• https://github.com/AbdulRehman393/AI-Important-Concepts-Lab-Assign-1

www.youtube.com/@khawajaabdulrehmansaeed5750

Create a dictionary to store student information with the following keys: name, age, roll_number, and grade. • Perform the following operations:

1. Print all keys and values.
2. Update the grade of the student.
3. Add a new key email with a value.
4. Delete the roll_number key.

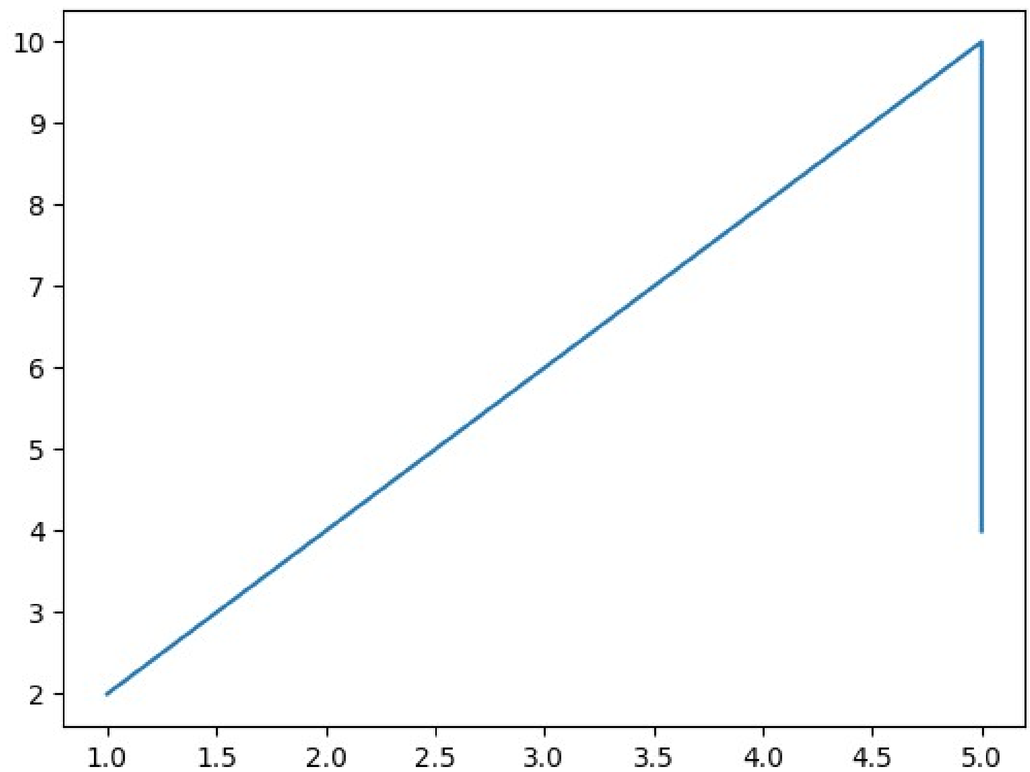
```
student = {"name":"Ahmad","age":20,"roll_number":222,"grade":"A"}
print(student)
student["grade"]="B"
print(student)
student["email"] = "khawajaabdulrehman222@gmail.com"
print(student)
del student["roll_number"]
print(student)
```

