|  |  |
| --- | --- |
| **Ex. No: 1** | **British Museum Search** |
| **14.07.2023** |

# Aim:

To implement British Museum Search algorithm and visualize the resulting path

# Algorithm:

1. Start at the current node. This is the node that the algorithm is currently exploring.
2. Randomly select a neighbor of the current node. This ensures that the algorithm explores the search space in a random order, which can help it to avoid getting stuck in local optima.
3. Move to the neighbor node. This updates the current node to the neighbor node that was selected in step 2.
4. Repeat steps 2 and 3 until a solution is found. This loop continues until a solution is found, or until the algorithm has explored all of the nodes in the search space.
5. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

d\_h = {

'S': {'A': 7.38, 'B': 6},

'A': {'S': np.inf, 'B': 6, 'D': 5},

'B': {'S': np.inf, 'A': 7.38, 'C': 7.58}, 'C': {'B': 6, 'E': np.inf},

'E': {'C': 7.58},

'D': {'A': 7.38, 'G': 0},

'G': {'D': 5}

}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos.

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red"

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def bms(d,start,end): q = [[0,start]] res = []

f = 0

while( len(q) != 0): temp = []

cp = q.pop(0) cn = cp[-1]

for i in d[cn].keys(): if i in cp:

continue el = cp + [i] temp.append(el)

temp.sort(key = lambda x:x[0]) q.append(el)

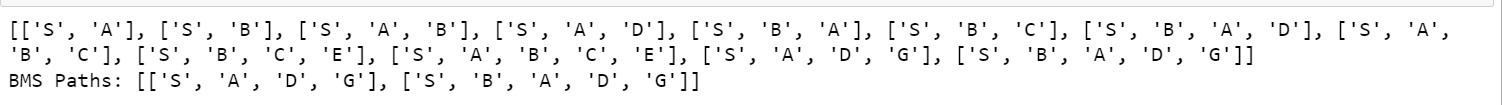
for i in temp: res.append(i)

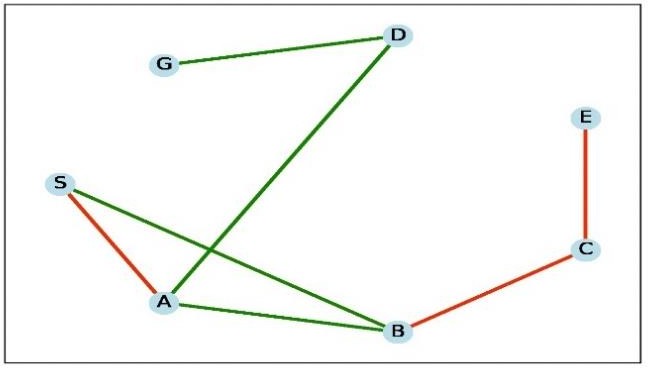
return res

FR = bms(d, 'S', 'G')

g=form\_graph(d,FR) Viz(g)

# Output:





**Performance Evaluation:**

|  |  |
| --- | --- |
| Total Enqueings | 10 |
| Time  Complexity | O(|V| \* |E| \* log(|V| \*  |E|)) |
| Space  Complexity | O(|V| \* |E|) |

# Result:

Therefore, we've successfully implemented the visualization for the BMS.

|  |  |
| --- | --- |
| **Ex. No: 2** | **Depth first search (DFS)** |
| **21.07.2023** |

# Aim:

To implement Depth first search algorithm and the resulting path is visualised.

# Algorithm:

1. Implement the DFS algorithm. This can be done either iteratively or recursively.
2. Create a data structure to store the resulting path. This could be a list, stack, or queue.
3. Modify the DFS algorithm to update the data structure with the resulting path. This can be done by adding the current node to the data structure each time the algorithm visits a new node.
4. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d = {

'S': {'A': 3, 'B': 5},

'A': {'S': 3, 'B': 4, 'D': 3},

'B': {'S': 5, 'A': 4, 'C': 4},

'C': {'B': 4, 'E': 6},

'E': {'C': 6},

'D': {'A': 3, 'G': 5},

'G': {'D': 5}

}

def form\_graph(d, FR): # FR is a function pertaining to defined search algos. global l

g = nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i, j)

l = []

# FR=BMS(d,'S','G',2)

# FR=dfs(d)

# FR=branch\_and\_bound(d,11)

# FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

# print(i) st = i[1:] tmp = []

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l += [tmp]

l

print((l)) return g

def ret\_edge(edge):

c = 0

# print(edge)

for i in g.edges: # print(i)

if (edge == list(i) or edge[::-1] == list(i)): break

c += 1

# edge\_col[c]="red" return c

def animate(i): edge\_col = []

for j in range(len(g.edges)): edge\_col += ["green"]

tmp = edge\_col.copy()

# for i in range(len(l)):

for j in (l[i]): tmp[ret\_edge(j)] = "blue"

print(tmp)

# node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g, pos)

node = nx.draw\_networkx\_nodes(g, pos, nodelist=list( g.nodes), node\_size=250, node\_color='lightblue')

edges = nx.draw\_networkx\_edges(

g, pos, edgelist=list(g.edges), edge\_color=tmp) return node, edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g = nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i, j)

print(g.edges)

fig = plt.gcf()

# nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len( l), interval=1000, repeat=True, blit=False)

# ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def dfs(d, start, end): queue = [[0, start]] path = []

while (len(queue) != 0): cp = queue.pop(0) cn = cp[-1]

if cn == end: break

else:

for i in d[cn].keys(): if i not in cp:

queue.append(cp+list(i)) queue = sorted(queue)

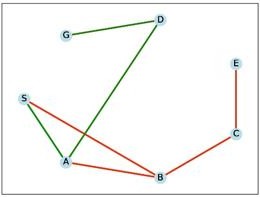
path.append(queue[0])

print(path) return path

FR = dfs(d, 'S', 'G')

g = form\_graph(d, FR) Viz(g)

# Output:



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 12 |
| Time  Complexity | O(V + E) |
| Space  Complexity | O(V) |

# Justification:

The time complexity of Depth-First Search (DFS) is O(V + E), where V is the number of vertices (nodes) in the graph, and E is the number of edges. In the worst case, DFS may visit all vertices and edges. This time complexity is generally efficient for small to moderately sized graphs. However, for graphs with a large number of vertices and edges, the time complexity can become a limiting factor, leading to a slower search process.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization for depth first search.

|  |  |
| --- | --- |
| **Ex. No: 3** | **Breadth First Search(BFS)** |
| **21.07.2023** |

# Aim:

To implement Breadth First Search algorithm and visualize the resulting path

# Algorithm:

1. Start at the current node. This is the node that the algorithm is currently exploring.
2. Initialize a queue.
3. Add the start node to the queue. While the queue is not empty:

Remove the first node from the queue.

1. If the node has not been visited:
2. Mark the node as visited.
3. Add all of the node's neighbors to the queue.
4. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos. global l

g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=branch\_and\_bound(d,11) #FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for j in FR:

for i in j: #print(i) print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print("l=",(l)) return g

def ret\_edge(edge):

c=0 #print(edge)

for i in g.edges:

#print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" print(c)

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

print("node to be changed",j) tmp[ret\_edge(j)]="red"

print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def bfs(d,start,end): queue=[[0,start]] fr=[]

goal=end gn=[] f=0

#print(queue) n\_g=0

f=n\_g while(len(queue)!=0):

print(queue) tmp=queue.copy() sl=[]

for j in range(len(tmp)): cp=queue.pop(0) n=cp[-1]

for i in d[n].keys(): # SAme loops as oracle search #print(i)

if i in cp:

continue el=cp+[i]

el[0]+=d[n][i] sl.append(el) sl=sorted(sl)

#sl.sort(key=lambda x: ord(x[-1])) if goal==i:

f=1 gn=el

queue=sl if f==1:

fr.append(queue.copy()) break

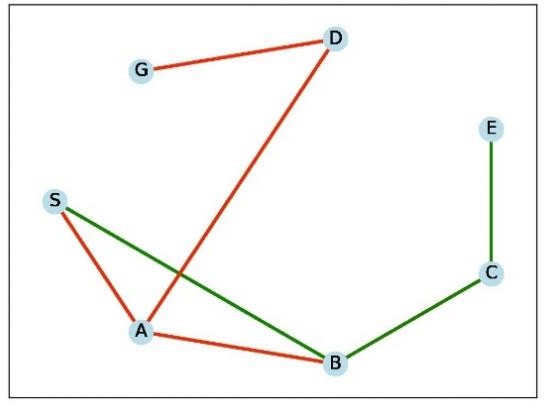
queue=queue[:] fr.append(queue.copy())

print(fr) print(gn) return fr

fr=bfs(d,'S','G') g=form\_graph(d,fr)

Viz(g)

# Output:



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 29 |
| Time  Complexity | O(V + E) |
| Space  Complexity | O(V) |

# Justification:

The choice to use the BFS (Breadth-First Search) algorithm is justified when the goal is to systematically explore and find the shortest path in unweighted graphs, making it well-suited for applications where finding the most efficient path or traversing a broad range of possibilities is essential.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the path for BFS.

|  |  |
| --- | --- |
| **Ex. No: 4** | **Beam search (BS)** |
| **28.07.2023** |

# Aim:

To implement Beam Search algorithm and visualize the resulting path

# Algorithm:

* 1. Initialize the beam with the start node and set the current node to the start node.
  2. Expand the current node and generate all its successors
  3. Score each successor using a heuristic function. The heuristic function estimates the likelihood of reaching the goal node from the current successor.
  4. Prune all but the top k successors, where k is the beam width.
  5. Select the highest-scored successor as the new current node.
  6. Repeat steps 2-5 until a goal node is reached or the search tree is fully explored.
  7. If a goal node is reached, backtrack through the search tree to construct the optimal solution.
  8. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos. global l

g=nx.Graph()

g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=branch\_and\_bound(d,11) #FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for j in FR:

for i in j: #print(i) print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print("l=",(l)) return g

def ret\_edge(edge):

c=0 #print(edge)

for i in g.edges:

#print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" print(c)

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

print("node to be changed",j) tmp[ret\_edge(j)]="red"

print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def beamsearch(d,start,end,bw): queue=[[0,start]]

fr=[] goal=end Beam=bw gn=[] f=0

#print(queue) n\_g=0

f=n\_g while(len(queue)!=0):

print(queue) tmp=queue.copy() sl=[]

for j in range(len(tmp)): cp=queue.pop(0) n=cp[-1]

for i in d[n].keys(): # SAme loops as oracle search #print(i)

if i in cp:

continue

el=cp+[i] el[0]+=d[n][i] sl.append(el) #queue.append(el)

sl.sort(key=lambda x: x[0]) if goal==i:

f=1 gn=el

queue=sl if f==1:

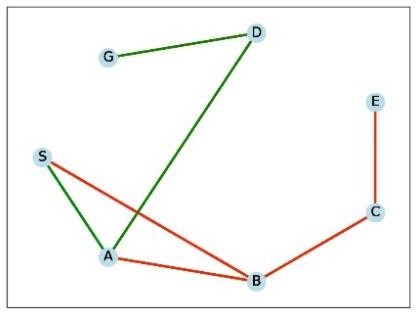
fr.append(queue.copy()) break

queue=queue[:Beam+2] fr.append(queue.copy())

print(fr) print(gn) return fr

fr=beamsearch(d,'S','G',2) g=form\_graph(d,fr)

Viz(g)



**Output:**

# Performance Comparison:

|  |  |
| --- | --- |
| Total Enqueings | 9 |
| Time  Complexity | O((B \* V) + (B \*  E)) |
| Space  Complexity | O(B \* V) |

**Justification:**

Beam Search offers a balance between efficiency and thoroughness in exploration, making it a practical choice for a variety of optimization and search problems. It's particularly valuable when resource constraints or the need for faster convergence make it challenging to use exhaustive search algorithms like BFS or DFS. The adjustable beam width and space efficiency of BS contribute to its applicability in a wide range of real-world problem-solving scenarios.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of resulting path for Beam Search.

|  |  |
| --- | --- |
| **Ex. No: 5** | **Hill Climbing** |
| **28.07.2023** |

# Aim:

To implement hill climbing algorithm and visualize the resulting path

# Algorithm:

1. Initialize the current state: Start with an arbitrary node in the graph as the current state.
2. Evaluate the current state: Calculate the cost or value of the current state. This could represent the distance from the current node to the goal node, the number of edges in the path, or any other relevant metric.
3. Generate neighbor states: Identify all possible successor nodes or neighbors of the current state. These represent the possible next steps in the path.
4. Select the best neighbor: Choose the neighbor state that has the lowest cost or highest value among the generated neighbors. This represents the most promising direction to move in.
5. Update the current state: Make the selected neighbor the new current state.
6. Repeat steps 2-5: Continue evaluating, generating neighbors, selecting the best neighbor, and updating the current state until a termination condition is met.
7. Termination condition: The algorithm terminates when one of the following conditions is met:
8. A goal node is reached.
9. No better neighbor can be found, indicating a local optimum has been reached.
10. A maximum number of iterations has been exceeded.
11. Construct the path: The final path is constructed by tracing back from the final current state to the initial state, following the parent-child relationships established during the search.
12. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d = {

'S': {'A': 3, 'B': 5},

'A': {'S': 3, 'B': 4, 'D': 3},

'B': {'S': 5, 'A': 4, 'C': 4},

'C': {'B': 4, 'E': 6},

'E': {'C': 6},

'D': {'A': 3, 'G': 5},

'G': {'D': 5}

}

def form\_graph(d, FR): # FR is a function pertaining to defined search algos.

global l

g = nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i, j)

l = []

# FR=BMS(d,'S','G',2)

# FR=dfs(d)

# FR=branch\_and\_bound(d,11)

# FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

# print(i) st = i[1:] tmp = []

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l += [tmp]

l

print((l)) return g

def ret\_edge(edge):

c = 0

# print(edge)

for i in g.edges: # print(i)

if (edge == list(i) or edge[::-1] == list(i)): break

c += 1

# edge\_col[c]="red" return c

def animate(i): edge\_col = []

for j in range(len(g.edges)): edge\_col += ["green"]

tmp = edge\_col.copy()

# for i in range(len(l)):

for j in (l[i]): tmp[ret\_edge(j)] = "blue"

print(tmp)

# node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g, pos)

node = nx.draw\_networkx\_nodes(g, pos, nodelist=list( g.nodes), node\_size=250, node\_color='lightblue')

edges = nx.draw\_networkx\_edges(

g, pos, edgelist=list(g.edges), edge\_color=tmp) return node, edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g = nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i, j)

print(g.edges)

fig = plt.gcf()

# nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len( l), interval=1000, repeat=True, blit=False)

# ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def hill\_climb(d, start, end): queue = [[0, start]]

path = []

while (len(queue) != 0): cp = queue.pop(0) sl = []

# print(cp) cn = cp[-1] if cn == end:

break else:

for i in d[cn].keys(): if i in cp:

continue

el = cp+list(i) el[0] = d[cn][i] sl.append(el)

# Perform Similar to stack by LIFO by reversing the sorted queue sl.sort(key=lambda x: x[0], reverse=True)

# print(sl)#same thing as DFS but with heuristics

for k in sl:

print(k) queue.insert(0, k)

path.append(queue[0]) print(path)

return path

d\_h = {

'S': {'A': 7.38, 'B': 6},

'A': {'S': np.inf, 'B': 6, 'D': 5},

'B': {'S': np.inf, 'A': 7.38, 'C': 7.58}, 'C': {'B': 6, 'E': np.inf},

'E': {'C': 7.58},

'D': {'A': 7.38, 'G': 0},

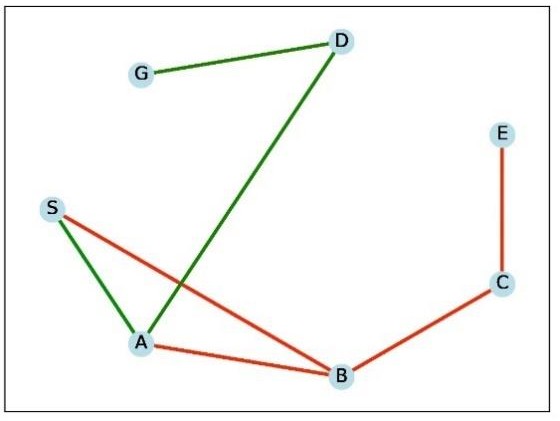
'G': {'D': 5}

}

FR = hill\_climb(d\_h, 'S', 'G') g = form\_graph(d, FR)

Viz(g)

# Output:



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | Exponential |
| Space  Complexity | O(B \* D) |

# Justification:

The use of the Hill Climbing (HC) algorithm is justified in scenarios where the primary objective is local optimization within a solution space. HC is chosen for its computational efficiency and simplicity, making it ideal for quickly converging to locally optimal solutions and refining them iteratively. Its low memory overhead and applicability to real-world problems, particularly in fields like machine learning, natural language processing, and constraint satisfaction, make it a practical choice when global optimality is not required, and local improvements are the focus.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the path for Hill Climbing.

