BFS

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self,u,v):
        self.graph[u].append(v)
    def BFS(self, s):
        visited = [False] * (max(self.graph) + 1)
        queue = []
        queue.append(s)
        visited[s] = True
        while queue:
            s = queue.pop(0)
            print (s, end = " ")
            for i in self.graph[s]:
                if visited[i] == False:
                    queue.append(i)
                    visited[i] = True
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print ("Following is Breadth First Traversal (starting from vertex 2)")
g.BFS(2)
```

DFS

```
from collections import defaultdict
class Graph:
    def __init__(self):
        self.graph = defaultdict(list)
    def addEdge(self, u, v):
        self.graph[u].append(v)
    def DFSUtil(self, v, visited):
        visited.add(v)
        print(v, end=' ')
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
    def DFS(self, v):
        visited = set()
        self.DFSUtil(v, visited)
g = Graph()
g.addEdge(0, 1)
g.addEdge(0, 2)
g.addEdge(1, 2)
g.addEdge(2, 0)
g.addEdge(2, 3)
g.addEdge(3, 3)
print("Following is DFS from (starting from vertex 2)")
g.DFS(2)
```

Best First Search

```
from queue import PriorityQueue
v = 14
graph = [[] for i in range(v)]
def best_first_search(source, target, n):
	visited = [0] * n
	visited = True
	pq = PriorityQueue()
	pq.put((0, source))
	while pq.empty() == False:
		u = pq.get()[1]
		print(u, end=" ")
		if u == target:
			break
		for v, c in graph[u]:
			if visited[v] == False:
				visited[v] = True
				pq.put((c, v))
	print()
def addedge(x, y, cost):
	graph[x].append((y, cost))
	graph[y].append((x, cost))
addedge(0, 1, 3)
addedge(0, 2, 6)
addedge(0, 3, 5)
addedge(1, 4, 9)
addedge(1, 5, 8)
addedge(2, 6, 12)
addedge(2, 7, 14)
addedge(3, 8, 7)
addedge(8, 9, 5)
addedge(8, 10, 6)
addedge(9, 11, 1)
addedge(9, 12, 10)
addedge(9, 13, 2)
source = 0
target = 9
best_first_search(source, target, v)
```

A*

```
def Astar(start, Goal):
        open_list = set(start)
        closed_list = set()
        g = {}          #store distance from starting node
        parents = {}  # parents contains an adjacency map of all node
        g[start] = 0   #distance of starting node from itself is zero
        #start is root node i.e it has no parent nodes
        #so start is set to its own parent node
        parents[start] = start
        while len(open_list) > 0:
            n = None
            #node with lowest f() is found
            for v in open_list:
                if n == None or g[v] + heuristic(v) < g[n] + heuristic(n):
                    n = v
            if n == Goal or Graph[n] == None:
                pass
            else:
                for (m, weight) in get_neighbors(n):
                    #nodes 'm' not in first and last set are added to first
                    #n is set its parent
                    if m not in open_list and m not in closed_list:
                        open_list.add(m)
                        parents[m] = n
                        g[m] = g[n] + weight
                    #for each node m,compare its distance from start i.e g(m) to the
                    #from start through n node
                    else:
                        if g[m] > g[n] + weight:
                            #update g(m)
                            g[m] = g[n] + weight
                            #change parent of m to n
                            parents[m] = n
                            #if m in closed set,remove and add to open
                            if m in closed_list:
                                closed_list.remove(m)
                                open_list.add(m)
            if n == None:
                print('Path does not exist!')
                return None
            # if the current node is the Goal
            # then we begin reconstructin the path from it to the start
            if n == Goal:
```

```
            path = []
            while parents[n] != n:
                path.append(n)
                n = parents[n]
            path.append(start)
            path.reverse()
            print('Path found: {}'.format(path))
            return path

        # remove n from the open_list, and add it to closed_list
        # because all of his neighbors were inspected
        open_list.remove(n)
        closed_list.add(n)
    print('Path does not exist!')
    return None

#define fuction to return neighbor and its distance
#from the passed node
def get_neighbors(v):
    if v in Graph:
        return Graph[v]
    else:
        return None
#for simplicity we ll consider heuristic distances given
#and this function returns heuristic distance for all nodes
def heuristic(n):
        H_dist = { 'A': 11, 'B': 6, 'C': 99, 'D': 1, 'E': 7, 'G': 0 }
        return H_dist[n]

#Describe your graph here
Graph = {
    'A': [('B', 2), ('E', 3)],
    'B': [('C', 1),('G', 9)],
    'C': None,
    'E': [('D', 6)],
    'D': [('G', 1)]
}
Astar('A', 'G')
```

Linear Regression

https://www.kaggle.com/akashsikarwar/simple-linear-regression

https://www.kaggle.com/akashsikarwar/height-weight-linear-regression-model

https://www.kaggle.com/akashsikarwar/udemyallfinanceaccounting-multi-linear-regression

Logistic Regression

https://www.kaggle.com/akashsikarwar/heart-disease-prediction-logistic-regression

Clustering (K means)

https://www.kaggle.com/andyxie/k-means-clustering-implementation-in-python

Or gate perceptron

```
import numpy as np
def unitStep(v):
	if v >= 0:
		return 1
	else:
		return 0

def perceptronModel(x, w, b):
	v = np.dot(w, x) + b
	y = unitStep(v)
	return y

# w1 = 1, w2 = 1, b = -0.5
def OR_logicFunction(x):
	w = np.array([1, 1])
	b = -0.5
	return perceptronModel(x, w, b)

# testing the Perceptron Model
```

```
test1 = np.array([0, 1])
test2 = np.array([1, 1])
test3 = np.array([0, 0])
test4 = np.array([1, 0])

print("OR({}, {}) = {}".format(0, 1, OR_logicFunction(test1)))
print("OR({}, {}) = {}".format(1, 1, OR_logicFunction(test2)))
print("OR({}, {}) = {}".format(0, 0, OR_logicFunction(test3)))
print("OR({}, {}) = {}".format(1, 0, OR_logicFunction(test4)))
```

Hill Climbing (Travelling Salesman Problem)

```
import random

def randomSolution(tsp):
    cities = list(range(len(tsp)))
    solution = []
    for i in range(len(tsp)):
        randomCity = cities[random.randint(0, len(cities) - 1)]
        solution.append(randomCity)
        cities.remove(randomCity)
    return solution

def routeLength(tsp, solution):
    routeLength = 0
    for i in range(len(solution)):
        routeLength += tsp[solution[i - 1]][solution[i]]
    return routeLength

def getNeighbours(solution):
    neighbours = []
    for i in range(len(solution)):
        for j in range(i + 1, len(solution)):
            neighbour = solution.copy()
            neighbour[i] = solution[j]
            neighbour[j] = solution[i]
            neighbours.append(neighbour)
    return neighbours

def getBestNeighbour(tsp, neighbours):
    bestRouteLength = routeLength(tsp, neighbours[0])
    bestNeighbour = neighbours[0]
    for neighbour in neighbours:
        currentRouteLength = routeLength(tsp, neighbour)
```

```
        if currentRouteLength < bestRouteLength:
            bestRouteLength = currentRouteLength
            bestNeighbour = neighbour
    return bestNeighbour, bestRouteLength

def hillClimbing(tsp):
    currentSolution = randomSolution(tsp)
    currentRouteLength = routeLength(tsp, currentSolution)
    neighbours = getNeighbours(currentSolution)
    bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    while bestNeighbourRouteLength < currentRouteLength:
        currentSolution = bestNeighbour
        currentRouteLength = bestNeighbourRouteLength
        neighbours = getNeighbours(currentSolution)
        bestNeighbour, bestNeighbourRouteLength = getBestNeighbour(tsp, neighbours)

    return currentSolution, currentRouteLength

def main():
    tsp = [
        [0, 400, 500, 300],
        [400, 0, 300, 500],
        [500, 300, 0, 400],
        [300, 500, 400, 0]
    ]

    print(hillClimbing(tsp))

if __name__ == "__main__":
    main()
```