|  |  |
| --- | --- |
| **Ex. No: 1-13** | SEARCH ALGORITHM VISUALIZATION |
| **14.07.2023** |

**Aim:**

To visualize different search algorithm and compare their performances.

**Program:**

"""

Author : Mukesh T P

"""

import time

import networkx as nx

import matplotlib.pyplot as plt

from matplotlib.animation import FuncAnimation

class Graph:

"""

Graph class for initializing and managing a graph.

Attributes:

graph: Dictionary where keys represent nodes, and values are lists of nodes connected to the key node.

weight: Dictionary where keys represent nodes, and values are lists of weights corresponding to edges connected to the key node.

heuristic: Dictionary where keys represent nodes, and values are heuristic values from the source to the goal.

"""

def \_\_init\_\_(self):

"""

Initializes the graph, weight, and heuristic dictionaries.

"""

self.graph = {}

self.weight = {}

self.heuristic = {}

def addEdge(self, o, d, w = 1):

"""

Adds an edge between two points in the graph.

Parameters:

o: Origin/start/current node.

d: Destination node.

w: Weight of the edge (default = 1).

"""

if o not in self.graph:

self.graph[o] = []

self.weight[o] = []

self.heuristic[o] = 100

if d not in self.graph:

self.graph[d] = []

self.weight[d] = []

self.heuristic[d] = 100

self.graph[o].append(d)

self.weight[o].append(w)

combined = sorted(zip(self.graph[o], self.weight[o]), key=lambda x: x[0])

self.graph[o], self.weight[o] = map(list, zip(\*combined))

self.graph[d].append(o)

self.weight[d].append(w)

combined = sorted(zip(self.graph[d], self.weight[d]), key=lambda x: x[0])

self.graph[d], self.weight[d] = map(list, zip(\*combined))

def addEdgeD(self, o, d, w = 1):

"""

Adds a directed edge between two points in the graph.

From o to d with weight w

Parameters:

o: Origin/start/current node.

d: Destination node.

w: Weight of the edge (default = 1).

"""

if o not in self.graph:

self.graph[o] = []

self.weight[o] = []

self.heuristic[o] = 100

self.graph[o].append(d)

self.weight[o].append(w)

combined = sorted(zip(self.graph[o], self.weight[o]), key=lambda x: x[0])

self.graph[o], self.weight[o] = map(list, zip(\*combined))

def addHeuristics(self, o, h):

"""

Adds heuristic value to the point mentioned.

Parameters:

o: Origin/start/current node.

h: Heuristic value (default value = 100).

"""

self.heuristic[o] = h

def \_\_str\_\_(self):

"""

Prints the graph, weight and hueristic

"""

return f"{self.graph}\n{self.weight}\n{self.heuristic}"

class Algorithm:

"""

This class contains searching techniques that can be used on a Graph.

Parameters:

g : graph

o : origin

d : destination

w : weight (default value = 1)

h : heuristics (default value = 100)

"""

def DFS(self, g, o, d):

"""

This implements Depth First Search on a given graph.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

visited = set()

stack = [(o, [o])] # Use a stack to store both the node and its path

total\_path = []

while stack:

node, path = stack.pop()

total\_path.append(path)

if node == d:

print(path)

return total\_path

if node not in visited:

visited.add(node)

for neighbor in sorted(g.graph[node], reverse=True):

if neighbor not in visited:

stack.append((neighbor, path + [neighbor]))

return None

def BFS(self, g, o, d):

"""

This implements Breadth First Search on a given graph.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

visited = set()

queue = [(o, [o])] # Use a queue to store both the node and its path

total\_path = []

while queue:

node, path = queue.pop(0)

total\_path.append(path)

if node == d:

print(path)

return total\_path

if node not in visited:

visited.add(node)

for neighbor in g.graph[node]:

if neighbor not in visited:

queue.append((neighbor, path + [neighbor]))

return None

def BMS(self, g, o, d):

"""

This implements British Museum Search on a given graph.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

paths = []

all\_paths = []

stack = [(o, [o])]

while stack:

node, path = stack.pop()

paths.append(path)

if node == d:

all\_paths.append(path)

for neighbor in g.graph[node]:

if neighbor not in path:

stack.append((neighbor, path + [neighbor]))

print(all\_paths)

return paths

def HC(self, g, o, d):

"""

