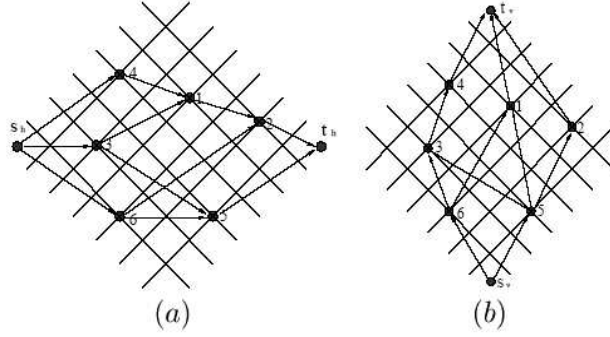


Figure 2: (a) The horizontal constraint graph $G_h(V, E)$ and (b) The vertical constraint graph $G_v(V, E)$ for the sequence pair $\langle 4\ 3\ 1\ 6\ 2\ 5 \rangle, \langle 6\ 3\ 5\ 4\ 1\ 2 \rangle$

$$\begin{aligned}
x_i \text{ is left to } x_j &\Leftrightarrow x_i \text{ is in the cone left of } x_j. \\
x_i \text{ is right to } x_j &\Leftrightarrow x_i \text{ is in the cone right of } x_j. \\
x_i \text{ is above } x_j &\Leftrightarrow x_i \text{ is in the cone above } x_j. \\
x_i \text{ is below } x_j &\Leftrightarrow x_i \text{ is in the cone below } x_j.
\end{aligned}$$

We define the relations to be nonreflexive ie. x_j does not lie in any of the cones formed by dividing the plane at x_j .

We can now prove the following lemma.

Lemma 2.2 *Given a sequence pair (X, Y) and two elements x_i and x_j , the following holds:*

$$x_i \in M^{bb}(x_j) \Leftrightarrow x_i \text{ is left to } x_j \quad (1)$$

$$x_i \in M^{ab}(x_j) \Leftrightarrow x_i \text{ is below } x_j \quad (2)$$

Proof We will only show (1) since the proof of (2) is equivalent.

" \Rightarrow ": Take the slop line from lower left to upper right going through x_j . By construction x_i will lie to the left of this line since x_i comes before x_j in the sequence X . Now add the sloping line from upper right to lower left going through x_j . Again by construction x_i must lie to the right of this line since x_i comes before x_j in Y . Now taking the intersection of these two areas it is clear that x_i must lie in the cone left of x_j ie. x_i is left to x_j .

" \Leftarrow ": x_i is left to $x_j \Rightarrow x_i$ is in the cone left of $x_j \Rightarrow X = \langle \dots x_i \dots x_j \dots \rangle \wedge Y = \langle \dots x_i \dots x_j \dots \rangle \Rightarrow x_i \in M^{bb}$. \square

We can now use Lemma 2.2 to construct the horizontal-constraint graph $G_h(V, E)$ as follows (see figure 2):

1. $V = \{s_h\} \cup \{t_h\} \cup \{v_i | i = 1, \dots, n\}$, where each v_i corresponds to the point x_i in the grid.
2. $E = \{(s_h, v_i) | i = 1, \dots, n\} \cup \{(v_i, t_h) | i = 1, \dots, n\} \cup \{(v_i, v_j) | x_i \text{ is left to } x_j\}$.
3. For each vertex v_i set its weight to the width of the corresponding module. s_h and t_h has weight zero.

The vertical-constraint graph is constructed similarly, by replacing the "is left to" with "is below". We now have two constraint graphs $G_h(V, E)$ and $G_v(V, E)$, which are vertex weighted directed acyclic graphs. Acyclicity follows from the fact that the two relations are transitive and nonreflexive. Assume a cycle existed in say G_h let v be one of the vertices in this cycle. The cycle would imply that v was left of itself.

To determine the coordinates of the blocks, we compute longest path trees in the two graphs.

We now prove the following

Theorem 2.3 *Let T_h be the longest path tree rooted at s_h in a horizontal-constraint graph $G_h(V, E)$. Let for each vertice $v \in T_h \setminus \{s_h, t_h\}$ the x -placement of the corresponding module be the weight of the path from the root of T_h to v . Let the y -placement of the modules be given by an equivalent construction on $G_v(V, E)$.*

Then the resulting placement is a correct placement optimal placement ie. no other placement exists without overlaps that satisfies the constraints given by the graphs and has a smaller bounding box.

Proof To see that the placement contains no overlaps, take two modules m_i and m_j with corresponding vertices v_i and v_j in the constraint graphs. Observe that there will always be an edge between v_i and v_j in exactly one of the two graphs (after maybe exchanging i and j). This follows from Lemma 2.2 since exactly one of the two cases will always be satisfied. Also observe that if two rectangular modules overlap, they will overlap in both x - and y -coordinates. It is thus enough to ensure that they do not overlap in one of the coordinates. Let (x^i, y^i) and (x^j, y^j) be the resulting (x, y) -placement of module m_i and m_j respectively. We now split into 2 cases:

1. There is an edge $e_{ij} = (v_i, v_j)$ in G_v but not in G_h .
2. There is an edge $e_{ij} = (v_i, v_j)$ in G_h but not in G_v .

Case 1 The edge e_{ij} is either part of T_h or not. In the first case $x^j = x^i + w_i$. In the second case $x_j \geq x_i + w_i$. In both cases there is no overlap of the two modules in x -coordinates and therefore no overlap in y neither.

Case 2 Same argument as for case 1 just for y .

We now need to show that the placement is optimal. The width and the height of the bounding box is determined by the length of the longest path from the source to the sink in $G_h(V, E)$ and $G_v(V, E)$ respectively. Lets look at $G_h(V, E)$. Every path from s_h to t_h through $G_h(V, E)$ corresponds to some horizontal ordering of the modules in the path. Every such path must be contained in the width of the bounding box which means that the longest such path must be contained therein. Or put differently the longest path in $G_v(V, E)$ is a lower limit of the width of the bounding box. The algorithm constructs a placement which had exactly this width, so it must be optimal in the x -direction. The same argument can be applied to $G_v(V, E)$ and the bounding box is thus as small as possible and the placement therefore optimal. \square

The construction of the constraint graphs runs in $O(n^2)$ time while the longest path computations can be done in $O(n + m)$ where n is the number of vertices and m the number of edges, which is bounded by $3n - 6$ in a planar graph. Thus the algorithm runs in $O(n^2)$ time.