|  |  |
| --- | --- |
| **Ex. No: 6** | **Oracle** |
| **18.08.2023** |

**Aim:** To implement oracle algorithm and visualise a graph for the resulting path

# Algorithm:

1. Start at the current node. This is the node that the algorithm is currently exploring.
2. select a neighbor where the cost is less of the current node. This ensures that the algorithm explores the search space , which can help it to avoid getting stuck in local optima.
3. Move to the neighbor node. This updates the current node to the neighbor node that was selected in step 2.
4. Repeat steps 2 and 3 until a solution is found. This loop continues until a solution is found, or until the algorithm has explored all of the nodes in the search space.
5. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos. global l

g=nx.Graph()

g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)): for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def oracle(d,start,end,m): q = [[0,start]]

res = [] f = 0

while( len(q) != 0): temp = []

cp = q.pop(0) cn = cp[-1]

for i in d[cn].keys(): if i in cp:

continue el = cp + [i]

el[0] = cp[0] + d[cn][i] temp.append(el)

temp.sort(key = lambda x:x[0]) q.append(el)

# q.sort(key = lambda x: x[0]) if (end == i):

f = f + 1

# print("temp : ",temp) # print("Queue : ",q)

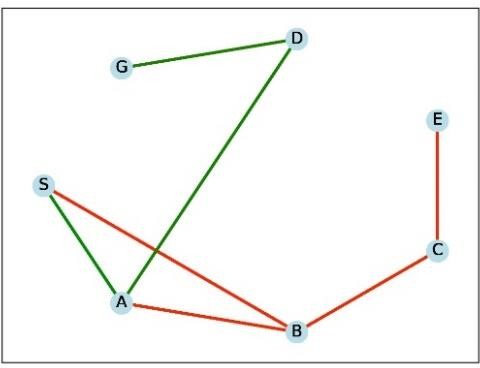
for i in temp: res.append(i)

if(m == f):

break return res

FR=oracle(d,'S','G',11) g=form\_graph(d,FR) Viz(g)

# Output:



**Performance Comparison:**

# Performance comparison:

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

**Justification:**

The Oracle Search (ORA) algorithm, while having a high time and space complexity in the worst case, is a suitable choice when exploring the entire solution space is necessary. It systematically explores paths, allowing it to find a solution with certainty when it exists, which is critical in applications like exhaustive search or verification tasks where completeness and accuracy are paramount. The algorithm's computational demands may be justified in scenarios where robustness and assurance in finding the correct solution outweigh concerns about time and memory resources.The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event

handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the path for Oracle.

|  |  |
| --- | --- |
| **Ex. No: 7** | **Oracle with cost and heuristics** |
| **18.08.2023** |

# Aim:

To implement oracle with cost and heuristics and visualize the resulting path

# Algorithm:

1. Start at the current node. This is the node that the algorithm is currently exploring.
2. select a neighbor where the cost and hueristics is less of the current node. This ensures that the algorithm explores the search space , which can help it to avoid getting stuck in local optima.
3. Move to the neighbor node. This updates the current node to the neighbor node that was selected in step 2.
4. Repeat steps 2 and 3 until a solution is found. This loop continues until a solution is found, or until the algorithm has explored all of the nodes in the search space.
5. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

d\_h = {

'S': {'A': 7.38, 'B': 6},

'A': {'S': np.inf, 'B': 6, 'D': 5},

'B': {'S': np.inf, 'A': 7.38, 'C': 7.58}, 'C': {'B': 6, 'E': np.inf},

'E': {'C': 7.58},

'D': {'A': 7.38, 'G': 0},

'G': {'D': 5}

}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos.

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)):

break c+=1

#edge\_col[c]="red" return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def oracle\_cost\_heur(d,start,end,d\_heuristic,max\_no\_goalcutoff): queue=[[0,start]]

res=[]

g='G'

f=0 n\_g=max\_no\_goalcutoff while (len(queue)!=0):

for i in d[currnode].keys():

if i in curr:continue # prevent accessing ancesstors to child tmp1=curr+list(i) tmp1[0]=curr[0]+d[currnode][i]+d\_heuristic[currnode][i] tmp.append(tmp1)

queue.append(tmp1)

queue.sort(key=lambda k1:k1[0]) # Should have used a priority queue but sort works !

if g==i:f+=1

for i in tmp:res.append(i) if(f==n\_g):

break return res

FR = oracle\_cost\_heur(d, 'S', 'G', d\_h, 2) g=form\_graph(d,FR)

Viz(g)

# This is the list of paths with index 0

# This is the last node in the traversed

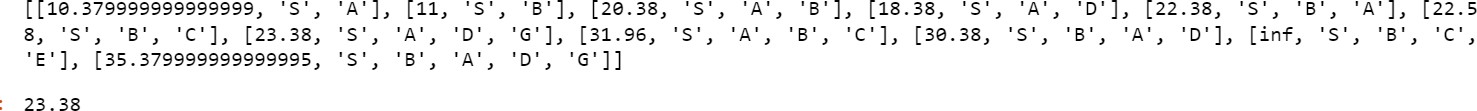
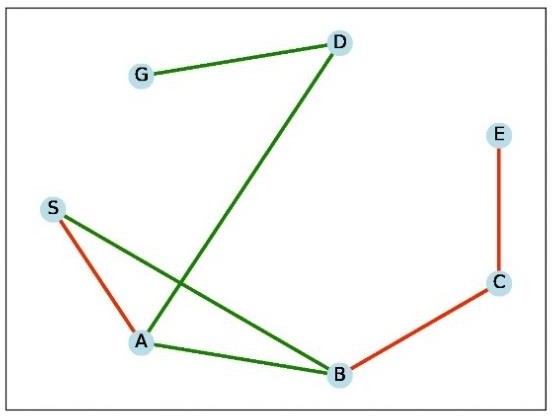
tmp=[]

curr=queue.pop(0) storing cumulative cost

currnode=curr[-1]

path

# Output:



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

The integration of heuristics with the Oracle Search (ORA-H) is a judicious choice when solving complex problems, striking a balance between computational efficiency and solution optimality. By incorporating problem-specific knowledge, heuristics guide the search toward promising solutions, reducing the time needed to find answers while retaining completeness. ORA-H's adaptability, robustness, and resource optimization make it a pragmatic approach for various applications, from pathfinding to optimization problems, where quickly achieving reliable results within limited resources is a priority.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of graph for oracle with cost and heuristics

|  |  |
| --- | --- |
| **Ex. No: 8** | **Branch and bound (BB)** |
| **25.08.2023** |

# Aim:

To implement branch and bound algorithm and visualize the resulting path

# Algorithm:

1. Start by creating a data structure to represent the graph. Also, initialize a variable to track the current solution, which will initially be empty.
2. Create a priority queue to store the nodes to be explored. The priority queue should be ordered by a heuristic function that estimates the distance to the goal node. A common heuristic function is the Manhattan distance, which calculates the sum of the absolute differences between the current node's coordinates and the goal node's coordinates.
3. Add the starting node to the priority queue with a heuristic value of 0. This indicates that the starting node is considered to be the closest to the goal node.
4. While the priority queue is not empty
5. While the priority queue is not empty, repeat the following steps:
   1. Remove the node with the smallest heuristic value from the priority queue. This is considered the most promising node to explore.
   2. If the removed node is the goal node, then the current solution is the optimal path. Stop the algorithm and return the solution.
   3. Expand the removed node by adding its neighbors to the priority queue. Calculate the heuristic value for each neighbor and add it to the priority queue along with the current node and the edge between them.
   4. If a neighbor has already been visited, skip it. This prevents the algorithm from exploring loops.
   5. If a neighbor's heuristic value is worse than the current solution, skip it. This is because the current solution is already known to be a valid path, so exploring a neighbor with a worse heuristic value is unlikely to lead to a better solution.
6. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

}

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos.

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def branch\_and\_bound(d,start,end,oracle): queue=[[0,start]]

path=[] isgoal=0 gn=[]

while(len(queue)!=0): sl=[] cp=queue.pop(0) cn=cp[-1]

if cn==end:

gn=el break

for i in d[cn].keys(): #print(queue)

if i in cp:

continue

elif cp[0]+d[cn][i]<=oracle: el=cp+list(i) el[0]+=d[cn][i] sl.append(el) sl.sort(key=lambda x:x[0]) queue.append(el)

sl.sort(key=lambda x:x[0]) for i in sl:path.append(i) #print(sl) #if(isgoal==1):break

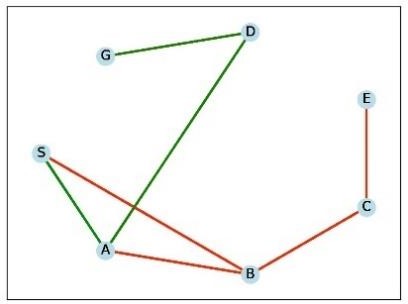
print(path) return path

FR=branch\_and\_bound(d,'S','G',11) g=form\_graph(d,FR)

Viz(g)

# Output:





**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 8 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

The Branch and Bound (BB) algorithm is a justified choice for solving a wide range of complex problems due to its ability to guarantee optimality by systematically exploring solution spaces while efficiently pruning unpromising branches. Its adaptability, versatility, and resource-aware approach make it well-suited for applications in optimization, scheduling, and other domains where finding the best or near-optimal solutions is of paramount importance, all while effectively balancing computational efficiency with solution quality.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the graph for branch and bound.

|  |  |
| --- | --- |
| **Ex. No: 9** | **Branch and bound with heuristics** |
| **22.09.2023** |

# Aim:

To implement branch and bound with estimated heuristics algorithm and visualize the resulting path

# Algorithm:

1. Initialize: Create a priority queue to store the nodes to be explored. Set the current node to the starting node and add it to the queue with an estimated cost of 0.
2. Explore: While the queue is not empty, remove the node with the lowest estimated cost from the queue. Expand the node by generating its successors. For each successor, calculate its estimated cost using the heuristic function. Add the successor to the queue with its estimated cost.
3. Check for goal: If the current node is the goal node, you have found the shortest path. Terminate the search and return the path.
4. Prune: If the estimated cost of the current node is greater than or equal to the current best solution, prune the current branch and discard the node. This is because the current branch cannot lead to a shorter path than the best solution found so far.
5. Repeat: Repeat steps 2 to 4 until the queue is empty or the goal node is found.
6. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

def form\_graph(d,FR): #FR is a function pertaining to defined search algos. global l

g=nx.Graph()

g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)): for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def branch\_and\_bound\_EL\_heur(d, start, end, oracle, d\_heuristic): queue = [[0, start]]

path = []

while len(queue) != 0: sl = []

cp = queue.pop(0) cn = cp[-1]

if cn == end: break

for i in d[cn].keys(): if i in cp:

continue

if cp[0] + d[cn][i] <= oracle : el = cp + [i]

el[0] += d[cn][i] + d\_heuristic[cn][i] sl.append(el)

sl.sort(key=lambda x: x[0]) queue.append(el)

sl.sort(key=lambda x: x[0]) for i in sl:

path.append(i)

return path

def oracle(d,d\_h,start,end,m): q = [[0,start]]

res = [] f = 0

while( len(q) != 0): temp = []

cp = q.pop(0) cn = cp[-1]

for i in d[cn].keys(): if i in cp:

continue el = cp + [i]

el[0] = cp[0] + d[cn][i] + d\_h[cn][i] temp.append(el)

temp.sort(key = lambda x:x[0]) q.append(el)

# q.sort(key = lambda x: x[0]) if (end == i):

f = f + 1

print("temp : ",temp)

print("Queue : ",q)

for i in temp: res.append(i)

if(m == f):

break path = []

for i in res:

if end in i:

path.append(i) return path[0][0]

d={

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6},

'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

}

d\_heuristic = {

'S': {'A': 7.38, 'B': 6},

'A': {'S': np.inf, 'B': 6, 'D': 5},

'B': {'S': np.inf, 'A': 7.38, 'C': 7.58}, 'C': {'B': 6, 'E': np.inf},

'E': {'C': 7.58},

'D': {'A': 7.38, 'G': 0},

'G': {'D': 5}

}

orac = oracle(d,d\_heuristic ,"S","G",1)

FR=branch\_and\_bound\_EL\_heur(d, "S", "G", orac, d\_heuristic) g=form\_graph(d,FR)

Viz(g)

# A white background with black text Description automatically generatedA diagram of a triangle with green lines and red dots Description automatically generatedOutput:

**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 8 |
| Time  Complexity | O(B \* D) |

|  |  |
| --- | --- |
| Space  Complexity | O(B \* D) |

# Justification:

The Branch and Bound with Heuristics (BB-H) algorithm is justified for its effective utilization of time and space resources in complex problem-solving. By combining branch pruning with heuristic guidance, BB-H efficiently narrows down the search space, significantly reducing the time required to find near-optimal solutions. The use of heuristics provides informed decision-making, allowing the algorithm to focus on promising paths and leading to timely results. Additionally, the algorithm's space complexity is manageable due to its focus on a limited set of paths at any given time. BB-H strikes a balance between computational efficiency, solution quality, and resource utilization, making it an ideal choice for time-constrained applications with limited memory resources.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the path branch and bound with heuristics.

|  |  |
| --- | --- |
| **Ex.NO: 10** | **Branch and Bound with extended list** |
| **22.09.2023** |

# Aim:

To implement branch and bound algorithm with extended list and visualize the resulting path

# Algorithm:

1. Initialization:
2. Create a priority queue to store nodes to be explored.
3. Initialize a list, called the extended list, to track visited nodes.
4. Set the current node to the starting node.
5. Calculate the current node's distance from the starting node (which is usually 0).
6. Exploration:
7. While the priority queue is not empty:
8. Remove the node with the minimum distance from the priority queue.
9. If the removed node is the goal node, the shortest path is found. Exit the loop.
10. Add the removed node to the extended list.
11. For each neighbor of the removed node:
12. If the neighbor is not in the extended list:
13. Calculate the neighbor's tentative distance from the starting node.
14. If the tentative distance is less than or equal to the neighbor's current distance from the starting node:
15. Update the neighbor's distance to the tentative distance.
16. Add the neighbor to the priority queue, along with its updated distance.
17. Output:
18. If the loop terminates without finding the goal node, no path exists.
19. If the goal node is found, reconstruct the shortest path by backtracking from the goal node to the starting node.
20. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

}

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

def form\_graph(d,FR): #FR is a function pertaining to defined search algos.

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red" return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def branch\_and\_bound\_EL(d, start, end, oracle): queue = [[0, start]]

path = [] is\_goal = 0 gn = []

vis = []

while len(queue) != 0: sl = []

cp = queue.pop(0) cn = cp[-1]

if cn == end:

gn = el break

for i in d[cn].keys(): if i in cp:

continue

if cp[0] + d[cn][i] <= oracle and i not in vis: vis.append(i)

el = cp + [i] el[0] += d[cn][i] sl.append(el)

sl.sort(key=lambda x: x[0]) queue.append(el)

sl.sort(key=lambda x: x[0]) for i in sl:

path.append(i)

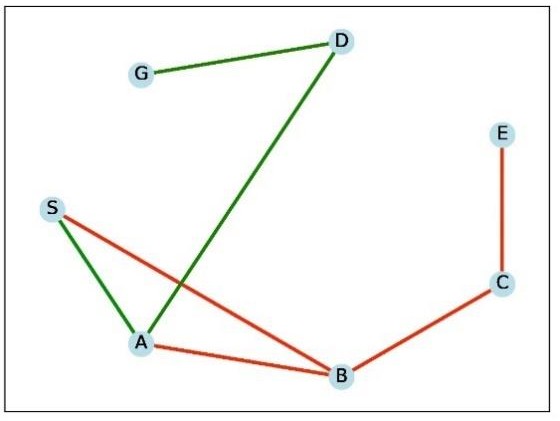
print("Visited List : ", vis) return path

FR=branch\_and\_bound\_EL(d,'S','G',11) g=form\_graph(d,FR)

Viz(g)

# A group of black letters Description automatically generatedOutput:

A black text on a white background  Description automatically generated



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 8 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

The Branch and Bound with Heuristics (BB-H) algorithm is justified for its effective utilization of time and space resources in complex problem-solving. By combining branch pruning with heuristic guidance, BB-H efficiently narrows down the search space, significantly reducing the time required to find near-optimal solutions. The use of heuristics provides informed decision-making, allowing the algorithm to focus on promising paths and leading to timely results. Additionally, the algorithm's space complexity is manageable due to its focus on a limited set of paths at any given time. BB-H strikes a balance between computational efficiency, solution quality, and resource utilization, making it an ideal choice for time-constrained applications with limited memory resources.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of path branch and bound with extended list.

|  |  |
| --- | --- |
| **Ex. No: 11** | **A\* algorithm** |
| **29.09.2023** |

# Aim:

To implement the a\* algorithm and visualize the resulting path.