This implements Hill Climbing on a given graph.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

path = []

total\_path = []

visited = set()

node = o

while node != d:

path.append(node)

visited.add(node)

neighbors = g.graph[node]

neighbor\_heuristics = [g.heuristic[neighbor] for neighbor in neighbors]

best\_neighbor = neighbors[neighbor\_heuristics.index(min(neighbor\_heuristics))]

if best\_neighbor in visited:

return total\_path

node = best\_neighbor

total\_path.append(list(path[:]))

path.append(d)

total\_path.append(list(path[:]))

print(path)

return total\_path

def BS(self, g, o, d, bw=1):

"""

This implements Beam Search on a given graph.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

bw : determines the beam width required (default value = 1)

"""

beam = [(g.heuristic[o], (o, [o]))]

total\_path = []

while beam:

beam.sort(key=lambda x: x[0])

best\_paths = beam[:bw]

beam = []

for misc, (node, path) in best\_paths:

total\_path.append(path)

if node == d:

print(path)

return total\_path

for neighbor in g.graph[node]:

if neighbor not in path:

heuristic\_score = g.heuristic[neighbor]

new\_path = path + [neighbor]

beam.append((heuristic\_score, (neighbor, new\_path)))

return None

def Oracle(self, g, o, d):

"""

Oracle search performing an exhaustive search to find all possible paths.

Returns a list of tuples, each containing a path and its cost.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

all\_paths = []

total\_path = []

stack = [(o, [], 0)] # (node, path, cost)

while stack:

current, path, cost = stack.pop()

total\_path.append(path+[current])

if current == d:

all\_paths.append((path + [current], cost))

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if neighbor not in path:

stack.append((neighbor, path + [current], cost + weight))

print(all\_paths)

return total\_path

def OracleH(self, g, o, d):

all\_paths = []

total\_path = []

stack = [(o, [], 0)] # (node, path, cost)

while stack:

current, path, cost = stack.pop()

total\_path.append(path+[current])

if current == d:

all\_paths.append((path + [current], cost))

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if neighbor not in path:

stack.append((neighbor, path + [current], cost + weight + g.heuristic[neighbor]))

print(all\_paths)

return total\_path

def BB(self, g, o, d):

"""

Branch and Bound algorithm to find the optimal path.

Returns the optimal path and its cost.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

best\_path = None

best\_cost = float('inf') # Initialize with positive infinity

# Priority queue implemented as a list of tuples (cost, node, path)

priority\_queue = [(0, o, [])]

total\_path = []

while priority\_queue:

# Find the path with the lowest cost in the priority queue

min\_index = 0

for i in range(1, len(priority\_queue)):

if priority\_queue[i][0] < priority\_queue[min\_index][0]:

min\_index = i

cost, current, path = priority\_queue.pop(min\_index)

total\_path.append(path+[current])

if current == d:

if cost < best\_cost:

best\_path = path + [current]

best\_cost = cost

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if neighbor not in path:

if cost+weight<=best\_cost:

# Add the neighbor to the priority queue with updated cost

priority\_queue.append((cost + weight, neighbor, path + [current]))

print(best\_path, best\_cost)

return total\_path

def EL(self, g, o, d):

"""

Branch and Bound algorithm with an extended list.

Returns the optimal path and its cost.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

best\_path = None

best\_cost = float('inf') # Initialize with positive infinity

# Priority queue implemented as a list of tuples (cost, node, path)

priority\_queue = [(0, o, [])]

total\_path = []

# Extended list to keep track of visited nodes

extended\_list = {node: False for node in g.graph}

while priority\_queue:

# Find the path with the lowest cost in the priority queue

min\_index = 0

for i in range(1, len(priority\_queue)):

if priority\_queue[i][0] < priority\_queue[min\_index][0]:

min\_index = i

cost, current, path = priority\_queue.pop(min\_index)

total\_path.append(path+[current])

if current == d:

if cost < best\_cost:

best\_path = path + [current]

best\_cost = cost

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if not extended\_list[current] and not extended\_list[neighbor]:

if cost+weight<=best\_cost:

# Add the neighbor to the priority queue with updated cost

priority\_queue.append((cost + weight, neighbor, path + [current]))

extended\_list[current] = True

print(best\_path, best\_cost)

return total\_path

def EH(self, g, o, d):

"""

Branch and Bound algorithm with estimated heuristics.

Returns the optimal path and its cost.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

best\_path = None

best\_cost = float('inf') # Initialize with positive infinity

# Priority queue implemented as a list of tuples (cost, node, path)

priority\_queue = [(0, o, [])]

total\_path = []

while priority\_queue:

# Find the path with the lowest cost in the priority queue

min\_index = 0

for i in range(1, len(priority\_queue)):

if priority\_queue[i][0] + g.heuristic[priority\_queue[i][1]] < priority\_queue[min\_index][0] + g.heuristic[priority\_queue[min\_index][1]]:

min\_index = i

cost, current, path = priority\_queue.pop(min\_index)

total\_path.append(path+[current])

if current == d:

if cost < best\_cost:

best\_path = path + [current]

best\_cost = cost

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if neighbor not in path:

if cost+weight+g.heuristic[current]<=best\_cost:

# Add the neighbor to the priority queue with updated cost

priority\_queue.append((cost + weight, neighbor, path + [current]))

print(best\_path, best\_cost)

return total\_path

def Astar(self, g, o, d):

"""

Branch and Bound algorithm with extended list and estimated heuristics.

Returns the optimal path and its cost.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

best\_path = None

best\_cost = float('inf') # Initialize with positive infinity

# Priority queue implemented as a list of tuples (cost, node, path)

priority\_queue = [(0, o, [])]

total\_path = []

# Extended list to keep track of visited nodes

extended\_list = {node: False for node in g.graph}

while priority\_queue:

# Find the path with the lowest cost in the priority queue

min\_index = 0

for i in range(1, len(priority\_queue)):

if priority\_queue[i][0] + g.heuristic[priority\_queue[i][1]] < priority\_queue[min\_index][0] + g.heuristic[priority\_queue[min\_index][1]]:

min\_index = i

cost, current, path = priority\_queue.pop(min\_index)

# Use the extended list to track visited nodes for the current path

visited = set(path)

total\_path.append(path+[current])

if current == d:

if cost < best\_cost:

best\_path = path + [current]

best\_cost = cost

else:

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if not extended\_list[current] and not extended\_list[neighbor] and neighbor not in visited:

if cost+weight+g.heuristic[current]<=best\_cost:

# Add the neighbor to the priority queue with updated cost

priority\_queue.append((cost + weight, neighbor, path + [current]))

# Mark the current node as visited in the global set

extended\_list[current] = True

print(best\_path, best\_cost)

return total\_path

def BestFirstSearch(self, g, o, d):

"""

Best-First Search algorithm.

Returns the optimal path.

Parameters:

g : is the object of class Graph

o : origin/start/current node

d : destination node

"""

best\_path = None

# Priority queue implemented as a list of tuples (heuristic, node, path)

priority\_queue = [(g.heuristic[o], o, [])]

total\_path = []

while priority\_queue:

# Find the path with the lowest heuristic value in the priority queue

min\_index = 0

for i in range(1, len(priority\_queue)):

if priority\_queue[i][0] < priority\_queue[min\_index][0]:

min\_index = i

heuristic, current, path = priority\_queue.pop(min\_index)

total\_path.append(path+[current])

if current == d:

# Destination reached, update best\_path

best\_path = path + [current]

print(best\_path)

return total\_path

else:

for neighbor in g.graph[current]:

if neighbor not in path:

# Add the neighbor to the priority queue with updated heuristic

priority\_queue.append((g.heuristic[neighbor], neighbor, path + [current]))

print(best\_path)

return total\_path

def AOstar(self, g, o, d):

open\_list = [(g.heuristic[o], o, [])]

closed\_list = []

total\_path = []

while open\_list:

open\_list.sort(key=lambda x: x[0])

h, current, path = open\_list.pop(0)

total\_path.append(path+[current])

if current == d:

print("Optimal path:", path + [current])

return total\_path

for neighbor, weight in zip(g.graph[current], g.weight[current]):

if neighbor not in path and neighbor not in closed\_list:

g\_value = len(path) + weight

h\_value = g.heuristic[neighbor]

f\_value = g\_value + h\_value

new\_path = path + [current]

open\_list.append((f\_value, neighbor, new\_path))

closed\_list.append(current)

print("No path found")

return None

class GraphVisualization:

def visualize\_traversal(self, g, o, d, traversal\_algorithm, bw=1):

# Create a networkx graph to visualize

G = nx.Graph()

for node, neighbors in g.graph.items():

for neighbor, weight in zip(neighbors, g.weight[node]):

G.add\_edge(node, neighbor, weight=weight)

# Determine the amount of time the algorithm runs

start\_time = time.time()

# Determine which traversal algorithm is being used and get the paths

if traversal\_algorithm.\_\_name\_\_ == "BS":

paths = traversal\_algorithm(g, o, d, bw)

else:

paths = traversal\_algorithm(g, o, d)

# Record the end time

end\_time = time.time()

# Calculate the elapsed time

elapsed\_time = end\_time - start\_time

# print(elapsed\_time)

elapsed\_time\_microseconds = elapsed\_time \* 1\_000\_000

print(f"Elapsed time: {elapsed\_time\_microseconds:.2f} microseconds\n\n")

# Choose a layout for the graph (e.g., planar layout)

pos = nx.planar\_layout(G) # You can choose a different layout if you prefer.

# Create a figure and axes for the visualization

fig, ax = plt.subplots()

# Define the update function for the animation

def update(frame):

ax.clear()

# Create node labels with both the node name and heuristic value

node\_labels = {node: f"{node}\nH:{g.heuristic[node]}" for node in G.nodes()}

# Draw the graph

nx.draw(G, pos, with\_labels=True, node\_size=700, font\_size=10, node\_color='lightblue', font\_color='black', font\_weight='bold', labels=node\_labels, ax=ax)

edge\_labels = {(node, neighbor): G[node][neighbor]['weight'] for node, neighbor in G.edges()}

nx.draw\_networkx\_edge\_labels(G, pos, edge\_labels=edge\_labels, label\_pos=0.5, font\_size=8, ax=ax)

# Highlight the path up to the current step

if frame < len(paths):

path = paths[frame]

path\_edges = [(path[i], path[i + 1]) for i in range(len(path) - 1)]

nx.draw\_networkx\_edges(G, pos, edgelist=path\_edges, edge\_color='red', width=2, ax=ax)

# Create an animation with the defined update function

ani = FuncAnimation(fig, update, frames=len(paths) + 1, repeat=False, interval=1000) # Adjust the interval to control animation speed

plt.show()

choice = input("Click Enter to continue with default values, else enter 1")

g = Graph()

algo = Algorithm()

if choice == '':

# Default graphs for testing

# Grpah 1

g.addEdge('S','A',3)

g.addEdge('S','B',5)

g.addEdge('A','B',4)

g.addEdge('A','D',3)

g.addEdge('D','G',5)

g.addEdge('B','C',4)

g.addEdge('C','E',6)

g.addHeuristics('S',10)

g.addHeuristics('A',7)

g.addHeuristics('B',6)

g.addHeuristics('C',7)

g.addHeuristics('D',5)

g.addHeuristics('E',4)

g.addHeuristics('G',0)

# Graph 2

# g.addEdge('A','F',3)

# g.addEdge('A','B',6)

# g.addEdge('F','G',1)

# g.addEdge('F','H',7)

# g.addEdge('G','I',3)

# g.addEdge('H','I',2)

# g.addEdge('I','E',5)

# g.addEdge('I','J',3)

# g.addEdge('J','E',5)

# g.addEdge('E','C',5)

# g.addEdge('E','D',8)

# g.addEdge('C','B',3)

# g.addEdge('C','D',1)

# g.addEdge('D','B',2)

# Graph 3

# g.addEdgeD('P','R',10)

# g.addEdgeD('P','A',4)

# g.addEdgeD('P','C',4)

# g.addEdgeD('A','M',3)

# g.addEdgeD('C','M',6)

# g.addEdgeD('C','R',2)

# g.addEdgeD('C','U',3)

# g.addEdgeD('M','U',5)

# g.addEdgeD('M','L',2)

# g.addEdgeD('L','N',5)

# g.addEdgeD('U','N',5)

# g.addEdgeD('N','S',6)

# g.addEdgeD('U','S',4)

# g.addEdgeD('R','E',5)

# g.addEdgeD('E','U',5)

# g.addEdgeD('E','S',1)

# g.addHeuristics('P',0)

# g.addHeuristics('R',0)

# g.addHeuristics('C',0)

# g.addHeuristics('A',0)

# g.addHeuristics('M',0)

# g.addHeuristics('L',0)

# g.addHeuristics('U',0)

# g.addHeuristics('N',0)

# g.addHeuristics('S',0)

# g.addHeuristics('E',0)

# Graph 4

# g.addEdge('P','R',10)

# g.addEdge('P','A',4)

# g.addEdge('P','C',4)

# g.addEdge('A','M',3)

# g.addEdge('C','M',6)

# g.addEdge('C','R',2)

# g.addEdge('C','U',3)

# g.addEdge('M','U',5)

# g.addEdge('M','L',2)

# g.addEdge('L','N',5)

# g.addEdge('U','N',5)

# g.addEdge('N','S',6)

# g.addEdge('U','S',4)

# g.addEdge('R','E',5)

# g.addEdge('E','U',5)

# g.addEdge('E','S',1)

# g.addHeuristics('P',10)

# g.addHeuristics('R',8)

# g.addHeuristics('C',6)

# g.addHeuristics('A',11)

# g.addHeuristics('M',9)

# g.addHeuristics('L',9)

# g.addHeuristics('U',4)

# g.addHeuristics('N',6)

# g.addHeuristics('S',0)

# g.addHeuristics('E',3)

else:

