Efficient Rerouting Algorithms for Congestion Mitigation

M. A. R. Chaudhry*, Z. Asad, A. Sprintson, and J. Hu
Department of Electrical and Computer Engineering
Texas A&M University, College Station, Texas

Abstract—Congestion mitigation and overflow avoidance are two of the major goals of the global routing stage. With a significant increase in the chip size and routing complexity, congestion and overflow have become critical issues in physical design automation. In this paper we present several routing algorithms for congestion reduction and overflow avoidance. Our methods are based on ripping up nets that go through the congested regions and replacing them with congestion-aware Steiner trees. We propose several efficient algorithms for finding congestion-aware Steiner trees and evaluate their performance using the ISPD routing benchmarks. We also show that the novel technique of network coding contributes to further improvements in routability and reduction of congestion. Accordingly, we propose an algorithm for identifying efficient congestion-aware network coding topologies and evaluate its performance.

I. INTRODUCTION

In almost any VLSI design flow, global routing is an essential stage that determines signal interconnections. Therefore, the capability of global router may significantly affect design turn-around time. Moreover, global routing results impact many circuit characteristics, such as power, timing, area, and signal integrity. Global routing poses major challenges in terms of the efficient computation of quality routes. In fact, most of the global routing problems, even special cases, tend to be NP-complete.

A significant increase in the chip size and routing complexity results in a larger number of nets that need to be routed, which, together with more stringent routing constraints, results in an increasing congestion, and, in some instances, overflow. In this work, we propose and evaluate efficient routing algorithms for congestion avoidance and overflow reduction. Our algorithms are designed for the rip-up-and-reroute phase of the global routing. At this stage, all the nets are already routed using standard global routing techniques, however, some of the nets need to be rerouted due to high congestion and overflow. Our techniques are based on ripping up the nets that go through congested regions and replacing them with congestion-aware Steiner trees. The proposed techniques facilitate even distribution of the routing load along the available routing areas. In addition, we use the novel technique of network coding for further reduction of congestion and the number of overflows.

A. Previous work

In the past decades, researchers have relentlessly strived to improve the performance of global routing algorithms (see e.g., [1], [2], [3], and references therein). To handle the complexity of large scale global routing, multilevel routing techniques are proposed in [4] and [5]. Recently proposed BoxRouter [6], [7] is based on progressive Integer Linear Programming (ILP) and rip-up-and-reroute techniques. A fast routing method is presented in [8]. Reference [9] proposes an approach based on the Lagrangian multiplier technique. An effective edge shifting technique is presented in [10]. Most of these previous works adopt the strategy of rip-up-and-reroute. However, they usually reroute one path at a time. In contrast, our method reroutes a Steiner tree for each multi-pin net.

B. Congestion-aware Steiner trees

The major goal of congestion-aware Steiner tree routing is to find a tree that connects the required set of nodes (referred to as terminals) while avoiding congested areas with a minimum penalty in terms of the total wirelength. In addition, the running time of the algorithm must scale well with the growing number of nets. This poses several challenges in terms of the algorithm design. The first challenge is to select a suitable cost function that adequately captures the local congestion conditions on the edges of the routing graph. Next, we need to devise an algorithm that finds a minimum cost tree with minimum running time. Since the this problem is NP-complete, the algorithm needs to use a suitable approximation technique. Finally, our technique must ensure that the overall performance of the rip-up-and-reroute phase is satisfactory in terms of congestion mitigation and overflow reduction. In this work, we evaluate several cost functions, which take into account various factors, such as density, overflow, and history. We propose several efficient algorithms for Steiner tree routing and compare their performance. Our algorithms are based on known approximation for the Steiner tree problem, heuristic methods, and artificial intelligence techniques.

C. Network coding

The basic idea of the network coding technique [11] is to enable the intermediate nodes to generate new signals by combining the signals arriving over their incoming wires. This is in contrast to the standard approach, in which each Steiner node can only forward or duplicate the incoming signals.