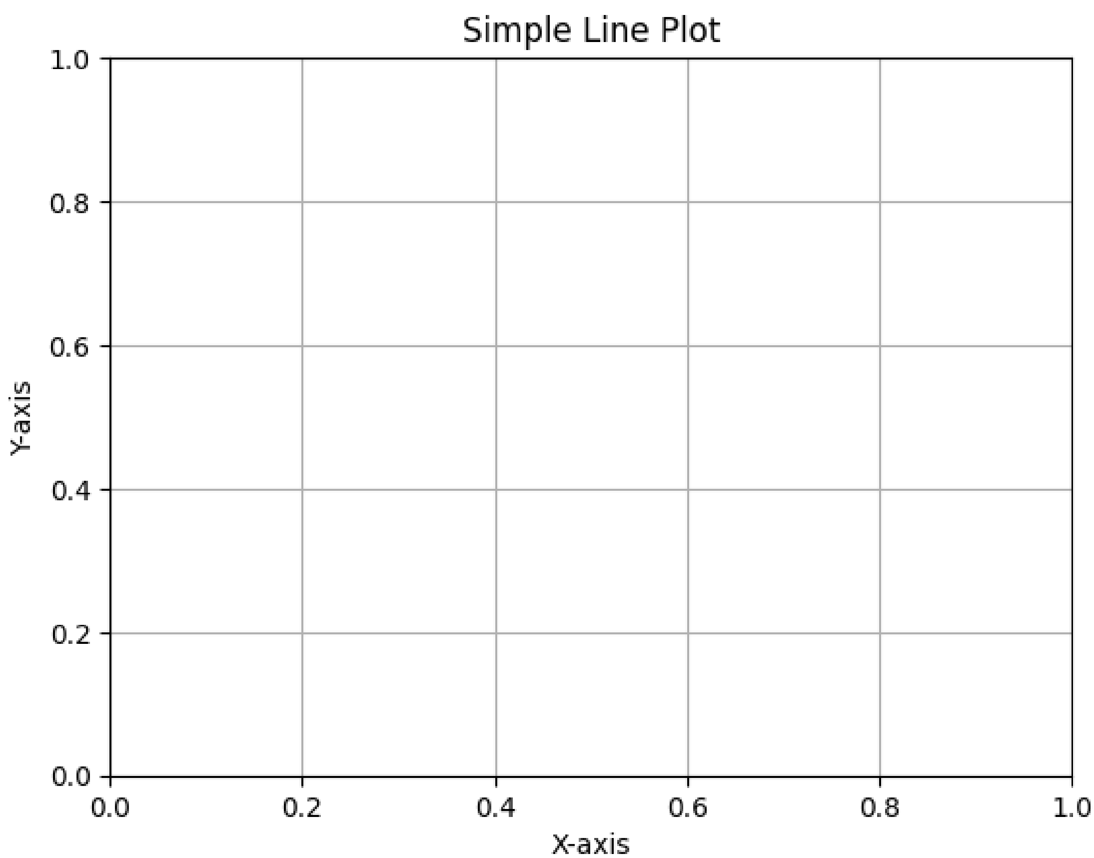
```
{'name': 'Ahmad', 'age': 20, 'roll_number': 222, 'grade': 'A'}
{'name': 'Ahmad', 'age': 20, 'roll_number': 222, 'grade': 'B'}
{'name': 'Ahmad', 'age': 20, 'roll_number': 222, 'grade': 'B',
'email': 'khawajaabdulrehman222@gmail.com'}
{'name': 'Ahmad', 'age': 20, 'grade': 'B', 'email':
'khawajaabdulrehman222@gmail.com'}
```

1. Import the Matplotlib library.
2. Create a simple line plot for the following data: • X = [1, 2, 3, 4, 5] • Y = [2, 4, 6, 8, 10]
3. Add:
4. Title: "Simple Line Plot"
5. Labels for the X-axis and Y-axis.
6. Grid lines.

```
import matplotlib.pyplot as plt

x = [1,2,3,4,5,5]
y = [2,4,6,8,10,4]
plt.plot(x,y)
plt.show()
plt.title("Simple Line Plot")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.grid(True)
plt.show()
```


10
9
8
7
6
5
4
3
2
1.0
1.5
2.0
2.5
3.0
3.5
4.0
4.5
5.0


Simple Line Plot
1.0
0.8
0.6
0.4
0.2
0.0
Y-axis
0.0
0.2
0.4
0.6
0.8
1.0
X-axis

For Water Jug problem

• Implement both BFS and DFS. • Compare their performance in terms of: o Number of steps taken. o Time taken to find the solution. o Memory usage.

