CHAPTER 5
WEIGHTED DISTANCE, MEDIAN AND PERIPHERY

In this chapter yet another set of weighted central structures namely, weighted distance \( wd(v) \), weighted median \( w\text{-median}(T) \), weighted peripheral vertex and weighted periphery are introduced. Parallel algorithms for these weighted central structures are designed, developed and analyzed. Algorithms to find distance between two vertices \( d(v,u) \), and to find distance of a vertex \( d(v) \) are developed as supportive algorithms.

5.1 Introduction

Let \( G \) be a finite, simple connected, undirected graph with vertex set \( V(G) \) and edge set \( E(G) \). Consider the following graph.

![Graph](https://via.placeholder.com/150)

Figure 5.1 A graph
It may be assumed that each vertex in the above graph is a city and each edge is a road connecting two cities. Suppose a fire service facility or police station is to be located. We want to have it in a place where the distance from the facility to the farthest place is the minimum. In figure 5.1, vertex 5 is the suitable place for such a facility. It may be noted that the distance (number of edges) of the farthest vertex is 4, which is the minimum.

Supposing a company wants to establish the head office at a vertex of the graph in figure 5.1 and branch offices in all other vertices of the graph. If vertex 5, the center of the graph is the location for the head office then the travel expenditure for the annual conference is $d(5)=4+3+2+1+1+2+3+(4*6) = 40$. Suppose we select vertex 8 as the head office then the expenditure would be $d(8)=7+6+5+4+3+2+1+(1*6) = 34$. Hence vertex 8 is a better location than vertex 5. Since $d(7)$ is also 34, it can be easily seen that vertices 7 and 8 are the only two vertices for which the $d$ value is the minimum. The choice of a vertex should be such that the sum of the distances from all the vertices to that particular vertex is the minimum.

In order to locate a service facility like company head office, post office, bank or power station, the authorities would want to minimize the total distance (expenditure). This problem is also a vertex-serves-vertex problem where the facility and customers are located on the vertices.
The distance between a pair of vertices \((u,v)\) denoted \(d(u,v)\) is the sum of the edge weights in the path from \(u\) to \(v\). The distance of a vertex \(v\), denoted \(d(v)\) is defined as the sum of distances from \(v\) to all the vertices of \(G\).

\[
d(v) = \sum_{u \in V} d(u, v)\]

Each vertex of \(G\) at which the distance function is minimized is called the median of \(G\). In the graph of figure 5.1 the medians are the vertices 7 and 8. The set of all medians form the median sub graph. The center and median of a graph need not be the same and can, in fact be disjoint as seen in figure 5.1.

These facts stress the importance of the central structures like distance, median and periphery. Since the approach is with weighted structures, weighted distance, weighted median and weighted periphery are defined.

The **weighted median** is defined by \(w\)-median\((T) = \min\{ w(d(v)) \mid v \in V(T) \}\). A vertex \(v\) is a **weighted peripheral vertex** if \(we(v) = w(\text{diam}(T))\). The **weighted periphery** is the set of all weighted peripheral vertices. In this chapter, parallel algorithms for these weighted central structures are developed and analyzed.
5.2 Distance

Before the weighted distance is taken for discussion, the prerequisites for the same are discussed first. In order to compute weighted distance for each vertex \( v \) it is required to compute the distance between any two vertices in the tree. Computing distance between any pair of vertices needs re-rooting the tree at a vertex and finding the distance from the root to every other vertex. So, two supportive parallel algorithms are developed. They are, the algorithm to find the distance from root and the algorithm to find all pair distances. Then the algorithm to compute the weighted distance is developed.

5.2.1 Parallel Algorithm to find the Distance from Root

The aim is to find the \( d(v) \) for each vertex in \( T \). In order to compute \( d(v) \), it is required to find \( d(v,u) \) for each pair \( (v,u) \).

Given a tree \( T \), the process of finding the distance between any pair of vertices \( u \) and \( v \), \( d(u,v) \) was already discussed in section 4.2. Now the task of writing a parallel algorithm for finding \( d(u,v) \) for each pair \( (u,v) \) is taken. This task is accomplished in two levels

i. At the first level, given a tree rooted at \( r \), the distance from root to every other vertex is found.
ii. The second level computes the distance between any pair of vertices $u$ and $v$, with the help of the Re-rooting algorithm and the work done at the first level.

Let $T$ be a tree rooted at $r$. The tree $T$ is represented by an array $p$ of length $n$ such that $p(i)=j$ if $j$ is the parent of $i$ in $T$. For the effective implementation of the parallel algorithm let $p(r)=r$. The problem is to determine the distance of each vertex from the root $r$.