Example 1: Consider routing instance depicted in Figure 1(a). In this example, we need to route two nets, one connecting source $s_1$ with terminals $t_1, t_2$, and $t_3$, and the other connecting source $s_2$ with the same set of terminals. The underlying routing graph is represented by a grid $G(V, E)$ as shown in Figure 1(a). Suppose that, due to congestion, each edge of this graph has a residual capacity of one unit, i.e.,
each edge can accommodate only a single wire. It is easy to verify that only one net can be routed without an overflow. For example, Figure 1(b) shows a possible routing of a net that connects \( s_1 \) with terminals \( t_1, t_2, t_3 \). Figure 1(c) shows that routing of both nets results in an overflow. In this example, two nets transmit different signals, \( a \) and \( b \), over separate Steiner trees. Figure 1(d) shows that the network coding approach allows to route both nets without violating edge capacities. With this approach, the terminal \( t_1 \) creates a new signal, \( a \oplus b \), which is delivered to terminals \( t_2 \) and \( t_3 \), while the signals \( a \) and \( b \) are delivered to terminals \( t_2 \) and \( t_3 \) directly. It is easy to verify that with this scheme each terminal can decode the two original symbols, \( a \) and \( b \).

The example above indicates that network coding can offer two distinguished advantages. First, it has a potential of solving very difficult cases. In some instances, the network coding technique allows to solve cases that cannot be routed by conventional techniques. Second, it can achieve a decrease in the total wirelength, and, at the same time, alleviate the routing congestion.

### D. Contribution

Our contributions are: First, we propose several algorithms for finding efficient congestion-aware Steiner trees and evaluate their performance on the ISPD98 benchmarks; Second, we focus on the novel technique of network coding and present an efficient algorithm for finding congestion-aware network coding topologies; Finally, we evaluate the performance of several cost functions that capture the local congestion on the routing edges.

## II. Model

In this work, we adopt the most commonly used global routing model. The routing topology is represented by a grid graph \( G(V, E) \), where each node \( v \in V \) corresponds to a global routing cell (GCell) \([7, 9]\) and each routing edge \( e \in E \) corresponds to a boundary between two adjacent GCells. A set \( S = \{n_1, n_2, \ldots, n_{|S|} \} \) of nets are to be routed on this graph. Each net \( n_i \in S \) is a set of terminal nodes that need to be connected by wires. If there is a wire connection between two adjacent GCells, the wire must cross their boundary and utilize the corresponding routing edge. Each routing edge \( e \in E \) has a certain routing capacity \( c(e) \) which determines the number of wires that can pass through this edge. We use \( \eta(e) \) to denote the number of wires that are currently using this edge.

For each edge \( e \in E \), the overflow \( ov(e) \) of \( e \) is defined as:

\[
ov(e) = \begin{cases} 
\eta(e) - c(e) & \text{if } \eta(e) > c(e) \\
0 & \text{otherwise.}
\end{cases}
\]

The maximum overflow \( ov_{\text{max}} \) is defined as:

\[
ovo_{\text{max}} = \max_{e \in E} ov(e).
\]

Total overflow \( ov_{\text{tot}} \) is defined as:

\[
ovo_{\text{tot}} = \sum_{e \in E} ov(e).
\]

Total wire-length \( wlen \) is defined as:

\[
wlen = \sum_{e \in E} \eta(e).
\]

The density \( d(e) \) of an edge \( e \in E \) is defined as:

\[
d(e) = \frac{\eta(e)}{c(e)}.
\]

Since we focus on the rerouting phase, we assume that for each net \( n_i \in S \) there exists a Steiner tree \( T_i \) which connects all the terminals in \( n_i \). Given a set of trees \( \{T_i | n_i \in T_i\} \) we can determine the values of \( ov(e) \) and \( \eta(e) \) for each edge \( e \in E \). Then, we find a set of nets \( S' \subseteq S \) that need to be rerouted as well as the new topologies for the rerouted nets. Our goal is to find topologies that minimize the maximum overflow \( ov_{\text{max}} \), the total overflow \( ov_{\text{tot}} \), and the total wirelength \( wlen \).

### III. CONGESTION-AWARE STEINER TREES

In this section we present tools and techniques for finding efficient Steiner trees that minimize congestion with a minimum penalty in terms of the overall wirelength. Our algorithms are intended for rip-up-and-reroute stage of the global routing.

