# Ahsanullah University of Science and Technology

Department of Computer Science and Engineering


Class Assignment: 3

# CSE 4108

# Artificial Intelligence

Submitted By:

Anika Tanzim 16.02.04.072

Date of Submission: **25th August 2020**

**Q:** 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.

**Answer:** **Input:** anika cse 3.73
rahim cse 3.0
karim mpe 3.0

**Major steps of processing:**

1. Dictionary is maintained after reading an input file where 'name' is taken as key and 'dept', 'cgpa' are taken as values for each line

```
mydict={}
f1=open("test.py", "r")
for l in f1:
  name, dept, cgpa =l.split("\t")
  print(name, dept, float(cgpa), end="\n")
  mydict[name]=[dept,cgpa]
f1.close
```

2. After taking user input, 'cgpa' is updated into the dictionary according to the key (name).

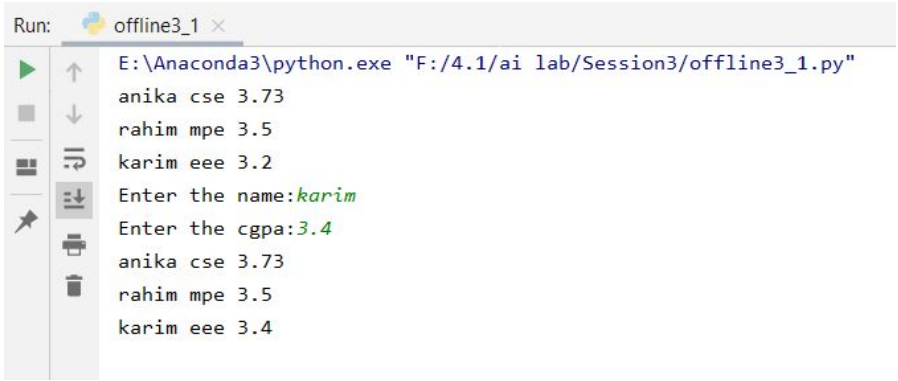
```
for (key,values) in mydict.items():
  first = values[0]
  second = values[1]
if(key==userName):
  second = userCgpa
  mydict[userName]=[first,userCgpa]
```

3. Then the whole file is re-written

```
f2=open("test.py", "w")

for (key,values) in mydict.items():
  name=str(key)
  dept=str(values[0])
  cgpa=str(values[1])
std=name+"\t"+dept+"\t"+cgpa
print(std, end="\n", file=f2)
print(name, dept, float(cgpa), end="\n")
```

**Output:**



Q: Implement in generic ways (as multi-modular and interactive systems) the Greedy Best-First and A* search algorithms in Prolog or in Python.

**Answer:** **Greedy Best-First Algorithm in Python:**

**Input:** For input, two dictionaries are maintained for the graph and the heuristic values.

```
graph= {  'i':[('a', 35),('b', 45)],
          'a':[('i', 35),('c', 22),('d',32)],  'b':[('d', 28),('e', 36),('i', 45),('f', 27)],
          'c': [('a', 22),('d', 31),('g', 47)], 'd':[('a', 32),('b', 28),('c', 31),('g', 30)],
          'e':[('b', 36),('g', 26)], 'f':[('b', 27)], 'g':[('c', 47),('d', 30),('e', 26)]  }
hfn = {'i' : 80, 'a' : 55, 'b' : 42, 'c' : 34, 'd' : 25, 'e' : 20, 'f' : 17, 'g' : 0}
```

**Major steps of processing:**

1. A priority queue (pq) is maintained where nodes are in ascending order in h values. For that a module **(gfsNode)** is maintained where a constructor (__int__)and a magic method (__lt__)is defined.

```
def __init__(self, name, priority):
 self.name = name
 self.priority = priority
 return
```

```
def __lt__(self, other):
  return self.priority < other.priority
```