Here, the pointer jumping technique is used. The pointer jumping technique is useful in general because it is simple and can effectively handle sub problems arising in many computational tasks. These sub problems are usually of a size small enough that the pointer jumping technique will allow optimal overall processing. It is also possible to use the pointer jumping technique in combination with other techniques to achieve optimality.

Let $p(1:n)$ be the parent array, where $p(i)$ holds the parent of vertex $i$ and $p(r)=r$. Let $ew(1:n)$, be the edge weight array, where $ew(i)$ holds the weight of edge$(i,p(i))$, and $ew(r)=0$. The aim is, for each vertex $i$, the distance from root $dfr(i)$ must be set equal to the sum of the weights of edges on the path from $i$ to the root $r$. The computation proceeds from vertex $i$, and moves towards the root, each time adding edge weights to the sum of vertex $i$. 

145
The formal algorithm to find the distance from root to each vertex \( i \) is given below.

**Algorithm DISTANCE-FROM-ROOT(r)**

**Input:**

1. The parent array \( p(1:n) \), where \( p(i) \) is the parent of vertex \( i \), and \( p(r) = r \).
2. The edge weight array \( ew(1:n) \), where \( ew(i) \) is the weight of the edge \((i, p(i))\) and \( ew(r) = 0 \)

**Output:** The array \( dfr(1:n) \), where \( dfr(i) \) holds the distance from root \( r \) to vertex \( i \)

**Begin**

1. for \( i = 1 \) to \( n \) and \( i \neq r \) do in parallel
   
   \( s(i) = p(i) \)
   
   \( dfr(i) = ew(i) \)
   
   while \( s(i) \neq r \) do
     
     \( dfr(i) = dfr(i) + dfr(s(i)) \)
     
     \( s(i) = s(s(i)) \)
   
   end

end parallel

**End**
The working of the above algorithm is explained with the help of the tree rooted at vertex 1 given in figure 5.2(a).

![Figure 5.2](image_url)

**Figure 5.2**

a) Given Tree  b) After First Interaction  c) After Second Interaction

The tree rooted at vertex 1 given in figure 5.2(a) can be represented as a parent array as follows:

```
1 2 3 4 5 6 7 8 9
p 1 1 1 2 2 3 3 4 6
```

And its edge weights can be stored in an array as given below.

Note that $ew(i)$ is the weight of edge $(i, p(i))$ and $ew(r) = 0$

```
1 2 3 4 5 6 7 8 9
ew 0 5 4 2 7 2 3 8 5
```

147
With the help of a parallel for statement the computation is initiated from each vertex except the root. Inside the loop first the parent array is copied in a temporary array \( s \), so that the parent array is not disturbed. Then the array \( dfr \) is initialized with the corresponding values of the array \( ew \). At this stage the contents of the array \( dfr \) will be

\[ \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline 
0 & 5 & 4 & 2 & 7 & 2 & 3 & 8 & 5 \\
\end{array} \]

After these preliminary tasks, a while loop is constructed to implement the pointer jumping technique. There are two steps inside the while loop. The first step adds the \( dfr \) value of the parent of vertex \( i \) to its current value of \( dfr \). The second step moves each vertex one level up by making parent of parent as its new parent. The following are the contents of the arrays \( dfr \) and \( s \) after the first iteration and the corresponding tree is given in figure 5.2(b).

\[ \begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
\hline 
\text{dfr} & 0 & 5 & 4 & 7 & 12 & 6 & 7 & 10 & 7 \\
\text{s} & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 2 & 3 \\
\end{array} \]

148
The tree after the second iteration is given in figure 5.2 (c), and the changes in \(dfr\) and \(s\) are given below.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>(dfr)</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>(s)</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

At this stage the while loop terminates since all the entries in the \(s\) array are 1 which is the root. It may be noted that the while loop will continue only when \(s(i) \neq r\). Now, the array \(dfr\) holds the result. That is \(dfr(i)\) is the distance from root 1 to each vertex \(i\).

**Theorem 5.1.** Given a tree \(T\) rooted at \(r\), the distance from \(r\) to each vertex \(v\) can be found in \(O(\log n)\) time using \(O(n)\) processors.

**Proof.** Let \(T\) be a tree rooted at \(r\). Let \(v \in V(T)\). By the definition of distance, and the use of pointer jumping technique initiated from \(v\), clearly the algorithm computes the distance from \(r\).