# Algorithm:

1. Initialize:

Create a set of open nodes (nodes to be explored) and a set of closed nodes (already explored nodes).

Set the current node to the starting node.

Calculate the heuristic cost (h) for the current node, which is an estimate of the distance from the current node to the goal node.

Calculate the total cost (f) for the current node, which is the sum of the actual cost (g) from the starting node to the current node and the heuristic cost (h).

1. Expand:

Add the current node to the closed set. Generate the neighbors of the current node. For each neighbor:

If the neighbor is not in the open set or in the closed set:

Calculate the actual cost (g) for the neighbor, which is the actual distance from the starting node to the neighbor.

Calculate the heuristic cost (h) for the neighbor.

Calculate the total cost (f) for the neighbor, which is the sum of the actual cost (g) and the heuristic cost (h).

Add the neighbor to the open set with its updated cost.

1. Check for goal:

If the current node is the goal node, stop and return the path.

1. Select the next node:

Remove the node with the lowest total cost (f) from the open set. This is the current node.

1. Repeat:

Repeat steps 2, 3, and 4 until the goal node is found or the open set is empty.

1. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d = {}

d\_h = {}

# Input for the 'd' dictionary while True:

try:

node = input("Enter a node (or 'done' to finish for 'd' dictionary):

")

if node == 'done': break

neighbors = {} while True:

neighbor = input(f"Enter a neighbor for node {node} (or 'done' to

finish): ")

if neighbor == 'done': break

weight = float(input(f"Enter the weight for the edge from {node}

to {neighbor}: "))

neighbors[neighbor] = weight d[node] = neighbors

except ValueError:

print("Invalid input. Please enter a valid number.")

# Input for the 'd\_h' dictionary while True:

try:

node = input("Enter a node (or 'done' to finish for 'd\_h' dictionary):

")

if node == 'done': break

heuristic = {} while True:

neighbor = input(f"Enter a neighbor for node {node} (or 'done' to

finish): ")

if neighbor == 'done': break

h\_value = float(input(f"Enter the heuristic value for node

{neighbor}: "))

heuristic[neighbor] = h\_value d\_h[node] = heuristic

except ValueError:

print("Invalid input. Please enter a valid number.")

# Print the resulting dictionaries print("d:")

print(d) print("d\_h:") print(d\_h)

def form\_graph(d,FR): #FR is a function pertaining to defined search algos.

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=[] #FR=BMS(d,'S','G',2)

#FR=dfs(d) #FR=branch\_and\_bound(d,11)

#FR=A\_star\_search(d,'S','G',d\_h,oracle\_cost\_heur(d,'S','G',d\_h,2)) for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp]

l

print((l)) return g

def ret\_edge(edge):

c=0

#print(edge)

for i in g.edges: #print(i)

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

#edge\_col[c]="red"

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() #for i in range(len(l)):

for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp)

#node=nx.draw\_shell(g, list(g.nodes),edge\_color=tmp, with\_labels=True, font\_weight='bold')

nx.draw\_networkx\_labels(g,pos) node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes),

node\_size=250,node\_color='lightblue') edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

#nx.draw\_shell(g, list(g.nodes), with\_labels=True, font\_weight='bold')

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

#ani.save("D://anim.mp4")

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def A\_star\_search(d, start, end, d\_heur, oracle\_cost\_heur): queue = [[0, start]]

path = [] isgoal = 0 gn = []

vis = []

while len(queue) != 0: sl = []

cp = queue.pop(0) cn = cp[-1]

for i in d[cn].keys():

if i in cp:

continue

elif cp[0] + d[cn][i] <= oracle\_cost\_heur and i not in vis: vis.append(i)

el = cp + [i]

el[0] += d[cn][i] + d\_heur[cn][i] sl.append(el)

sl.sort(key=lambda x: x[0]) queue.append(el)

if cn == end:

isgoal = 1 gn = el break

sl.sort(key=lambda x: x[0]) for i in sl:

path.append(i) if isgoal == 1:

break print(vis) return path

def oracle\_cost\_heur(d,start,end,d\_heuristic,max\_no\_goalcutoff): queue=[[0,start]]

res=[]

g='G'

f=0 n\_g=max\_no\_goalcutoff while (len(queue)!=0):

tmp=[]

curr=queue.pop(0) # This is the list of paths with index 0 storing cumulative cost

currnode=curr[-1] # This is the last node in the traversed

path

for i in d[currnode].keys():

if i in curr:continue # prevent accessing ancesstors to child tmp1=curr+list(i) tmp1[0]=curr[0]+d[currnode][i]+d\_heuristic[currnode][i] tmp.append(tmp1)

queue.append(tmp1)

queue.sort(key=lambda k1:k1[0]) # Should have used a priority

queue but sort works !

if g==i:f+=1

for i in tmp:res.append(i) if(f==n\_g):break

goal\_node=[] for i in res:

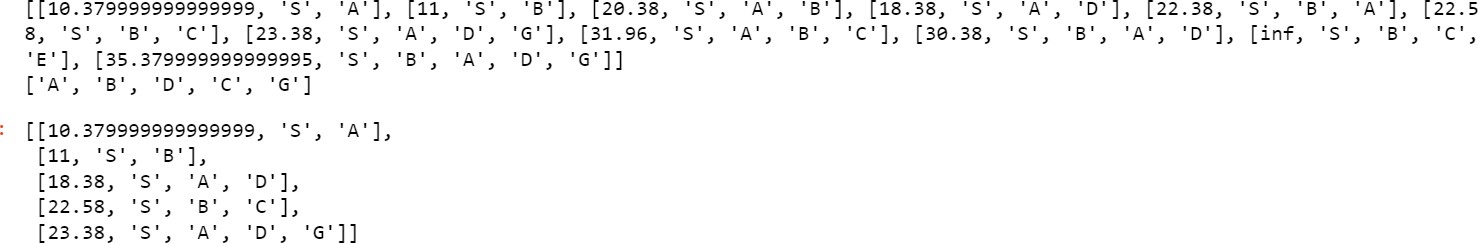
if end in i:goal\_node.append(i) print(goal\_node[0][0])

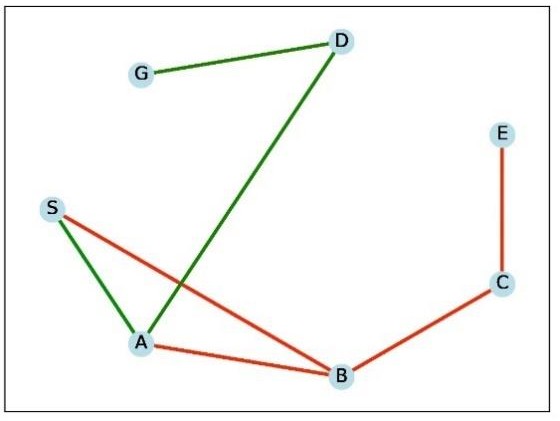
return goal\_node[0][0]

FR = A\_star\_search(d, 'S', 'G', d\_h, oracle\_cost\_heur(d, 'S', 'G', d\_h, 2)) g=form\_graph(d,FR)

Viz(g)

# Output:





**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

The A\* Search Algorithm is justified for use due to its ability to provide optimal solutions efficiently. By prioritizing promising paths using heuristics, A\* strikes a balance between completeness and speed, making it applicable to a wide range of real-world problems, including pathfinding, puzzle-solving, and robotics. Its adaptability, informed search nature, and memory-efficient variations further enhance its suitability for different scenarios. A\* allows users to customize heuristics, making it a versatile and powerful tool in finding the most efficient paths or solutions in various domains while guaranteeing optimality when an admissible heuristic is employed.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers

efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the resulting path for A \*.

|  |  |
| --- | --- |
| **Ex. No: 12** | **AO\* Algorithm** |
| **13.10.2023** |

# Aim:

To implement AO\* algorithm and visualize the resulting path

# Algorithm:

* 1. Initialize an empty list sub\_list to hold potential extensions of the current path.
  2. Pop the first path from the queue, representing the current path and its last node (choosen\_node).
  3. For each child node of the choosen\_node in the graph:
  4. If the child node is already in the current path, continue to the next child.
  5. If the child node is already visited, continue to the next child. iii. Create a new path by adding the child node to the current path.
  6. Update the cost of the new path by adding the edge weight from the choosen\_node to the child node.
  7. Update the heuristics of the new path appropriately, taking into account the difference in heuristics between the child node and the previous node in the path. vi. Enqueue the new path in the sub\_list. vii. Use the priority\_queue function to maintain a priority order in the queue based on the path cost. viii. Mark the child node as visited. ix. Increment the enqueuing count (no\_enqueue).
  8. x. If the child node is the goal, set flag to 1 and store the goal path.
  9. Sort the sub\_list in ascending order based on the path cost.
  10. Create a copy sub\_list2 of sub\_list and use the priority\_queue function to maintain a priority order in sub\_list based on the path cost.
  11. Extend the Final\_result list with the paths from sub\_list.
  12. If flag is 1, break the loop.
  13. If flag is not set to 1 after the loop, return None to indicate that no path to the goal was found.
  14. Find the index of the goal\_path in the Final\_result list and return all paths from the start to the goal.
  15. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

}

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

d\_h={

'S':{'A':7.38,'B':6},

'A':{'S':np.inf,'B':6,'D':5},

'B':{'S':np.inf,'A':7.38,'C':7.58}, 'C':{'B':6,'E':np.inf},

'E':{'C':7.58},

'D':{'A':7.38,'G':0}, 'G':{'D':5}

}

def form\_graph(d,FR):

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=

for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp] print((l))

return g

def ret\_edge(edge): c=0

for i in g.edges:

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp) nx.draw\_networkx\_labels(g,pos)

node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes), node\_size=250,node\_color='lightblue')

edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def ao\_star(graph, start, goal, cost, heuristics): visited = set()

queue = [(start, [start], 0, heuristics[start])]

while queue:

queue.sort(key=lambda x: x[3]) # Sort by heuristic value (node, path, path\_cost, heuristic\_cost) = queue.pop(0)

if node == goal:

print(f"Goal reached: {' -> '.join(path)}")

return

if node not in visited: visited.add(node)

for neighbor in graph.graph[node]: if neighbor not in path:

new\_cost = path\_cost + cost[neighbor] new\_heuristic = heuristics[neighbor] new\_path = path + [neighbor] queue.append((neighbor, new\_path, new\_cost,

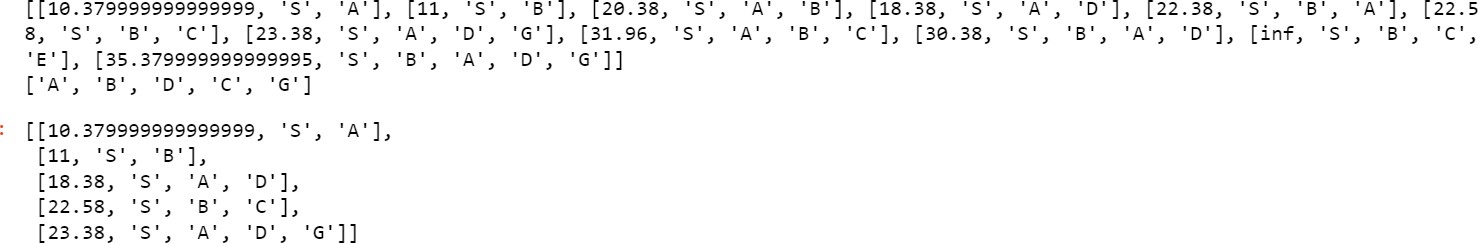
new\_heuristic))

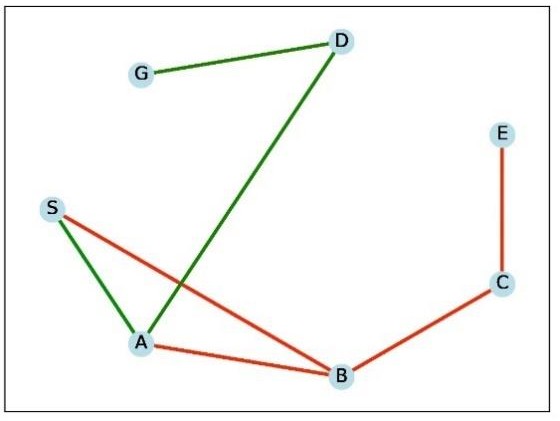
print("Goal not found") ao\_star(d,’s’,’g’,d\_h,11)

FR= ao\_star(d,’s’,’g’,d\_h,11) g=form\_graph(d,FR)

Viz(g)

# Output:





**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

This code will perform AO\* search with the specified max\_iterations parameter and return a list of solutions.

Instead of stopping the search when a goal is found, continue searching while periodically checking if the goal is still the best solution found so far.

Maintain a list of solutions found during the search, and if a new solution with a lower cost is found, update the list.

Use a parameter, such as a time limit or a number of iterations, to determine when to stop the search and return the best solution found so far.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of the path for AO \*.

|  |  |
| --- | --- |
| **Ex. No: 13** | **Best First Search** |
| **20.10.2023** |

# Aim:

To implement the best first search algorithm and visualize the resulting path

# Algorithm:

1. Initialize: Create an open list and a closed list. The open list will store the nodes that you have not yet visited, and the closed list will store the nodes that you have already visited.
2. Add the start node to the open list: Place the starting node of the graph into the open list.
3. While the open list is not empty:
   1. Remove the node with the lowest f-value from the open list: The f-value is a combination of the g-value (the distance from the start node) and the h-value (an estimate of the distance to the goal node).
   2. Expand the node: Add the neighbors of the current node to the open list. If a neighbor has already been visited, skip it.
   3. Update the g-value of each neighbor: Set the g-value of each neighbor to the g-value of the current node plus the distance between the current node and the neighbor.
   4. Update the h-value of each neighbor: If the neighbor is not the goal node, update its h-value using a heuristic function. The heuristic function should estimate the distance from the neighbor to the goal node.
   5. Check if the goal node has been reached: If the current node is the goal node, then the path has been found. Backtrack through the parent nodes to get the path from the start node to the goal node.
4. If the open list is empty and the goal node has not been found, then there is no path from the start node to the goal node.
5. Implement a visualization function. This function should take the resulting path as input and display it in a meaningful way. This could be done using a graph visualization library such as NetworkX or Matplotlib.

# Program:

import numpy as np import pandas as pd

import networkx as nx

import matplotlib.animation as animation import matplotlib.pyplot as plt

from IPython.display import HTML

d={

}

'S':{'A':3,'B':5},

'A':{'S':3,'B':4,'D':3},

'B':{'S':5,'A':4,'C':4},

'C':{'B':4,'E':6}, 'E':{'C':6}, 'D':{'A':3,'G':5}, 'G':{'D':5}

d\_h={

'S':{'A':7.38,'B':6},

'A':{'S':np.inf,'B':6,'D':5},

'B':{'S':np.inf,'A':7.38,'C':7.58}, 'C':{'B':6,'E':np.inf},

'E':{'C':7.58},

'D':{'A':7.38,'G':0}, 'G':{'D':5}

}

def form\_graph(d,FR):

global l g=nx.Graph() g.add\_nodes\_from(d) for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

l=

for i in FR:

#print(i) st=i[1:] tmp=[]

for c in range(len(i[1:])):

tmp.append(st[c:c+2]) tmp.pop(-1)

l+=[tmp] print((l)) return g

def ret\_edge(edge):

c=0

for i in g.edges:

if(edge==list(i) or edge[::-1]==list(i)): break

c+=1

return c

def animate(i): edge\_col=[]

for j in range(len(g.edges)): edge\_col+=["green"]

tmp=edge\_col.copy() for j in (l[i]):

tmp[ret\_edge(j)]="blue" print(tmp) nx.draw\_networkx\_labels(g,pos)

node=nx.draw\_networkx\_nodes(g, pos,nodelist=list(g.nodes), node\_size=250,node\_color='lightblue')

edges=nx.draw\_networkx\_edges(g,pos,edgelist=list(g.edges),edge\_color=tmp) return node,edges

def Viz(g):

global pos

pos = nx.shell\_layout(g) g=nx.Graph() g.add\_nodes\_from(d)

for i in d.keys():

for j in d[i].keys(): g.add\_edge(i,j)

print(g.edges)

fig=plt.gcf()

ani = animation.FuncAnimation(fig, animate, frames=len(l), interval=1000, repeat=True,blit=False)

plt.savefig('plotgraph.png', dpi=300, bbox\_inches='tight') plt.show()

def bestfirst(d,s,e): queue=[[0,s]] res=[]

f=0 while(len(queue)!=0):

u=queue.pop(0) if(u==e):

break else:

cn=u[-1]

for i in d[cn].keys():

if i in u:

continue if i=='G':

f=1 else:

t1=u+[i] t1[0]+=d[cn][i] queue.append(t1)

queue.sort(key=lambda k:k[0]) # priority q must be used but nevertheless the same time complexity though

res+=[u] if(f==1):

