```
class Node(object):
    def __init__(self, name,h=0,parent=None):
        """Assumes name is a string"""
        self.name = name
        self.parent = parent
        self.h=h
    def getName(self):
        return self.name
    def getHeuristic(self):
        return self.h
    def __str__(self):
        return self.name

class Edge(object):
    def __init__(self, src, dest):
        """Assumes src and dest are nodes"""
        self.src = src
        self.dest = dest
        self.dest.parent = src
    def getSource(self):
        return self.src
    def getDestination(self):
        return self.dest
    def __str__(self):
        return self.src.getName() + '->' + self.dest.getName()

class WeightedEdge(Edge):
    def __init__(self, src, dest, weight = 1.0):
        """Assumes src and dest are nodes, weight a float"""
        self.src = src
        self.dest = dest
        self.weight = weight
    def getWeight(self):
        return self.weight
    def __str__(self):
        return self.src.getName() + '->(' + str(self.weight) + ')'\
               + self.dest.getName()
```

```
class Digraph(object):
    #nodes is a list of the nodes in the graph
    #edges is a dict mapping each node to a list of its children
    def __init__(self):
        self.nodes = []
        self.edges = {}
    def addNode(self, node):
        if node in self.nodes:
            raise ValueError('Duplicate node')
        else:
            self.nodes.append(node)
            self.edges[node] = []
    def addEdge(self, edge):
        src = edge.getSource()
        dest = edge.getDestination()
        if not(src in self.nodes and dest in self.nodes):
            raise ValueError('Node not in graph')
        self.edges[src].append(dest)
    def adjacent_nodes(self, node):
        return self.edges[node]
    def hasNode(self, node):
        return node in self.nodes
    def __str__(self):
        result = ''
        for src in self.nodes:
            for dest in self.edges[src]:
                result = result + '[('+src.getName() + ','+str(src.getHeuristic())+')->('\
                         + dest.getName() + ','+str(dest.getHeuristic())+')]\n'
        return result[:-1] #omit final newline
```

```
#Heuristics data
Cities = [('Arad',366),('Zerind',374),('Timisoara',329),
          ('Sibiu',253),('Oradea',380),('Lugoj',244),('Fagaras',176),
          ('Rimniu Vilcea',193),('Mehadia',241),('Pitesti',101),
          ('Dobreta',242),('Craiova',160),('Bucharest',0),
          ('Giurgiu',77),('Urziceni',80),('Hirsova',151),
          ('Eforie',161),('Vasliu',199),('Iasi',226),('Neamt',234)]
```

```
def search(graph,start, end):
    return HillClimbing(graph, start, end)

def testGraph():
    nodes = []
    for city,h in Cities:
        nodes.append(Node(city,h))
    g = Digraph()
    for n in nodes:
        g.addNode(n)
    g.addEdge(WeightedEdge(nodes[0],nodes[1],75))
    g.addEdge(WeightedEdge(nodes[0],nodes[2],118))
    g.addEdge(WeightedEdge(nodes[0],nodes[3],140))
    g.addEdge(WeightedEdge(nodes[1],nodes[4],71))
    g.addEdge(WeightedEdge(nodes[4],nodes[3],151))
    g.addEdge(WeightedEdge(nodes[2],nodes[5],111))
    g.addEdge(WeightedEdge(nodes[3],nodes[6],99))
    g.addEdge(WeightedEdge(nodes[3],nodes[7],80))
    g.addEdge(WeightedEdge(nodes[6],nodes[12],211))
    g.addEdge(WeightedEdge(nodes[7],nodes[9],97))
    g.addEdge(WeightedEdge(nodes[7],nodes[11],146))
    g.addEdge(WeightedEdge(nodes[9],nodes[12],101))
    # not complete
    print g
    print "Search Result (Hill Climbing):", search(g,nodes[0],nodes[12])


# Find the minimum heuristic for each Node and expand it.
def HillClimbing(graph,start,end):
    """Assumes start and end are nodes"""
    print "\n start:",start
    print "end:", end
    bestNode = start
    loopCount=0
    while True:
        if bestNode == end :
            print "\n Found the goal : "
            return bestNode
        L = graph.adjacent_nodes(bestNode)
        print "\n L: ",
        for each in L:
            print each,
        nodeEval=1000
        nextNode=None
        loopCount += 1
        if loopCount > 10:
            print "Cycle in graph, no solution."
            return None
        if len(L) != 0:
            for node in L:
                if nodeEval > node.h:
                    nodeEval = node.h
                    nextNode = node
            if bestNode.h < nodeEval:
                print "\n Local optmal solution."
                return bestNode
            else:
                bestNode = nextNode
        else:
            print "\n All nodes searched. No solution found."
            return None

testGraph()
```