# Create your own graph

while(True):

choice = input("Enter\n 1 for Edge (Source, Destination, Cost),\n 2 for Heuristic (Source, Value) [Default Hueristics = 100],\n 0 to exit\n")

if choice == 1:

g.addEdge(input("Enter Source : "), input("Enter Destination : ", int(input("Enter Weight : "))))

elif choice == 2:

g.addHeuristics(input("Enter Source : "), int(input("Enter Heuristics : ")))

elif choice == 0:

break

else:

print("Wrong option Dummy")

source = input("Enter Source node : ")

destination = input("Enter Destination node : ")

while True:

choice = input("\n\nEnter between\nDFS\nBFS\nBMS\nHC\nBS\nOracle\nOracleH\nBB\nEL\nEH\nAstar\nAOstar\nBestFirstSearch\nExit\n\n")

if choice == "DFS":

paths = algo.DFS(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.DFS)

elif choice == "BFS":

paths = algo.BFS(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.BFS)

elif choice == "BMS":

paths = algo.BMS(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.BMS)

elif choice == "HC":

paths = algo.HC(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.HC)

elif choice == "BS":

bw = int(input("Enter beam width: "))

paths = algo.BS(g, source, destination, bw)

GraphVisualization().visualize\_traversal(g, source, destination, algo.BS, bw)

elif choice == "Oracle":

paths = algo.Oracle(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.Oracle)

elif choice == "OracleH":

paths = algo.OracleH(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.OracleH)

elif choice == "BB":

paths = algo.BB(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.BB)

elif choice == "EL":

paths = algo.EL(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.EL)

elif choice == "EH":

paths = algo.EH(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.EH)

elif choice == "Astar":

paths = algo.Astar(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.Astar)

elif choice == "AOstar":

paths = algo.AOstar(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.AOstar)

elif choice == "BestFirstSearch":

paths = algo.BestFirstSearch(g, source, destination)

GraphVisualization().visualize\_traversal(g, source, destination, algo.BestFirstSearch)

elif choice == "Exit":

print("Program terminated")

break

else:

print("Wrong option Dummy.")

"""

Author : Mukesh T P

"""

**Output:**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

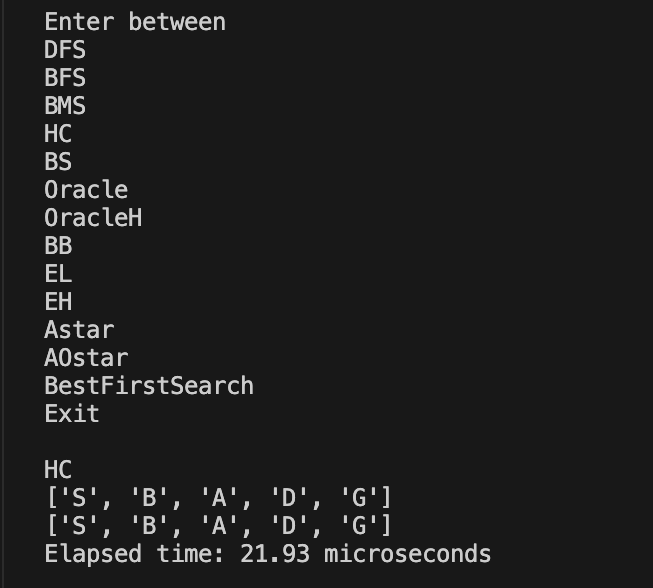
**A screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generated**

**A computer screen shot of a black screen

Description automatically generatedA screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