2.2.2 Longest common sequence based transformation

As we saw in 2.2.1 the translation from sequence pair to actual placement can be done in $O(n^2)$ time based on constructing constraint graphs and then calculating longest paths in these graphs. We now look at another approach based on longest common subsequences as described in [1]. We start with a few definitions.

Definition 2.4 *A weighted sequence is a sequence whose elements are in a given set S , while every element $s_i \in S$ has a weight. Let $w(s_i) \geq 0$ denote the weight of s_i .*

Definition 2.5 *Given a sequence X , a subsequence Z is a sequence whose elements appear in X in the same order ie. if $Z = \langle z_1 z_2 \dots z_n \rangle$ then $X = \langle \dots z_1 \dots z_2 \dots \dots z_n \dots \rangle$.*

Definition 2.6 *Given two weighted sequences X and Y , a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .*

Definition 2.7 *The length of a common subsequence $Z = \langle z_1 \dots z_n \rangle$ is*

$$\sum_{i=1}^n w(z_i)$$

The longest common subsequence for a sequence pair is thus the common subsequence of the two sequences with the maximum length. In the following let **LCS** denote the longest common subsequence and $lcs(X, Y)$ denote the length of the longest common subsequence of X and Y .

Lemma 2.8 *Let there be given a sequence pair (X, Y) and let $G_h(V, E)$ be the associated horizontal-constraint graph. Then*

$$v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_k \text{ path in } G_h(V, E) \Leftrightarrow \langle v_1 v_2 \dots v_k \rangle \text{ is a common subsequence of } (X, Y),$$

where we have removed a possible s_h at the beginning of the path and a possible t_h at the end.

Proof Here we use the fact that if there is an edge between two vertices $v_i, v_j \in G_h(V, E)$ then by construction v_i is left of v_j . We now have:

$$\begin{aligned} p = v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_k \text{ path in } G_h(V, E) &\Leftrightarrow \\ v_i \text{ is left to } v_{i+1}, \quad i = 1, \dots, k-1 &\Leftrightarrow \\ v_i \in M^{bb}(v_{i+1}), \quad i = 1, \dots, k-1 &\Leftrightarrow \\ X = \langle \dots v_i \dots v_{i+1} \dots \rangle \wedge Y = \langle \dots v_i \dots v_{i+1} \dots \rangle, \quad i = 1, \dots, k-1 &\Leftrightarrow \\ X = \langle \dots v_1 \dots v_2 \dots \dots v_k \dots \rangle \wedge Y = \langle \dots v_1 \dots v_2 \dots \dots v_k \dots \rangle &\Leftrightarrow \\ Z = \langle v_1 v_2 \dots v_k \rangle \text{ is a common subsequence of } (X, Y). & \end{aligned}$$

□

Similarly we can prove

Lemma 2.9 *Let there be given a sequence pair (X, Y) and let $G_v(V, E)$ be the associated horizontal-constraint graph. Then*

$v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_k$ path in $G_h(V, E) \Leftrightarrow \langle v_1 v_2 \dots v_k \rangle$ is a common subsequence of (X^R, Y) ,

where X^R is the reverse of X and where we have removed a possible s_h at the beginning of the path and a possible t_h at the end.

Now we are ready to prove the following lemma.

Lemma 2.10 *Let B be a blockset and $\forall b \in B : w(b) = \text{width of } b$. Let $b \in B$ and let (X, Y) be a sequence pair of the form $(X, Y) = (X_1 b X_2, Y_1 b Y_2)$. Then $\text{lcs}(X_1, Y_1)$ is the x-coordinate of the block b in an optimal placement and $\text{lcs}(X, Y)$ is the total width of an optimal block placement.*

Proof We again need to show that the placement is correct and optimal. Given Theorem 2.3 it is enough to prove that the x-placements produced are the same as for those produced by the graph-based algorithm described in section 2.2.1. That is, we need to prove that the longest path from vertex s_h to the vertex v_b representing b in graph $G_h(V, E)$ is equal to $\text{lcs}(X_1, Y_1)$.

Let $p = s_h \rightsquigarrow v_1 \rightsquigarrow v_2 \rightsquigarrow \dots \rightsquigarrow v_n \rightsquigarrow v_b$ be the the longest path in $G_h(V, E)$ from s_h to v_b . First note that the same path p' ending at v_n must have all nodes lying X_1 and Y_1 . Now by Lemma 2.8 the sequence $Z = \langle v_1 \dots v_n \rangle$ is a common subsequence of (X_1, Y_1) . Now because of the way w was chosen, the weight of the subsequence Z must be equal to the length of the path p' . Z must be a longest common subsequence since again by Lemma 2.8 if there were a longer subsequence Z' of (X_1, Y_1) we would have a corresponding path in $G_h(V, E)$ where if you added v_b to the end would be longer than p . That is $\text{lcs}(X_1, Y_1) = \text{length}(p)$. That is every block b is placed at the same x-coordinate as with the graph algorithm. \square

Similarly we can conclude the following lemma

Lemma 2.11 *Let B be a blockset and $\forall b \in B : w(b) = \text{width of } b$. Let $b \in B$ and let (X, Y) be a sequence pair of the form $(X, Y) = (X_1 b X_2, Y_1 b Y_2)$. Then $\text{lcs}(X_1, Y_1)$ is the x-coordinate of the block b in an optimal placement and $\text{lcs}(X, Y)$ is the total width of an optimal block placement.*

Now by the two preceding lemmas we have the following theorem.

Theorem 2.12 *LCS computation can be applied to determine the x and y coordinates of each block and the width and height of the block placement for a given sequence pair. Therefore, the optimal packing can be obtained by computing LCS.*

2.3 A simple algorithm for computing LCS

In this section we describe the algorithm 1 proposed in [1]. It is pretty straight forward and has a running time of $O(n^2)$ for evaluation a sequence pair placement. In the next section we will describe a more sophisticated algorithm.