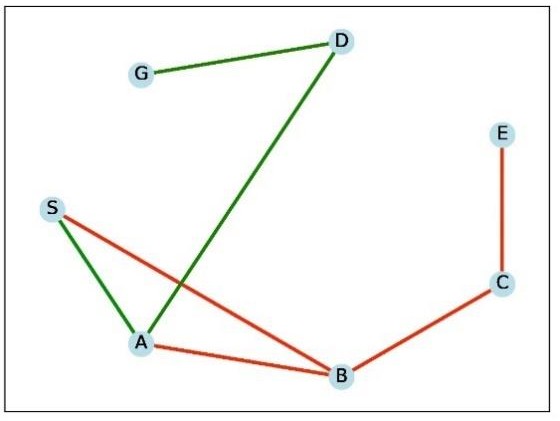
break

return res bestfirst(d,'S','G')

FR= bestfirst(d,'S','G')

g=form\_graph(d,FR) Viz(g)

# Output:



**Performance Comparison:**

|  |  |
| --- | --- |
| Total Enqueings | 6 |
| Time  Complexity | O(B \* D) |
| Space  Complexity | O(B \* D) |

# Justification:

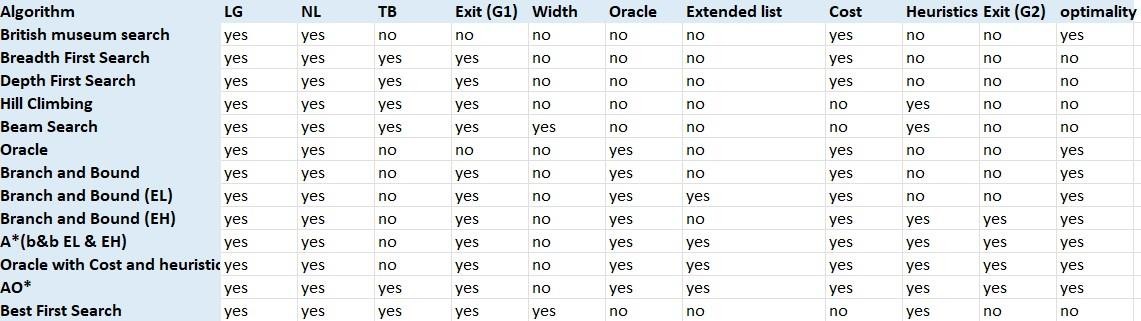
The Best-First Search (Best) Algorithm is a heuristic-based search method used for various applications, primarily for finding efficient paths or solutions in a domain. It begins with an initial state and explores the successors by prioritizing them based on their heuristic values, which represent an estimate of the cost or distance to the goal. The algorithm maintains a priority queue of successor states and selects the one with the lowest heuristic value for further exploration. This process continues until the goal state is reached or no more successors are available.

The use of HTML, CSS (Bootstrap), and Brython for webpage development is justified due to their strengths and capabilities. HTML and CSS are fundamental web technologies that ensure standardization and responsive design, while Bootstrap simplifies styling and provides responsive components. Brython facilitates the integration of Python on the client side, enabling developers to write Python code for event handling. This combination offers efficiency, cross-browser compatibility, a separation of concerns, and the ability to maintain clean and readable code, making it a well-rounded approach to web development.

# Result:

Therefore, we've successfully implemented the visualization of resulting path for best first search.

**PERFORMANCE EVALUATION AND JUSTIFICATION:**



**Result:**

Thus all search algorithms are implemented, visualized and tabulated.

|  |  |
| --- | --- |
| **Ex. No: 14** | **MIN-MAX ALGORITHM** |
| **27.10.2023** |

# Aim:

To implement min max algorithm and visualize the path.

# Algorithm:

1. Generate the game tree: Start with the current game state as the root node. Generate all possible legal moves for the current player, creating child nodes for each move. Repeat this process for each child node until the game ends or a depth limit is reached.
2. Evaluate terminal nodes: Assign a utility value to each terminal node (leaf node) in the tree. This value represents the outcome of the game from the perspective of the current player. For example, in a game of chess, a checkmate position would have a high positive utility value for the player who checkmated their opponent, while a stalemate position would have a value of zero.
3. Backpropagate utility values: For each non-terminal node, recursively calculate its utility value based on the utility values of its child nodes. If the current player is the maximizing player, choose the child node with the highest utility value. If the current player is the minimizing player, choose the child node with the lowest utility value.
4. Select the best move: The utility value of the root node represents the best possible outcome for the current player, given that the opponent plays optimally. The move that leads to this utility value is the best move for the current player.

# Program:

import math import networkx as nx import matplotlib.pyplot as plt import numpy as np

MAX = np.inf MIN = -np.inf

arr = [8, 7, 3, 9, 9, 8, 2, 4, 1, 8, 8, 9, 9, 9, 3, 4] lim =

int(math.log(len(arr), 2))

G = nx.Graph() pos = {} labels = {}

def add\_children(graph, parent\_index, depth, is\_max\_node, alpha=None, beta=None):

if depth == lim:

return arr[parent\_index] if is\_max\_node:

best = MIN for i in range(2):

child\_index = parent\_index \* 2 + i

child\_value = add\_children(graph, child\_index, depth + 1, False, alpha, beta)

best = max(best, child\_value)

# Create nodes and edges graph.add\_node(child\_index, value=arr[child\_index]) graph.add\_edge(parent\_index, child\_index)

# Positioning and labels for visualization pos[child\_index] = (depth, -child\_index) labels[child\_index] = arr[child\_index]

if alpha is not None and best >= beta: break

if alpha is not None:

alpha = max(alpha, best)

else:

return best

best = MAX

for i in range(2):

child\_index = parent\_index \* 2 + i

child\_value = add\_children(graph, child\_index, depth + 1,True,alpha, beta)

best = min(best, child\_value)

# Create nodes and edges graph.add\_node(child\_index, value=arr[child\_index]) graph.add\_edge(parent\_index, child\_index)

# Positioning and labels for

visualization pos[child\_index] = (depth, -

child\_index) labels[child\_index] = arr[child\_index] if beta is not None and best <= alpha:

break if beta is not None: beta = min(beta, best)

return best

# Visualize the alpha-beta pruning tree add\_children(G, 0, 0, True, MIN, MAX) plt.figure(figsize=(10, 8)) nx.draw(G, pos=pos, labels=labels, with\_labels=True, node\_color='lightblue', node\_size=800, font\_size=10) plt.title("Alpha-Beta Pruning Decision Tree") plt.show()

# Clear the graph and visualize the min-max tree G.clear() pos.clear() labels.clear()

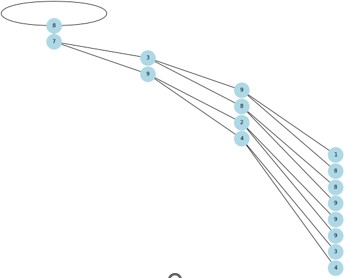
G.add\_node(0, value=arr[0]) add\_children(G, 0, 0, True)

plt.figure(figsize=(10, 8))

nx.draw(G, pos=pos, labels=labels, with\_labels=True, node\_color='lightblue', node\_size=800, font\_size=10) plt.title("Min-Max Decision Tree") plt.show()

# Output:





**Result:**

Therefore, we've successfully implemented visualization for min max algorithm.

|  |  |
| --- | --- |
| **Ex. No: 15** | **Alpha Beta Pruning** |
| **27.10.2023** |

# Aim:

To implement the alpha beta pruning algorithm and visualise the algorithm.

# Algorithm:

1. Initialization:

Start with the initial game state.

Initialize two variables, alpha and beta, to negative infinity and positive infinity, respectively.

1. Move Generation:

Generate all possible legal moves for the current player.

1. Move Evaluation and Recursion:

For each generated move:

Simulate the move by applying it to the current game state.

Evaluate the resulting game state using a utility function (heuristic) to determine its value. Check if the game is in a terminal state (win, loss, or draw) or a depth limit is reached. If so, return the utility value.

If not, recursively apply the Alpha-Beta Pruning algorithm for the next player (opponent), passing the current game state, alpha, and beta as parameters.

1. Alpha-Beta Pruning:

During the recursive call for the opponent (minimizing player):

Update beta with the minimum of beta and the utility value returned by the recursive call. If alpha is greater than or equal to beta, prune the remaining branches, meaning skip evaluating the rest of the generated moves for the current player.

1. Best Move Selection:

Once all moves have been evaluated or pruned:

If it's the current player's turn (maximizing player), select the move with the highest utility value and update alpha accordingly.

If it's the opponent's turn (minimizing player), select the move with the lowest utility value and update beta accordingly.

1. Backtracking and Final Move Selection:

Backtrack up the recursion tree, propagating the updated alpha and beta values.

At the top level, the move with the highest utility value (for the current player) represents the best move to play.

# Program:

import math

import networkx as nx

import matplotlib.pyplot as plt import numpy as np

MAX = np.inf MIN = -np.inf

arr = [8, 7, 3, 9, 9, 8, 2, 4, 1, 8, 8, 9, 9, 9, 3, 4]

lim = int(math.log(len(arr), 2))

G = nx.Graph() pos = {} labels = {}

def add\_children(graph, parent\_index, depth, is\_max\_node, alpha=None, beta=None):

if depth == lim:

return arr[parent\_index] if is\_max\_node:

best = MIN

for i in range(2):

child\_index = parent\_index \* 2 + i

child\_value = add\_children(graph, child\_index, depth + 1, False, alpha, beta)

best = max(best, child\_value)

# Create nodes and edges graph.add\_node(child\_index, value=arr[child\_index]) graph.add\_edge(parent\_index, child\_index)

# Positioning and labels for

visualization pos[child\_index] = (depth, -

child\_index) labels[child\_index] = arr[child\_index] if alpha is not None and best >= beta:

break

if alpha is not None: alpha = max(alpha, best)

else:

return best

best = MAX

for i in range(2):

child\_index = parent\_index \* 2 + i

child\_value = add\_children(graph, child\_index, depth + 1, True,

alpha, beta)

best = min(best, child\_value)

# Create nodes and edges graph.add\_node(child\_index, value=arr[child\_index]) graph.add\_edge(parent\_index, child\_index)

# Positioning and labels for

visualization pos[child\_index] = (depth, -

child\_index) labels[child\_index] = arr[child\_index] if beta is not None and best <= alpha:

break if beta is not None: beta = min(beta, best)

return best

# Visualize the alpha-beta pruning tree add\_children(G, 0, 0, True, MIN, MAX) plt.figure(figsize=(10, 8)) nx.draw(G, pos=pos, labels=labels, with\_labels=True, node\_color='lightblue', node\_size=800, font\_size=10) plt.title("Alpha-Beta Pruning Decision Tree") plt.show()

# Clear the graph and visualize the min-max tree G.clear() pos.clear() labels.clear()

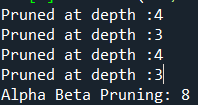
G.add\_node(0, value=arr[0]) add\_children(G, 0, 0, True)

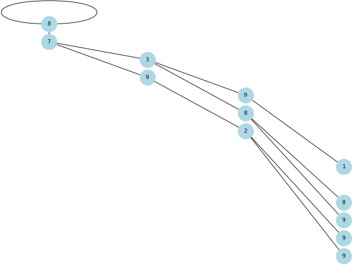
plt.figure(figsize=(10, 8))

nx.draw(G, pos=pos, labels=labels, with\_labels=True, node\_color='lightblue', node\_size=800, font\_size=10)

plt.title("Min-Max Decision Tree") plt.show()

# Output:





**Result:** Therefore, we've successfully implemented the visualization for alpha beta pruning.

|  |  |
| --- | --- |
| **Ex. No: 16** | **CARLA** |
| **27.10.2023** |

# Aim:

To implement CARLA simulation.

# Algorithm:

* 1. Create a CARLA Server: Set up a CARLA server on a machine with sufficient computational power, preferably a cloud-based virtual machine. This server will host the CARLA simulation environment.
  2. Develop a Client Application: Create a client application that can connect to the CARLA server over the internet. This application will be responsible for interacting with the simulation environment, sending commands, and receiving updates.
  3. Establish Communication: Establish a communication channel between the client application and the CARLA server. This can be achieved using a network socket or a real-time communication protocol like WebRTC.
  4. Implement Simulation Control: Implement mechanisms in the client application to control the CARLA simulation. This includes spawning vehicles, setting traffic rules, and modifying the environment.
  5. Handle Simulation Updates: Implement mechanisms in the client application to receive and process simulation updates from the CARLA server. This includes vehicle positions, traffic events, and sensor data.
  6. Visualize Simulation: Implement visualization techniques in the client application to display the CARLA simulation environment. This can be done using simple 2D graphics or more sophisticated 3D rendering libraries.
  7. Optimize for CPU Usage: Since GPU processing is not available, optimize the client application to minimize CPU usage. This may involve using efficient data structures, minimizing data transfers, and employing lightweight visualization techniques.
  8. Deploy and Test: Deploy the client application and CARLA server to separate machines over the internet. Test the communication, control, and visualization capabilities to ensure a seamless simulation experience.

# Program:

Code Snippet:

Generating NPC in the simulator:

#!/usr/bin/env python

"""Script to generate traffic in the simulation"""

import glob import os import sys import time

try:

sys.path.append(glob.glob('../carla/dist/carla-\*%d.%d-%s.egg' % ( sys.version\_info.major,

sys.version\_info.minor,

'win-amd64' if os.name == 'nt' else 'linux-x86\_64'))[0]) except IndexError:

pass

import carla

from carla import VehicleLightState as vls

import argparse import logging

from numpy import random

def get\_actor\_blueprints(world, filter, generation): bps = world.get\_blueprint\_library().filter(filter)

if generation.lower() == "all": return bps

# If the filter returns only one bp, we assume that this one needed # and therefore, we ignore the generation

if len(bps) == 1: return bps

try:

int\_generation = int(generation)

# Check if generation is in available generations if int\_generation in [1, 2]:

bps = [x for x in bps if int(x.get\_attribute('generation')) == int\_generation]

return bps else:

print(" Warning! Actor Generation is not valid. No actor will be

spawned.")

except:

return []

print(" Warning! Actor Generation is not valid. No actor will be spawned.")

return []

def main():

argparser = argparse.ArgumentParser( description= doc )

argparser.add\_argument( '--host', metavar='H', default='127.0.0.1',

help='IP of the host server (default: 127.0.0.1)') argparser.add\_argument(

'-p', '--port', metavar='P', default=2000, type=int,

help='TCP port to listen to (default: 2000)') argparser.add\_argument(

'-n', '--number-of-vehicles', metavar='N',

default=30, type=int,

help='Number of vehicles (default: 30)') argparser.add\_argument(

'-w', '--number-of-walkers', metavar='W',

default=10, type=int,

help='Number of walkers (default: 10)') argparser.add\_argument(

'--safe', action='store\_true',

help='Avoid spawning vehicles prone to accidents') argparser.add\_argument(

'--filterv', metavar='PATTERN', default='vehicle.\*',

help='Filter vehicle model (default: "vehicle.\*")') argparser.add\_argument(

'--generationv', metavar='G', default='All',

help='restrict to certain vehicle generation (values: "1","2","All" - default: "All")')

argparser.add\_argument( '--filterw', metavar='PATTERN',

default='walker.pedestrian.\*',

help='Filter pedestrian type (default: "walker.pedestrian.\*")') argparser.add\_argument(

'--generationw',

metavar='G', default='2',

help='restrict to certain pedestrian generation (values: "1","2","All"

- default: "2")') argparser.add\_argument(

'--tm-port', metavar='P', default=8000, type=int,

help='Port to communicate with TM (default: 8000)') argparser.add\_argument(

'--asynch', action='store\_true',

help='Activate asynchronous mode execution') argparser.add\_argument(

'--hybrid', action='store\_true',

help='Activate hybrid mode for Traffic Manager') argparser.add\_argument(

'-s', '--seed', metavar='S', type=int,

help='Set random device seed and deterministic mode for Traffic Manager')

argparser.add\_argument( '--seedw', metavar='S', default=0, type=int,

help='Set the seed for pedestrians module') argparser.add\_argument(

'--car-lights-on', action='store\_true', default=False,

help='Enable automatic car light management') argparser.add\_argument(

'--hero', action='store\_true', default=False,

help='Set one of the vehicles as hero') argparser.add\_argument(

'--respawn', action='store\_true', default=False,

help='Automatically respawn dormant vehicles (only in large maps)') argparser.add\_argument(