A parallel 'for' statement initiates the work on \(n\) processors in parallel. Within the 'for' statement the first two statements are primitive assignments and take \(O(1)\) unit time. The while loop is used to implement the pointer jumping technique and takes \(O(\log n)\) time. Thus the algorithm DISTANCE-FROM-ROOT(r) takes \(O(\log n)\) time using \(O(n)\) processors.
5.2.2 Parallel Algorithm to find All Pairs Distance

Now, the problem is to find the distance between each pair of vertices \((u,v)\). Let \(d(1:n,1:n)\) be the distance matrix, where \(d(i,j)\) is the distance from vertex \(i\) to \(j\). The aim is to write a parallel algorithm to find the matrix \(d\). To achieve this the algorithms \(RE\text{-}ROOT\) and \(DISTANCE\text{-}FROM\text{-}ROOT\) are used. The logic goes this way: for each vertex \(v\), re-root the tree at \(v\), compute the distance of each vertex from the new root, and store the result in the corresponding row of \(d\).

The algorithm to compute the distance matrix \(d\) is given below.

Algorithm ALL-PAIRS-DISTANCE

Input:

1. The parent array \(p(1:n)\), where \(p(i)\) is the parent of vertex \(i\), and \(p(r)=r\).
2. The edge weight array \(ew(1:n)\), where \(ew(i)\) is the weight of the edge \((i,p(i))\) and \(ew(r)=0\).

Output: The distance matrix \(d\)

Begin

1. For \(r=1\) to \(n\) do in parallel
   
   call RE-ROOT\((r)\)
The working of the All-Pairs-Distance algorithm is explained using the tree given in figure 5.2(a). The algorithm receives this tree as a parent array $p$ and an edge weight array $ew$. A parallel for loop with $1 \leq r \leq n$ is initiated to perform the following steps in parallel on $n$ vertices.

i. Re-root $T$ at $r$

This step calls $RE$-$ROOT$ algorithm for each $r$. As a result of this step the trees given in figure 5.3 are got. It may be noted that when $r=1$, it is nothing but the input tree given in figure 5.2(a). Hence in total there are nine trees constructed in parallel.

ii. Compute distance from root

For each tree rooted at $r$, $1 \leq r \leq n$, the $DISTANCE$-$FROM$-$ROOT$ algorithm is called in parallel. For each tree, this algorithm returns an array $dfr(1:n)$, where $dfr(i)$ is the distance of vertex $i$ from the corresponding root $r$. The results of this step are listed in figure 5.4.
Figure 5.3 Tree rooted at r, when  

i) $r=2$  

ii) $r=3$  

iii) $r=4$  

iv) $r=5$  

v) $r=6$  

vi) $r=7$  

vii) $r=8$  

viii) $r=9$
Figure 5.4 Results of DISTANCE-FROM-ROOT algorithm
iii. Store $dfr$ in $d$

For each tree rooted at $r$, $1 \leq r \leq n$, the $n$ values stored in the array $dfr$ are copied to the $r^{th}$ row of the matrix $d$. This is done in parallel with the help of a parallel for statement. The distance matrix constructed by this step is given in table 5.1.

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>5</td>
<td>4</td>
<td>7</td>
<td>12</td>
<td>6</td>
<td>7</td>
<td>15</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>0</td>
<td>9</td>
<td>2</td>
<td>7</td>
<td>11</td>
<td>12</td>
<td>10</td>
<td>16</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
<td>9</td>
<td>0</td>
<td>11</td>
<td>16</td>
<td>2</td>
<td>3</td>
<td>19</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>7</td>
<td>2</td>
<td>11</td>
<td>0</td>
<td>9</td>
<td>13</td>
<td>14</td>
<td>8</td>
<td>18</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>7</td>
<td>16</td>
<td>9</td>
<td>0</td>
<td>18</td>
<td>19</td>
<td>17</td>
<td>23</td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>11</td>
<td>2</td>
<td>13</td>
<td>18</td>
<td>0</td>
<td>5</td>
<td>21</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>12</td>
<td>3</td>
<td>14</td>
<td>19</td>
<td>5</td>
<td>0</td>
<td>22</td>
<td>10</td>
</tr>
<tr>
<td>8</td>
<td>15</td>
<td>10</td>
<td>19</td>
<td>8</td>
<td>17</td>
<td>21</td>
<td>22</td>
<td>0</td>
<td>26</td>
</tr>
<tr>
<td>9</td>
<td>11</td>
<td>16</td>
<td>7</td>
<td>18</td>
<td>23</td>
<td>5</td>
<td>10</td>
<td>26</td>
<td>0</td>
</tr>
</tbody>
</table>

Table 5.1 Distance matrix of figure 3.1
Theorem 5.2. Given a tree $T$ rooted at $r$, the distance between all pairs of vertices can be found in $O(\log n)$ time using $O(n^2)$ processors on the CREW PRAM.

Proof. Clearly the algorithm computes $d(u,v)$ for all pairs of vertices $u$ and $v$.

A parallel 'for' statement is used to initiate the computations on $n$ processors in parallel. Within the for statement first the RE-ROOT algorithm is called, which takes $O(1)$ unit time using $O(n)$ processors. Next the DISTANCE-FROM-ROOT algorithm is called, which takes $O(\log n)$ time using $O(n)$ processors. Finally a parallel 'for' statement is used to perform an assignment statement with $O(1)$ unit time using $O(n)$ processors. Thus the overall time taken to find the distance between all pairs of vertices using the parallel algorithm ALL-PAIRS-DISTANCE is $O(\log n)$ and the number of processors used is $O(n^2)$.

5.2.3 Parallel Algorithm to find the Distance of a Vertex

The distance of a vertex $v$, $d(v)$, is the sum of the distances from $v$ to every other vertex in $T$

$$d(v) = \sum d(v,u); \ u \in V(T)$$
This is nothing but the sum of row \( v \) of distance matrix \( d \). Figure 5.5 and table 5.2 represent the distance of each vertex \( v \).

![Diagram](attachment:image.png)

**Figure 5.5** \( d(v) \) for each \( v \)

<table>
<thead>
<tr>
<th>( v )</th>
<th>( d(v) )</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>67</td>
</tr>
<tr>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>3</td>
<td>71</td>
</tr>
<tr>
<td>4</td>
<td>82</td>
</tr>
<tr>
<td>5</td>
<td>121</td>
</tr>
<tr>
<td>6</td>
<td>81</td>
</tr>
<tr>
<td>7</td>
<td>92</td>
</tr>
<tr>
<td>8</td>
<td>138</td>
</tr>
<tr>
<td>9</td>
<td>116</td>
</tr>
</tbody>
</table>

**Table 5.2** Vertex \( v \) and \( d(v) \)
The algorithm to compute the distance of a vertex $r$, where $r$ is the root of the tree $T$, is given below.

**Algorithm DISTANCE($r$)**

**Input:**

1. The parent array $p(1:n)$, where $p(i)$ is the parent of vertex $i$, and $p(r)=r$.

2. The edge weight array $ew(1:n)$, where $ew(i)$ is the weight of the edge $(i,p(i))$ and $ew(r)=0$

**Output:** The distance of $r$, $d(r)$

**Begin**

1. Call ALL-PAIRS-DISTANCE algorithm to find the distance matrix $d$

2. For $i=1$ to $n$ do in parallel

   $td(i) = d(r,i)$

   end parallel

3. $m=n/2$

4. While $m>0$ do

   For $i=1$ to $m$ do in parallel

   $td(i) = td(2*i-1) + td(2*i)$

   end parallel

   $m=m/2$
end while
5. return td(1)
End

The distance of all the vertices can be found by calling the DISTANCE algorithm as follows:

For i=1 to n do in parallel
d(i) = DISTANCE(i)
End parallel

Theorem 5.3. Given a tree $T$ rooted at $r$, the distance of $r$, $d(r)$, can be found in $O(\log n)$ time using $O(n^2)$ processors on the CREW PRAM.

Proof. Clearly the algorithm computes the distance of $r$, $d(r)$.

Step1 calls the ALL-PAIRS-DISTANCE algorithm which takes $O(\log n)$ time using $O(n^2)$ processors. Step2 takes $O(1)$ unit time using $O(n)$ processors. Step3 and step5 are primitive assignments with $O(1)$ unit time. Step4 needs $O(\log n)$ time with $O(n/2)$ processors. Thus the DISTANCE($r$) algorithm takes $O(\log n)$ time using $O(n^2)$ processors. \qed
5.3 Weighted Distance

Once the distance matrix, \( d(i,j) \) is computed, the weighted distance between vertices \( i \) and \( j \) can be computed as follows.

\[
wd(i,j) = d(i,j) \times w(j)
\]

Where, \( w(j) \) is the vertex weight of \( j \). From \( wd(i,j) \), the weighted distance of any vertex \( i \) in the tree \( T \) can be computed. The weighted distance of a vertex \( v \) is defined as the sum of weighted distances from \( v \) to every other vertex in the tree.

\[
wd(v) = \sum wd(v,u); \ u \in V
\]

