Wireless Sensor Networks – Extended Lab 2 Report
Gabriel Planarization & New Routing Algorithm

By:
Kiran Kanukurthy
Amlan Bhattacharya

Task 1 - Gabriel Planarization:

A graph where all pairs of Gabriel neighbors are connected with an edge is called the *Gabriel graph*. Two points A and B are said to be *Gabriel neighbors* if their *diametrical sphere* (i.e. the sphere such that AB is its diameter) does not contain any other points (see Figure 1).

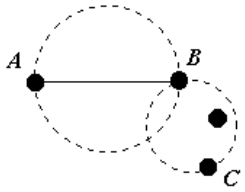


Figure 1: *Gabriel neighbors.*

As Figure 1 demonstrates, the points A and B are Gabriel neighbors, whereas B and C are not. The Gabriel graph can be computed in the following way. For every potential pair of neighbors A and B we verify if any other point X is contained in the diametrical sphere:

$$\text{distance}^2(A, B) > \text{distance}^2(A, X) + \text{distance}^2(B, X)$$

Note that a point on the sphere is considered as contained in sphere.

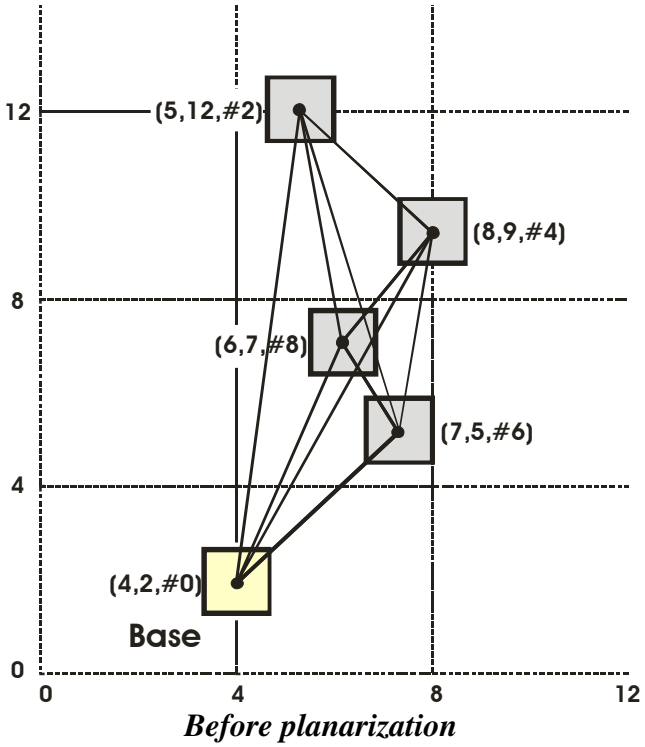
C code for 'planerize' routine:

```
/*---------------------------------------planerization routine-------------------------------------*/
/*
Note:
        c is for current node (the node which calls this routine i.e. myx myy node)
        i is for a potential neighbor of c
        j is any other node that could be inside/outside the circle with dci as its diameter
        d_ci_square = square of distance between nodes "c" and "i"
        d_cj_square = square of distance between nodes "c" and "j"
        d_ij_square = square of distance between nodes "i" and "j"
        node[i].neigbor =1 if "i" is a neighbor of "c"

*/
void planerize( )
{
        int i ,j ;
        int d_ci_square, d_ij_square, d_cj_square;

        for ( i = 0 ; i <= MAX_NNODES  ; i++){
                node[i].neigbour = 1 ;     //make all pairs connected before using Gabriel method
        }

        //Debug code
        printf("The node table before Gabriel Planarization....\r\n");
        printf ("-----------------------------------------------\r\n");
        for ( i = 0;i<=4;i++){
                printf(" node address = %d ",node[i].address);
                printf(" node (x, y) = (%d, %d) ",node[i].x, node[i].y);
                printf(" node neighbor value = %d \r\n ",node[i].neigbour);
                printf("\r\n");
        }
        printf ("-----------------------------------------------\r\n");
        //Debug code

         for ( i = 0 ; i <= MAX_NNODES ; i++){

                d_ci_square  =(node[i].x - myx)*(node[i].x - myx) +  (node[i].y - myy)*(node[i].y - myy);

                for ( j = 0; j <= MAX_NNODES ; j++){

                        d_cj_square = (node[j].x - myx)*(node[j].x - myx)+  (node[j].y - myy)*(node[j].y -
                                                                                                myy);
```

