Ahsanullah University of Science and Technology

**Department of Computer Science And Engineering**

**Course No:** 4108

**Course Title:** Artificial Intelligence Lab

**AssignmentNo:**03

**Session:** Spring2020

**Date of Submission:**15th march, 2021

**Submitted To:**

Tonmoy Hossain Lecturer, Department Of CSE

Rayhan Ahmed Lecturer, Department Of CSE

**Submitted by:**

**Name**: Faiza Anan Noor

**Year**: 4th

**Semester**: 1st

**Lab Group**: B2

**ID**: 17.01.04.093

# Lab Exercise :

## Questions :

1) Write a Python program that reads the file created as demonstrated into a dictionary taking ‘name’ as the
key and a list consisting of ‘dept’ and ‘cgpa’ as the value for each line. Make changes in some ‘cgpa’ and then
write back the whole file.
2) Implement in generic ways (as multi-modular and interactive systems) the Greedy Best-First and A* search
algorithms in Prolog and in Python.

## Ans to the Question no. 1 :

### Code in Python:

```
print("The contents of the file before updation: ")

f=open('input.txt','r')

d = {}

flag=0

for line in f:

   temp = line.strip().split()

   d[temp[0]]=[temp[1],temp[2]]

   print(line)

inp = input('Enter Name of the entry to change: ')

for k,v in d.items():

   if k==inp:
       d[k][1]=input('Enter the new CGPA of that entry: ')
       flag=0
       break
   else:

       flag=1

f.close()

if flag==0:
   print("The contents of the file after updation: ")
   f = open('input.txt','w')
   for k, v in d.items():
       x = str(k) + ' ' + str(v[0]) +' ' + str(v[1]) + '\n'
       print(x)
       f.write(x)
   f.close()
else:
     print("NO SUCH ENTRY FOUND,CHANGE NOT POSSIBLE")
```

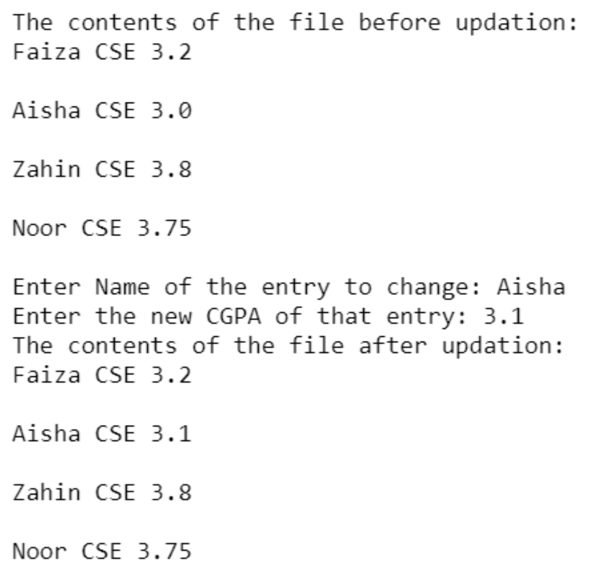
## Explanation :

For this question we were asked to create a program that reads the file created as demonstrated into a dictionary taking ‘name’ as the key and a list consisting of ‘dept’ and ‘cgpa’ as the value for each line. Then we were asked to changes in some ‘cgpa’ and then write back the whole file.

The working procedure for this is explained below:

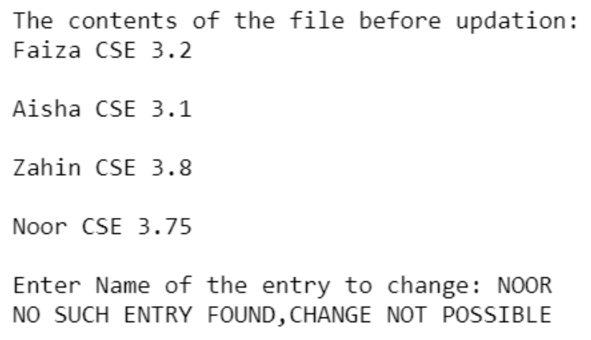
- Here, firstly we initialized an empty dictionary.
- We had an input file where we kept the name, dept and cgpa of some students.
- We iterated through each line of the file and then split them into dept and cgpa and mapped them into the dictionary.
- After that we took an input of whose cgpa we wanted to change.
- If we found the name in the dictionary, we took the cgpa as input. This cgpa will be the updated cgpa of the particular name we took as input.
- If we couldn’t find the name, we gave an alert or showed a message saying that an entry with the name couldn’t be found and thus change wasn’t possible.
- Lastly, we updated the contents of the dictionary with our desired CGPA and then wrote the changes into the same file and then printed out the contents of the file.

## Python Output :



```
The contents of the file before updation:
Faiza CSE 3.2

Aisha CSE 3.0

Zahin CSE 3.8

Noor CSE 3.75

Enter Name of the entry to change: Aisha
Enter the new CGPA of that entry: 3.1
The contents of the file after updation:
Faiza CSE 3.2

Aisha CSE 3.1

Zahin CSE 3.8

Noor CSE 3.75
```

Fig1.1: If an entry was found with the input name



```
The contents of the file before updation:
Faiza CSE 3.2

Aisha CSE 3.1

Zahin CSE 3.8

Noor CSE 3.75

Enter Name of the entry to change: NOOR
NO SUCH ENTRY FOUND,CHANGE NOT POSSIBLE
```

Fig1.2: If an entry wasn’t found with the input name

- Here in this case, firstly we printed the existing contents of the file and stored it into a dictionary. Then we took the name of the student whose cgpa we want to change
- If we found the name in the dictionary, we took the cgpa as input
- If we couldn’t find the name, we gave an alert that an entry with the name couldn’t be found and thus change wasn’t possible
- Lastly, we updated the contents of the dictionary with our desired CGPA and then wrote the changes into the same file and then printed out the contents of the file.

# Ans to the Q no. 2

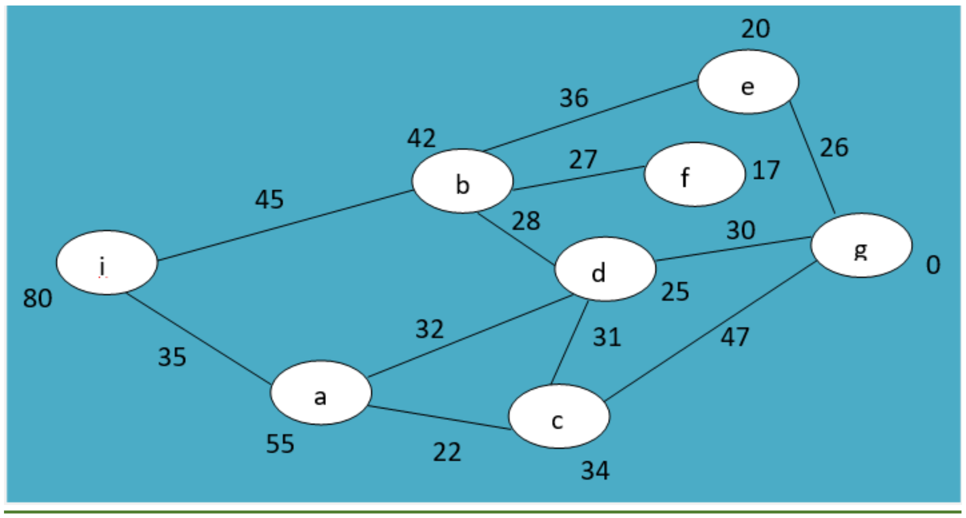


Fig 2.1: The graph for our problem

## GBFS:

**Code In Python:**

```
from queue import PriorityQueue


graph = [[(2, 22), (3, 32), (8, 35)],
      [(3, 28),(4, 36), (5, 27), (8, 45)],
      [(0, 22), (3, 31), (6, 47)],
      [(0, 32), (2, 31), (1, 28), (6, 30)],
      [(1, 36), (6, 26)],
        [(1, 27)],
        [(2, 47), (3, 30), (4, 26)],
        [],
        [(0, 35), (1, 45)]]

h = [('a',55),('b', 42),('c', 34),('d', 25),('e', 20),('f', 17),('g' ,0),('h', 99999),('i', 80)]

Q = PriorityQueue()

def gbfs_exec():

    startSymb='a'
    intstartSymb=ord(startSymb)

    source=str(input('Enter the starting symbol:: '))
    source=ord(source)-intstartSymb
    goal=str(input('Enter the goal symbol:: '))
    goal=ord(goal)-intstartSymb



    parent = [-1] * len(graph)
    visited = [False] * len(graph)

    Q.put((h[source][1], source))
```

```
visited[source]=True
while(Q.empty()==False):
    x=Q.get()
    if(x[1]==goal):
        break
    for adj in graph[x[1]]:
        v = adj[0]
        if(visited[v]==False):
            Q.put((h[v][1],v))
            visited[v]=True
            parent[v]=x[1]

tmp = goal
cost = 0
print("The Possible Path Is :: ")
while (parent[tmp]!=-1):
    x = parent[tmp]
    v = tmp
    print(chr(ord('a')+tmp))
    for adj in graph[x]:
        if(adj[0]==v):
            cost+=adj[1]
            break
    tmp = parent[tmp]
    print(chr(ord('a')+tmp))
    print("Total Cost Found :: ", cost)
gbfs_exec()
```

## Explanation:

The working procedure is explained in points below:

1. Firstly, we used two tuples called **“graph”** and **“h”** to store the adjacent vertices of each of our nodes and the heuristics value respectively in a sequential way. So the node called a is the first vertex, the node called b is the 2nd vertex and so on.

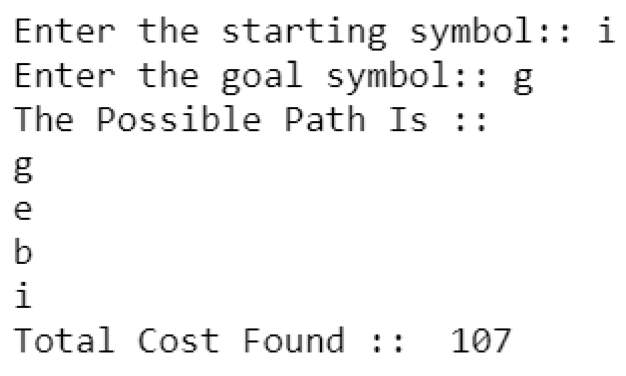
2. A Priority Queue (PQ), which contains nodes in ascending order of h-values, is maintained which was called **“Q”** in our case. For this we imported PriorityQueue module from Queue.
3. We also kept two arrays to denote what the parent of a particular node is, that is where it came from. And also to denote whether a node has been visited or not. For this the two arrays were **“parent “** and **“visited”** respectively.
4. Then we took inputs of our source and goal node using variables **“source”** and **“goal”.**
5. Then we pushed the value of the source nodes heuristic value and also the name of the source itself into the priority queue and marked its visited value as true.
6. Then we analyzed the contents of the priority queue as long as its not empty. We iterated through the adjacent nodes of the initial staring node and then we pushed the adjacent nodes and their heuristic values into the Queue.
7. Then we continued our iteration by popping off the node that has the smallest heuristic value since and also analyzed its adjacent nodes in the same way. We also marked each node as visited once we visited them and also updated the parent values accordingly.
8. Then we printed out our all possible paths by printing out the contents of the **“parent”** array and also calculated the sum and printed it.

9. This method gave us the solution that wasn't the most optimal since it followed greedy approach by choosing an approach that seemed the best in a particular instance but didn't guarantee the least cost in the long run.

So in short ,the most fundamental points are:

I. The process terminates when the destination node is placed in the Priority Queue, and consequently, selected for visit.
II. The 1st node from the Queue is selected repeatedly, and each time the tree, the Queue and the Parent arrays are updated:
III. The node in the tree is marked visited and its neighbors from the graph are added to the tree as its children, while no repeated node is allowed in the tree;
IV. The node itself is deleted from the PQ, but its children are added to the Priority Queue .
V. The Possible Path/Parent array is straightened up to the root from the selected node.

## Python Output :



```
Enter the starting symbol:: i
Enter the goal symbol:: g
The Possible Path Is ::
g
e
b
i
Total Cost Found ::  107
```

- Here in this case we specified out start and goal symbol
- Then we executed our function needed to find out the most optimal path from our start to goal state which was i->b->e->g
- Here out of all paths, the cost for this specific path was not the least and it was calculated as 107.
- As opposed to A star, this gave a solution which wasn't optimal and didn't guarantee the least cost.