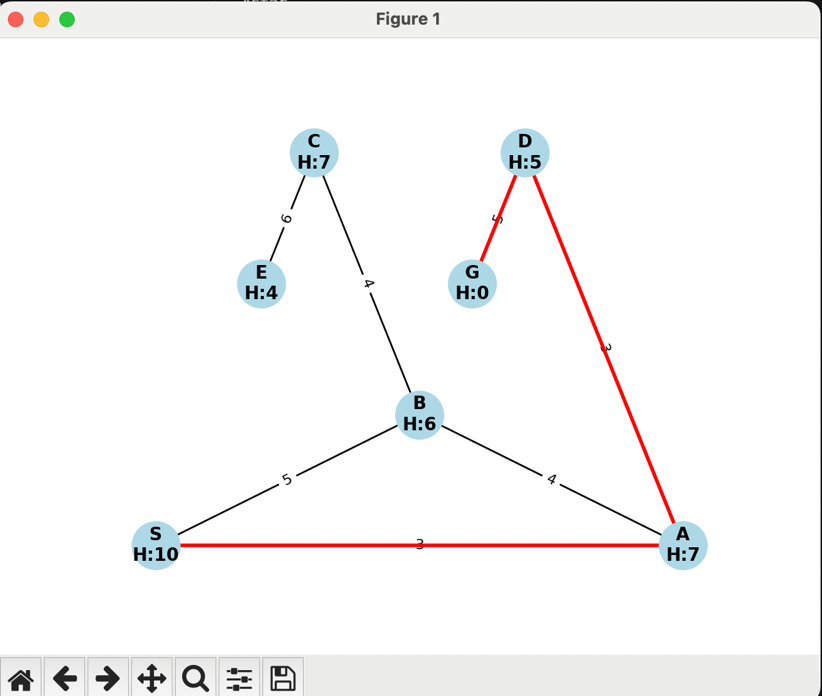
Description automatically generated**

**A screenshot of a computer

Description automatically generatedA computer screen shot of a black screen

Description automatically generated**

**A computer screen shot of a black screen

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA computer screen shot of a black screen

Description automatically generated**

**A computer screen shot of a black screen

Description automatically generated**

**A screenshot of a computer

Description automatically generatedA computer screen shot of a black screen

Description automatically generated**

**Comparison:**

**A table of letters in a grid

Description automatically generated**

**Justification:**

1. **AO\* (10.97 microseconds):**
   * AO\* stands out as the fastest algorithm, with the shortest execution time.
   * This performance can be attributed to AO\*'s use of an effective heuristic and its efficient pruning and caching strategies, which reduce redundant work.
2. **A\* (15.97 microseconds):**
   * A\* also demonstrates efficient performance, slightly slower than AO\* but faster than most other algorithms.
   * The effectiveness of A\* is due to its admissible heuristic and an optimal pathfinding strategy.
3. **BFS (18.12 microseconds) and DFS (19.07 microseconds):**
   * BFS and DFS are classic search algorithms and demonstrate relatively good performance.
   * The differences in execution times can be attributed to the nature of the problem and the specific characteristics of the search space. BFS explores the breadth of the search space, while DFS explores depth-first.
4. **BestFS (21.93 microseconds) and HC (21.93 microseconds):**
   * BestFS and HC have the same execution time, suggesting that they may share some similarities in their search strategies.
   * The execution time is competitive, indicating that these algorithms efficiently explore the search space.
5. **BS (26.23 microseconds) and BMS (27.89 microseconds):**
   * BS and BMS exhibit similar execution times, with BMS being slightly slower.
   * The variations can be attributed to the specific problem instances and the inherent differences in their search strategies.
6. **Oracle (29.8 microseconds) and OracleH (29.8 microseconds):**
   * Oracle and OracleH have identical execution times, indicating that they likely share a common approach.
   * These algorithms are among the slower ones in this comparison, which might be due to their problem-specific nature or computational requirements.
7. **BB (36.24 microseconds):**
   * Branch and Bound (BB) is slower than the faster algorithms, primarily because it exhaustively explores and evaluates all possible solutions, leading to longer runtimes.
8. **EH (38.15 microseconds) and EL (35.76 microseconds):**
   * EH and EL, being variants of Branch and Bound, show relatively slower execution times. Their additional complexities in search strategy contribute to these longer runtimes.

I have used Python to write all the logic and the visualization because of the following

 **Readability and Maintainability:** Python is known for its clean and readable syntax. This makes it easier to write, understand, and maintain code. When working on complex algorithms, code readability is crucial for collaboration and future reference.

 **Rich Ecosystem:** Python boasts a vast ecosystem of libraries and frameworks for various tasks. You can find libraries for data manipulation, visualization and more. This rich ecosystem can save you time and effort when implementing and visualizing algorithms.