#### A. Cost functions

Our algorithms associate each edge \( e \in E \) in the graph with a cost function \( \rho(e) \) which captures its congestion and overflow. The cost of the entire tree is then defined as the total cost of all of its edges. Our goal is to identify trees that go through congested regions and replace them by trees or network coding topologies that use less congested areas.

In this work we consider several cost functions. First, we propose a cost assignment where the cost of an overflowed edge is a polynomial function of the sum of its density and overflow, whereas the cost of an edge without overflow is a polynomial function of its density. Formally, the proposed cost function is defined as follows:

\[
\rho(e) = \begin{cases} 
(d(e) + ov(e))^{\alpha} & \text{if } ov(e) > 0 \\
d(e)\alpha & \text{otherwise.}
\end{cases}
\]

Here, \( \alpha \) is a constant which determines the relative penalty for the congested edges.
Second, we use the cost assignment function proposed by [9]. With this cost assignment, the cost of an edge is an exponential function of its density:

\[ \rho(e) = \begin{cases} 
\exp(\beta \cdot (d(e) - 1)) & \text{if } d(e) > 1 \\
\frac{1}{d(e)} & \text{otherwise.} 
\end{cases} \]  

(2)

Here \( \beta \) is a constant which determines the penalty for overflowed edges.

The third function assigns costs based on the history [7], [9]. Specifically, each edge is associated with a parameter \( h(e) \) that specifies the number of times the edge has been overflowed during the previous iterations of the algorithm. That is, each time the edge with overflow is used, the parameter \( h_e \) is incremented by one. Then, the modified cost \( \rho'(e) \) of the edge is defined as follows:

\[ \rho'(e) = 1 + h_e \cdot \rho(e). \]  

(3)

Here, \( \rho(e) \) is either the polynomial function (Equation 1) or exponential function (Equation 2). If the density of the edge is less or equal to one, the parameter \( h_e \) is set to one.

B. Previous work on Steiner tree routing

The Steiner tree problem has proven to be NP-complete [12]. It is a well studied problem in the computer science and a wealth of heuristic and approximate solutions have been developed for it. The best known approximation algorithm has an Approximation ratio of 1.55 (i.e., the cost of the tree returned by the algorithm is less than two times the optimum) [13]. The best known approximations require significant computation time so we have used computationally feasible and easy to implement approximation and heuristic solution for constructing Steiner trees.

C. Algorithms for finding congestion-aware trees

As mentioned above, our goal is to rip up and reroute nets that use congested edges of \( G(V, E) \). For each net \( n \in S \) which has been ripped up, we need to find an alternative Steiner tree, with low congestion edges. We denote the set of terminals of \( n \) by \( T = \{t_1, t_2, \ldots, t|T|\} \).

In this section we describe five algorithms for finding congestion-aware Steiner trees. The first three algorithms use well-known techniques (see e.g., [14], [15], [16]) while the last two are based on the intelligent search techniques [17]. The performance of the algorithms is evaluated in Section V.

1) Algorithm 1: This algorithm approximates minimum cost Steiner tree by using a shortest path tree. A shortest path tree is a union of the shortest paths between terminal \( t_1 \) and each terminal \( t_i \in T \setminus \{t_1\} \). A shortest path tree can be identified by a single invocation of the Dijkstra’s algorithm. However, the cost of the tree may be significantly larger than the optimum.

2) Algorithm 2: This algorithm constructs the tree in an iterative manner, such that in iteration \( j \) it finds a shortest path between terminal \( t_1 \) and terminal \( t_i \in T \setminus \{t_1\} \). More specifically, we first find a shortest path \( P_1 \) between \( t_1 \) and \( t_2 \) with respect to the original costs. Then, we assign a zero cost to all edges that belong to \( P_1 \) and find a shortest path \( P_2 \) between \( t_1 \) and \( t_3 \) with respect to modified costs. The idea behind this algorithm is to encourage sharing of the edges between different paths. That is, if an edge \( e \) belongs to \( P_1 \), it can be used in \( P_2 \) without additional cost. In general, when finding a shortest path to terminal \( t_i \), all edges that belong to paths of previously processed terminals are assigned a zero cost. This algorithm requires \( |T| - 1 \) iterations of the Dijkstra’s algorithm, but it typically returns a lower cost tree than Algorithm 1.