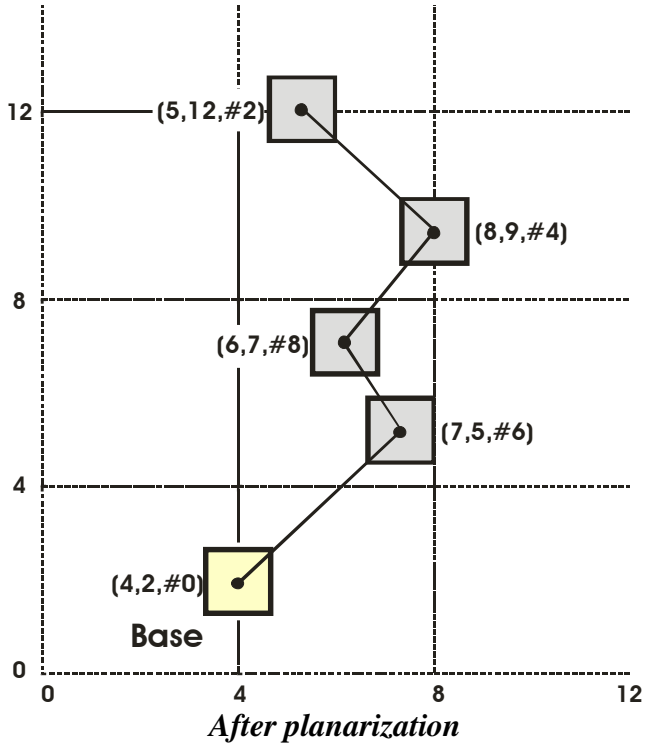
```
                        d_ij_square =  (node[j].x - node[i].x)*(node[j].x - node[i].x) +  (node[j].y -
                                                                node[i].y)*(node[j].y - node[i].y);

                         if (d_ci_square > (d_cj_square + d_ij_square)) {
                               node[i].neigbour = 0 ;
                        }
                        if (d_ci_square == 0) node[i].neigbour = 0;  //current node cannot be its neighbor
                }
        }

        //Debug code
        printf("The node table after Gabriel Planarization....\r\n");
        printf ("_____________________________________________\r\n");
        for ( i = 0;i<=4;i++){
                printf(" node address = %d ",node[i].address );
                printf(" node x y = (%d, %d) ",node[i].x, node[i].y );
                printf(" node neighbor value = %d \r\n ",node[i].neighbor );
                printf("\r\n");
        }
        printf ("_____________________________________________\r\n");
        //Debug code

         return ;
}
/*----------------------------End of planerization routine-----------------------------------------*/
```

What does the code do?

The algorithm starts with all possible pairs of nodes as neighbors i.e. an edge between every possible pair of nodes. Hence, all nodes will have their node[i].neigbor value set to '1'. The decision criterion is applied and edges are removed based on the outcome. When an edge is removed, the node[i].neigbor value is set to '0'. At the end of the routine, the only nodes with their node[i].neigbor value set to '1' are the true neighbors of the current node.



Before planarization



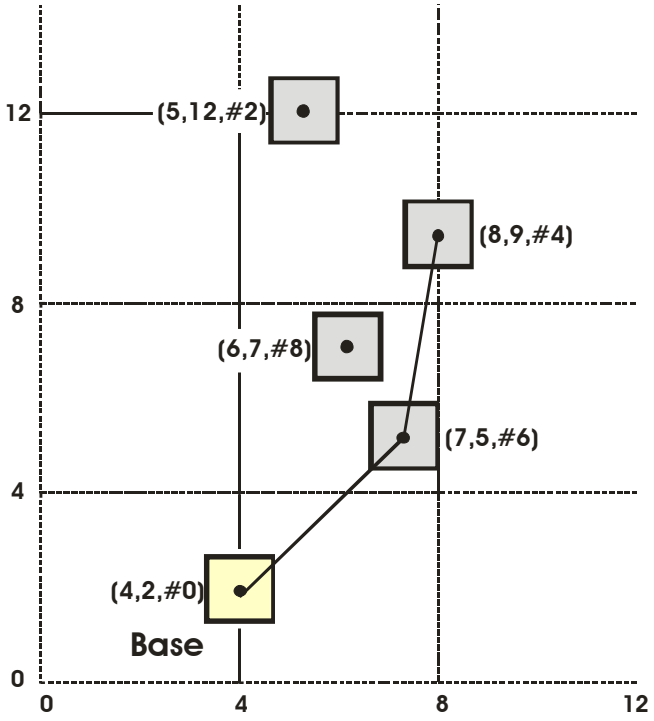
After planarization

'planerize' in action:

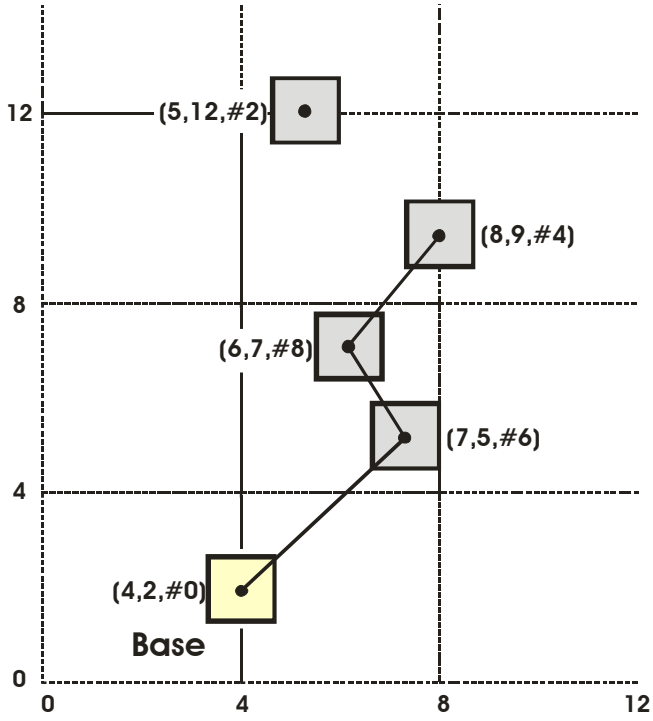
The following two scenarios emphasize how 'planerize' works.

Scenario 1: Switch is pressed on 'node 4' or 'node 2' so that 'node 4' is made to route a packet to the base station. 'findRoute' routine uses the node[i].neigbor values and determines the route. Both

'discoverNodes' and 'planerize' routines set the node[i].neigbor values and the following illustration depicts the behavioral difference between the above two routines.

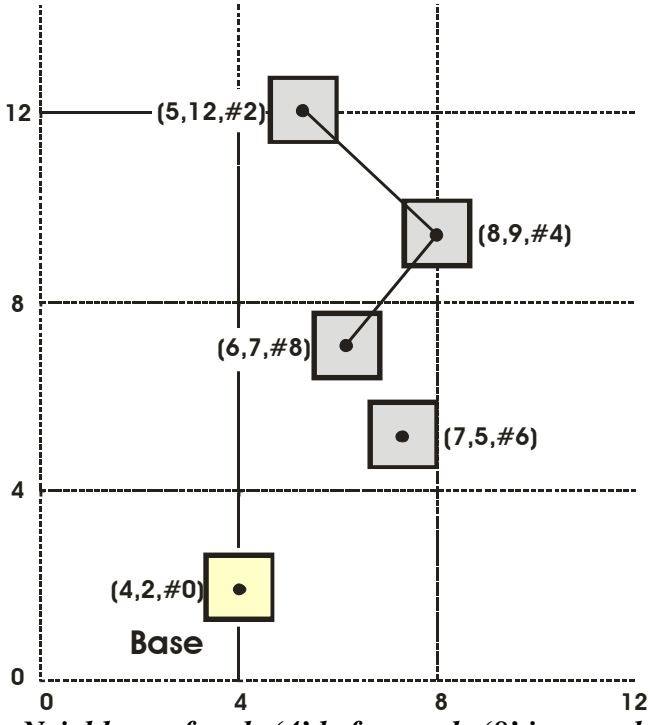


Routing based on 'discoverNodes'

