AIM:

Identification of the problem, defining PEAS Description and problem formulation.

S/W Used: Case Study

Video Presentation:
Meeting with Geetika Kaushik[CSE - 2020]-20221106_233657-Meeting Recording.mp4

GitHub Link: Geetika-2001/Artificial-Intelligence-Project (github.com)

Specifying Task Environment:

Due to the Covid 19 Pandemic and it's associated risks, there is a shortage of vaccines in many cities. There is a provided list of cities and their distances. The user has to find the optimal route for the Medical Van to return to the starting city after visiting every city on the list.

Problem Formulation:

Let's call the office(vaccine formation take place here) (A)and the 3 cities (B) (C) (D) respectively. We initialize the problem state by {A} means the Medical Van departed from it's office. When it visited city-B, the problem state is updated to {A, B}, where the order of elements in { } is considered. When the Van visited all the cities, {A, B, C, D} in this case, the departed point A is automatically added to the state which means {A, B, C, D, A}.

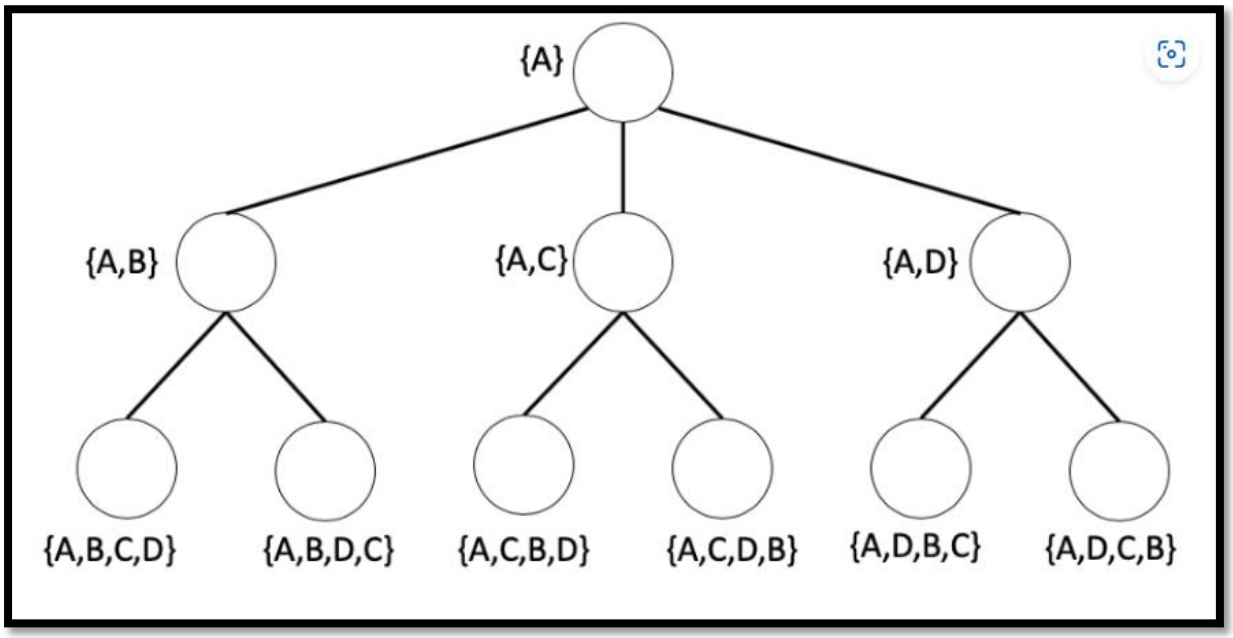
Therefore,

INITIAL STATE of this problem is {A} and

FINAL STATE(goal) is {A, X1, X2, X3, A} where travelled distance is minimized.

PATH COST:

Taking each state as a node of a tree structure, we can represent this TSP as the following tree search problem.



PEAS:

PEAS stands for **Performance, Environment, Actuators, Sensors**. They help define the task environment for an intelligent agent. Hence, PEAS is an important representation system for defining an Artificial Intelligence model

Performance- Fastest Delivery, Right temperature

Environment- City Roads

Actuator-Driver , Medical Centre Representatives

Sensors- Temperature, Effectiveness

2) Depth-first search

The depth-first search algorithm starts at the root node and explores as deep as possible along each branch before taking backtracking. In our TSP, when a state node with all city labels is visited, its total distance is memorized. This information will later be used to define the shortest path.

Let VISIT be a ***stack*** to save visited nodes, PATH be a set to save distances from the root node to the goal. The depth-first algorithm can be written as

Depth-first algorithm for TSP

1. Push root node to VISIT
2. If VISIT is empty, then go to 7.
3. Pop a node from VISIT, calls N.
4. If N is the goal node, then append distance from root to N to PATH.
5. Push all child nodes of N to VISIT.
6. Go to 2.
7. Get the minimum value from PATH, output the result. Done.

3) Solution using A* Algorithm:

A-algorithm (A*-algorithm)

In the A algorithm search, we use the information of distance from the present visit node to the goal as a heuristic function, h(X). Let g(X) be the distance from the root node to node-X. In this case, we consider the priority of node visit order by f(X)=g(X)+h(X).

In real-world problems, it is impossible to obtain the exact value of h(X). In that case, an estimation value of h(X), h'(X), is used. However, setting h'(X) takes risks in falling into a local optimal answer. To prevent this problem, choosing h'(X) which h'(X)≤h(X) for all X is recommended. In this case, it is known as A*-algorithm and it can be shown that the obtained answer is the global optimal answer.

In our experiment described in the following part, we are setting h'(X) as the sum of the minimum distance of all possible routes from each city which is not presented in the current visit node label, and the present city. For example, if the present node is {A, D}, then city-B and C is not present in the label. Therefore, h'(D)=min(all possible route distance from C)+min(all possible route distance from B)+min(all possible route distance from D).

Let VISIT be a ***list*** to save visited nodes. The A-algorithm can be written as

A-algorithm for TSP

1. Append root node to VISIT
2. If VISIT is empty, then search fails. Exit.
3. Get a node from the head of VISIT, calls N.
4. If N is the goal node, then search succeeds, output the result. Exit.
5. Push all child nodes of N to VISIT. Sort all elements X of VISIT in ascending order of f(X)=g(X)+h(X)
6. Go to 2.

Comparison of All possible algorithms:

Number of cities	Depth-level	Breadth-first	Depth-first	Best-first	A* Algorithm
3	1	5	5	2	2
4	2	16	16	5	5
5	3	65	65	18	7
6	4	326	326	87	15
7	5	1957	1957	518	15
8	6	13700	13700	3621	66

Implementation:

```
# Evaluate Function for A Algorithm
def get_bound(label):
f = 0
for i in range(0,len(label)-1):
f = f+graph[label[i]-1][label[i+1]-1]
remain = city.difference(set(label))
remain = list(remain)remain.append(label[-1])
for i in remain:
f = f+min_bound[i-1]
if len(remain)==2:
f=0label.append(remain[0])
label.append(1)
```

```
for i in range(0,len(label)-1):
f = f+graph[label[i]-1][label[i+1]-1]
return f

# Search for GOAL (for A algorithm/Best-first Algorithm)
tree = node(number=1)tree.label.append(1)tree.cost=0
for i in range(len(Gs)):
print("----------i=%d----------"%i)
graph = Gs[i]
NN = len(graph)
for idx in range(NN):
graph[idx][idx] = float('inf')
city = range(1,len(graph)+1)
city = set(city)
min_bound = [min(graph[i])
for i in range(len(graph))]
tree = node(number=1)
tree.label.append(1)
tree.cost=0
visit=[]
visit.append(tree)
count = 0
fcnt = 0
ans = 0
while len(visit)>0:
N = visit[0]
if len(N.label)==(len(city)-2):
fcnt = fcnt+1
del(visit[0])
count = count+1
child_list = set(city).difference(set(N.label))
if len(child_list)==0:
ans = 1
break
for c in child_list:
N.add_child(number=c)
tmp = N.child
```

```
for i in tmp:
visit.append(i)
visit = sorted(visit,key= lambda x:x.cost)
if ans==1:
print("RESULT:",N.label,N.cost)
b = solve(count,NN-2)
print("d=%d ,N= %d ,  b*=%f"%(NN-2,count,b))
print("ROUTEs: %d"%(fcnt))
resultsA.append((NN-2,count,b))
else:print("FAILED")
```