'--no-rendering', action='store\_true',

default=False,

help='Activate no rendering mode')

args = argparser.parse\_args()

logging.basicConfig(format='%(levelname)s: %(message)s', level=logging.INFO)

vehicles\_list = [] walkers\_list = [] all\_id = []

client = carla.Client(args.host, args.port) client.set\_timeout(10.0) synchronous\_master = False

random.seed(args.seed if args.seed is not None else int(time.time()))

try:

world = client.get\_world()

traffic\_manager = client.get\_trafficmanager(args.tm\_port) traffic\_manager.set\_global\_distance\_to\_leading\_vehicle(2.5) if args.respawn:

traffic\_manager.set\_respawn\_dormant\_vehicles(True) if args.hybrid:

traffic\_manager.set\_hybrid\_physics\_mode(True) traffic\_manager.set\_hybrid\_physics\_radius(70.0)

if args.seed is not None: traffic\_manager.set\_random\_device\_seed(args.seed)

settings = world.get\_settings() if not args.asynch:

traffic\_manager.set\_synchronous\_mode(True) if not settings.synchronous\_mode:

synchronous\_master = True settings.synchronous\_mode = True settings.fixed\_delta\_seconds = 0.05

else:

synchronous\_master = False

else:

print("You are currently in asynchronous mode. If this is a traffic simulation, \

you could experience some issues. If it's not working correctly, switch to synchronous \

mode by using traffic\_manager.set\_synchronous\_mode(True)")

if args.no\_rendering: settings.no\_rendering\_mode = True

world.apply\_settings(settings)

blueprints = get\_actor\_blueprints(world, args.filterv, args.generationv)

blueprintsWalkers = get\_actor\_blueprints(world, args.filterw, args.generationw)

if args.safe:

blueprints = [x for x in blueprints if x.get\_attribute('base\_type') == 'car']

blueprints = sorted(blueprints, key=lambda bp: bp.id)

spawn\_points = world.get\_map().get\_spawn\_points() number\_of\_spawn\_points = len(spawn\_points)

if args.number\_of\_vehicles < number\_of\_spawn\_points: random.shuffle(spawn\_points)

elif args.number\_of\_vehicles > number\_of\_spawn\_points:

msg = 'requested %d vehicles, but could only find %d spawn points' logging.warning(msg, args.number\_of\_vehicles,

number\_of\_spawn\_points)

args.number\_of\_vehicles = number\_of\_spawn\_points

# @todo cannot import these directly. SpawnActor = carla.command.SpawnActor SetAutopilot = carla.command.SetAutopilot FutureActor = carla.command.FutureActor

# # Spawn vehicles # batch = []

hero = args.hero

for n, transform in enumerate(spawn\_points): if n >= args.number\_of\_vehicles:

break

blueprint = random.choice(blueprints) if blueprint.has\_attribute('color'):

color = random.choice(blueprint.get\_attribute('color').recommended\_values)

blueprint.set\_attribute('color', color) if blueprint.has\_attribute('driver\_id'):

driver\_id = random.choice(blueprint.get\_attribute('driver\_id').recommended\_values)

blueprint.set\_attribute('driver\_id', driver\_id) if hero:

blueprint.set\_attribute('role\_name', 'hero') hero = False

else:

blueprint.set\_attribute('role\_name', 'autopilot')

together

# spawn the cars and set their autopilot and light state all

batch.append(SpawnActor(blueprint, transform)

.then(SetAutopilot(FutureActor, True,

traffic\_manager.get\_port())))

for response in client.apply\_batch\_sync(batch, synchronous\_master): if response.error:

logging.error(response.error) else:

vehicles\_list.append(response.actor\_id)

# Set automatic vehicle lights update if specified if args.car\_lights\_on:

all\_vehicle\_actors = world.get\_actors(vehicles\_list) for actor in all\_vehicle\_actors:

traffic\_manager.update\_vehicle\_lights(actor, True)

run

# # Spawn Walkers # # some settings

percentagePedestriansRunning = 0.0 # how many pedestrians will

percentagePedestriansCrossing = 0.0 # how many pedestrians will

walk through the road

if args.seedw: world.set\_pedestrians\_seed(args.seedw) random.seed(args.seedw)

# 1. take all the random locations to spawn spawn\_points = []

for i in range(args.number\_of\_walkers): spawn\_point = carla.Transform()

loc = world.get\_random\_location\_from\_navigation() if (loc != None):

spawn\_point.location = loc spawn\_points.append(spawn\_point)

# 2. we spawn the walker object batch = []

walker\_speed = []

for spawn\_point in spawn\_points:

walker\_bp = random.choice(blueprintsWalkers) # set as not invincible

if walker\_bp.has\_attribute('is\_invincible'): walker\_bp.set\_attribute('is\_invincible', 'false')

# set the max speed

if walker\_bp.has\_attribute('speed'):

if (random.random() > percentagePedestriansRunning): # walking

walker\_speed.append(walker\_bp.get\_attribute('speed').recom

mended\_values[1])

else:

# running walker\_speed.append(walker\_bp.get\_attribute('speed').recom

mended\_values[2])

else:

print("Walker has no speed") walker\_speed.append(0.0)

batch.append(SpawnActor(walker\_bp, spawn\_point)) results = client.apply\_batch\_sync(batch, True) walker\_speed2 = []

for i in range(len(results)): if results[i].error:

logging.error(results[i].error) else:

walkers\_list.append({"id": results[i].actor\_id}) walker\_speed2.append(walker\_speed[i])

walker\_speed = walker\_speed2

# 3. we spawn the walker controller batch = []

walker\_controller\_bp = world.get\_blueprint\_library().find('controller.ai.walker')

for i in range(len(walkers\_list)): batch.append(SpawnActor(walker\_controller\_bp, carla.Transform(),

walkers\_list[i]["id"]))

results = client.apply\_batch\_sync(batch, True) for i in range(len(results)):

if results[i].error: logging.error(results[i].error)

else:

walkers\_list[i]["con"] = results[i].actor\_id

# 4. we put together the walkers and controllers id to get the objects from their id

for i in range(len(walkers\_list)): all\_id.append(walkers\_list[i]["con"]) all\_id.append(walkers\_list[i]["id"])

all\_actors = world.get\_actors(all\_id)

# wait for a tick to ensure client receives the last transform of the walkers we have just created

if args.asynch or not synchronous\_master: world.wait\_for\_tick()

else:

world.tick()

# 5. initialize each controller and set target to walk to (list is [controler, actor, controller, actor ...])

# set how many pedestrians can cross the road world.set\_pedestrians\_cross\_factor(percentagePedestriansCrossing) for i in range(0, len(all\_id), 2):

# start walker all\_actors[i].start()

# set walk to random point all\_actors[i].go\_to\_location(world.get\_random\_location\_from\_naviga

tion())

# max speed all\_actors[i].set\_max\_speed(float(walker\_speed[int(i/2)]))

print('spawned %d vehicles and %d walkers, press Ctrl+C to exit.' % (len(vehicles\_list), len(walkers\_list)))

# Example of how to use Traffic Manager parameters traffic\_manager.global\_percentage\_speed\_difference(30.0)

while True:

if not args.asynch and synchronous\_master: world.tick()

else:

world.wait\_for\_tick()

finally:

if not args.asynch and synchronous\_master: settings = world.get\_settings() settings.synchronous\_mode = False settings.no\_rendering\_mode = False settings.fixed\_delta\_seconds = None world.apply\_settings(settings)

print('\ndestroying %d vehicles' % len(vehicles\_list)) client.apply\_batch([carla.command.DestroyActor(x) for x in

vehicles\_list])

# stop walker controllers (list is [controller, actor, controller, actor ...])

for i in range(0, len(all\_id), 2): all\_actors[i].stop()

print('\ndestroying %d walkers' % len(walkers\_list)) client.apply\_batch([carla.command.DestroyActor(x) for x in all\_id])

time.sleep(0.5) if name == ' main ':

try:

main()

except KeyboardInterrupt: pass

finally:

print('\ndone.')

Python code for randomizing the climate:

#!/usr/bin/env python

"""

CARLA Dynamic Weather:

Connect to a CARLA Simulator instance and control the weather. Change Sun position smoothly with time and generate storms occasionally.

"""

import glob import os import sys

try:

sys.path.append(glob.glob('../carla/dist/carla-\*%d.%d-%s.egg' % ( sys.version\_info.major,

sys.version\_info.minor,

'win-amd64' if os.name == 'nt' else 'linux-x86\_64'))[0]) except IndexError:

pass import carla

import argparse import math

def clamp(value, minimum=0.0, maximum=100.0): return max(minimum, min(value, maximum))

class Sun(object):

def init (self, azimuth, altitude): self.azimuth = azimuth self.altitude = altitude

self.\_t = 0.0

def tick(self, delta\_seconds): self.\_t += 0.008 \* delta\_seconds self.\_t %= 2.0 \* math.pi

self.azimuth += 0.25 \* delta\_seconds self.azimuth %= 360.0

self.altitude = (70 \* math.sin(self.\_t)) - 20

def str (self):

return 'Sun(alt: %.2f, azm: %.2f)' % (self.altitude, self.azimuth)

class Storm(object):

def init (self, precipitation):

self.\_t = precipitation if precipitation > 0.0 else -50.0 self.\_increasing = True

self.clouds = 0.0

self.rain = 0.0

self.wetness = 0.0

self.puddles = 0.0

self.wind = 0.0

self.fog = 0.0

def tick(self, delta\_seconds):

delta = (1.3 if self.\_increasing else -1.3) \* delta\_seconds self.\_t = clamp(delta + self.\_t, -250.0, 100.0) self.clouds = clamp(self.\_t + 40.0, 0.0, 90.0)

self.rain = clamp(self.\_t, 0.0, 80.0) delay = -10.0 if self.\_increasing else 90.0

self.puddles = clamp(self.\_t + delay, 0.0, 85.0) self.wetness = clamp(self.\_t \* 5, 0.0, 100.0)

self.wind = 5.0 if self.clouds <= 20 else 90 if self.clouds >= 70 else

40

self.fog = clamp(self.\_t - 10, 0.0, 30.0) if self.\_t == -250.0:

self.\_increasing = True if self.\_t == 100.0:

self.\_increasing = False

def str (self):

return 'Storm(clouds=%d%%, rain=%d%%, wind=%d%%)' % (self.clouds, self.rain, self.wind)

class Weather(object):

def init (self, weather): self.weather = weather

self.\_sun = Sun(weather.sun\_azimuth\_angle, weather.sun\_altitude\_angle) self.\_storm = Storm(weather.precipitation)

def tick(self, delta\_seconds): self.\_sun.tick(delta\_seconds) self.\_storm.tick(delta\_seconds) self.weather.cloudiness = self.\_storm.clouds self.weather.precipitation = self.\_storm.rain

self.weather.precipitation\_deposits = self.\_storm.puddles self.weather.wind\_intensity = self.\_storm.wind self.weather.fog\_density = self.\_storm.fog self.weather.wetness = self.\_storm.wetness self.weather.sun\_azimuth\_angle = self.\_sun.azimuth self.weather.sun\_altitude\_angle = self.\_sun.altitude

def str (self):

return '%s %s' % (self.\_sun, self.\_storm)

def main():

argparser = argparse.ArgumentParser( description= doc )

argparser.add\_argument( '--host', metavar='H', default='127.0.0.1',

help='IP of the host server (default: 127.0.0.1)') argparser.add\_argument(

'-p', '--port', metavar='P', default=2000, type=int,

help='TCP port to listen to (default: 2000)') argparser.add\_argument(

'-s', '--speed', metavar='FACTOR', default=1.0, type=float,

help='rate at which the weather changes (default: 1.0)') args = argparser.parse\_args()

speed\_factor = args.speed update\_freq = 0.1 / speed\_factor

client = carla.Client(args.host, args.port) client.set\_timeout(2.0)

world = client.get\_world()

weather = Weather(world.get\_weather()) elapsed\_time = 0.0

while True:

timestamp = world.wait\_for\_tick(seconds=30.0).timestamp elapsed\_time += timestamp.delta\_seconds

if elapsed\_time > update\_freq: weather.tick(speed\_factor \* elapsed\_time) world.set\_weather(weather.weather) sys.stdout.write('\r' + str(weather) + 12 \* ' ') sys.stdout.flush()

elapsed\_time = 0.0

if name == ' main ': main()

Python code for manual user control:

#!/usr/bin/env python

# Allows controlling a vehicle with a keyboard. """

Welcome to CARLA manual control.

Use ARROWS or WASD keys for control.

W : throttle

S : brake

A/D : steer left/right

Q : toggle reverse

Space : hand-brake

P : toggle autopilot

M : toggle manual transmission

,/. : gear up/down

CTRL + W : toggle constant velocity mode at 60 km/h

L : toggle next light type SHIFT + L : toggle high beam

Z/X : toggle right/left blinker

I : toggle interior light

TAB : change sensor position

` or N : next sensor

[1-9] : change to sensor [1-9]

G : toggle radar visualization

C : change weather (Shift+C reverse) Backspace : change vehicle

O : open/close all doors of vehicle

T : toggle vehicle's telemetry

V : Select next map layer (Shift+V reverse)

B : Load current selected map layer (Shift+B to unload)

R : toggle recording images to disk

CTRL + R : toggle recording of simulation (replacing any previous) CTRL + P : start replaying last recorded simulation

CTRL + + : increments the start time of the replay by 1 second (+SHIFT

= 10 seconds)

CTRL + - : decrements the start time of the replay by 1 second (+SHIFT

= 10 seconds)

"""

F1 : toggle HUD

H/? : toggle help

ESC : quit

from future import print\_function

#

==============================================================================

# find carla module

-- #

==============================================================================

import glob import os import sys

try:

sys.path.append(glob.glob('../carla/dist/carla-\*%d.%d-%s.egg' % ( sys.version\_info.major,

sys.version\_info.minor,

'win-amd64' if os.name == 'nt' else 'linux-x86\_64'))[0]) except IndexError:

pass

#

==============================================================================

# imports

-- #

==============================================================================

import carla

from carla import ColorConverter as cc

import argparse import collections import datetime import logging import math

import random import re import weakref

try:

import pygame

from pygame.locals import KMOD\_CTRL from pygame.locals import KMOD\_SHIFT from pygame.locals import K\_0

from pygame.locals import K\_9

from pygame.locals import K\_BACKQUOTE from pygame.locals import K\_BACKSPACE from pygame.locals import K\_COMMA

from pygame.locals import K\_DOWN from pygame.locals import K\_ESCAPE from pygame.locals import K\_F1 from pygame.locals import K\_LEFT from pygame.locals import K\_PERIOD from pygame.locals import K\_RIGHT from pygame.locals import K\_SLASH from pygame.locals import K\_SPACE from pygame.locals import K\_TAB from pygame.locals import K\_UP from pygame.locals import K\_a

from pygame.locals import K\_b from pygame.locals import K\_c from pygame.locals import K\_d from pygame.locals import K\_f from pygame.locals import K\_g from pygame.locals import K\_h from pygame.locals import K\_i from pygame.locals import K\_l from pygame.locals import K\_m from pygame.locals import K\_n from pygame.locals import K\_o from pygame.locals import K\_p from pygame.locals import K\_q from pygame.locals import K\_r from pygame.locals import K\_s from pygame.locals import K\_t from pygame.locals import K\_v from pygame.locals import K\_w from pygame.locals import K\_x from pygame.locals import K\_z from pygame.locals import K\_MINUS

from pygame.locals import K\_EQUALS except ImportError:

raise RuntimeError('cannot import pygame, make sure pygame package is installed')

try:

import numpy as np except ImportError:

raise RuntimeError('cannot import numpy, make sure numpy package is installed')

#

==============================================================================

# Global functions

--

#

==============================================================================

def find\_weather\_presets():

rgx = re.compile('.+?(?:(?<=[a-z])(?=[A-Z])|(?<=[A-Z])(?=[A-Z][a-z])|$)')

name = lambda x: ' '.join(m.group(0) for m in rgx.finditer(x))

presets = [x for x in dir(carla.WeatherParameters) if re.match('[A-Z].+',

x)]

return [(getattr(carla.WeatherParameters, x), name(x)) for x in presets]

def get\_actor\_display\_name(actor, truncate=250):

name = ' '.join(actor.type\_id.replace('\_', '.').title().split('.')[1:]) return (name[:truncate - 1] + u'\u2026') if len(name) > truncate else name

def get\_actor\_blueprints(world, filter, generation): bps = world.get\_blueprint\_library().filter(filter)

if generation.lower() == "all": return bps

# If the filter returns only one bp, we assume that this one needed # and therefore, we ignore the generation

if len(bps) == 1: return bps

try:

int\_generation = int(generation)

# Check if generation is in available generations if int\_generation in [1, 2]:

bps = [x for x in bps if int(x.get\_attribute('generation')) == int\_generation]

return bps else:

print(" Warning! Actor Generation is not valid. No actor will be

spawned.")

except:

return []

print(" Warning! Actor Generation is not valid. No actor will be spawned.")

return []

#

==============================================================================

# World

--

#

==============================================================================

class World(object):

def init (self, carla\_world, hud, args): self.world = carla\_world

self.sync = args.sync self.actor\_role\_name = args.rolename try:

self.map = self.world.get\_map() except RuntimeError as error:

print('RuntimeError: {}'.format(error))

print(' The server could not send the OpenDRIVE (.xodr) file:') print(' Make sure it exists, has the same name of your town, and

is correct.')