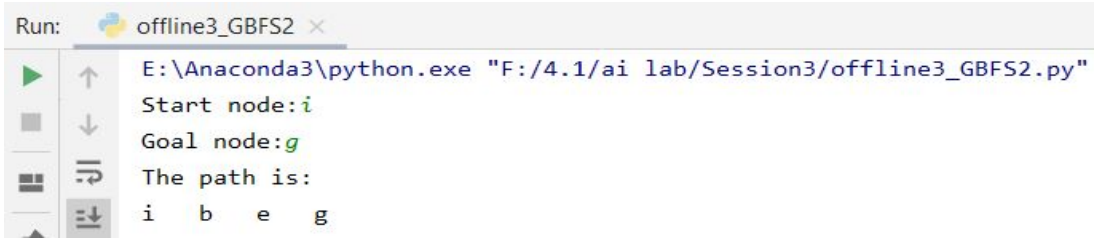
2. The process is based on the evaluation function f(n) =h(n)
3. A list of path (final_path) is maintained to keep track of the solution.
4. After selecting the starting node and the goal node, the process begins by putting the starting node in pq.
5. A dictionary (visited) is also maintained to mark the nodes as visited. At first, all the nodes are marked as not visited(false).
6. The process terminates when the goal node is placed in the PQ, and consequently, selected for a visit. The process continues while the pq is not empty otherwise.

```
while not (pq.empty()):
  v = pq.get()
  node = v.name
  visited[node] = True
  if (node == goal):
     final_path.append(node)
     break
  else:
     for (k, v) in graph.items():
        if k == node:
           j = 0
```

```
     while (j < len(v)):
        newV = v[j][0]
        if (visited[newV] == False):
           newVFound = True
           for (a, b) in hfn.items():
              if (a == newV):
                 pq.put(n.Nodes(a, b))
                 break
        else:
           newVFound = False
        j=j+1
if (newVFound == True):
  final_path.append(node)
```

7. When the node is found as a key in the dictionary, its values are next to be found as the neighbors. If the neighbor (newV) is not visited, the pq as well as the final_path will be updated.
8. The first node in the pq is selected always and the pq and final_path are updated. The nodes are marked as visited. And the node itself is deleted from the pq.

**Output:**




**A* search Algorithm in Python:**

**Input:** For input, two dictionaries are maintained for the graph and the heuristic values.

graph= { **'i'**:[(**'a'**, 35),(**'b'**, 45)],
**'a'**:[(**'i'**, 35),(**'c'**, 22),(**'d'**,32)], **'b'**:[(**'d'**, 28),(**'e'**, 36),(**'i'**, 45),(**'f'**, 27)],
**'c'**: [(**'a'**, 22),(**'d'**, 31),(**'g'**, 47)], **'d'**:[(**'a'**, 32),(**'b'**, 28),(**'c'**, 31),(**'g'**, 30)],
**'e'**:[(**'b'**, 36),(**'g'**, 26)], **'f'**:[(**'b'**, 27)], **'g'**:[(**'c'**, 47),(**'d'**, 30),(**'e'**, 26)] }

hfn = {**'i'** : 80, **'a'** : 55, **'b'** : 42, **'c'** : 34, **'d'** : 25, **'e'** : 20, **'f'** : 17, **'g'** : 0}

**Major steps of processing:**

1. A priority queue (pq) is maintained where nodes are in ascending order in h values. For that a module **(astarNode)** is maintained where a constructor (__int__)and a magic method (__lt__)is defined. Here, parent is also added to pq unlike greedy best first search.

```
def __init__(self, name, parent,priority):
  self.name = name
  self.parent = parent
  self.priority = priority
  return
```

```
def __lt__(self, other):
  return self.priority < other.priority
```

2. The process is based on the evaluation function f(n) =h(n) + g(n) ; where g(n) = an actual path cost from initial node to node n and h(n) = estimated cost of the cheapest path from n to the goal.
3. A list of path (final_path) is maintained to keep track of the solution.
4. A list (track ) and a dictionary (parent) is maintained to track the nodes and parent.
5. After selecting the starting node and the goal node, the process begins by putting the starting node in pq.
6. A dictionary (visited) is also maintained to mark the nodes as visited. At first, all the nodes are marked as not visited(false).

7. The process terminates when the goal node is placed in the PQ, and consequently, selected for a visit. The process continues while the pq is not empty otherwise.

```
while not (pq.empty()):
  v = pq.get()
  node = v.name
  visited[node] = True
  if (node == goal):
     final_path.append(node)
     break
  else:
     for (k, v) in graph.items():
        if k == node:
           j = 0
           while (j < len(v)):
              newV = v[j][0]
              parent[newV]=k
              cost = v[j][1]
```

```
if (visited[newV] == False):
            newVFound = True
            track.append((node, newV)) #parent
            for (a, b) in hfn.items():
               if (a == newV):
                  pq.put(n.Nodes(newV,  parent[newV], b
+ cost))
                  break
         else:
            newVFound = False
         j=j+1
if(newVFound == True):
   final_path.append(node)
```

8. When the node is found as a key in the dictionary, its values are next to be found as the neighbors. If the neighbor (newV) is not visited, the pq as well as the final_path will be updated.
9. The pq will be updated with a total of heuristic value and the path cost. Because of the __lt__ method, pq will be in ascending order. And parent will also be tracked.
10. The first node in the pq is selected always and the pq and final_path are updated. The nodes are marked as visited. And the node itself is deleted from the pq. The optimal solution is considered here.

Output: