```
#ifndef DFS_H_
#define DFS_H_

class dfsGraph
{
public:
    dfsGraph(int vN);
        //Constructor
    void addEdge(cities v, cities w);
        //Adds a new edge to the graph
    void dfsTraversal(cities start);
        //Depth first traversal
    void printEdges();
        //Prints out the discovery and cross edges
private:
    int vNum; //number of vertices
    vector<cities> *adjList; //adjacency list
    vector<edge> backEdges; //vector of the back edges
    vector<edge> discoveryEdges; //vector of the discovery edges

    void dfsUtil(cities v, bool visited[], int& tDist);
        //Recursive function for the Depth first traversal
    int findDistBtwn(cities v1, cities v2);
        //Find the distance between 2 vertices
        //returns int of the distance
    void bubbleSort(cities v, vector<cities>& a);
        //Vector bubble sort (least to greatest)
};

dfsGraph::dfsGraph(int vN)
{
    vNum = vN;
    adjList = new vector<cities>[vNum];

    addEdge(Seattle, Chicago);
    addEdge(Seattle, Denver);
    addEdge(Seattle, SanFrancisco);
    addEdge(SanFrancisco, Seattle);
    addEdge(SanFrancisco, Denver);
    addEdge(SanFrancisco, LosAngeles);
    addEdge(Denver, Seattle);
    addEdge(Denver, SanFrancisco);
    addEdge(Denver, LosAngeles);
    addEdge(Denver, KansasCity);
    addEdge(Denver, Chicago);
    addEdge(Chicago, Seattle);
    addEdge(Chicago, Denver);
    addEdge(Chicago, KansasCity);
    addEdge(Chicago, NewYork);
    addEdge(Chicago, Boston);
    addEdge(Boston, Chicago);
    addEdge(Boston, NewYork);
    addEdge(NewYork, Boston);
    addEdge(NewYork, Chicago);
    addEdge(NewYork, Atlanta);
    addEdge(NewYork, KansasCity);
    addEdge(LosAngeles, SanFrancisco);
    addEdge(LosAngeles, Denver);
    addEdge(LosAngeles, KansasCity);
    addEdge(LosAngeles, Dallas);
    addEdge(KansasCity, LosAngeles);
    addEdge(KansasCity, Denver);
    addEdge(KansasCity, Chicago);
    addEdge(KansasCity, NewYork);
    addEdge(KansasCity, Atlanta);
    addEdge(KansasCity, Dallas);
    addEdge(Atlanta, NewYork);
    addEdge(Atlanta, KansasCity);
    addEdge(Atlanta, Dallas);
    addEdge(Atlanta, Houston);
    addEdge(Atlanta, Miami);
    addEdge(Dallas, LosAngeles);
    addEdge(Dallas, KansasCity);
    addEdge(Dallas, Atlanta);
    addEdge(Dallas, Houston);
    addEdge(Houston, Dallas);
    addEdge(Houston, Atlanta);
    addEdge(Houston, Miami);
    addEdge(Miami, Atlanta);
    addEdge(Miami, Houston);
}

void dfsGraph::addEdge(cities v, cities w)
{
    adjList[v].push_back(w);
}

void dfsGraph::dfsTraversal(cities start)
{
    int totalDist = 0;
    bool *visited = new bool[vNum];
    for(int i = 0; i < vNum; i++)
    {
        visited[i] = false;
    }

    dfsUtil(start, visited, totalDist);
    cout << "\nTotal Distance Traveled: " << totalDist << " miles.\n";
}

void dfsGraph::dfsUtil(cities v, bool visited[], int& tDist)
{
    visited[v] = true;
    cout << enumGet(v) << endl;

    bubbleSort(v, adjList[v]);

    vector<cities>::iterator i;
    for(i = adjList[v].begin(); i != adjList[v].end(); i++)
    {
        if(!visited[*i])
        {
            discoveryEdges.push_back(edge(v, *i));
            tDist += findDistBtwn(v, *i);
            dfsUtil(*i, visited, tDist);
        }
        else
        {
            backEdges.push_back(edge(v, *i));
        }
    }
}

int dfsGraph::findDistBtwn(cities v1, cities v2)
{
    int distBtwn;
    int j = 0;
    while (j < 23)
    {
        if((distances[j].city1 == v1 && distances[j].city2 == v2) ||
           (distances[j].city2 == v1 && distances[j].city1 == v2))
        {
            distBtwn = distances[j].distance;
            break;
        }
        else
        {
            j++;
        }
    }
    return distBtwn;
}

void dfsGraph::bubbleSort(cities v, vector<cities>& a)
{
      bool swap = true;
      while(swap)
      {
        swap = false;
        int size = a.size();
        for (int i = 0; i < size -1; i++)
        {
            if (findDistBtwn(v, a[i]) > findDistBtwn(v, a[i+1]))
            {
                cities temp = a[i];
                a[i] = a[i+1];
                a[i+1] = temp;
                swap = true;
            }
        }
    }
}

void dfsGraph::printEdges()
{
    cout << "\nDiscovery Edges:";
    vector<edge>::iterator i;
    for(i = discoveryEdges.begin(); i != discoveryEdges.end(); i++)
    {
        cout << endl << enumGet(i->city1) << "->" << enumGet(i->city2);
    }

    cout << "\n\nBack Edges:";
    for(i = backEdges.begin(); i != backEdges.end(); i++)
    {
        cout << endl << enumGet(i->city1) << "->" << enumGet(i->city2);
    }
}

#endif /* DFS_H_ */

```