sys.exit(1) self.hud = hud self.player = None

self.collision\_sensor = None self.lane\_invasion\_sensor = None self.gnss\_sensor = None self.imu\_sensor = None self.radar\_sensor = None self.camera\_manager = None

self.\_weather\_presets = find\_weather\_presets() self.\_weather\_index = 0

self.\_actor\_filter = args.filter self.\_actor\_generation = args.generation self.\_gamma = args.gamma

self.restart() self.world.on\_tick(hud.on\_world\_tick) self.recording\_enabled = False self.recording\_start = 0 self.constant\_velocity\_enabled = False self.show\_vehicle\_telemetry = False self.doors\_are\_open = False self.current\_map\_layer = 0 self.map\_layer\_names = [

carla.MapLayer.NONE, carla.MapLayer.Buildings, carla.MapLayer.Decals, carla.MapLayer.Foliage, carla.MapLayer.Ground, carla.MapLayer.ParkedVehicles, carla.MapLayer.Particles, carla.MapLayer.Props, carla.MapLayer.StreetLights, carla.MapLayer.Walls,

carla.MapLayer.All

]

def restart(self): self.player\_max\_speed = 1.589

self.player\_max\_speed\_fast = 3.713

# Keep same camera config if the camera manager exists.

cam\_index = self.camera\_manager.index if self.camera\_manager is not None else 0

cam\_pos\_index = self.camera\_manager.transform\_index if self.camera\_manager is not None else 0

# Get a random blueprint.

blueprint = random.choice(get\_actor\_blueprints(self.world, self.\_actor\_filter, self.\_actor\_generation))

blueprint.set\_attribute('role\_name', self.actor\_role\_name) if blueprint.has\_attribute('terramechanics'):

blueprint.set\_attribute('terramechanics', 'true') if blueprint.has\_attribute('color'):

color = random.choice(blueprint.get\_attribute('color').recommended\_values)

blueprint.set\_attribute('color', color) if blueprint.has\_attribute('driver\_id'):

driver\_id = random.choice(blueprint.get\_attribute('driver\_id').recommended\_values)

blueprint.set\_attribute('driver\_id', driver\_id) if blueprint.has\_attribute('is\_invincible'):

blueprint.set\_attribute('is\_invincible', 'true') # set the max speed

if blueprint.has\_attribute('speed'): self.player\_max\_speed =

float(blueprint.get\_attribute('speed').recommended\_values[1]) self.player\_max\_speed\_fast =

float(blueprint.get\_attribute('speed').recommended\_values[2])

# Spawn the player.

if self.player is not None:

spawn\_point = self.player.get\_transform() spawn\_point.location.z += 2.0

spawn\_point.rotation.roll = 0.0

spawn\_point.rotation.pitch = 0.0 self.destroy()

self.player = self.world.try\_spawn\_actor(blueprint, spawn\_point) self.show\_vehicle\_telemetry = False self.modify\_vehicle\_physics(self.player)

while self.player is None:

if not self.map.get\_spawn\_points():

print('There are no spawn points available in your map/town.')

scene.')

print('Please add some Vehicle Spawn Point to your UE4

sys.exit(1)

spawn\_points = self.map.get\_spawn\_points()

spawn\_point = random.choice(spawn\_points) if spawn\_points else

carla.Transform()

self.player = self.world.try\_spawn\_actor(blueprint, spawn\_point) self.show\_vehicle\_telemetry = False self.modify\_vehicle\_physics(self.player)

# Set up the sensors.

self.collision\_sensor = CollisionSensor(self.player, self.hud) self.lane\_invasion\_sensor = LaneInvasionSensor(self.player, self.hud) self.gnss\_sensor = GnssSensor(self.player)

self.imu\_sensor = IMUSensor(self.player) self.camera\_manager = CameraManager(self.player, self.hud,

self.\_gamma)

self.camera\_manager.transform\_index = cam\_pos\_index self.camera\_manager.set\_sensor(cam\_index, notify=False) actor\_type = get\_actor\_display\_name(self.player) self.hud.notification(actor\_type)

if self.sync:

self.world.tick() else:

self.world.wait\_for\_tick()

def next\_weather(self, reverse=False): self.\_weather\_index += -1 if reverse else 1 self.\_weather\_index %= len(self.\_weather\_presets) preset = self.\_weather\_presets[self.\_weather\_index] self.hud.notification('Weather: %s' % preset[1]) self.player.get\_world().set\_weather(preset[0])

def next\_map\_layer(self, reverse=False): self.current\_map\_layer += -1 if reverse else 1 self.current\_map\_layer %= len(self.map\_layer\_names) selected = self.map\_layer\_names[self.current\_map\_layer] self.hud.notification('LayerMap selected: %s' % selected)

def load\_map\_layer(self, unload=False):

selected = self.map\_layer\_names[self.current\_map\_layer] if unload:

self.hud.notification('Unloading map layer: %s' % selected) self.world.unload\_map\_layer(selected)

else:

self.hud.notification('Loading map layer: %s' % selected) self.world.load\_map\_layer(selected)

def toggle\_radar(self):

if self.radar\_sensor is None:

self.radar\_sensor = RadarSensor(self.player) elif self.radar\_sensor.sensor is not None:

self.radar\_sensor.sensor.destroy() self.radar\_sensor = None

def modify\_vehicle\_physics(self, actor):

#If actor is not a vehicle, we cannot use the physics control try:

physics\_control = actor.get\_physics\_control() physics\_control.use\_sweep\_wheel\_collision = True actor.apply\_physics\_control(physics\_control)

except Exception: pass

def tick(self, clock): self.hud.tick(self, clock)

def render(self, display): self.camera\_manager.render(display) self.hud.render(display)

def destroy\_sensors(self): self.camera\_manager.sensor.destroy() self.camera\_manager.sensor = None self.camera\_manager.index = None

def destroy(self):

if self.radar\_sensor is not None: self.toggle\_radar()

sensors = [

self.camera\_manager.sensor, self.collision\_sensor.sensor, self.lane\_invasion\_sensor.sensor, self.gnss\_sensor.sensor, self.imu\_sensor.sensor]

for sensor in sensors:

if sensor is not None: sensor.stop() sensor.destroy()

if self.player is not None: self.player.destroy()

#

==============================================================================

# KeyboardControl

--

#

==============================================================================

class KeyboardControl(object):

"""Class that handles keyboard input."""

def init (self, world, start\_in\_autopilot): self.\_autopilot\_enabled = start\_in\_autopilot self.\_ackermann\_enabled = False self.\_ackermann\_reverse = 1

if isinstance(world.player, carla.Vehicle): self.\_control = carla.VehicleControl()

self.\_ackermann\_control = carla.VehicleAckermannControl() self.\_lights = carla.VehicleLightState.NONE world.player.set\_autopilot(self.\_autopilot\_enabled) world.player.set\_light\_state(self.\_lights)

elif isinstance(world.player, carla.Walker): self.\_control = carla.WalkerControl() self.\_autopilot\_enabled = False

self.\_rotation = world.player.get\_transform().rotation else:

raise NotImplementedError("Actor type not supported") self.\_steer\_cache = 0.0

world.hud.notification("Press 'H' or '?' for help.", seconds=4.0)

def parse\_events(self, client, world, clock, sync\_mode): if isinstance(self.\_control, carla.VehicleControl):

current\_lights = self.\_lights for event in pygame.event.get():

if event.type == pygame.QUIT: return True

elif event.type == pygame.KEYUP:

if self.\_is\_quit\_shortcut(event.key): return True

elif event.key == K\_BACKSPACE: if self.\_autopilot\_enabled:

world.player.set\_autopilot(False) world.restart() world.player.set\_autopilot(True)

else:

world.restart() elif event.key == K\_F1:

world.hud.toggle\_info()

elif event.key == K\_v and pygame.key.get\_mods() & KMOD\_SHIFT: world.next\_map\_layer(reverse=True)

elif event.key == K\_v: world.next\_map\_layer()

elif event.key == K\_b and pygame.key.get\_mods() & KMOD\_SHIFT: world.load\_map\_layer(unload=True)

elif event.key == K\_b: world.load\_map\_layer()

elif event.key == K\_h or (event.key == K\_SLASH and pygame.key.get\_mods() & KMOD\_SHIFT):

world.hud.help.toggle() elif event.key == K\_TAB:

world.camera\_manager.toggle\_camera()

elif event.key == K\_c and pygame.key.get\_mods() & KMOD\_SHIFT: world.next\_weather(reverse=True)

elif event.key == K\_c: world.next\_weather()

elif event.key == K\_g: world.toggle\_radar()

elif event.key == K\_BACKQUOTE: world.camera\_manager.next\_sensor()

elif event.key == K\_n: world.camera\_manager.next\_sensor()

elif event.key == K\_w and (pygame.key.get\_mods() & KMOD\_CTRL): if world.constant\_velocity\_enabled:

world.player.disable\_constant\_velocity() world.constant\_velocity\_enabled = False world.hud.notification("Disabled Constant Velocity

Mode")

7, 0, 0))

at 60 km/h")

else:

world.player.enable\_constant\_velocity(carla.Vector3D(1

world.constant\_velocity\_enabled = True world.hud.notification("Enabled Constant Velocity Mode

elif event.key == K\_o: try:

if world.doors\_are\_open: world.hud.notification("Closing Doors") world.doors\_are\_open = False world.player.close\_door(carla.VehicleDoor.All)

else:

world.hud.notification("Opening doors") world.doors\_are\_open = True world.player.open\_door(carla.VehicleDoor.All)

except Exception: pass

elif event.key == K\_t:

if world.show\_vehicle\_telemetry: world.player.show\_debug\_telemetry(False) world.show\_vehicle\_telemetry = False world.hud.notification("Disabled Vehicle Telemetry")

else:

try:

Telemetry")

index\_ctrl) KMOD\_CTRL):

world.player.show\_debug\_telemetry(True) world.show\_vehicle\_telemetry = True world.hud.notification("Enabled Vehicle

except Exception: pass

elif event.key > K\_0 and event.key <= K\_9: index\_ctrl = 0

if pygame.key.get\_mods() & KMOD\_CTRL: index\_ctrl = 9

world.camera\_manager.set\_sensor(event.key - 1 - K\_0 + elif event.key == K\_r and not (pygame.key.get\_mods() &

world.camera\_manager.toggle\_recording()

elif event.key == K\_r and (pygame.key.get\_mods() & KMOD\_CTRL): if (world.recording\_enabled):

client.stop\_recorder() world.recording\_enabled = False world.hud.notification("Recorder is OFF")

else:

client.start\_recorder("manual\_recording.rec") world.recording\_enabled = True world.hud.notification("Recorder is ON")

elif event.key == K\_p and (pygame.key.get\_mods() & KMOD\_CTRL): # stop recorder

client.stop\_recorder() world.recording\_enabled = False

# work around to fix camera at start of replaying current\_index = world.camera\_manager.index world.destroy\_sensors()

# disable autopilot self.\_autopilot\_enabled = False

world.player.set\_autopilot(self.\_autopilot\_enabled) world.hud.notification("Replaying file

'manual\_recording.rec'")

# replayer client.replay\_file("manual\_recording.rec",

world.recording\_start, 0, 0)

world.camera\_manager.set\_sensor(current\_index) elif event.key == K\_MINUS and (pygame.key.get\_mods() &

KMOD\_CTRL):

if pygame.key.get\_mods() & KMOD\_SHIFT: world.recording\_start -= 10

else:

world.recording\_start -= 1 world.hud.notification("Recording start time is %d" %

(world.recording\_start))

KMOD\_CTRL):

elif event.key == K\_EQUALS and (pygame.key.get\_mods() &

if pygame.key.get\_mods() & KMOD\_SHIFT: world.recording\_start += 10

else:

world.recording\_start += 1 world.hud.notification("Recording start time is %d" %

(world.recording\_start))

if isinstance(self.\_control, carla.VehicleControl): if event.key == K\_f:

# Toggle ackermann controller

self.\_ackermann\_enabled = not self.\_ackermann\_enabled world.hud.show\_ackermann\_info(self.\_ackermann\_enabled) world.hud.notification("Ackermann Controller %s" %

("Enabled" if

self.\_ackermann\_enabled else "Disabled"))

if event.key == K\_q:

if not self.\_ackermann\_enabled:

self.\_control.gear = 1 if self.\_control.reverse

else -1

else:

self.\_ackermann\_reverse \*= -1 # Reset ackermann control self.\_ackermann\_control =

carla.VehicleAckermannControl()

elif event.key == K\_m: self.\_control.manual\_gear\_shift = not

self.\_control.manual\_gear\_shift

self.\_control.gear = world.player.get\_control().gear world.hud.notification('%s Transmission' %

('Manual' if self.\_control.manual\_gear\_shift else 'Automatic'))

elif self.\_control.manual\_gear\_shift and event.key ==

K\_COMMA:

K\_PERIOD:

KMOD\_CTRL:

mode and could " simulation")

self.\_control.gear = max(-1, self.\_control.gear - 1) elif self.\_control.manual\_gear\_shift and event.key ==

self.\_control.gear = self.\_control.gear + 1

elif event.key == K\_p and not pygame.key.get\_mods() &

if not self.\_autopilot\_enabled and not sync\_mode: print("WARNING: You are currently in asynchronous

"experience some issues with the traffic

self.\_autopilot\_enabled = not self.\_autopilot\_enabled world.player.set\_autopilot(self.\_autopilot\_enabled) world.hud.notification(

else 'Off')) KMOD\_CTRL:

KMOD\_SHIFT:

'Autopilot %s' % ('On' if self.\_autopilot\_enabled elif event.key == K\_l and pygame.key.get\_mods() &

current\_lights ^= carla.VehicleLightState.Special1 elif event.key == K\_l and pygame.key.get\_mods() &

current\_lights ^= carla.VehicleLightState.HighBeam elif event.key == K\_l:

# Use 'L' key to switch between lights: # closed -> position -> low beam -> fog if not self.\_lights &

carla.VehicleLightState.Position:

world.hud.notification("Position lights") current\_lights |= carla.VehicleLightState.Position

else:

world.hud.notification("Low beam lights") current\_lights |= carla.VehicleLightState.LowBeam

if self.\_lights & carla.VehicleLightState.LowBeam: world.hud.notification("Fog lights") current\_lights |= carla.VehicleLightState.Fog

if self.\_lights & carla.VehicleLightState.Fog: world.hud.notification("Lights off") current\_lights ^= carla.VehicleLightState.Position current\_lights ^= carla.VehicleLightState.LowBeam current\_lights ^= carla.VehicleLightState.Fog

elif event.key == K\_i:

current\_lights ^= carla.VehicleLightState.Interior elif event.key == K\_z:

current\_lights ^= carla.VehicleLightState.LeftBlinker elif event.key == K\_x:

current\_lights ^= carla.VehicleLightState.RightBlinker

if not self.\_autopilot\_enabled:

if isinstance(self.\_control, carla.VehicleControl): self.\_parse\_vehicle\_keys(pygame.key.get\_pressed(),

clock.get\_time())

self.\_control.reverse = self.\_control.gear < 0 # Set automatic control-related vehicle lights if self.\_control.brake:

current\_lights |= carla.VehicleLightState.Brake else: # Remove the Brake flag

current\_lights &= ~carla.VehicleLightState.Brake if self.\_control.reverse:

current\_lights |= carla.VehicleLightState.Reverse else: # Remove the Reverse flag

current\_lights &= ~carla.VehicleLightState.Reverse

if current\_lights != self.\_lights: # Change the light state only if necessary

\_lights))

ol) controller.

)

self.\_lights = current\_lights world.player.set\_light\_state(carla.VehicleLightState(self.