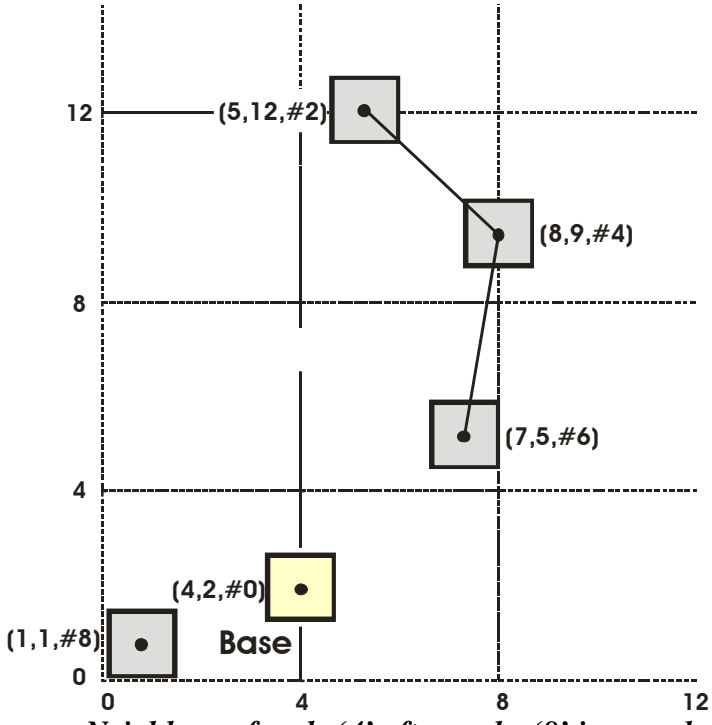


Routing based on 'planerize'

Scenario 2: 'node 8' is moved farther away from 'node 4' to the location (1, 1) and a change in neighbors of 'node 4' is noticed. Packets from 'node 4' to base will now be routed through 'node 6' instead of through 'node 8'.



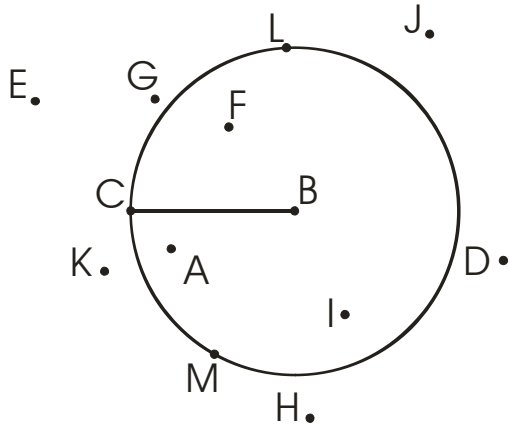
Neighbors of node '4' before node '8' is moved



Neighbors of node '4' after node '8' is moved

Task 2 - New Routing algorithm:
Our routing algorithm is a greedy distance algorithm based on an elimination strategy. The algorithm works in two stages. The nodes who are potential recipients of the packet, neighbors, are first decided. One among the neighbors is then chosen to route the packet to the base station.

Stage 1: Elimination of nodes and choosing the neighbors
Nodes that are in the opposite direction from the base station and/or are farther from the base station than the current node are eliminated. In other words, all nodes lying outside the circle with base as the center and radius d_{cb} are eliminated as illustrated below.



The set of nodes {D, E, G, H, J, K, L, M} are eliminated and Node 'C' will have the set of nodes {A, B, I, F} as its neighbors.
Note that the base station 'B' is also included in the set. Also note that the nodes 'L', 'M' that are on the circle are eliminated. This is to prevent the scenario when nodes located on the circle pass the packet between themselves and the base station never receives the packet. In the case when all nodes are present on the circle, they route the packet directly to the base station, which is intuitive.

Stage 2: Choosing the recipient of packet from set of neighbors
We employ greedy distance criterion here in choosing the recipient. The node that is nearest to us among the set of neighbors is chosen as the recipient of the packet. In the above case, 'C' chooses 'A' as the recipient and routes its packet to 'A'. This is done to economize the energy consumption of the whole network. By increasing the number of hops and reducing the hop distance, more nodes are involved in routing and energy consumption of the whole network is reduced. It is to be noted that one can also use RSSI and not distance in choosing the recipient.