Let there be given a blocklist $B = \{1, \dots, n\}$ and the input sequence pair (X, Y) . The following datastructures are used in the algorithm. A block position array $P[b], b = 1, \dots, n$ is used to record the x - or y -coordinate depending on whether $w(b)$ represents the width or the height of b respectively. The indices of the block b in X and Y are recorded in the array $match[b], b = 1, \dots, n$ such that $match[b].x = i \Leftrightarrow b$ has index i in X and $match[b].y = i \Leftrightarrow b$ has index i in Y . Finally the length array $L[1..n]$ is used to record the length of candidates of the longest common subsequence. That is $L[i]$ records the longest substring seen so far, where $Y[i]$ is the last element. The algorithm is as follows (see figure 3):

Algorithm 1:

```

1 Initialize_Match_Array match;
2 Initialize_Length_Array L with 0;
3 for i = 1 to n
4   do b = X[i];
5     p = match[b].y;
6     P[b] = L[p];
7     t = P[b] + w(b);
8     for j = p to n
9       do if(t > L[j])
10          then L[j] = t;
11          else break;
12 return L[n];

```

We now show that the algorithm is correct.

Theorem 2.13 *Algorithm 1 correctly returns $lcs(X, Y)$ and the position of each block is correctly recorded in $P[1, ..n]$.*

Proof The proof is an induction over i to prove that the positions of the blocks in the subsequence $X[1..i]$ is recorded correctly. That is the placements recorded for each block b is the same placement as the one computed by solving the corresponding lcs problem.

For the base case where $i = 1$ this holds trivially since $L[1..n] = 0$ and therefor $P[1] = 0$ which is the correct placement for the first block.

Now assume that the positions off all the blocks in the subsequence $X[1..k]$ are placed correctly. Let $i = k + 1$ and let b_{k+1} denote the block $X[k + 1]$. Also let p_{k+1} denote the index of b_{k+1} in the sequence Y . That is

$$\begin{aligned}
match[b_{k+1}].x &= k + 1 \\
match[b_{k+1}].y &= p_{k+1}
\end{aligned}$$

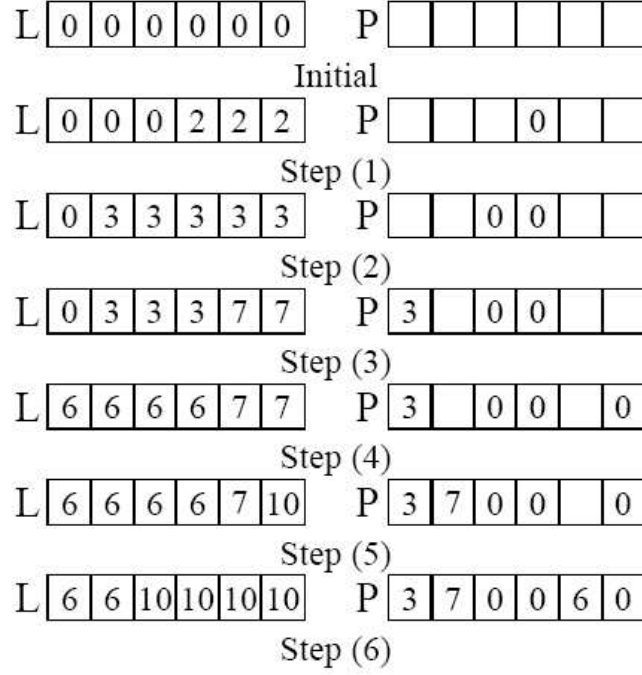


Figure 3: A sample run of algorithm 1 on the running example $\langle 431625 \rangle, \langle 635412 \rangle$. The $w(i)$ here refers to corresponding width. P : position of blocks

Now according to lemma 2.10 the position of block b_{k+1} is $lcs(X[1..k], Y[1..(p_{k+1} - 1)])^1$. We want to show that the value that gets recorded into $P[b]$ is actually this value. Let the last element of the LCS for this problem be b_l . Thus, $match[b_l].x \leq k$ and $match[b_l].y \leq p_{k+1}$. We now have that the position of b_{k+1} is:

$$lcs(X[1..k], Y[1..(p_{k+1} - 1)]) = lcs(X[1..(match[b_l].x - 1)], Y[1..(match[b_l].y - 1)]) + w(b_l)$$

Note that $lcs(X[1..(match[b_l].x - 1)], Y[1..(match[b_l].y - 1)])$ is the position of the block b_l .

Now since all the positions of the blocks in $X[1..k]$ are correctly recorded by the induction assumption, the value $lcs(X[1..(match[b_l].x - 1)], Y[1..(match[b_l].y - 1)])$ must be the same as the one recorded in $P[b_l]$. That is

$$P[b_l] = lcs(X[1..(match[b_l].x - 1)], Y[1..(match[b_l].y - 1)])$$

We then have that the placement of b_{k+1} is given by $P[b_l] + w(b_l)$.

The code from line 7 to 11 guarantees that this position is recorded in $L[p_{k+1}]$. Because of line 9, it is clear that when the algorithm ends $L[n]$ will have recorded the largest common substring seen.

It can be seen that the values in the locations of L monotonically rise. The locations of L can hence be divided into intervals, where each interval has same value stored in them. It can also be seen that just the index of the first location in an interval and the corresponding value can

¹if we were solving for the y-position we would use lemma 2.11

be used to define the corresponding interval. Now, one can use a *balanced search tree*(BST) to manage the intervals efficiently. The following algorithm uses the BST-based approach and improves the time complexity of the previous algorithm.

2.4 A faster algorithm for computing LCS

Algorithm 2:

```

1 Initialize_Match_Array match;
2 Initialize BST with a node (0,0);
3 for  $i = 1$  to  $n$ 
4   do  $b = X[i]$ ;
5      $p = match[b].y$ ;
6      $P[b] = \text{find\_BST}(p)$ ;
7      $t = P[b] + w(b)$ ;
8     insert( $p, t$ ) to BST;
9     discard the nodes with greater index than  $p$  and less length than  $t$ ;
10 return find_BST( $n$ );
```

find_BST(p):

```

1 find the greatest index in BST which is less than  $p$ ;
2 return the corresponding length;
```

Theorem 2.14 *Algorithm 2 functions like algorithm 1, and requires only $O(n \log n)$ time and $O(n)$ space.*

Proof It is sufficient to prove that **find_BST(p)** always returns the corresponding $L[p]$ in algorithm-1. In step 10 of algorithm-1, the $L[j]$ is updated only if $t > L[j]$. As the values in L array are monotonically rising, the locations that are updated by step 10 of algorithm-1 are those with index more than p and the corresponding L value less than t . In case of BST-based approach of algorithm-2, exactly the nodes corresponding to the locations (except p , whose corresponding node is inserted) that are updated in algorithm-1 are those deleted from the BST (is present). Also, in any iteration of algorithm-1, the locations of L , which are not unchanged have their corresponding nodes in BST, if present, unaffected. Hence, when looking up for a value in BST, by definition of **find_BST(p)**, it returns the corresponding value that would have been stored in $L[p]$ by algorithm-1.

Another simpler explanation would be to observe that, the locations of L could be divided into intervals, where each interval has same value stored in them. It can be seen that only the

index of the first location in each interval is stored in BST and the corresponding length is equivalent to the value of the interval. This proves that the BST-based approach is equivalent to algorithm-1 and hence, results in $lcs(X, Y)$. \square

Now, we use amortized analysis to prove that the algorithm takes $O(n \log n)$ time. Line 6 and Line 8 of the algorithm needs $O(\log n)$ time. In line 9, the BST deleted some elements and rebalances the BST after every deletion. This needs $O(\log n)$ time. Since, there can be maximum n insertions into BST in any run of the algorithm, there can be maximum n deletions and hence the amortized cost contributed by line 9 to the algorithm is $O(\log n)$. As, line 3 results in n iterations, the time complexity of algorithm-2 is $O(n \log n)$.

The space requirement is $O(n)$, as each one among the constant number of datastructures used in the algorithm needs at most $O(n)$ space.

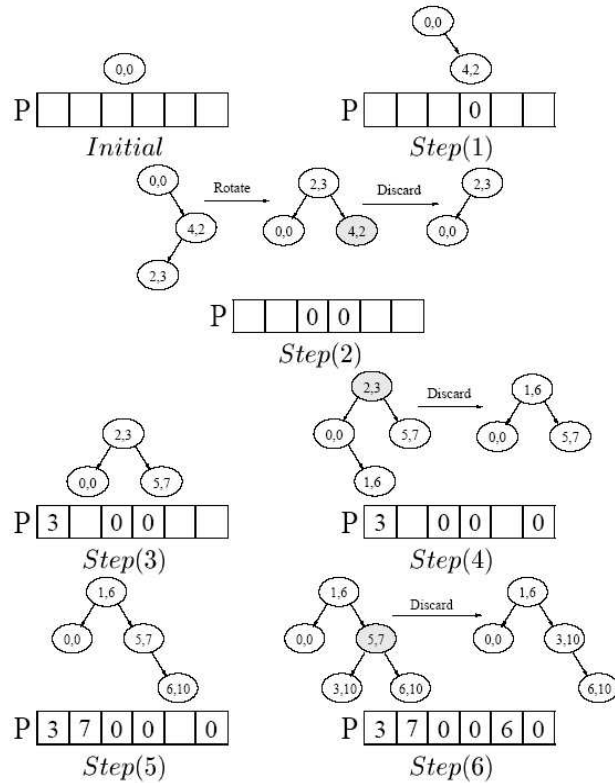


Figure 4: A sample run of the BST-based LCS computation algorithm on the running example $(\langle 431625 \rangle, \langle 635412 \rangle)$. The $w(i)$ here refers to corresponding width. P : position of blocks

Figure 4 shows the steps taken by algorithm-2 on the running example $(\langle 431625 \rangle, \langle 635412 \rangle)$.

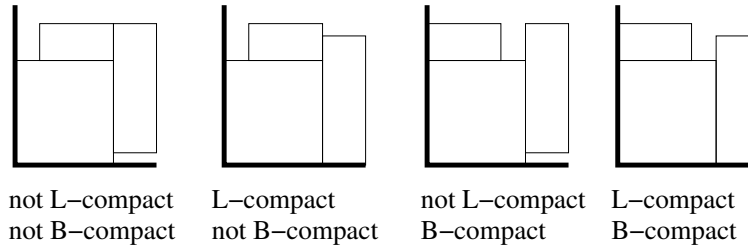


Figure 5: Illustrating L-compactness and B-compactness

2.5 An $O(n \log \log n)$ algorithm for FAST-SP

An improved $O(n \log \log n)$ algorithm for fast sequence pair evaluation, which uses a special priority queue with bound n , was presented in [7]. It was also shown in that paper that the FAST-SP can even handle additional constraints like pre-placed modules at the same worst-case complexity. The discussion of that method being beyond the scope of this report, the interested reader is referred to [7].

3 Floorplanning using O-trees

3.1 L-, B- and LB-compact placements

A placement is L-Compact iff there is no block which can be moved downwards with the other block fixed without causing the moved block to overlap with some other block or moving outside the bounding rectangle. In figure 1 the third and fourth placements is L-compact.

In a similar fashion a placement is L-Compact iff there is no block which can be moved to the left with the other block fixed without causing the moved block to overlap with some other block or moving outside the bounding rectangle. In figure 5 the second and fourth placement is B-compact.

A placement is LB-compact if and only if the placement is both L- and B-compact. The fourth placement in figure 5 is LB-compact. To make an arbitrary placement LB-compact we can iteratively move blocks down and left as much as possible without causing overlap and keeping all blocks within the bounding rectangle. Note that the bounding rectangle of the final LB-compact placement will be equal to or smaller than the original. A placement which is LB-compact is admissible.

3.2 Constraint graphs

A constraint graph is a graph where each block is a vertice and there are additional four vertices called B, T, L and R for bottom, top, left and right edge of the smallest rectangle covering all the blocks. Given two vertices there will be an edge between them iff there can be placed a vertical or horizontal line between the corresponding blocks without passing other blocks. This means that constraint graphs can only describe slicing placements.

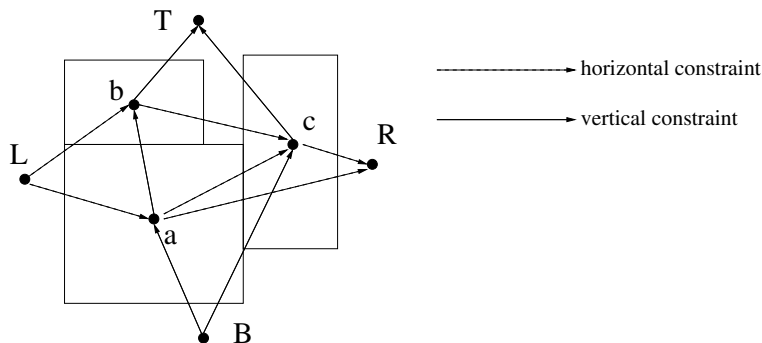


Figure 6: A constraint graph

If we only consider edges between two nodes going from left to right and from bottom to top and divides the edges in horizontal and vertical constraints called E_h and E_v , we get two planar graphs called G_h and G_v . Recall that a planar graph contains at most $3|V| - 6$ nodes for $|V| > 3$.

The weight of an edge between two vertices is equal to zero iff the two corresponding modules are adjacent to each other. The weight is positive and equal to the distance between the blocks if the two blocks are not adjacent. In figure 6 only the edges (b, c) , (a, R) and (B, c) have weight different from 0. Note that the placement in figure 6 is L-compact, but not B-compact. A placement is fully described by a vertical and horizontal constraint graph and every placement can be described by such a pair of constraint graphs.

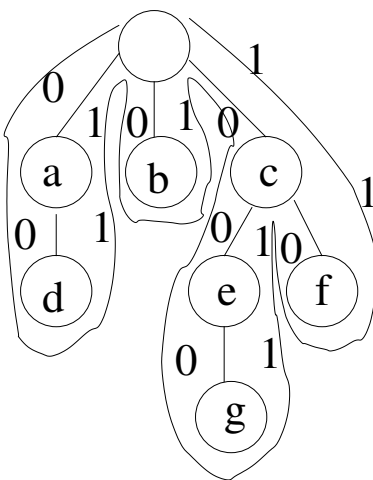
3.3 O-trees

A O-tree is described by a data structure (\mathbf{T}, π) where the tree topology \mathbf{T} is a list of descents and ascents, and π is a list of labels in the order a depth first search visits the nodes in the tree. One can create the tree from data by starting at the root and making the first label the subnode of the root. For each bit following the first (which has to be a 0, for a descent) create a new subnode if the bit is a 0 and place the next label at the new subnode. If the next bit is a 1, ascent to the parent node and execute the next bit. Repeat until there are no more labels left, in which case the rest of the bits has to be ones so one ends at the root node when the bit list is empty.

To create the data structure from the tree, one has to start at the root, and descent to the first subnode while placing a 0 in the bit string and the label of the subnode in the list of label. If the current node has got an unvisited subnode place the next subnodes label in the label list and place a 1 in the bit list and proceed from the subnode. If the current node has no unvisited subnodes place a 0 in the bit list and proceed from the parent of the current node.

In figure 7 a tree is shown with the corresponding data structure shown next to the place where the data is constructed from.

A n node tree requires n labels and to make the n labels distinct one has to use $\lceil \lg n \rceil$ per node. To store the bit list containing n ascents and descents $2n$ is needed, but the first descent and the last ascent are given, so they can be ignored giving a total requirement of $2n - 2$ bit.



Figur 7: The tree representing the encoding $(00110100011011, adbcegf)$

3.4 O-trees and placement

A horizontal O-tree corresponds to a placement where the root is the left side of the bounding rectangle of the placement. The block corresponding to a node has all the blocks corresponding to its subnodes lying to the right of it. Further its right side x -coordinate is equal to its childrens rightside x -coordinates. When describing the placement in an O-tree one makes sure that different subtrees are ordered such that the ordering of blocks in the label list describes the ordering of their y -coordinates.

If given an horizontal O-tree one can transform it to a placement by visiting it in a depth first search, placing the root in the lower, left corner and when encountering a node in the O-tree add it to the placement with the x -coordinate of its left side equal to the x -coordinate of the right side of its parent node and as low as possible. As one always places a block as low as possible the placement will always be B-compact, but not necessarily L-compact which is shown in figure 8. A vertical O-tree on the other hand will always be L-compact, but not necessarily B-compact.

An O-tree is said to be admissible if it transforms to admissible placement. The placement in figure 8 is admissible, but the counterpart with the same constraint graph in figure 9 is not admissible.

3.5 Number of O-trees

The number of O-trees is equal to the number of permutations of the label list, which is $n!$ times the number of possible bit strings which is $\frac{1}{n} \binom{2n-2}{n-1}$.

The second term can be reformulated and by using the Stirling appoximation:

$$\frac{1}{n} \binom{2n-2}{n-1} \approx \frac{4^{n-1}}{n\sqrt{\pi n}} + o\left(\frac{4^{n-1}}{n^{2.5}}\right) = O\left(\frac{4^{n-1}}{n^{1.5}}\right)$$

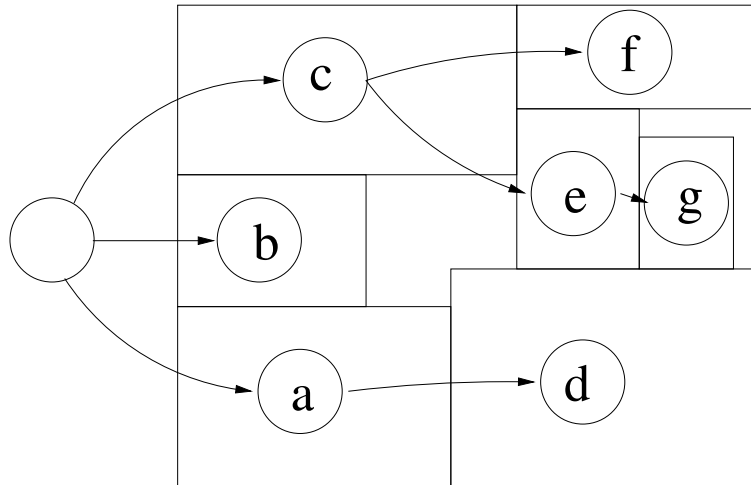


Figure 8: The tree representing the encoding $(00110100011011, adbcegf)$ and the corresponding placement

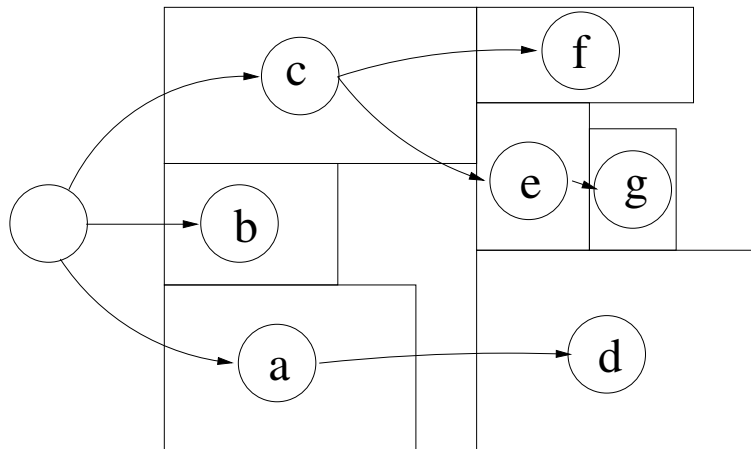


Figure 9: A non-admissible counterpart to the O-tree and placement in figure 4

3.6 Conversion between O-trees and constraint graphs

When one needs to convert an horizontal O-tree to the corresponding constraint graph, one lets the root node of the O-tree have the x -coordinate 0 and the width 0. Then we place every child of a node on the right side of the parent with no space in between. If B_i is the child of B_j , let: $x_j = x_i + w_i$

Remember when a node got more than one child, the permutation π decides the vertical position of the children, so the first child mentioned in π (which is the successor of the parent in π) will be placed beneath the next children of the parent mentioned in π . In figure 8 one can see that e is placed beneath f because it comes before f in the permutation $adbcegf$.