5.3.1 Evaluating Weighted Distance

Table 4.3 gives the weighted distance matrix \( wd(i,j) \) for the tree in figure 3.1. From this table the weighted distance of each vertex \( wd(i) \) can be computed. For any vertex \( i \), the \( wd(i) \) is the sum of the elements in the row \( i \). Table 5.3 and figure 5.6 give the weighted distance \( wd(i) \) for each vertex \( i \).
Table 5.3 Vertex $v$ and $wd(v)$

<table>
<thead>
<tr>
<th>$v$</th>
<th>$wd(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>16400</td>
</tr>
<tr>
<td>2</td>
<td>16400</td>
</tr>
<tr>
<td>3</td>
<td>17200</td>
</tr>
<tr>
<td>4</td>
<td>19200</td>
</tr>
<tr>
<td>5</td>
<td>23400</td>
</tr>
<tr>
<td>6</td>
<td>19600</td>
</tr>
<tr>
<td>7</td>
<td>22000</td>
</tr>
<tr>
<td>8</td>
<td>33600</td>
</tr>
<tr>
<td>9</td>
<td>26600</td>
</tr>
</tbody>
</table>

Figure 5.6 $wd(v)$ for each $v$
As far as the large example tree in figure 3.8 is concerned the distance matrix is given in table 4.5 and weighted distance between any pair of vertices $i$ and $j$, is given in table 4.6 as a weighted distance matrix.

After the computation of the row sums the weighted distance, $wd(i)$ of each vertex $i$ is given in table 5.4.

<table>
<thead>
<tr>
<th>$V$</th>
<th>$wd(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4366</td>
</tr>
<tr>
<td>2</td>
<td>4546</td>
</tr>
<tr>
<td>3</td>
<td>5226</td>
</tr>
<tr>
<td>4</td>
<td>5654</td>
</tr>
<tr>
<td>5</td>
<td>5086</td>
</tr>
<tr>
<td>6</td>
<td>5274</td>
</tr>
<tr>
<td>7</td>
<td>4806</td>
</tr>
<tr>
<td>8</td>
<td>6666</td>
</tr>
<tr>
<td>9</td>
<td>6130</td>
</tr>
<tr>
<td>10</td>
<td>6254</td>
</tr>
<tr>
<td>11</td>
<td>6814</td>
</tr>
<tr>
<td>12</td>
<td>6178</td>
</tr>
<tr>
<td>13</td>
<td>5692</td>
</tr>
<tr>
<td>14</td>
<td>6388</td>
</tr>
<tr>
<td>15</td>
<td>6842</td>
</tr>
<tr>
<td>16</td>
<td>6106</td>
</tr>
<tr>
<td>17</td>
<td>6444</td>
</tr>
<tr>
<td>18</td>
<td>5490</td>
</tr>
<tr>
<td>19</td>
<td>5926</td>
</tr>
<tr>
<td>20</td>
<td>7224</td>
</tr>
<tr>
<td>21</td>
<td>6722</td>
</tr>
<tr>
<td>22</td>
<td>8142</td>
</tr>
<tr>
<td>23</td>
<td>7134</td>
</tr>
<tr>
<td>24</td>
<td>7942</td>
</tr>
<tr>
<td>25</td>
<td>6812</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>$V$</th>
<th>$wd(v)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>7302</td>
</tr>
<tr>
<td>27</td>
<td>7188</td>
</tr>
<tr>
<td>28</td>
<td>7078</td>
</tr>
<tr>
<td>29</td>
<td>7226</td>
</tr>
<tr>
<td>30</td>
<td>7744</td>
</tr>
<tr>
<td>31</td>
<td>7054</td>
</tr>
<tr>
<td>32</td>
<td>7066</td>
</tr>
<tr>
<td>33</td>
<td>8056</td>
</tr>
<tr>
<td>34</td>
<td>9032</td>
</tr>
<tr>
<td>35</td>
<td>9474</td>
</tr>
<tr>
<td>36</td>
<td>9398</td>
</tr>
<tr>
<td>37</td>
<td>9122</td>
</tr>
<tr>
<td>38</td>
<td>8240</td>
</tr>
<tr>
<td>39</td>
<td>8004</td>
</tr>
<tr>
<td>40</td>
<td>8594</td>
</tr>
<tr>
<td>41</td>
<td>9114</td>
</tr>
<tr>
<td>42</td>
<td>9912</td>
</tr>
<tr>
<td>43</td>
<td>9186</td>
</tr>
<tr>
<td>44</td>
<td>10294</td>
</tr>
<tr>
<td>45</td>
<td>9502</td>
</tr>
<tr>
<td>46</td>
<td>8712</td>
</tr>
<tr>
<td>47</td>
<td>9558</td>
</tr>
<tr>
<td>48</td>
<td>10752</td>
</tr>
<tr>
<td>49</td>
<td>11572</td>
</tr>
<tr>
<td>50</td>
<td>10672</td>
</tr>
</tbody>
</table>

Table 5.4 Vertex $v$ and $wd(v)$ for the tree in figure 3.8
5.3.2 Parallel Algorithm to find the Weighted Distance

The algorithm to compute the weighted distance of a vertex \( r \), where \( r \) is the root of the tree \( T \) is given below.