# Apply control

if not self.\_ackermann\_enabled: world.player.apply\_control(self.\_control)

else:

world.player.apply\_ackermann\_control(self.\_ackermann\_contr # Update control to the last one applied by the ackermann

self.\_control = world.player.get\_control()

# Update hud with the newest ackermann control world.hud.update\_ackermann\_control(self.\_ackermann\_control

elif isinstance(self.\_control, carla.WalkerControl): self.\_parse\_walker\_keys(pygame.key.get\_pressed(),

clock.get\_time(), world)

world.player.apply\_control(self.\_control)

def \_parse\_vehicle\_keys(self, keys, milliseconds): if keys[K\_UP] or keys[K\_w]:

if not self.\_ackermann\_enabled:

self.\_control.throttle = min(self.\_control.throttle + 0.01,

1.00)

else:

self.\_ackermann\_control.speed += round(milliseconds \* 0.005,

2) \* self.\_ackermann\_reverse else:

if not self.\_ackermann\_enabled: self.\_control.throttle = 0.0

if keys[K\_DOWN] or keys[K\_s]:

if not self.\_ackermann\_enabled:

self.\_control.brake = min(self.\_control.brake + 0.2, 1) else:

self.\_ackermann\_control.speed -= min(abs(self.\_ackermann\_control.speed), round(milliseconds \* 0.005, 2)) \* self.\_ackermann\_reverse

self.\_ackermann\_control.speed = max(0, abs(self.\_ackermann\_control.speed)) \* self.\_ackermann\_reverse

else:

if not self.\_ackermann\_enabled: self.\_control.brake = 0

steer\_increment = 5e-4 \* milliseconds if keys[K\_LEFT] or keys[K\_a]:

if self.\_steer\_cache > 0: self.\_steer\_cache = 0

else:

self.\_steer\_cache -= steer\_increment elif keys[K\_RIGHT] or keys[K\_d]:

if self.\_steer\_cache < 0: self.\_steer\_cache = 0

else:

self.\_steer\_cache += steer\_increment

else:

self.\_steer\_cache = 0.0

self.\_steer\_cache = min(0.7, max(-0.7, self.\_steer\_cache)) if not self.\_ackermann\_enabled:

self.\_control.steer = round(self.\_steer\_cache, 1) self.\_control.hand\_brake = keys[K\_SPACE]

else:

self.\_ackermann\_control.steer = round(self.\_steer\_cache, 1)

def \_parse\_walker\_keys(self, keys, milliseconds, world): self.\_control.speed = 0.0

if keys[K\_DOWN] or keys[K\_s]: self.\_control.speed = 0.0

if keys[K\_LEFT] or keys[K\_a]: self.\_control.speed = .01 self.\_rotation.yaw -= 0.08 \* milliseconds

if keys[K\_RIGHT] or keys[K\_d]: self.\_control.speed = .01 self.\_rotation.yaw += 0.08 \* milliseconds

if keys[K\_UP] or keys[K\_w]:

self.\_control.speed = world.player\_max\_speed\_fast if pygame.key.get\_mods() & KMOD\_SHIFT else world.player\_max\_speed

self.\_control.jump = keys[K\_SPACE] self.\_rotation.yaw = round(self.\_rotation.yaw, 1)

self.\_control.direction = self.\_rotation.get\_forward\_vector()

@staticmethod

def \_is\_quit\_shortcut(key):

return (key == K\_ESCAPE) or (key == K\_q and pygame.key.get\_mods() & KMOD\_CTRL)

#

==============================================================================

# HUD

-- #

==============================================================================

class HUD(object):

def init (self, width, height): self.dim = (width, height)

font = pygame.font.Font(pygame.font.get\_default\_font(), 20) font\_name = 'courier' if os.name == 'nt' else 'mono'

fonts = [x for x in pygame.font.get\_fonts() if font\_name in x] default\_font = 'ubuntumono'

mono = default\_font if default\_font in fonts else fonts[0] mono = pygame.font.match\_font(mono)

self.\_font\_mono = pygame.font.Font(mono, 12 if os.name == 'nt' else

14)

self.\_notifications = FadingText(font, (width, 40), (0, height - 40)) self.help = HelpText(pygame.font.Font(mono, 16), width, height) self.server\_fps = 0

self.frame = 0

self.simulation\_time = 0 self.\_show\_info = True self.\_info\_text = []

self.\_server\_clock = pygame.time.Clock()

self.\_show\_ackermann\_info = False

self.\_ackermann\_control = carla.VehicleAckermannControl()

def on\_world\_tick(self, timestamp): self.\_server\_clock.tick()

self.server\_fps = self.\_server\_clock.get\_fps() self.frame = timestamp.frame self.simulation\_time = timestamp.elapsed\_seconds

def tick(self, world, clock): self.\_notifications.tick(world, clock) if not self.\_show\_info:

return

t = world.player.get\_transform() v = world.player.get\_velocity() c = world.player.get\_control()

compass = world.imu\_sensor.compass

heading = 'N' if compass > 270.5 or compass < 89.5 else '' heading += 'S' if 90.5 < compass < 269.5 else ''

heading += 'E' if 0.5 < compass < 179.5 else '' heading += 'W' if 180.5 < compass < 359.5 else ''

colhist = world.collision\_sensor.get\_collision\_history()

collision = [colhist[x + self.frame - 200] for x in range(0, 200)] max\_col = max(1.0, max(collision))

collision = [x / max\_col for x in collision]

vehicles = world.world.get\_actors().filter('vehicle.\*') self.\_info\_text = [

'Server: % 16.0f FPS' % self.server\_fps, 'Client: % 16.0f FPS' % clock.get\_fps(), '',

'Vehicle: % 20s' % get\_actor\_display\_name(world.player, truncate=20),

'Map: % 20s' % world.map.name.split('/')[-1], 'Simulation time: % 12s' %

datetime.timedelta(seconds=int(self.simulation\_time)), '',

'Speed: % 15.0f km/h' % (3.6 \* math.sqrt(v.x\*\*2 + v.y\*\*2 +

v.z\*\*2)),

u'Compass:% 17.0f\N{DEGREE SIGN} % 2s' % (compass, heading), 'Accelero: (%5.1f,%5.1f,%5.1f)' %

(world.imu\_sensor.accelerometer),

'Gyroscop: (%5.1f,%5.1f,%5.1f)' % (world.imu\_sensor.gyroscope), 'Location:% 20s' % ('(% 5.1f, % 5.1f)' % (t.location.x,

t.location.y)),

'GNSS:% 24s' % ('(% 2.6f, % 3.6f)' % (world.gnss\_sensor.lat, world.gnss\_sensor.lon)),

'Height: % 18.0f m' % t.location.z, '']

if isinstance(c, carla.VehicleControl): self.\_info\_text += [

('Throttle:', c.throttle, 0.0, 1.0),

('Steer:', c.steer, -1.0, 1.0),

('Brake:', c.brake, 0.0, 1.0), ('Reverse:', c.reverse), ('Hand brake:', c.hand\_brake),

('Manual:', c.manual\_gear\_shift),

'Gear: %s' % {-1: 'R', 0: 'N'}.get(c.gear, c.gear)] if self.\_show\_ackermann\_info:

self.\_info\_text += [ '',

'Ackermann Controller:',

' Target speed: % 8.0f km/h' % (3.6\*self.\_ackermann\_control.speed),

]

elif isinstance(c, carla.WalkerControl): self.\_info\_text += [

('Speed:', c.speed, 0.0, 5.556), ('Jump:', c.jump)]

self.\_info\_text += [ '',

'Collision:', collision, '',

'Number of vehicles: % 8d' % len(vehicles)] if len(vehicles) > 1:

self.\_info\_text += ['Nearby vehicles:']

distance = lambda l: math.sqrt((l.x - t.location.x)\*\*2 + (l.y - t.location.y)\*\*2 + (l.z - t.location.z)\*\*2)

vehicles = [(distance(x.get\_location()), x) for x in vehicles if x.id != world.player.id]

for d, vehicle in sorted(vehicles, key=lambda vehicles: vehicles[0]):

if d > 200.0: break

vehicle\_type = get\_actor\_display\_name(vehicle, truncate=22) self.\_info\_text.append('% 4dm %s' % (d, vehicle\_type))

def show\_ackermann\_info(self, enabled): self.\_show\_ackermann\_info = enabled

def update\_ackermann\_control(self, ackermann\_control): self.\_ackermann\_control = ackermann\_control

def toggle\_info(self):

self.\_show\_info = not self.\_show\_info

def notification(self, text, seconds=2.0): self.\_notifications.set\_text(text, seconds=seconds)

def error(self, text):

self.\_notifications.set\_text('Error: %s' % text, (255, 0, 0))

def render(self, display): if self.\_show\_info:

info\_surface = pygame.Surface((220, self.dim[1])) info\_surface.set\_alpha(100) display.blit(info\_surface, (0, 0))

v\_offset = 4

bar\_h\_offset = 100

bar\_width = 106

for item in self.\_info\_text:

if v\_offset + 18 > self.dim[1]: break

if isinstance(item, list): if len(item) > 1:

points = [(x + 8, v\_offset + 8 + (1.0 - y) \* 30) for

x, y in enumerate(item)]

points, 2)

pygame.draw.lines(display, (255, 136, 0), False,

item = None v\_offset += 18

elif isinstance(item, tuple):

if isinstance(item[1], bool):

6))

item[1] else 1)

8), (bar\_width, 6))

rect\_border, 1)

rect = pygame.Rect((bar\_h\_offset, v\_offset + 8), (6, pygame.draw.rect(display, (255, 255, 255), rect, 0 if

else:

rect\_border = pygame.Rect((bar\_h\_offset, v\_offset + pygame.draw.rect(display, (255, 255, 255),

f = (item[1] - item[2]) / (item[3] - item[2]) if item[2] < 0.0:

rect = pygame.Rect((bar\_h\_offset + f \* (bar\_width

- 6), v\_offset + 8), (6, 6))

else:

rect = pygame.Rect((bar\_h\_offset, v\_offset + 8),

(f \* bar\_width, 6))

pygame.draw.rect(display, (255, 255, 255), rect) item = item[0]

255))

if item: # At this point has to be a str.

surface = self.\_font\_mono.render(item, True, (255, 255,

display.blit(surface, (8, v\_offset))

v\_offset += 18 self.\_notifications.render(display) self.help.render(display)

#

==============================================================================

# FadingText

-- #

==============================================================================

class FadingText(object):

def init (self, font, dim, pos): self.font = font

self.dim = dim self.pos = pos self.seconds\_left = 0

self.surface = pygame.Surface(self.dim)

def set\_text(self, text, color=(255, 255, 255), seconds=2.0): text\_texture = self.font.render(text, True, color) self.surface = pygame.Surface(self.dim)

self.seconds\_left = seconds self.surface.fill((0, 0, 0, 0))

self.surface.blit(text\_texture, (10, 11))

def tick(self, \_, clock):

delta\_seconds = 1e-3 \* clock.get\_time()

self.seconds\_left = max(0.0, self.seconds\_left - delta\_seconds) self.surface.set\_alpha(500.0 \* self.seconds\_left)

def render(self, display): display.blit(self.surface, self.pos)

#

==============================================================================

# HelpText

-- #

==============================================================================

class HelpText(object):

"""Helper class to handle text output using pygame""" def init (self, font, width, height):

lines = doc .split('\n') self.font = font self.line\_space = 18

self.dim = (780, len(lines) \* self.line\_space + 12)

self.pos = (0.5 \* width - 0.5 \* self.dim[0], 0.5 \* height - 0.5 \* self.dim[1])

self.seconds\_left = 0

self.surface = pygame.Surface(self.dim) self.surface.fill((0, 0, 0, 0))

for n, line in enumerate(lines):

text\_texture = self.font.render(line, True, (255, 255, 255)) self.surface.blit(text\_texture, (22, n \* self.line\_space)) self.\_render = False

self.surface.set\_alpha(220)

def toggle(self):

self.\_render = not self.\_render

def render(self, display): if self.\_render:

display.blit(self.surface, self.pos)

#

==============================================================================

# CollisionSensor

--

#

==============================================================================

class CollisionSensor(object):

def init (self, parent\_actor, hud): self.sensor = None

self.history = [] self.\_parent = parent\_actor self.hud = hud

world = self.\_parent.get\_world()

bp = world.get\_blueprint\_library().find('sensor.other.collision') self.sensor = world.spawn\_actor(bp, carla.Transform(),

attach\_to=self.\_parent)

# We need to pass the lambda a weak reference to self to avoid circular

# reference.

weak\_self = weakref.ref(self) self.sensor.listen(lambda event:

CollisionSensor.\_on\_collision(weak\_self, event))

def get\_collision\_history(self):

history = collections.defaultdict(int) for frame, intensity in self.history:

history[frame] += intensity return history

@staticmethod

def \_on\_collision(weak\_self, event): self = weak\_self()

if not self:

return

actor\_type = get\_actor\_display\_name(event.other\_actor) self.hud.notification('Collision with %r' % actor\_type) impulse = event.normal\_impulse

intensity = math.sqrt(impulse.x\*\*2 + impulse.y\*\*2 + impulse.z\*\*2) self.history.append((event.frame, intensity))

if len(self.history) > 4000: self.history.pop(0)

#

==============================================================================

# LaneInvasionSensor

-- #

==============================================================================

class LaneInvasionSensor(object):

def init (self, parent\_actor, hud): self.sensor = None

# If the spawn object is not a vehicle, we cannot use the Lane Invasion Sensor

if parent\_actor.type\_id.startswith("vehicle."): self.\_parent = parent\_actor

self.hud = hud

world = self.\_parent.get\_world() bp =

world.get\_blueprint\_library().find('sensor.other.lane\_invasion') self.sensor = world.spawn\_actor(bp, carla.Transform(),

attach\_to=self.\_parent)

# We need to pass the lambda a weak reference to self to avoid

circular

# reference.

weak\_self = weakref.ref(self) self.sensor.listen(lambda event:

LaneInvasionSensor.\_on\_invasion(weak\_self, event))

@staticmethod

def \_on\_invasion(weak\_self, event): self = weak\_self()

if not self:

return

lane\_types = set(x.type for x in event.crossed\_lane\_markings) text = ['%r' % str(x).split()[-1] for x in lane\_types] self.hud.notification('Crossed line %s' % ' and '.join(text))

#

==============================================================================

# GnssSensor

-- #

==============================================================================

class GnssSensor(object):

def init (self, parent\_actor): self.sensor = None self.\_parent = parent\_actor self.lat = 0.0

self.lon = 0.0

world = self.\_parent.get\_world()

bp = world.get\_blueprint\_library().find('sensor.other.gnss') self.sensor = world.spawn\_actor(bp,

carla.Transform(carla.Location(x=1.0, z=2.8)), attach\_to=self.\_parent)

# We need to pass the lambda a weak reference to self to avoid circular

# reference.

weak\_self = weakref.ref(self)

self.sensor.listen(lambda event: GnssSensor.\_on\_gnss\_event(weak\_self,

event))

@staticmethod

def \_on\_gnss\_event(weak\_self, event): self = weak\_self()

if not self:

return

self.lat = event.latitude self.lon = event.longitude

#

==============================================================================

# IMUSensor

-- #

==============================================================================

class IMUSensor(object):

def init (self, parent\_actor): self.sensor = None self.\_parent = parent\_actor

self.accelerometer = (0.0, 0.0, 0.0)

self.gyroscope = (0.0, 0.0, 0.0)

self.compass = 0.0

world = self.\_parent.get\_world()

bp = world.get\_blueprint\_library().find('sensor.other.imu') self.sensor = world.spawn\_actor(

bp, carla.Transform(), attach\_to=self.\_parent)

# We need to pass the lambda a weak reference to self to avoid circular

# reference.

weak\_self = weakref.ref(self) self.sensor.listen(

lambda sensor\_data: IMUSensor.\_IMU\_callback(weak\_self, sensor\_data))

@staticmethod

def \_IMU\_callback(weak\_self, sensor\_data): self = weak\_self()

if not self:

return

limits = (-99.9, 99.9)

self.accelerometer = (

max(limits[0], min(limits[1], sensor\_data.accelerometer.x)), max(limits[0], min(limits[1], sensor\_data.accelerometer.y)), max(limits[0], min(limits[1], sensor\_data.accelerometer.z)))

self.gyroscope = ( max(limits[0], min(limits[1],

math.degrees(sensor\_data.gyroscope.x))), max(limits[0], min(limits[1],

math.degrees(sensor\_data.gyroscope.y))), max(limits[0], min(limits[1],

math.degrees(sensor\_data.gyroscope.z))))

self.compass = math.degrees(sensor\_data.compass)

#

==============================================================================

# RadarSensor

-- #

==============================================================================

class RadarSensor(object):

def init (self, parent\_actor): self.sensor = None self.\_parent = parent\_actor

bound\_x = 0.5 + self.\_parent.bounding\_box.extent.x bound\_y = 0.5 + self.\_parent.bounding\_box.extent.y bound\_z = 0.5 + self.\_parent.bounding\_box.extent.z

self.velocity\_range = 7.5 # m/s world = self.\_parent.get\_world() self.debug = world.debug

bp = world.get\_blueprint\_library().find('sensor.other.radar') bp.set\_attribute('horizontal\_fov', str(35)) bp.set\_attribute('vertical\_fov', str(20))

self.sensor = world.spawn\_actor( bp,

carla.Transform(

carla.Location(x=bound\_x + 0.05, z=bound\_z+0.05), carla.Rotation(pitch=5)),

attach\_to=self.\_parent)

# We need a weak reference to self to avoid circular reference. weak\_self = weakref.ref(self)

self.sensor.listen(

lambda radar\_data: RadarSensor.\_Radar\_callback(weak\_self, radar\_data))

@staticmethod

def \_Radar\_callback(weak\_self, radar\_data): self = weak\_self()

if not self:

return

# To get a numpy [[vel, altitude, azimuth, depth],...[,,,]]:

# points = np.frombuffer(radar\_data.raw\_data, dtype=np.dtype('f4')) # points = np.reshape(points, (len(radar\_data), 4))

current\_rot = radar\_data.transform.rotation for detect in radar\_data:

azi = math.degrees(detect.azimuth) alt = math.degrees(detect.altitude)