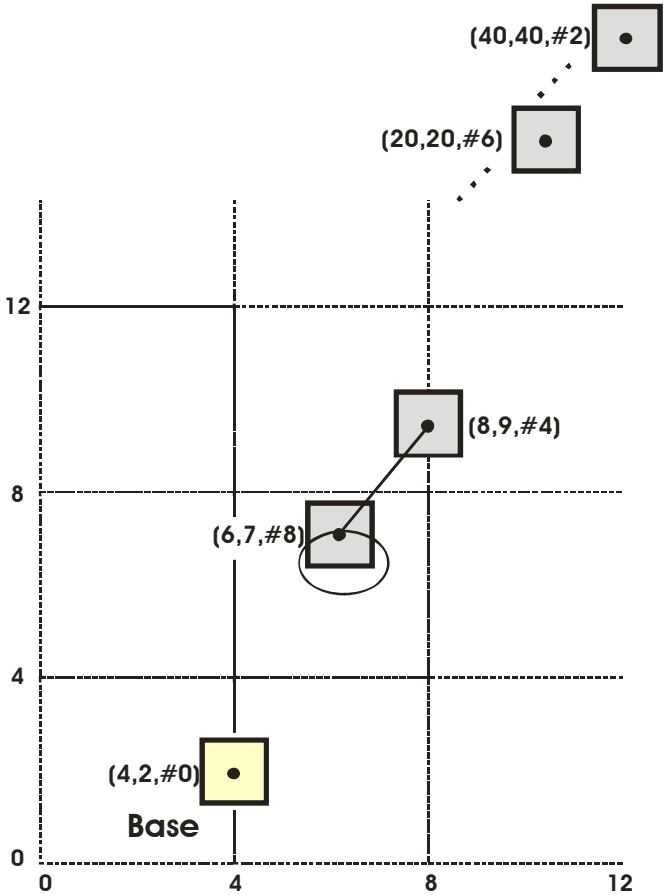
C code for 'findRouteNew' routine:

```
/*-----------------------------findRouteNew routine-----------------------------------------------*/
/*
Note:
        c is current node (the node which calls this routine i.e. myx myy node)
        i is potential neighbor of c
        j is any node from the set of neighbors
        b is base station
        d_cb_square = square of distance between nodes "c" and "b"
        d_ib_square = square of distance between nodes "i" and "b"
        d_cj_square = square of distance between nodes "c" and "j"
        node[i].neigbor =1 if "i" is a neighbor of "c"

*/
int findRouteNew ()
{
        int i,j,d_cb_sq,d_ib_sq,n,d_cj_sq;
        int min = 1000;

        for ( i = 0 ; i <= MAX_NNODES  ; i++){
                node[i].neigbour = 0 ;
        }
```

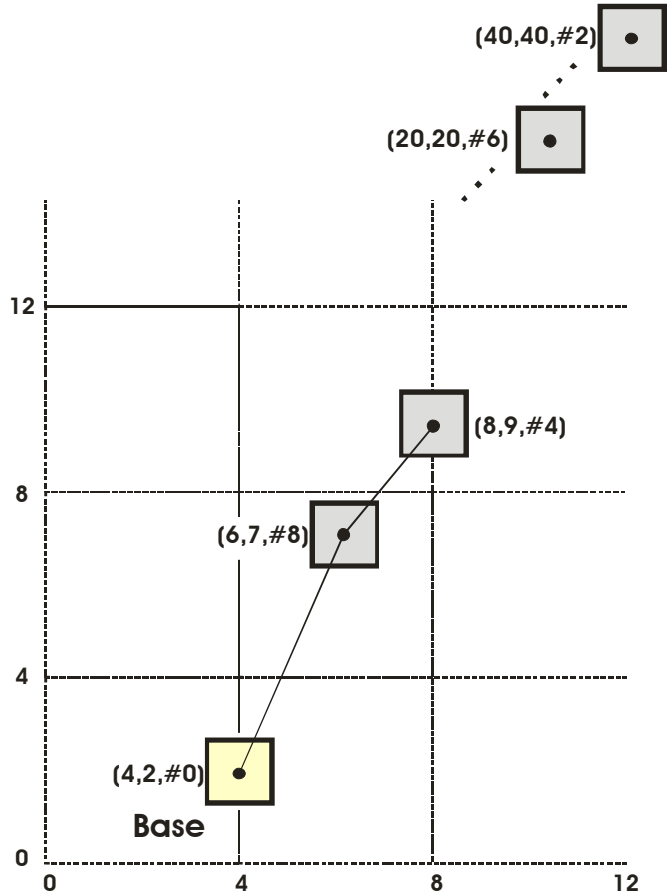
```
//eliminate nodes in opposite direction and/or farther away from base station than current node
        for (i=0;i<=MAX_NNODES;i++)
        {
                d_cb_sq  =  (basex - myx)*(basex - myx) + (basey - myy)*(basey - myy);
                d_ib_sq  =  (node[i].x - basex)*(node[i].x - basex) + (node[i].y - basey)*(node[i].y -
                                                                                                basey);
                if ((d_cb_sq - d_ib_sq) > 0) node[i].neigbour = 1;
        }
//choose a recipient from the set of neighbors that was created above
//choose the one closest to the current node
        for(j=0;j<=MAX_NNODES;j++)
        {
                 if (node[j].neigbour == 1)
                {
                        d_cj_sq  =   (node[j].x - myx)*(node[j].x - myx) +  (node[j].y - myy)*(node[j].y -
                                                                                                myy);

                        if (min >=  d_cj_sq)
                        {
                                n = j;
                                min = d_cj_sq;
                        }
                }
        }
         // printf(" the nearest node is %d \r\n", node[n].address ); //debug code
         return n;
}
/*----------------------------End of findRouteNew routine----------------------------------------*/
```