**Algorithm W-DISTANCE**(r)

**Input:**

1. The parent array \( p(1:n) \), where \( p(i) \) is the parent of vertex \( i \), and \( p(r)=r \).
2. The edge weight array \( ew(1:n) \), where \( ew(i) \) is the weight of the edge \( (i,p(i)) \) and \( ew(r)=0 \)

**Output:** Weighted distance of \( r \), \( wd(r) \)

**Begin**

1. Call ALL-PAIRS-DISTANCE algorithm to find the distance matrix \( d \)
2. For \( i=1 \) to \( n \) do in parallel
   
   \[ twd(i) = d(r,i) \times w(i) \]

   end parallel
3. \( m=n/2 \)
4. While \( m>0 \) do
   
   For \( i=1 \) to \( m \) do in parallel
   
   \[ twd(i) = twd(2\times i-1) + twd(2\times i) \]
end parallel

\( m = m/2 \)

5. end while

6. Return twd(1)

End

The weighted distance of all the vertices can be found by calling \( W\text{-DISTANCE} \) as follows:

For \( i = 1 \) to \( n \) do in parallel

\[ \text{wd}(i) = \text{W-DISTANCE}(i) \]

End parallel

Theorem 5.4: Given a weighted tree \( T \) rooted at \( r \), the weighted distance of \( r \) can be found in \( O(\log n) \) time using \( O(n^2) \) processors on the CREW PRAM.

Proof. Clearly the algorithm computes the weighted distance of \( r \).

Step 1 calls the \( ALL-PAIRS-DISTANCE \) algorithm which takes \( O(\log n) \) time using \( O(n^2) \) processors. Step 2 takes \( O(1) \) unit time using \( O(n) \) processors. Step 3 and step 5 are primitive assignments with \( O(1) \) unit time. Step 4 needs \( O(\log n) \) time with \( O(n/2) \) processors. Thus the \( W\text{-DISTANCE} \) algorithm takes \( O(\log n) \) time using \( O(n^2) \) processors.

\[ \square \]
5.4 **Weighted Median**

The median of a graph is the vertex at which the distance function \(d(v)\) is minimized. The weighted median of a weighted graph is the vertex at which the weighted distance is minimized.

In figure 5.5, the minimum distance, \(d(v)\) is for the vertex 1, which is the median and is given in figure 5.7(a). Similarly in figure 5.6, the minimum weighted distance, \(wd(v)\) is for the vertices 1 and 2, hence the vertices 1 and 2 are the weighted medians and are marked in figure 5.7(b).

![Figure 5.7 a) Median b) Weighted Median](image)
From table 5.4, the weighted median of the large example tree is vertex 1 with \( wd(1) = 4366 \).

### 5.4.1 Evaluating Weighted Median

Finding weighted median \( w\text{-median}(T) \) of a weighted tree needs the following steps.

a. For each vertex \( v \in V \), find \( wd(v) \).

b. Find a vertex \( u \) for which the weighted distance is the minimum.

c. Declare such \( u \) as the weighted median.

These steps are explained with the help of the example tree.

**a. Finding \( wd(v) \) for each \( v \in V \)**

Weighted distance, \( wd(v) \), for each \( v \) can be found by calling the \( W\text{-DISTANCE} \) algorithm. The weighted distances for all the vertices are given in table 5.4.
b. Finding the minimum weighted distance

The standard approach can be used to find the minimum weighted distance in $O(\log n)$ time. The minimum weighted distance is 4366 as per table 5.4.

c. Weighted median

As per definition, the weighted median of a graph or tree is the vertex with minimum weighted distance. Hence, the weighted median of the example tree is vertex 1 as it has the minimum weighted distance.

5.4.2 Parallel Algorithm to find the Weighted Median

The formal algorithm to find the weighted median is given below.