## A Star:

**Python Code :**

```
from queue import PriorityQueue

h =  [('a',55),('b', 42),('c', 34),('d', 25),('e', 20),('f', 17),('g' ,0),('h', 99999),('i', 80)]


graph = [ [(2, 22), (3, 32), (8, 35)],

[(8, 45), (4, 36), (5, 27), (3, 28)],

[(0, 22), (3, 31), (6, 47)],

[(0, 32), (2, 31), (1, 28), (6, 30)],

[(1, 36), (6, 26)],

[(1, 27)],

[(2, 47), (3, 30), (4, 26)],

[],

[(0, 35), (1, 45)] ]


Q = PriorityQueue()
```

```
def a_star():
    startSymb='a'
    intstartSymb=ord(startSymb)

    source=str(input('Enter starting symbol:: '))
    source=ord(source)-intstartSymb

    goal=str(input('Enter starting symbol: '))
    goal=ord(goal)-intstartSymb

    parent = [-1] * len(graph)
    g = [inf] * len(graph)
    g[source] = 0
    Q.put((g[source] + h[source][1], source))
    while (Q.empty() == False):
        tmp = Q.get()
        u = tmp[1]
        for adj in graph[u]:
            v = adj[0]
            if (g[u] + adj[1] < g[v]):
                g[v] = g[u] + adj[1]
                Q.put((g[u] + adj[1] + h[u][1], v))
                parent[v] = u
    tmp = goal
    sum = 0
    print("Possible path is: ")
    while (parent[tmp] != -1):
        u = parent[tmp]
        v = tmp
        print(chr(ord('a') + tmp))
        for adj in graph[u]:
            if (adj[0] == v):
                sum += adj[1]
                break
        tmp = parent[tmp]
    print(chr(ord('a') + tmp))
    print("Total path cost:: ", sum)

a_star()
```

## Explanation:

The working procedure is given below in points:

1. Firstly, we used two tuples **called "graph" and "h"** to store the adjacent vertices of each of our nodes and the heuristics value respectively in a sequential way. So the node called a is the first vertex, the node called b is the 2nd vertex and so on.

2. A Priority Queue (PQ), which contains nodes in ascending order of h-values, is maintained which was called **"Q"** in our case. For this we imported Priority Queue module from Queue.
3. We also kept two arrays to denote what the parent of a particular node is, that is where it came from. And also to denote whether a node has been visited or not. For this the two arrays were **"parent "** and **"visited"** respectively.
4. Then we took inputs of our source and goal node using variables **"source"** and **"goal".**

5. Here we kept an array g which is used to store the actual path cost from initial node to node n.

**Evaluation function for this problem**:

f(n) = g(n) + h(n), where

g(n) = an actual path cost from initial node to node n,

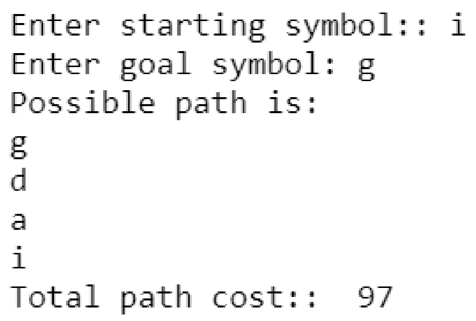
h(n) = estimated cost of the cheapest path from n to the goal.

6. Then we pushed the value of the  sum of the source node's heuristic value and the actual path cost from initial node to goal node. So basically we summed h(source) and g(source) and pushed them into the priority queue along with the name of the source itself and marked its visited value as true.

7. Then we analyzed the contents of the priority queue as long as its not empty. We iterated through the adjacent nodes of the initial staring node and then we pushed the adjacent nodes and their values of h(n)+g(n) into the Queue.

8. Then we continued our iteration by popping off the node that has the smallest value of h(n)+g(n) and also analyzed its adjacent nodes in the same way. We also marked each node as visited once we visited them and also updated the parent values accordingly.
**9.** Then we printed out our all possible paths by printing out the contents of the "parent" array and we also printed out the sum by adding up the cost required for the traversal from start to finish nodes.
**10.** This method gave us the solution that was the most optimal and it also managed to guarantee the least cost in the long run.

## Python Output:



```
Enter starting symbol:: i
Enter goal symbol: g
Possible path is:
g
d
a
i
Total path cost::  97
```

- Here in this case we specified out start and goal symbol
- Then we executed our function needed to find out the most optimal path from our start to goal state which was i->a->d->g
- Here out of all paths, the cost for this specific path was the last and it was calculated as 97.
- As opposed to GBFS, this gave a much more optimal and less cost.