# The 0.25 adjusts a bit the distance so the dots can # be properly seen

fw\_vec = carla.Vector3D(x=detect.depth - 0.25) carla.Transform(

carla.Location(), carla.Rotation(

pitch=current\_rot.pitch + alt, yaw=current\_rot.yaw + azi, roll=current\_rot.roll)).transform(fw\_vec)

def clamp(min\_v, max\_v, value):

return max(min\_v, min(value, max\_v))

norm\_velocity = detect.velocity / self.velocity\_range # range [-1,

1]

r = int(clamp(0.0, 1.0, 1.0 - norm\_velocity) \* 255.0)

g = int(clamp(0.0, 1.0, 1.0 - abs(norm\_velocity)) \* 255.0)

b = int(abs(clamp(- 1.0, 0.0, - 1.0 - norm\_velocity)) \* 255.0) self.debug.draw\_point(

radar\_data.transform.location + fw\_vec, size=0.075,

life\_time=0.06, persistent\_lines=False, color=carla.Color(r, g, b))

#

==============================================================================

# CameraManager

-- #

==============================================================================

class CameraManager(object):

def init (self, parent\_actor, hud, gamma\_correction): self.sensor = None

self.surface = None

self.\_parent = parent\_actor self.hud = hud self.recording = False

bound\_x = 0.5 + self.\_parent.bounding\_box.extent.x bound\_y = 0.5 + self.\_parent.bounding\_box.extent.y bound\_z = 0.5 + self.\_parent.bounding\_box.extent.z Attachment = carla.AttachmentType

if not self.\_parent.type\_id.startswith("walker.pedestrian"): self.\_camera\_transforms = [

(carla.Transform(carla.Location(x=-2.0\*bound\_x, y=+0.0\*bound\_y, z=2.0\*bound\_z), carla.Rotation(pitch=8.0)), Attachment.SpringArmGhost),

(carla.Transform(carla.Location(x=+0.8\*bound\_x, y=+0.0\*bound\_y, z=1.3\*bound\_z)), Attachment.Rigid),

(carla.Transform(carla.Location(x=+1.9\*bound\_x, y=+1.0\*bound\_y, z=1.2\*bound\_z)), Attachment.SpringArmGhost),

(carla.Transform(carla.Location(x=-2.8\*bound\_x, y=+0.0\*bound\_y, z=4.6\*bound\_z), carla.Rotation(pitch=6.0)), Attachment.SpringArmGhost),

(carla.Transform(carla.Location(x=-1.0, y=-1.0\*bound\_y, z=0.4\*bound\_z)), Attachment.Rigid)]

else:

self.\_camera\_transforms = [ (carla.Transform(carla.Location(x=-2.5, z=0.0),

carla.Rotation(pitch=-8.0)), Attachment.SpringArmGhost), (carla.Transform(carla.Location(x=1.6, z=1.7)),

Attachment.Rigid),

(carla.Transform(carla.Location(x=2.5, y=0.5, z=0.0), carla.Rotation(pitch=-8.0)), Attachment.SpringArmGhost),

(carla.Transform(carla.Location(x=-4.0, z=2.0), carla.Rotation(pitch=6.0)), Attachment.SpringArmGhost),

(carla.Transform(carla.Location(x=0, y=-2.5, z=-0.0), carla.Rotation(yaw=90.0)), Attachment.Rigid)]

{}],

self.transform\_index = 1 self.sensors = [

['sensor.camera.rgb', cc.Raw, 'Camera RGB', {}],

['sensor.camera.depth', cc.Raw, 'Camera Depth (Raw)', {}], ['sensor.camera.depth', cc.Depth, 'Camera Depth (Gray Scale)',

['sensor.camera.depth', cc.LogarithmicDepth, 'Camera Depth

(Logarithmic Gray Scale)', {}],

['sensor.camera.semantic\_segmentation', cc.Raw, 'Camera Semantic Segmentation (Raw)', {}],

['sensor.camera.semantic\_segmentation', cc.CityScapesPalette, 'Camera Semantic Segmentation (CityScapes Palette)', {}],

['sensor.camera.instance\_segmentation', cc.CityScapesPalette, 'Camera Instance Segmentation (CityScapes Palette)', {}],

['sensor.camera.instance\_segmentation', cc.Raw, 'Camera Instance Segmentation (Raw)', {}],

['sensor.lidar.ray\_cast', None, 'Lidar (Ray-Cast)', {'range':

'50'}],

]

['sensor.camera.dvs', cc.Raw, 'Dynamic Vision Sensor', {}], ['sensor.camera.rgb', cc.Raw, 'Camera RGB Distorted',

{'lens\_circle\_multiplier': '3.0',

'lens\_circle\_falloff': '3.0',

'chromatic\_aberration\_intensity': '0.5',

'chromatic\_aberration\_offset': '0'}], ['sensor.camera.optical\_flow', cc.Raw, 'Optical Flow', {}], ['sensor.camera.normals', cc.Raw, 'Camera Normals', {}],

world = self.\_parent.get\_world() bp\_library = world.get\_blueprint\_library() for item in self.sensors:

bp = bp\_library.find(item[0])

if item[0].startswith('sensor.camera'): bp.set\_attribute('image\_size\_x', str(hud.dim[0])) bp.set\_attribute('image\_size\_y', str(hud.dim[1])) if bp.has\_attribute('gamma'):

bp.set\_attribute('gamma', str(gamma\_correction)) for attr\_name, attr\_value in item[3].items():

bp.set\_attribute(attr\_name, attr\_value) elif item[0].startswith('sensor.lidar'):

self.lidar\_range = 50

for attr\_name, attr\_value in item[3].items(): bp.set\_attribute(attr\_name, attr\_value) if attr\_name == 'range':

self.lidar\_range = float(attr\_value)

item.append(bp) self.index = None

def toggle\_camera(self):

self.transform\_index = (self.transform\_index + 1) % len(self.\_camera\_transforms)

self.set\_sensor(self.index, notify=False, force\_respawn=True)

def set\_sensor(self, index, notify=True, force\_respawn=False): index = index % len(self.sensors)

needs\_respawn = True if self.index is None else \ (force\_respawn or (self.sensors[index][2] !=

self.sensors[self.index][2])) if needs\_respawn:

1])

if self.sensor is not None: self.sensor.destroy() self.surface = None

self.sensor = self.\_parent.get\_world().spawn\_actor( self.sensors[index][-1], self.\_camera\_transforms[self.transform\_index][0], attach\_to=self.\_parent, attachment\_type=self.\_camera\_transforms[self.transform\_index][

# We need to pass the lambda a weak reference to self to avoid # circular reference.

weak\_self = weakref.ref(self) self.sensor.listen(lambda image:

CameraManager.\_parse\_image(weak\_self, image)) if notify:

self.hud.notification(self.sensors[index][2]) self.index = index

def next\_sensor(self): self.set\_sensor(self.index + 1)

def toggle\_recording(self): self.recording = not self.recording

self.hud.notification('Recording %s' % ('On' if self.recording else

'Off'))

def render(self, display):

if self.surface is not None: display.blit(self.surface, (0, 0))

@staticmethod

def \_parse\_image(weak\_self, image): self = weak\_self()

if not self:

return

if self.sensors[self.index][0].startswith('sensor.lidar'): points = np.frombuffer(image.raw\_data, dtype=np.dtype('f4')) points = np.reshape(points, (int(points.shape[0] / 4), 4)) lidar\_data = np.array(points[:, :2])

lidar\_data \*= min(self.hud.dim) / (2.0 \* self.lidar\_range) lidar\_data += (0.5 \* self.hud.dim[0], 0.5 \* self.hud.dim[1]) lidar\_data = np.fabs(lidar\_data) # pylint: disable=E1111 lidar\_data = lidar\_data.astype(np.int32)

lidar\_data = np.reshape(lidar\_data, (-1, 2)) lidar\_img\_size = (self.hud.dim[0], self.hud.dim[1], 3) lidar\_img = np.zeros((lidar\_img\_size), dtype=np.uint8) lidar\_img[tuple(lidar\_data.T)] = (255, 255, 255) self.surface = pygame.surfarray.make\_surface(lidar\_img)

elif self.sensors[self.index][0].startswith('sensor.camera.dvs'): # Example of converting the raw\_data from a carla.DVSEventArray # sensor into a NumPy array and using it as an image dvs\_events = np.frombuffer(image.raw\_data, dtype=np.dtype([

('x', np.uint16), ('y', np.uint16), ('t', np.int64), ('pol',

np.bool)]))

dvs\_img = np.zeros((image.height, image.width, 3), dtype=np.uint8) # Blue is positive, red is negative

dvs\_img[dvs\_events[:]['y'], dvs\_events[:]['x'],

dvs\_events[:]['pol'] \* 2] = 255

self.surface = pygame.surfarray.make\_surface(dvs\_img.swapaxes(0,

1))

elif

self.sensors[self.index][0].startswith('sensor.camera.optical\_flow'): image = image.get\_color\_coded\_flow()

array = np.frombuffer(image.raw\_data, dtype=np.dtype("uint8")) array = np.reshape(array, (image.height, image.width, 4)) array = array[:, :, :3]

array = array[:, :, ::-1]

self.surface = pygame.surfarray.make\_surface(array.swapaxes(0, 1)) else:

image.convert(self.sensors[self.index][1])

array = np.frombuffer(image.raw\_data, dtype=np.dtype("uint8")) array = np.reshape(array, (image.height, image.width, 4)) array = array[:, :, :3]

array = array[:, :, ::-1]

self.surface = pygame.surfarray.make\_surface(array.swapaxes(0, 1)) if self.recording:

image.save\_to\_disk('\_out/%08d' % image.frame)

#

==============================================================================

# game\_loop()

-- #

==============================================================================

def game\_loop(args): pygame.init() pygame.font.init() world = None original\_settings = None

try:

client = carla.Client(args.host, args.port) client.set\_timeout(2000.0)

sim\_world = client.get\_world() if args.sync:

original\_settings = sim\_world.get\_settings() settings = sim\_world.get\_settings()

if not settings.synchronous\_mode: settings.synchronous\_mode = True settings.fixed\_delta\_seconds = 0.05

sim\_world.apply\_settings(settings)

traffic\_manager = client.get\_trafficmanager() traffic\_manager.set\_synchronous\_mode(True)

if args.autopilot and not sim\_world.get\_settings().synchronous\_mode: print("WARNING: You are currently in asynchronous mode and could "

"experience some issues with the traffic simulation")

display = pygame.display.set\_mode( (args.width, args.height), pygame.HWSURFACE | pygame.DOUBLEBUF)

display.fill((0,0,0)) pygame.display.flip()

hud = HUD(args.width, args.height) world = World(sim\_world, hud, args)

controller = KeyboardControl(world, args.autopilot)

if args.sync:

sim\_world.tick() else:

sim\_world.wait\_for\_tick()

clock = pygame.time.Clock() while True:

if args.sync:

sim\_world.tick() clock.tick\_busy\_loop(60)

if controller.parse\_events(client, world, clock, args.sync): return

world.tick(clock) world.render(display) pygame.display.flip()

finally:

if original\_settings: sim\_world.apply\_settings(original\_settings)

if (world and world.recording\_enabled):

client.stop\_recorder()

if world is not None: world.destroy()

pygame.quit()

#

==============================================================================

# main()

-- #

==============================================================================

def main():

argparser = argparse.ArgumentParser( description='CARLA Manual Control Client')

argparser.add\_argument( '-v', '--verbose', action='store\_true', dest='debug',

help='print debug information') argparser.add\_argument(

'--host', metavar='H', default='127.0.0.1',

help='IP of the host server (default: 127.0.0.1)') argparser.add\_argument(

'-p', '--port', metavar='P', default=2000, type=int,

help='TCP port to listen to (default: 2000)') argparser.add\_argument(

'-a', '--autopilot', action='store\_true', help='enable autopilot')

argparser.add\_argument( '--res',

metavar='WIDTHxHEIGHT', default='1280x720',

help='window resolution (default: 1280x720)') argparser.add\_argument(

'--filter', metavar='PATTERN', default='vehicle.\*',

help='actor filter (default: "vehicle.\*")')

argparser.add\_argument( '--generation', metavar='G', default='2',

help='restrict to certain actor generation (values: "1","2","All" - default: "2")')

argparser.add\_argument( '--rolename', metavar='NAME', default='hero',

help='actor role name (default: "hero")') argparser.add\_argument(

'--gamma', default=2.2, type=float,

help='Gamma correction of the camera (default: 2.2)') argparser.add\_argument(

'--sync', action='store\_true',

help='Activate synchronous mode execution') args = argparser.parse\_args()

args.width, args.height = [int(x) for x in args.res.split('x')]

log\_level = logging.DEBUG if args.debug else logging.INFO logging.basicConfig(format='%(levelname)s: %(message)s', level=log\_level)

logging.info('listening to server %s:%s', args.host, args.port)

print( doc )

try:

game\_loop(args)

except KeyboardInterrupt: print('\nCancelled by user. Bye!')

if name == ' main ':

main()

# OUTPUT:

Default Spectator View:



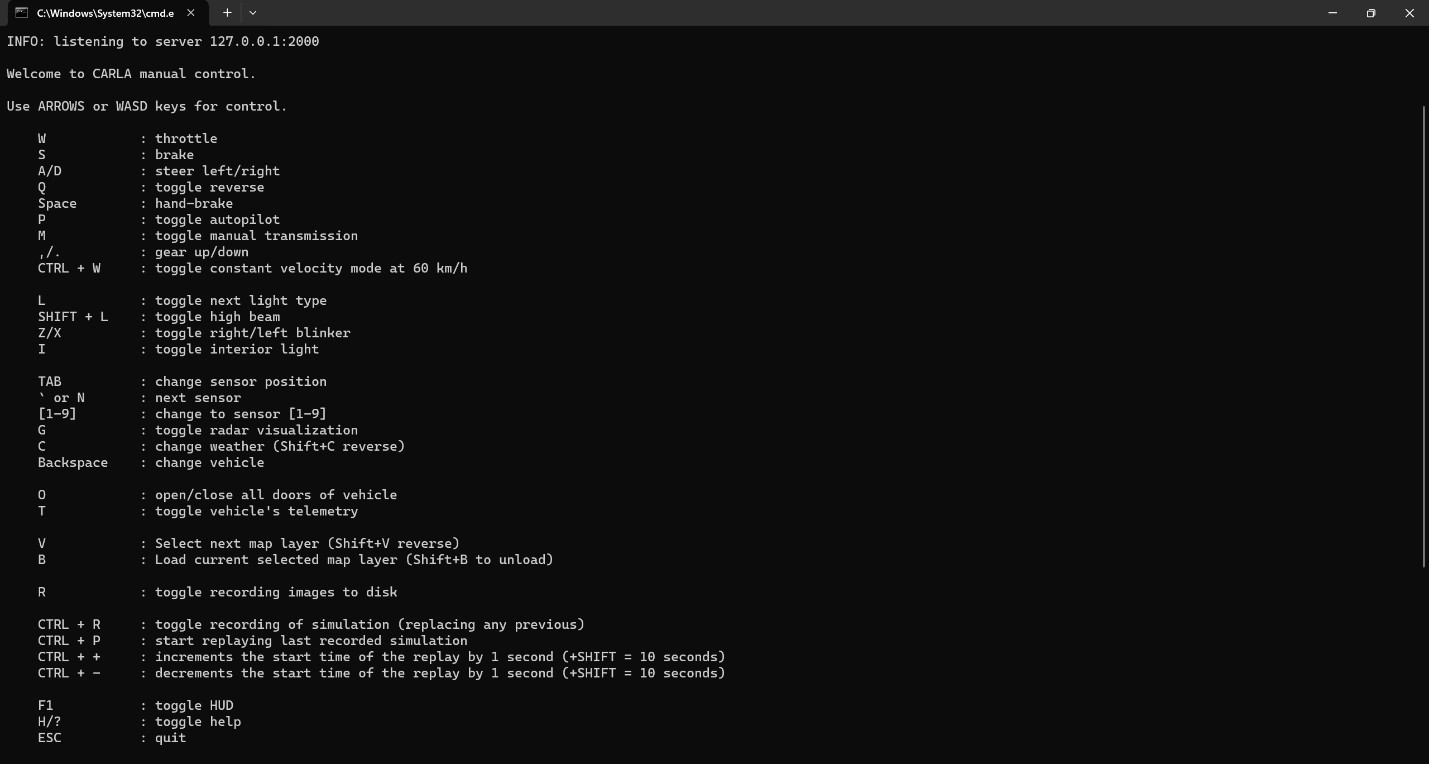
Spectator view after importing NPC in the simulator:



Manual Control by the User:



Instruction for the manual controls:



# Result:

Therefore, we've successfully implemented carla.