Algorithm W-MEDIAN

Input:

1. The parent array $p(1:n)$, where $p(i)$ is the parent of vertex $i$, and $p(r)=r$.
2. The edge weight array $ew(1:n)$, where $ew(i)$ is the weight of the edge $(i,p(i))$ and $ew(r)=0$
Output: The weighted median, \( w\text{-median}(T) \)

Begin

1. For \( i=1 \) to \( n \) do in parallel
   \[
   wd(i) = W\text{-DISTANCE}(i)
   \]
   end parallel

2. Find in parallel a vertex \( v \) such that
   \[
   wd(v) = \min \{ wd(i) ; 1 \leq i \leq n \}
   \]

3. \( w\text{-median}(T) = v \)

End

Theorem 5.5: The weighted median of a weighted tree can be found in \( O(\log n) \) time using \( O(n^3) \) processors on CREW PRAM.

Proof. Clearly the algorithm computes the weighted median.

Step 1 calls \( W\text{-DISTANCE} \) algorithm in parallel on \( n \) processors and takes \( O(\log n) \) time using \( O(n^3) \) processors. Step 2 clearly needs \( O(\log n) \) time using \( O(n/2) \) processors. Step 3 is a primitive assignment and takes \( O(1) \) unit time. Thus the overall time taken by \( W\text{-MEDIAN} \) algorithm to find \( w\text{-median} \) is \( O(\log n) \) time using \( O(n^3) \) processors.
5.5 Weighted Periphery

A vertex $v$ is a peripheral vertex if $e(v) = diam(T)$, and the periphery is the set of all peripheral vertices. Similarly, a vertex $v$ is a weighted peripheral vertex if $we(v) = w\cdot diam(T)$, and the weighted periphery is the set of all weighted peripheral vertices. To identify the weighted peripheral vertices we need the weighted eccentricities of vertices and the weighted diameter of the tree. In chapter 4 the algorithms to find the weighted eccentricities and the weighted diameter were discussed.

Table 5.5 gives the eccentricities and weighted eccentricities of the vertices of figure 3.1.
\begin{center}
\begin{tabular}{|c|c|c|}
\hline
V  & e(v) & we(v) \\
\hline
1  & 15 & 6000 \\
2  & 16 & 4800 \\
3  & 19 & 8000 \\
4  & 18 & 5400 \\
5  & 23 & 6900 \\
6  & 21 & 9000 \\
7  & 22 & 9500 \\
8  & 26 & 8500 \\
9  & 26 & 11500 \\
\hline
\end{tabular}
\end{center}

\textit{Table 5.5 e(v) and we(v) of figure 3.1}

\begin{center}
\begin{tikzpicture}
  \node {1} [grow=up, sibling distance=5em, level distance=3em]
    child {node {2} [grow=up, sibling distance=4em,level distance=2em]
      child {node {4} [level distance=1em]
        child {node {6} [level distance=1em]
          child {node {8} [level distance=1em]}
          child {node {6} [level distance=3em]
            child {node {100}}}}
          child {node {5} [level distance=3em]
            child {node {100}}}}
      child {node {3} [level distance=4em]
        child {node {7} [level distance=1em]
          child {node {2} [level distance=1em]}
          child {node {7}}}}}
    child {node {3} [grow=up, sibling distance=4em,level distance=2em]
      child {node {2} [level distance=1em]
        child {node {4} [level distance=1em]
          child {node {6} [level distance=1em]
            child {node {8} [level distance=1em]}
            child {node {5} [level distance=3em]
              child {node {100}}}}
          child {node {5} [level distance=3em]
            child {node {100}}}}}
      child {node {3} [level distance=4em]
        child {node {7} [level distance=1em]
          child {node {2} [level distance=1em]}
          child {node {7}}}}}
\end{tikzpicture}
\end{center}

\textit{Figure 5.8 a) Periphery  b) Weighted periphery}
As per table 5.5 the maximum eccentricity is 26. The diameter of the tree is 26. So the peripheral vertices are 8 and 9. Hence the periphery of the tree is \{8,9\}.

The maximum weighted eccentricity is 11500. The weighted diameter of the tree is 11500. So the weighted peripheral vertex is 9. Hence the weighted periphery of the tree is \{9\}.

5.5.1 Parallel Algorithm to find the Weighted Peripheral Vertex

The algorithm to find the weighted peripheral vertex is given below.

Algorithm W-PERIPHERAL-VERTEX

Input:

1. The distance matrix d(1:n,1:n), where d(i,j) is the distance from vertex i to j and d(i,i)=0
2. The array w(1:n), where w(i) is the weight of vertex i

Output: The weighted peripheral vertex, wpv(T)

Begin

1. Call W-ECCENTRICITY algorithm to find we(v) of each vertex v
2. Call W-DIAMETER algorithm to find the w-diam(T)
3. Find in parallel a vertex v such that
\[ \text{we}(v) = w\text{-diam}(T) \]

4. \( \text{wpv}(T) = v \)

End

**Theorem 5.6:** The weighted peripheral vertex of a weighted tree can be found in \( O(\log n) \) time using \( O(n) \) processors on CREW PRAM.

**Proof.** Correctness of the algorithm immediately follows from Theorems 4.11 and 4.13.

Step 1 calls the \textit{W-ECCENTRICITY} algorithm, which takes \( O(\log n) \) time using \( n \) processors. Step 2 calls the \textit{W-DIAMETER} algorithm which takes \( O(\log n) \) time using \( O(n) \) processors. Step 3 clearly needs \( O(1) \) time using \( n \) processors. Step 4 is a primitive assignment and takes \( O(1) \) unit time. Thus the algorithm \textit{W-PERIPHERAL-VERTEX} takes \( O(\log n) \) time using \( O(n) \) processors. \( \Box \)

### 5.5.2 Parallel Algorithm to find the Weighted Periphery

The algorithm to find the weighted peripheral vertex can be slightly modified to give the weighted periphery of the tree. The following is the algorithm to produce the weighted periphery.
Algorithm W-PERIPHERY

Input:
1. The distance matrix $d(1:n,1:n)$, where $d(i,j)$ is the distance from vertex $i$ to $j$ and $d(i,i)=0$
2. The array $w(1:n)$, where $w(i)$ is the weight of vertex $i$

Output: The set of vertices that forms the weighted periphery

Begin
1. Call W-ECCENTRICITY algorithm to find $we(v)$ of each vertex $v$
2. Call W-DIAMETER algorithm to find the $w$-diam($T$)
3. for $i=1$ to $n$ do in parallel
   if $we(i) = w$-diam($T$) then print $i$
End

Theorem 5.7: The weighted periphery of a weighted tree can be found in $O(\log n)$ time using $O(n)$ processors on CREW PRAM.

Proof. By Theorems 4.11 and 4.13 with straightforward parallel comparisons, the correctness of the algorithm immediately follows.

Step 1 calls the W-ECCENTRICITY algorithm, which takes $O(\log n)$ time using $n$ processors. Step 2 calls the W-DIAMETER algorithm which takes $O(\log n)$ time using $O(n)$ processors. Step 3 clearly needs $O(1)$ time using $n$ processors. Thus the algorithm W-PERIPHERY takes $O(\log n)$ time using $O(n)$ processors.
5.6 Summary

In this chapter parallel algorithms for finding a few weighted central structures are developed. Standard parallel techniques like pointer jumping were used. Parallel algorithms were developed for finding the distance from root to a vertex, finding the distance between all pairs of vertices, weighted distance of a vertex \( v \), weighted median, weighted peripheral vertex, and weighted periphery. Table 5.6 provides a summary of the asymptotic bounds of the algorithms developed in this chapter.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>Section</th>
<th>Time</th>
<th>Processor</th>
<th>PRAM model</th>
</tr>
</thead>
<tbody>
<tr>
<td>DISTANCE-FROM-ROOT</td>
<td>5.2.1</td>
<td>( O(\log n) )</td>
<td>( O(n) )</td>
<td>CREW</td>
</tr>
<tr>
<td>ALL-PAIRS-DISTANCE</td>
<td>5.2.2</td>
<td>( O(\log n) )</td>
<td>( O(n^2) )</td>
<td>CREW</td>
</tr>
<tr>
<td>DISTANCE</td>
<td>5.2.3</td>
<td>( O(\log n) )</td>
<td>( O(n^2) )</td>
<td>CREW</td>
</tr>
<tr>
<td>W-DISTANCE</td>
<td>5.3.2</td>
<td>( O(\log n) )</td>
<td>( O(n^2) )</td>
<td>CREW</td>
</tr>
<tr>
<td>W-MEDIAN</td>
<td>5.4.2</td>
<td>( O(\log n) )</td>
<td>( O(n^3) )</td>
<td>CREW</td>
</tr>
<tr>
<td>W-PERIPHERAL-VERTEX</td>
<td>5.5.1</td>
<td>( O(\log n) )</td>
<td>( O(n) )</td>
<td>CREW</td>
</tr>
<tr>
<td>W-PERIPHERY</td>
<td>5.5.3</td>
<td>( O(\log n) )</td>
<td>( O(n) )</td>
<td>CREW</td>
</tr>
</tbody>
</table>

Table 5.6 Parallel algorithms presented in this chapter