It is generally true, so when two blocks are overlapping in the x -direction their mutual order in the permutation decides their mutual order in the y -direction. So after one has figured out the x -coordinate of each block, one figures out the y -coordinate of a block in the order that they are mentioned in the permutation.

The y -coordinate is the height of the current block plus the largest y -coordinate of all of the blocks mentioned before the current block in the permutation and having an overlap in the x -coordinate with the current block. If there is no overlapping blocks and the current is given the y -coordinate 0.

For a vertical O-tree the procedure is the same, but one starts by placing the root node of the tree at y equal to 0 and giving each block an y -coordinate according to its own height and the placement of its parent. After that one assigns the x -coordinates as one did the y -coordinates for the horizontal O-tree. Note that the algorithm always transforms a vertical O-tree to a B-compact placement and a horizontal O-tree to a L-compact O-tree.

The algorithm beneath transforms an O-tree to a pair of constraint graph and a placement, by maintaining a contour graph:

Algorithm OT2OCG

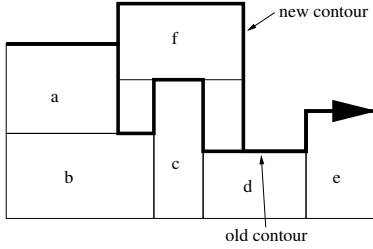
Input: O-tree (\mathbf{T}, π) (where $|\mathbf{T}| = 2n - 2$ and $|\pi| = n$)

Output: orthogonal constraint graph $G = (V, E)$ and a placement x, y (where $|x| = n$ and $|y| = n$)

```

perm = 1 // Entry in the permutation label list
contour = NULL
current_contour = 0
for code = 0 to 2n-2
    if  $\mathbf{T}[\text{code}] = 0$  // We consider only the descents
        current_block =  $\pi[\text{perm}]$ 
        if current_contour != 0
            then  $x[\text{current\_block}] = x[\text{current\_contour}] + w[\text{current\_block}]$ 
            else  $x[\text{current\_block}] = 0$ 
         $y[\text{current\_block}] = \text{find\_max\_y}(\text{contour}, \text{current\_block})$ 

```



Figur 10: Illustrating how the contour changes, when adding the next block

```

update_constraint_graph(G, contour, current_block)
update_contour(contour, current_block)
current_contour = current_block
else current_contour = prev[current_contour]

```

This algorithm uses the fact that when finding the y -coordinate of a block, the running time will be linear in the number of blocks already placed, which is bounded by the total number of blocks, but the running time can be amortized to a constant by maintaining a contour of the blocks already placed. This contour structure is a double linked list, where the entries is ordered in the order they are encountered following the contour from left to right.

In figure 10, when wanting to insert the block f , one can see that the suggested placement of f suggest the y -coordinate will be determined by the blocks b , c and e , which is true.

The algorithm `Algorithm OT2OCG` have a linear running time in the number of blocks, because the `for` loop executes $2n - 2$ times. In each loop we perform some conditional statements, `find_max_y`, `update_constraint_graph` and `update_contour`.

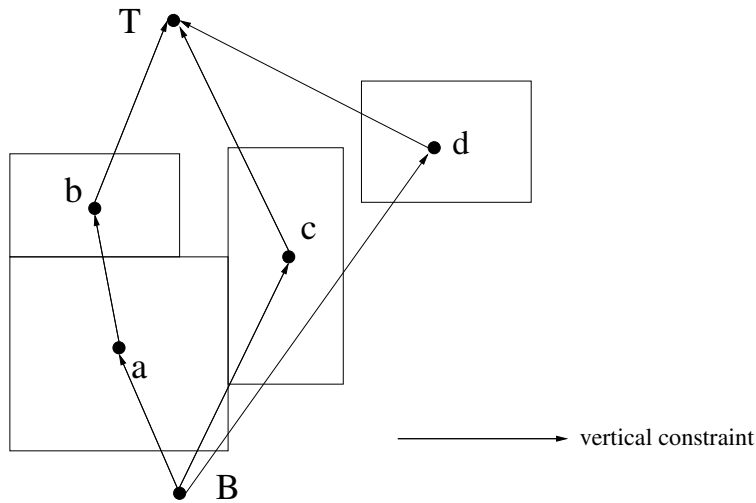
The conditional statements clearly has constant running time.

`find_max_y` has a constant running time, because one has the contour point, where the next block should be inserted. Then place the block at the highest y -coordinate of the contour points overlapping with the block in the x -coordinate. After inserting the algorithm uses `update_contour` to update the contour as suggested in figure 10, which can be done in amortized constant time for a double linked list.

`update_constraint_graph` builds up the constraint graphs by inserting the block as a vertex, when the block is placed. Each block is placed once and then the edges from existing nodes corresponding to already placed blocks are added. When the algorithm terminates the constraint graphs are still planar there will at most $3n - 6$ edges and as the algorithm never removes edges it is linear in the number of blocks, leading to the conclusion that the whole algorithm is running in linear time.

The algorithm for converting a constraint graph to an O-tree works as follows. It finds a SPST in one of the constraint graphs, which will contain only zero weight edges visiting all the nodes. Note that this algorithm will not work on the constraint graph in figure 11, as there is no zero weight edge leading to the block d even though the edge from d to T has zero weight.

The algorithm can be modified so it can handle cases with 'floating' blocks. After having run the original algorithm one should examine the edges, starting with the one with lowest weight,



Figur 11: A constraint graph the algorithm cannot convert, unless the algorithm is modified

and when it finds a nonzero weight edge leading from a marked node to an unmarked node it should add it to the O-tree, mark it and proceed. As the number of edges is linear in the number of blocks, because the constraint graph is a planar graph, the running time will still be linear. Note that this problem will not pop up in the **AOT** algorithm nor in the **deterministic algorithm**.