3) Algorithm 3: This is a standard approximation algorithm with the approximation ratio of 2 (i.e., the cost of the tree returned by the algorithm is less than two times the optimum). Specifically, with this algorithm we find a shortest path between each pair of terminal nodes of \( n \). Then, we construct a complete graph \( G' \) in which nodes are comprised of the terminal nodes of \( n \) and each edge \((v, u) \in G'\) is assigned the length of the shortest path between \( v \) and \( u \) in \( G \). The algorithm then finds a minimum spanning tree \( \Theta \) in \( G' \). Next, each edge in \( \Theta \) is substituted by the corresponding shortest path in \( G \), which results in Steiner tree in \( G \) that connects all terminals in \( T \).

4) Algorithms 4 and 5: The next two algorithms, referred to as Algorithms 4 and 5 are intelligent search based algorithms. Our approach is inspired by the \( A^* \) algorithm, which belongs to the general class of best-first graph search algorithms and uses a heuristic function to determine the order of visiting the nodes of the graph. Specifically, for each node \( v \in G \) we define \( p(v) \) to be the maximum Manhattan distance between node \( v \) and a terminal \( t_i \in T \) which has not been processed yet by the algorithm. The Manhattan distance between \( v \) and \( t_i \) is defined as the minimum number of hops that separate \( v \) and \( t_i \) in \( G \). Then, Algorithms 4 and 5 follow the same steps as Algorithms 1 and 2, respectively, with the addition of the heuristic function \( p(v) \) to the current cost estimate.

IV. NETWORK CODING TECHNIQUE

In this paper, we use the network coding technique in order to achieve further improvement in terms of minimizing congestion and reducing the number of overflows. The network coding technique enables, under certain conditions, to share edges that belong to different nets. For example, for the graph depicted in Figure 1(c) there are two minimum Steiner trees, one transmitting signal \( a \) from source \( s_1 \) and the second transmitting signal \( b \) from source \( s_2 \). These two trees clash at the middle edge (emanating from \( t_1 \)), resulting in an overflow. This conflict can be resolved by coding at node \( t_1 \), which effectively allows two trees to share edges. Our methods will identify pairs of nets that share a large number of common terminals and then apply network coding technique to reduce overflow.

A. Previous work on network coding

The problem of routing of multiple nets with shared terminals is related to the problem of establishing efficient multicast connections in communication networks. The network coding technique was proposed in a seminal paper by Ahlswede et al. [11]. It was shown in [11] and [18] that the capacity of the multicast networks, i.e., the number of packets that can be sent simultaneously from the source node \( s \) to all terminals is equal to the minimum size of a cut that separates \( s \) from each terminal. Li et. al. [18] proved that linear network codes
are sufficient for achieving the capacity of the network. In a subsequent work, Koetter and Méard [19] developed an algebraic framework for network coding. This framework was used by Ho et al. [20] to show that linear network codes can be efficiently constructed through a randomized algorithm. Jaggi et al. [21] proposed a deterministic polynomial-time algorithm for finding feasible network codes in multicast networks. An initial study of applicability of network coding for improving the routability of VLSI designs appears in [22]. In [23] Gulati et al. used network coding for improving the routability of FPGAs, focusing on finding the nets that are suitable for network coding. To the best of our knowledge, this is the first work that proposes efficient algorithms for finding the congestion-aware network coding topologies for VLSI circuits.

B. Network coding algorithm

We proceed to present the algorithm we use for constructing congestion-aware coding networks that reduces congestion and overflow. Unlike the Steiner tree algorithms, for each net \( n_i \in S \) it is important to distinguish between the source node \( s_i \) of \( n_i \) and the rest of the terminals in \( T_i \). In particular, the node that generates the signal and drives the net is referred to as the source. Our algorithm assumes that the location of the source is known for each net.

