Python code for A\* Algorithm

#SATYAM PARASHAR

from queue import PriorityQueue

def astar(start, goal, h, graph):

frontier = PriorityQueue()

frontier.put((0, start))

came\_from = {}

cost\_so\_far = {}

came\_from[start] = None

cost\_so\_far[start] = 0

while not frontier.empty():

\_, current = frontier.get()

if current == goal:

break

for next\_node in graph.neighbors(current):

new\_cost = cost\_so\_far[current] + graph.cost(current, next\_node)

if next\_node not in cost\_so\_far or new\_cost < cost\_so\_far[next\_node]:

cost\_so\_far[next\_node] = new\_cost

priority = new\_cost + h(next\_node, goal)

frontier.put((priority, next\_node))

came\_from[next\_node] = current

# Build path from start to goal

path = []

node = goal

while node != start:

path.append(node)

node = came\_from[node]

path.append(start)

path.reverse()

print("Shortest path:", path)

print("Cost of shortest path:", cost\_so\_far[goal])

return came\_from, cost\_so\_far

OUTPUT:

Shortest path: [A, B, C, E, H]

Cost of shortest path: 12

Hill Climbing Algorithm

#SATYAM PARASHAR

import random

def hill\_climbing(start\_state, goal\_state, evaluate\_state):

current\_state = start\_state

while True:

neighbors = get\_neighbors(current\_state)

neighbor\_states = [apply\_move(current\_state, move) for move in neighbors]

neighbor\_states.append(current\_state)

neighbor\_states = [state for state in neighbor\_states if state is not None]

neighbor\_states.sort(key=lambda state: evaluate\_state(state, goal\_state))

if neighbor\_states[0] == current\_state:

return current\_state

current\_state = neighbor\_states[0]

def get\_neighbors(state):

# Returns a list of possible moves from the current state

# For example, if the state is [1, 2, 3, 4, 5], the neighbors might be [[2, 1, 3, 4, 5], [1, 3, 2, 4, 5], [1, 2, 3, 5, 4]]

# Each move is represented as a list that represents the new order of the elements in the list

neighbors = []

for i in range(len(state)):

for j in range(i+1, len(state)):

neighbor = state.copy()

neighbor[i], neighbor[j] = neighbor[j], neighbor[i]

neighbors.append(neighbor)

return neighbors

def apply\_move(state, move):

# Returns the new state that results from applying the given move to the current state

# If the move is not valid (e.g., it tries to swap an element with itself), returns None

new\_state = state.copy()

try:

i, j = move

new\_state[i], new\_state[j] = new\_state[j], new\_state[i]

return new\_state

except:

return None

def evaluate\_state(state, goal\_state):

# Returns a score that measures how close the current state is to the goal state

# In this example, we simply count the number of elements that are in the correct position

score = 0

for i in range(len(state)):

if state[i] == goal\_state[i]:

score += 1

return score

# Example usage

start\_state = [1, 3, 2, 4, 5]

goal\_state = [1, 2, 3, 4, 5]

final\_state = hill\_climbing(start\_state, goal\_state, evaluate\_state)

print("Final state:", final\_state)

print("Number of correct elements:", evaluate\_state(final\_state, goal\_state))

OUTPUT:

Final state: [1, 3, 2, 4, 5]

Number of correct elements: 3

MinMax Algorithm

#SATYAM PARASHAR

import math

def minimax(node, depth, maximizing\_player, evaluate):

if depth == 0 or node is None:

return evaluate(node)

if maximizing\_player:

value = -math.inf

for child in get\_children(node):

value = max(value, minimax(child, depth-1, False, evaluate))

return value

else:

value = math.inf

for child in get\_children(node):

value = min(value, minimax(child, depth-1, True, evaluate))

return value

def get\_children(node):

# Returns a list of child nodes for the given node

# In this example, we represent the node as a list of integers

# The children are generated by adding either 1 or -1 to a random element in the list

children = []

for i in range(len(node)):

for delta in [-1, 1]:

child = node.copy()

child[i] += delta

children.append(child)

return children

def evaluate(node):

# Returns a score that measures the value of the given node

# In this example, we simply return the sum of the elements in the node

return sum(node)

# Example usage

root = [0, 0, 0, 0]

maximizing\_player = True

depth = 3

best\_value = minimax(root, depth, maximizing\_player, evaluate)

print("Best value:", best\_value)

OUTPUT:

Best value: 6

Uinform Cost Search Algorithm

#SATYAM PARASHAR

from queue import PriorityQueue

def uniform\_cost\_search(problem):

frontier = PriorityQueue()

frontier.put((0, problem.initial\_state))

explored = set()

while not frontier.empty():

cost, state = frontier.get()

if problem.goal\_test(state):

return (state, cost)

explored.add(state)

for action, next\_state, step\_cost in problem.get\_successors(state):

if next\_state not in explored:

new\_cost = cost + step\_cost

frontier.put((new\_cost, next\_state))

return None

class Problem:

def \_\_init\_\_(self, initial\_state, goal\_state, successors\_fn, step\_cost\_fn):

self.initial\_state = initial\_state

self.goal\_state = goal\_state

self.successors\_fn = successors\_fn

self.step\_cost\_fn = step\_cost\_fn

def goal\_test(self, state):

return state == self.goal\_state

def get\_successors(self, state):

return [(action, next\_state, self.step\_cost\_fn(state, action, next\_state))

for action, next\_state in self.successors\_fn(state)]

def successors\_fn(state):

return [("right", state+1), ("left", state-1)]

def step\_cost\_fn(state, action, next\_state):

return abs(next\_state - state)

# Example usage

problem = Problem(initial\_state=0, goal\_state=10, successors\_fn=successors\_fn, step\_cost\_fn=step\_cost\_fn)

solution = uniform\_cost\_search(problem)

print("Solution:", solution)

OUTPUT:

Solution: (10, 10)