This algorithm uses a depth first search, which uses less memory while having the same running time as the traditional breadth first. Note that a L-compact constraint graph will be converted to a horizontal O-tree with zero weight edges and a B-compact constraint graph will lead to a vertical O-tree with all edges having weights zero.

Algorithm CG2OT

Input: constraint graph $G = (V, E)$

Output: O-tree (\mathbf{T}, π) (where $|\mathbf{T}| = 2n - 2$ and $|\pi| = n$)

set all marks to false

perm = 0

code = 0

DFS-traversal of the graph

$n = \text{current_node}$

$p = \text{parent}[n]$

if !mark[n] && $w[e(p, n)] = 0$

then

mark[n] = true

```

T[perm++] = 0
π[code++] = n
for c ∈ children[n]
    traverse(c)
T[perm++] = 1

```

As the algorithm visits each node in the constraint graph once and only doing operations with constant running time, the total running time is linear in the number of blocks.

3.7 Making an O-tree admissible

If one encounters an O-tree that is nonadmissible one can make it admissible by using the fact that converting a horizontal O-tree always yeilds a L-compact placement and the algorithm transforms a vertical O-tree to a B-compact placement. If one repeats those actions one will obtain an admissible O-tree in the end.

The algorithm look like this:

Algorithm AOT

Input: O-tree T

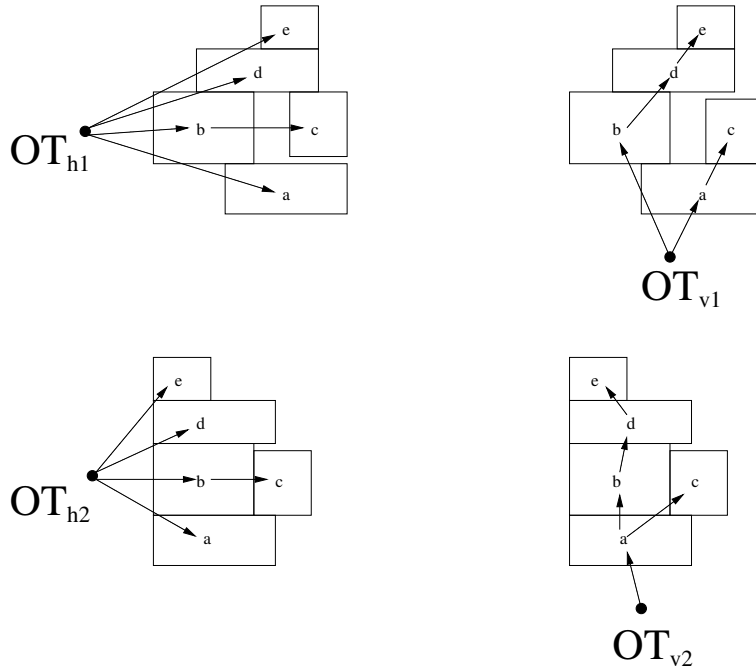
Output: Admissible O-tree T

```

change = true
while changed
    changed = false
     $G_y = \text{OT2OCG}(T)$ 
     $T_y = \text{CG2OT}(G_y)$ 
     $G_x = \text{OT2OCG}(T_y)$ 
     $T_x = \text{CG2OT}(G_x)$ 
    if  $T \neq T_x$ 
        then
             $T = T_x$ 
            change = true
return  $T$ 

```

According to [2] the **Algorithm AOT** has linear worst case running time. In figure 12 can be seen how the **Algorithm AOT** transforms the horizontal O-tree (0101001101, $edbca$) via a constraint graph to the vertical O-tree (0011000111, $acbde$). It transforms to the horizontal O-tree (0101001101, $edbca$), which transforms to the vertical O-tree ($acbde$, 0010001111)



Figur 12: Making an O-tree admissible

3.8 Floorplanning Algorithm using O-Tree

This section explains a deterministic floorplanning algorithm based on the O-Tree representation introduced in the previous sections. Given a fixed sequence of blocks to be placed the algorithm deterministically finds a placement. We will also explain a placement construction algorithm which is a variant of the deterministic floorplanning. The construction algorithm can be used to construct a heuristic initial placement. Search strategies like simulated-annealing, genetic-algorithms, and tabu-search can be used to find a sub-optimal solution from the placement given by the construction algorithm. The construction algorithm is also deterministic for a fixed sequence of blocks.

Lets first explain three procedures which will be used by the deterministic floorplanning and the construction algorithm.

In the Cost function, w_1 and w_2 are weight for the area and wirelength respectively. If either one of area and wirelength alone needs to be considered, one could assign zero value to the weight of the not-required value. Assigning normalized 0.5 to both of them would give equal weight to both values: A and W . The normalization is needed in order to ensure that even though A and W are measured in different units they will be given same priority.

Cost Function

Input: Original O-tree (T, π) , Interconnection details (N)

Output: Cost of the corresponding placement C

Using AOT obtain an admissible O-tree T'

Using OT2OCG obtain a placement P for (T', π)

Find the area A of the smallest bounding rectangle of the placement P

Using interconnection details in N , find the wirelength W required by P using half-perimeter method.

$$C = w_1 * A + w_2 * W$$

return C

The delete function removes specified block from the O-tree.

Delete Funtion

Input: Original O-tree (T, π) and a block to be removed B_i

Output: Removes the block B_i from the input O-tree

Delete the two bits corresponding to B_i in T .

Remove the block B_i from the sequence π .

return

The *insert* function is similar to Delete function, but adds a block at a specified position of an O-tree.

The perturb function moves a specified block to the position in the O-tree where the cost function is minimum.

Perturb Funtion

Input: Original O-tree (T, π) and a block B_i

Output: New O-Tree with B_i moved to a best possible position

Remove $((T, \pi), B_i)$

The current O-tree has $n-1$ blocks. *Insert* the B_i block at a position in the current O-tree where the cost is minimum.

Repeat the above two steps on the corresponding orthogonal O-tree.

return

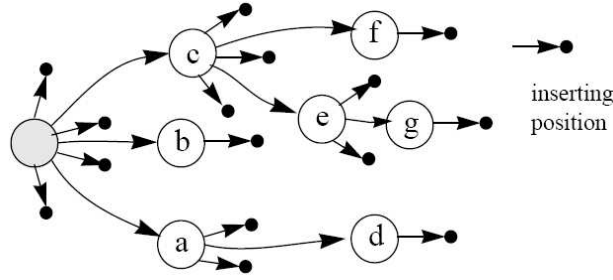


Figure 13: Possible insertion positions while perturbing

There will be $2n-1$ possible insertion positions in any O-tree with n nodes. In Fig. 13 an example O-tree with 8 nodes and the corresponding 15 possible insertion positions are marked. The perturbing might result in a non-admissible O-tree. As the cost function by default converts any input O-tree into an AOT, the perturbation function need not worry whether the perturbed O-tree is AOT or not.