The first step of the algorithm to identify the subset \( S' \) of \( S \) that includes nets that go through edges with overflow. Second, we identify pairs of nets in \( S' \) that share at least four common terminals. Thus, we check, for each such pair of nets \((n_i, n_j)\) whether we can replace the Steiner trees for \( n_i \) and \( n_j \) by a more efficient routing topology with respect to congestion and overflow.

More specifically, let \((n_i, n_j)\) be a pair of nets in \( S' \) that share at least four terminals. We denote the set of terminals shared by \( n_i \) and \( n_j \) by \( T_{ij} \). We also denote by \( T_{ij}^t \) the set of terminals in \( T_i \) that do not belong to \( T_{ij} \) and by \( T_{ij}^s \) the set of terminals in \( T_j \) that do not belong to \( T_{ij} \). Next, we find two congestion-aware Steiner trees \( \Phi_i \) and \( \Phi_j \) that connect \( s_i \) to \( T_{ij}^t \) and \( s_j \) to \( T_{ij}^s \). These trees can be identified by one of the algorithms presented in Section III. The cost of edges in the graph is updated after finding \( \Phi_i \) and \( \Phi_j \). Finally, we find a congestion-aware network coding topology \( \Theta \) that connects \( s_i \) and \( s_j \) to the common set of terminals \( T_{ij} \).

The network coding topology \( \Theta \) is identified through the following procedure:

1) Determine the shortest distance between \( s_i \) and each terminal in \( T_{ij} \);
2) For each terminal \( t \in T_{ij} \) let \( d(t) \) the the sum of the shortest distances between \( s_i \) and \( t \), and \( s_j \) and \( t \);
3) Set \( \Theta \leftarrow \emptyset \);
4) For all terminals \( t \in T_{ij} \) in the increasing order of \( d(t) \) do:
   a) Find two disjoint paths \( P_i \) and \( P_j \) of minimum total cost, that connect \( s_i \) and \( t \), and \( s_j \) and \( t \), respectively. Such paths can be determined by the algorithm due to Suurballe and Tarjan [24];
   b) Assign a zero cost to all edges that belong to \( P_i \) and \( P_j \);

c) Set \( \Theta \leftarrow \Theta \cup E(P_i) \cup E(P_j) \), where \( E(P_i) \) and \( E(P_j) \) are the sets of edges that belong to paths \( P_i \) and \( P_j \), respectively;
5) Return \( \Theta \).

Finally, we determine whether the total cost of \( \Theta, \Phi_i, \) and \( \Phi_j \) is less than the total cost of the original Steiner trees for \( n_i \) and \( n_j \). If there is a reduction in terms of cost, the two original Steiner trees are replaced by \( \Theta, \Phi_i, \) and \( \Phi_j \).

V. PERFORMANCE EVALUATION

We have evaluated the performance of our algorithms using the ISPD98 routing benchmarks [25]. In all the experiments, we first run the Steiner tree tool Flute [26] in order to determine the initial routing of all the nets in the benchmark. Next, we perform an iterative procedure, referred to as Phase I, which processes each net and checks whether an alternative Steiner tree of lower cost exists and if yes, rips up the existing tree and replaces it with the alternative one. The procedure uses one of the algorithms described in Section III. The procedure terminates when four subsequent iterations yield the same number of overflows.

Next, we check whether the application of network coding technique can further reduce the number of overflows. This procedure is referred to as the NC Phase. We first identify pairs of nets that have overflowed edges and share at least four terminals. We then apply the network coding algorithm presented in Section IV to find an alternative network coding topology and perform rip-up and reroute if such a topology
faster than their counterparts (Algorithm 1 and Algorithm 2, respectively). This is due to the fact that intelligent search methods speed up the search by preferring nodes closer to the destination.

Table I. Performance evaluation using Algorithm 3 on ISPD98 benchmarks.

<table>
<thead>
<tr>
<th>Polynomial Cost</th>
<th>Exponential Cost</th>
<th>History Based Cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha_{tot}$</td>
<td>$\alpha_{max}$</td>
<td>$\omega_{tot}$</td>
</tr>
<tr>
<td>ibm1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm2</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm3</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm5</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm6</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm7</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm8</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm9</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>ibm10</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Table II. Performance evaluation of different cost functions (using Algorithm 3 for Phase 1) on ISPD98 benchmarks.

