MONKEY BANANA PROBLEM

```
move(state(middle,onbox,middle,hasnot),
   grasp,
   state(middle,onbox,middle,has)).
move(state(P,onfloor,P,H),
   climb,
   state(P,onbox,P,H)).
move(state(P1,onfloor,P1,H),
   drag(P1,P2),
   state(P2,onfloor,P2,H)).
move(state(P1,onfloor,B,H),
   walk(P1,P2),
   state(P2,onfloor,B,H)).
canget(state(_,_,_,has)).
canget(State1) :-
   move(State1,_,State2),
   canget(State2).
```

Travelling Salesperson Problem with Hill Climbing

```
import random
import math
def calculate_distance(city1, city2):
    # Calculate the Euclidean distance between two cities
    x1, y1 = city1
    x2, y2 = city2
    return math.sqrt((x2 - x1)**2 + (y2 - y1)**2)
def calculate_total_distance(route, cities):
    # Calculate the total distance of a given route
    total_distance = 0
    for i in range(len(route)):
        city1 = cities[route[i]]
        city2 = cities[route[(i + 1) % len(route)]]
        total_distance += calculate_distance(city1, city2)
    return total_distance
def generate_random_route(cities):
    # Generate a random route that visits all cities
    route = list(range(len(cities)))
    random.shuffle(route)
    return route
def hill_climbing(cities, iterations):
    # Perform hill climbing to find the shortest route
    current_route = generate_random_route(cities)
    current_distance = calculate_total_distance(current_route, cities)
    for _ in range(iterations):
        # Generate a neighboring route by swapping two cities
        neighbor_route = current_route.copy()
        index1 = random.randint(0, len(neighbor_route) - 1)
        index2 = random.randint(0, len(neighbor_route) - 1)
        neighbor_route[index1], neighbor_route[index2] =
neighbor_route[index2], neighbor_route[index1]
        # Calculate the distance of the neighboring route
        neighbor_distance = calculate_total_distance(neighbor_route, cities)
        # Update the current route if the neighbor is better
        if neighbor_distance < current_distance:
            current_route = neighbor_route
            current_distance = neighbor_distance
    return current_route, current_distance
# Example usage
cities = [(0, 0), (1, 5), (3, 2), (6, 3), (8, 1)]
iterations = 1000

shortest_route, shortest_distance = hill_climbing(cities, iterations)
print("Shortest route:", shortest_route)
print("Shortest distance:", shortest_distance)
```

BFS DFS PROGRAM

```
#BFS_DFS
from collections import deque

# Breadth-First Search (BFS)
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        print(vertex, end=" ")

        for neighbor in graph[vertex]:
            if neighbor not in visited:
                queue.append(neighbor)
                visited.add(neighbor)

# Depth-First Search (DFS)
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()

    visited.add(start)
    print(start, end=" ")

    for neighbor in graph[start]:
        if neighbor not in visited:
            dfs(graph, neighbor, visited)

# Example graph representation as an adjacency list
graph = {
    'A': ['B', 'C'],
    'B': ['D', 'E'],
    'C': ['F'],
    'D': [],
    'E': ['F'],
    'F': []
}

print("BFS traversal:")
bfs(graph, 'A')

print("\nDFS traversal:")
dfs(graph, 'A')
```

AO* ALGORITHM

```
import heapq

# Node class to represent each state in the search
class Node:
    def _init_(self, state, g_cost, h_cost):
        self.state = state
        self.g_cost = g_cost  # cost from the start node
        self.h_cost = h_cost  # heuristic cost
        self.f_cost = g_cost + h_cost  # total estimated cost

    def _lt_(self, other):
        return self.f_cost < other.f_cost

# AO* algorithm implementation
def ao_star(start, goal, heuristic_func, successors_func):
    open_set = []
    closed_set = set()

    # Create the start node
    start_node = Node(start, 0, heuristic_func(start, goal))
    heapq.heappush(open_set, start_node)

    while open_set:
        current_node = heapq.heappop(open_set)

        if current_node.state == goal:
            return current_node

        closed_set.add(current_node.state)

        for successor in successors_func(current_node.state):
            g_cost = current_node.g_cost + 1  # Assuming uniform
cost between nodes
            h_cost = heuristic_func(successor, goal)
            f_cost = g_cost + h_cost
            child_node = Node(successor, g_cost, h_cost)

            if successor in closed_set:
                continue
            # Check if the successor is already in the open set with
a lower cost
            for node in open_set:
                if node.state == successor and node.f_cost <=
f_cost:
                    break
            else:
                heapq.heappush(open_set, child_node)

    return None  # No path found

# Example usage
def heuristic(state, goal):
    # Example heuristic function: Manhattan distance
    x1, y1 = state
    x2, y2 = goal
    return abs(x1 - x2) + abs(y1 - y2)

def successors(state):
    # Example successors function: 4-connected grid world
    x, y = state
    successors = [(x + 1, y), (x - 1, y), (x, y + 1), (x, y - 1)]
    return [(x, y) for x, y in successors if 0 <= x < 5 and 0 <= y <
5]

start_state = (0, 0)
goal_state = (4, 4)

result = ao_star(start_state, goal_state, heuristic, successors)

if result:
    path = [result.state]
    while result.state != start_state:
        result = result.parent
        path.append(result.state)
    path.reverse()
    print("Path found:", path)
else:
    print("No path found.")
```

Simulated Annealing Implementation

```
import random
import math
# Objective function example (you can define your own)
def objective_function(x):
    return x**2
# Simulated Annealing implementation
def simulated_annealing(initial_state, initial_temperature, cooling_rate,
num_iterations):
    current_state = initial_state
    best_state = initial_state
    for i in range(num_iterations):
        temperature = initial_temperature * math.exp(-cooling_rate * i)
        # Generate a random neighbor state
        neighbor_state = generate_neighbor(current_state)
        # Calculate the objective function values for current and neighbor
states
        current_cost = objective_function(current_state)
        neighbor_cost = objective_function(neighbor_state)
        # Decide whether to move to the neighbor state
        if neighbor_cost < current_cost:
            current_state = neighbor_state
        else:
            # Calculate the acceptance probability based on the cost
difference and temperature
            acceptance_prob = math.exp((current_cost - neighbor_cost) /
temperature)
            if random.random() < acceptance_prob:
                current_state = neighbor_state
        # Update the best state if necessary
        if objective_function(current_state) < objective_function(best_state):
            best_state = current_state
    return best_state
# Helper function to generate a neighbor state (you can define your own)
def generate_neighbor(state):
    return state + random.uniform(-1, 1)
# Example usage
initial_state = 5
initial_temperature = 100.0
cooling_rate = 0.01
num_iterations = 1000

best_state = simulated_annealing(initial_state, initial_temperature,
cooling_rate, num_iterations)

print("Best state:", best_state)
print("Objective function value:", objective_function(best_state))
```