Now, let's explain the deterministic floorplanning algorithm, which given a fixed sequence of the blocks finds a deterministic placement which optimal with respect to the heuristic strategy used by it. The deterministic algorithm uses the construction algorithm to obtain an initial placement. The details about the initial construction algorithm follows.

Deterministic Floorplanning Algorithm

Input: A sequence of blocks with corresponding width, height, and πn positions. Input-Output (I/O) pad positions. A list of nets, where each net is a set of size atleast two and contains elements from the set of input blocks and I/O pins.

Output: Position for each blocks

Obtain an initial O-tree T using the Construction algorithm.

$\text{mincost} = \text{Cost}(T, \pi)$

$\text{min_T} = (T, \pi)$

For each block B_i

$\text{Perturb}((T, \pi), B_i)$

$\text{newcost} = \text{Cost}(T, \pi)$

if ($\text{newcost} < \text{mincost}$)

$\text{min_T} = (T, \pi)$

$\text{mincost} = \text{newcost}$

set $(T, \pi) = \text{min_T}$

Output placement for (T, π)

The above deterministic algorithm heuristically perturbs the O-tree in a fixed sequence and results selects a best solution among the configurations reached through the perturbations. The advantage of having a deterministic algorithm may be that its implementation is simple and easy (as specified in [2]). But, the above algorithm will clearly result *only* in a local optimum value. But that's the cost for having a deterministic procedure. The search strategies like simulated annealing have more chance for resulting in better placement, if not in a global optimum. The deterministic algorithm has two nested loops. Each loop has an $O(n)$ time complexity. The cost function inside the two nested loops has an $O(n)$ complexity. Hence, the overall time complexity is $O(n^3)$

Constructive Algorithm

Input: A sequence of blocks with corresponding width, height, and πn positions. Input-Output (I/O) pad positions. A list of nets, where each net is a set of size atleast two and contains elements from the set of input blocks and I/O pins.

Output: An O-tree (T, π) representing a heuristic initial placement.

set $T = \{\}$, $\pi = \langle \rangle$

Same as the main loop in the deterministic floorplanning algorithm, where replace Cost by partialCost function

Output (T, π)

Similar to the deterministic algorithm, the constructive algorithm also has $O(n^3)$ time complexity.

3.9 An $O(n^2)$ deterministic algorithm using an enhanced perturbation

An $O(n^2)$ deterministic algorithm for floorplanning using the O-tree representation was presented in [4]. The basic idea is to reduce the complexity of the perturb function by approximating the best insertion position. When each block is perturbed, rather than doing $O(n)$ insertions, the enhanced perturbation procedure estimates the cost of all possible positions in $O(n)$ time. This brings down the deterministic procedure complexity to $O(n^2)$. Although the approximation might result in poorer placements, the authors of [4] have claimed using experimental results that the tradeoff is worth making. As a detailed explanation of enhanced perturbation procedure is beyond the scope of this report, the interested reader is referred to [4].

4 Comparison

In this section, as in [5] and [6], we compare the three different non-slicing floorplan representations: Sequence-Pair, Fast Sequence-Pair and O-Tree.

Definition 4.1 P-admissible[3]: A representation is P-admissible , if it satisfies the following properties.

1. the number of solutions represented is finite.
2. every solution represented is feasible. Hence, given a P-admissible representation, there is no need to check whether it is valid or not. For example, in case of O-Tree one need to have an admissible O-Tree before evaluating the placement. Hence, O-Tree is not P-admissible
3. rectangle packing and hence the evaluation can be done in polynomial time.
4. the best placement is always in the solution space.

In table 1, a comparison of the representations is listed. As can be inferred from the table, the advantage of O-Tree is that it just needs linear time for evaluation, while its disadvantage is that it is not P-admissible.

Table 1: Comparison of different representations [5] [6]

Representation	SP[3]	FAST-SP[1]	O-Tree[2]
Representation size (bits)	$2n(\log n)$	$2n(\log n)$	$n(2 + \log n)$
Is P-admissible?	Yes	Yes	No
Need Sequence encoding?	Yes	Yes	Yes
Construct constraint graphs for packing?	Yes	No	Yes
Time for Evaluation	$O(n^2)$	Amortized $O(n \log n)^*$	Amortized $O(n)$

*Improved to amortized $O(n \log \log n)$, using advanced datastructures, in [7].

Litteratur

- [1] Xiaoping Tang, Ruiqi Tian, D.F. Wong, *Fast Evaluation of Sequence Pair in Block Placement by Longest Common Subsequence Computation*, DATE (2000)
- [2] Pei-Ning Guo, Toshihiko Takahashi, Chung-Kuan Cheng, and Takeshi Yoshimura: *Floorplanning Using a Tree Representation*. In IEEE Transaction on CAD, **20.2**, (2001) 281-289
- [3] Hiroshi Murata, Kunihiro Fujiyoshi, Shigetoshi Nakatake, and Yoji Kajitani: *VLSI Module Placement Based on Rectangle-Packing by the Sequence-Pair*, In IEEE Transaction on CAD, **15.12**, (1996): 1518-1524
- [4] Yingxin Pang, Chung-Kuan Cheng, and Takeshi Yoshimura: *An Enhanced Perturbing Algorithm for Floorplan Design Using the O-tree Representation*. In Proceedings of International Symposium on Physical Design, (2000) 168-173
- [5] Yun-Chih Chang, Yao-Wen Chang, Guang-Ming Wu, and Shu-Wei Wu: *B*-Trees: A New Representation for Non-Slicing Floorplans*. In Proceedings of Design Automation Conference, (2000) 458-463
- [6] Ji-Ming Lin, and Yao-Wen Chang: *TCG: A Transitive Closure Graph-Based Representation for Non-Slicing Floorplans*. In Proceedings of Design Automation Conference, (2001) 764-769
- [7] Xiaoping Tang, and D.F. Wong: *FAST-SP: A Fast Algorithm for Block Placement based on Sequence Pair*. In Proceedings of Asia South Pacific Design Automation Conference, (2001) 521-526