 **Visualization Capabilities:** Python offers numerous libraries for data visualization, such as Matplotlib, Seaborn, and Plotly. These tools make it convenient to create graphical representations of your algorithms, helping you better understand their behavior and results.

 **Cross-Platform Compatibility:** Python is a cross-platform language, which means your code can run on different operating systems without significant modifications. This flexibility is valuable when you want your code to be accessible to a broad audience.

**Result:**

Successfully visualized different search algorithm and compared their performances.

|  |  |
| --- | --- |
| **Ex. No: 14-15** | GAMING ALGORITHMS |
| **27.10.2023** |

**Aim:**

**To show gaming algorithms.**

**Program:**

import math

def minimax(curDepth, nodeIndex, maxTurn, scores, targetDepth, branchingFactor):

if curDepth == targetDepth:

return scores[nodeIndex]

if maxTurn:

maxVal = float('-inf')

for i in range(branchingFactor):

childIndex = nodeIndex \* branchingFactor + i

childScore = minimax(curDepth + 1, childIndex, False, scores, targetDepth, branchingFactor)

maxVal = max(maxVal, childScore)

return maxVal

else:

minVal = float('inf')

for i in range(branchingFactor):

childIndex = nodeIndex \* branchingFactor + i

childScore = minimax(curDepth + 1, childIndex, True, scores, targetDepth, branchingFactor)

minVal = min(minVal, childScore)

return minVal

def minimax\_alpha\_beta(curDepth, nodeIndex, maxTurn, scores, targetDepth, branchingFactor, alpha, beta):

if curDepth == targetDepth:

return scores[nodeIndex]

if maxTurn:

maxVal = float('-inf')

for i in range(branchingFactor):

childIndex = nodeIndex \* branchingFactor + i

childScore = minimax\_alpha\_beta(curDepth + 1, childIndex, False, scores, targetDepth, branchingFactor, alpha, beta)

maxVal = max(maxVal, childScore)

alpha = max(alpha, childScore)

if beta <= alpha:

print("Pruned")

break

return maxVal

else:

minVal = float('inf')

for i in range(branchingFactor):

childIndex = nodeIndex \* branchingFactor + i

childScore = minimax\_alpha\_beta(curDepth + 1, childIndex, True, scores, targetDepth, branchingFactor, alpha, beta)

minVal = min(minVal, childScore)

beta = min(beta, childScore)

if beta <= alpha:

print("Pruned")

break

return minVal

base\_input = [int(i) for i in input("Enter scores (space-separated): ").split()]

branchingFactor = int(input("Enter branching factor: "))

targetDepth = int(math.log(len(base\_input), branchingFactor))

result = minimax(0, 0, True, base\_input, targetDepth, branchingFactor)

print("Minimax value:", result)

result = minimax\_alpha\_beta(0, 0, True, base\_input, targetDepth, branchingFactor, float('-inf'), float('inf'))

print("Minimax value with alpha-beta pruning:", result)

**Output:**

**A black screen with white text

Description automatically generated**

**PERFORMANCE COMPARISON:**

|  |  |  |
| --- | --- | --- |
| **Algorithm** | **Time Complexity** | **Space Complexity** |
| **MINIMAX** | O(bm)  b is the branching factor  m is the maximum depth of the tree | Dependent on the depth of the recursion. |
| **ALPHA BETA PRUNING** | O(bm/2) (best case)  O(bm) (worst case) | Similar to Minimax but with reduced memory consumption due to pruning. |

**JUSTIFICATION FOR THE PERFORMANCE:**

The Minimax algorithm and its enhanced version, the Alpha-Beta Pruning algorithm, both serve the purpose of determining the optimal decision in a two-player game scenario.

While Minimax exhaustively explores the entire game tree to find the optimal move, Alpha-Beta Pruning enhances Minimax by pruning subtrees that do not affect the final decision, leading to a more efficient search process.

- Minimax guarantees the optimal solution in two-player games. However, it may be inefficient for large game trees, as it explores all possible moves without considering the potential irrelevance of some branches.

- Alpha-Beta Pruning enhances the Minimax algorithm by eliminating subtrees that do not affect the final decision, leading to a more efficient search process. It can significantly reduce the number of nodes evaluated, making it more practical for larger game trees.