<table>
<thead>
<tr>
<th>Ver. Cap</th>
<th>Hor. Cap</th>
<th>MaizeRouter</th>
<th>Phase 1 + NC Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ibm1</td>
<td>11</td>
<td>43</td>
<td>190378</td>
</tr>
<tr>
<td>ibm3</td>
<td>21</td>
<td>31</td>
<td>45326</td>
</tr>
<tr>
<td>ibm8</td>
<td>17</td>
<td>28</td>
<td>649</td>
</tr>
<tr>
<td>ibm9</td>
<td>10</td>
<td>24</td>
<td>4006</td>
</tr>
</tbody>
</table>

Table IV. Improvement over MaizeRouter for modified (congested) IBM benchmarks using Algorithm 3 for Phase 1 and polynomial cost function (see Equation 1).

In the first set of experiments we evaluated the performance of three cost functions mentioned in Section III on the ISPD98 benchmarks using Algorithm 3 for Phase 1. For cost function given by Equation 1 we used $\alpha \geq 10$, whereas for cost function given by Equation 2 we used $\beta \geq 50$ and for cost function given by Equation 3, $\rho(e)$ was a polynomial cost function. The results are shown in Table II. Polynomial cost function showed the best performance in terms of overflows.

In another set of experiments we tried to solve slightly modified ISPD98 benchmark files. The modification included reducing the vertical and horizontal capacity by one unit. For Phase 1 we used Algorithm 3 and then applied the NC Phase.
Polynomial cost function given by Equation 1 was used to check the performance on these more congested cases. The results are given in Table III.

We have also conducted the same experiment on several selected benchmarks on the output of MaizeRouter [10]. In these experiments, we iteratively reduced the vertical and horizontal capacity of the ISP98 benchmarks until we got overflows while running them through MaizeRouter. Then, we used the output of MaizeRouter as input to Algorithm 3 and after that we have applied the NC Phase. The cost function used was polynomial with $\alpha = 10$. The results are shown in Table IV. The results demonstrate that Algorithm 3 performs well and can contribute to further reduction of the number of overflows.

VI. CONCLUSIONS

In this paper we presented several efficient technique for rip-up-and-reroute state of the global routing process. Our techniques is based on ripping up nets that go through highly congested areas and rerouting them using efficient Steiner tree algorithms. We have have considered several efficient Steiner tree algorithms as well as several weight functions that capture congestion and overflow. We have also studied the application of the novel technique of network coding and showed that it can efficiently handle difficult routing cases, facilitating reduction in the number of overflows.

REFERENCES


TABLE III

<table>
<thead>
<tr>
<th></th>
<th>Flute Phase 1</th>
<th>NC Phase</th>
<th>Successful NC pairs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$w_{tota}$</td>
<td>$w_{max}$</td>
<td>$w_{tota}$</td>
</tr>
<tr>
<td>ibm1</td>
<td>5257</td>
<td>14</td>
<td>60209</td>
</tr>
<tr>
<td>ibm2</td>
<td>5619</td>
<td>22</td>
<td>106193</td>
</tr>
<tr>
<td>ibm3</td>
<td>2308</td>
<td>16</td>
<td>145777</td>
</tr>
<tr>
<td>ibm4</td>
<td>4833</td>
<td>15</td>
<td>162844</td>
</tr>
<tr>
<td>ibm5</td>
<td>5</td>
<td>4</td>
<td>410038</td>
</tr>
<tr>
<td>ibm6</td>
<td>5397</td>
<td>24</td>
<td>276012</td>
</tr>
<tr>
<td>ibm7</td>
<td>4665</td>
<td>13</td>
<td>363678</td>
</tr>
<tr>
<td>ibm8</td>
<td>5400</td>
<td>13</td>
<td>403502</td>
</tr>
<tr>
<td>ibm9</td>
<td>8545</td>
<td>14</td>
<td>411524</td>
</tr>
<tr>
<td>ibm10</td>
<td>7103</td>
<td>20</td>
<td>547475</td>
</tr>
</tbody>
</table>