'findRouteNew' in action:
The following two scenarios emphasize how 'findRouteNew' works.

Scenario 1: This scenario illustrates the behavioral difference in between 'findRoute' and 'findRouteNew' routines. Nodes '6' and '2' are moved from their original positions to (20,20) and (40,40) respectively. Switch is pressed on 'node 4' so that 'node 4' is made to route a packet to the base station.

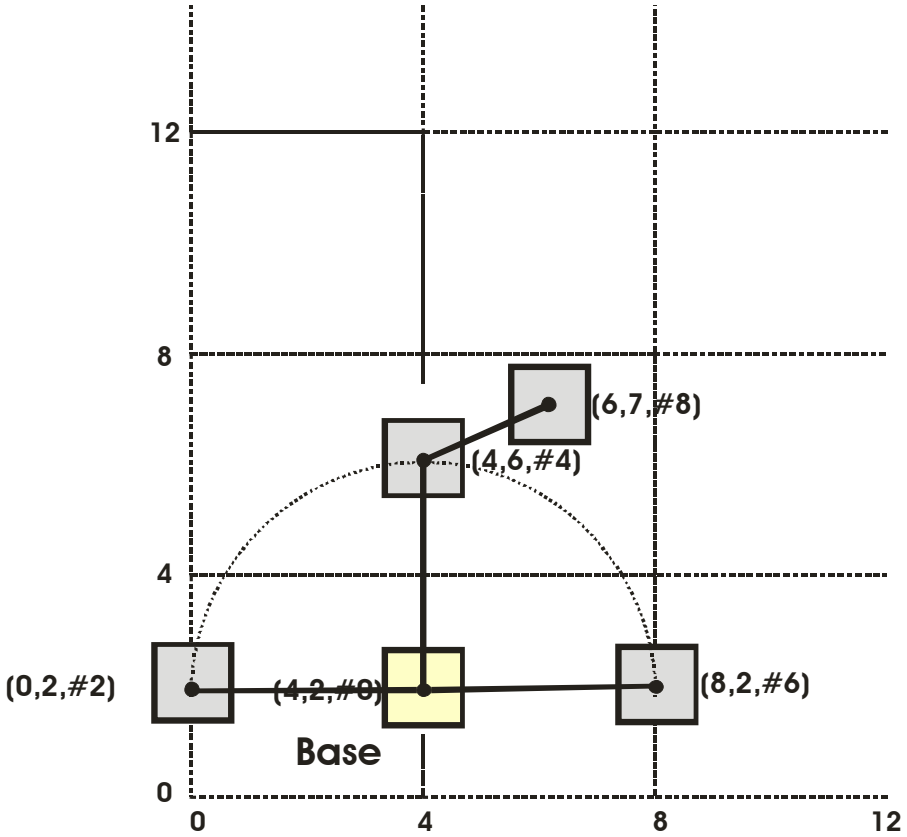


Packet stuck at node '8' using findRoute



Packet routed to base using findRouteNew

Scenario 2: This scenario illustrates the case when nodes are present on a circle and base station is inside the circle.



Nodes '2', '4', '6' route directly to base, node '8' routes through node '4'