**JUSTIFICATION FOR THE CHOICE OF PROGRAMMING MEDIUM:**

Python is a popular choice for implementing minimax and alpha-beta pruning algorithms due to its simplicity, readability, and extensive libraries for data structures. Its dynamic typing and ease of debugging make it a practical language for quickly prototyping and experimenting with complex algorithms, which is essential for AI and game theory implementations like minimax and alpha-beta pruning.

|  |  |
| --- | --- |
| **Ex. No: 16** | CARLA |
| **03.11.2023** |

**Aim:**

Use CARLA to simulate any experimental CARLA setup for autonomous vehicles.

**Reference GitHub link:**

<https://github.com/Geniussh/Self-Driving-Car-Projects/tree/main/Motion%20Planning>

**Explaination:**

**Agent:**

Your agent serves as the core decision-making entity in your autonomous driving system. It integrates various modules to navigate through complex environments:

1. **Behavioural Planner:** This component is responsible for high-level decision-making. It manages scenarios like stop signs at T-junctions using a state machine. It transitions between different states based on the current situation and predefined rules.
2. **Local Planner:** This module generates detailed trajectories or paths for the vehicle. By utilizing a spiral path generation technique based on given waypoints and employing optimization techniques from the scipy library, it ensures the vehicle follows smooth and feasible routes.
3. **Collision Checker:** Safety is paramount. This module verifies that the paths generated by the local planner are free from collisions. By implementing circle-based collision checking, it provides an additional layer of security.
4. **Path Selector:** Once potential paths are generated, this module evaluates them based on an objective function. This ensures the selection of the safest and most feasible path for execution.
5. **Velocity Planner:** This component is crucial for determining how fast the vehicle should travel along the selected path. It considers various scenarios, including stop signs and dynamic obstacles, to generate a velocity profile for the controller.

**Environment:**

The CARLA simulator provides a virtual urban environment that accurately mimics real world driving scenarios. It includes streets, intersections, vehicles, pedestrians, and other objects. This serves as the testing ground for your autonomous driving system.

**Sensor:**

The setup relies on LIDAR measurements as a key sensory input. These measurements provide a detailed 3D representation of the environment, which is vital for tasks like generating the occupancy grid.

**Algorithmic Model:**

The motion planning stack combines various algorithms, enabling your agent to make informed decisions:

1. **State Machines:** Used in the behavioural planner, state machines allow for structured decision-making by transitioning between predefined states based on the current situation.
2. **Optimization with scipy:** The local planner leverages optimization techniques from the scipy library to generate smooth and optimal paths based on given waypoints.
3. **Circle-based Collision Checking:** This algorithm ensures that the generated paths are collision-free by checking for intersections with obstacles represented as circles.
4. **Objective Function Evaluation:** The path selector uses objective functions to evaluate and rank potential paths, ensuring the selection of the most suitable path for execution.
5. **Dynamic Obstacle Handling for Velocity Planning:** The velocity planner adapts the speed profile based on dynamic obstacles and other scenarios, ensuring safe and efficient motion.

**Interactions with the CARLA Environment:**

1. **Sensor Data Reception:** The agent receives LIDAR measurements, which serve as a critical source of environmental perception.
2. **Behavioural Planner Operations:** The agent uses the behavioural planner to handle specific scenarios, such as stop signs at T-junctions. State transitions occur as needed to navigate through intersections.
3. **Local Path Generation:** The local planner uses optimization techniques to generate detailed paths based on provided waypoints. This ensures precise vehicle navigation.
4. **Collision Checking:** The collision checker verifies that the generated paths are free from collisions, adding an additional layer of safety to the planning process.
5. **Path Selection:** The path selector evaluates potential paths based on an objective function, ultimately choosing the best-suited path for execution.
6. **Velocity Profile Generation:** The velocity planner determines the speed profile along the selected path, considering various scenarios like stop signs and dynamic obstacles.
7. **Controller Commands:** The controller updates the vehicle's control commands (e.g., throttle, brake, steering) based on the computed path and velocity profile, ensuring the planned motion is executed accurately.

**Occupancy Grid Generator:**

This component utilizes LIDAR measurements to create an occupancy grid belief map. The inverse scanner measurement model and iterative logodds updates contribute to an accurate representation of the environment's occupancy probabilities.

**Mission Planner:**

The mission planner incorporates Dijkstra's and A\* search algorithms to plan routes on a road network in Berkeley, California. The use of OSMNX and NetworkX libraries for generating Open Street Map data and determining the correct shortest path serves as a solid foundation for route planning.

**Screenshots/Simulations:**

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

**Result:**

Therefore, successfully simulated an experimental setup for autonomous vechiles using CARLA.