Solution:

Jug 1: 4 liters capacity

Jug 2: 3 liters capacity

Goal: Get exactly 2 liters in either jug

Your Results Algorithm Steps Taken Time (seconds) Memory (bytes) Solution Path BFS 4 0.000139 1736 (0,0) → (0,3) → (3,0) → (3,3) → (4,2) DFS 4 0.000066 992 (0,0) → (0,3) → (3,0) → (3,3) → (4,2)

Row 0: A: (0,0) - Start B: (0,1) C: (0,2) D: (0,3) E: (0,4) Row 1: F: (1,0) G: (1,2) Row 2: H: (2,0) I: (2,1) J: (2,2) K: (2,4) L: (2,5) Row 3: M: (3,0) N: (3,1) O: (3,3) P: (3,4) Q: (3,5) Row 4: R: (4,0) S: (4,1) T: (4,2) U: (4,3) V: (4,5) Row 5: W: (5,1) X: (5,3) Y: (5,4) - Goal The walls are the blue tiles. This means you cannot move to (0,5), (1,1), (1,3), (1,4), (1,5), (2,3), (3,2), (4,4), (5,0), (5,2), (5,5).

1. A Search Algorithm Components *A* search requires: g(n): The cost from the start node to node n. In this grid, each move (up, down, left, right) has a cost of 1. h(n): The heuristic cost (estimated cost) from node n to the goal node. For a grid, the Manhattan distance is a common and admissible heuristic. Manhattan Distance = abs(n.row - goal.row) + abs(n.col - goal.col) f(n): The total estimated cost of the path through node n to the goal. f(n) = g(n) + h(n) Our goal node Y is at (5,4).
2. Let's Calculate Heuristics (h(n)) for each node: Node Coordinates h(n) (to Y=(5,4)) A (0,0) abs(0-5)+abs(0-4) = 9 B (0,1) abs(0-5)+abs(1-4) = 8 C (0,2) abs(0-5)+abs(2-4) = 7 D (0,3) abs(0-5)+abs(3-4) = 6 E (0,4) abs(0-5)+abs(4-4) = 5 F (1,0) abs(1-5)+abs(0-4) = 8 G (1,2) abs(1-5)+abs(2-4) = 6 H (2,0) abs(2-5)+abs(0-4) = 7 I (2,1) abs(2-5)+abs(1-4) = 6 J (2,2) abs(2-5)+abs(2-4) = 5 K (2,4) abs(2-5)+abs(4-4) = 3 L (2,5) abs(2-5)+abs(5-4) = 4 M (3,0) abs(3-5)+abs(0-4) = 6 N (3,1) abs(3-5)+abs(1-4) = 5 O (3,3) abs(3-5)+abs(3-4) = 3 P (3,4) abs(3-5)+abs(4-4) = 2 Q (3,5) abs(3-5)+abs(5-4) = 3 R (4,0) abs(4-5)+abs(0-4) = 5 S (4,1) abs(4-5)+abs(1-4) = 4 T (4,2) abs(4-5)+abs(2-4) = 3 U (4,3) abs(4-5)+abs(3-4) = 2 V (4,5) abs(4-5)+abs(5-4) = 2 W (5,1) abs(5-5)+abs(1-4) = 3 X (5,3) abs(5-5)+abs(3-4) = 1 Y (5,4) abs(5-5)+abs(4-4) = 0
3. A Search Steps (Detailed Walkthrough)* We'll use two lists: open_list (nodes to explore) and closed_list (nodes already explored). Each entry in the open_list will be (f_score, g_score, node, parent_node). Start: Node A (0,0). g(A) = 0, h(A) = 9. So, f(A) = 9. open_list = [(9, 0, 'A', None)] closed_list = [] came_from = {} (to reconstruct path) Iteration 1: Pop A from open_list. A is now current. Add A to closed_list. Neighbors of A: B (0,1), F (1,0) To B: g(B) = g(A) + 1 = 1. h(B) = 8. f(B) = 1 + 8 = 9. open_list gets (9, 1, 'B', 'A') To F: g(F) = g(A) + 1 = 1. h(F) = 8. f(F) = 1 + 8 = 9. open_list gets (9, 1, 'F', 'A') open_list = [(9, 1, 'B', 'A'), (9, 1, 'F', 'A')] (sorted by f, then g) Iteration 2: Pop B (or F, let's choose B as it's typically ordered) from open_list. Current is B. Add B to closed_list. Neighbors of B: A (already in closed), C (0,2) To C: g(C) = g(B) + 1 = 2. h(C) = 7. f(C) = 2 + 7 = 9. open_list gets (9, 2, 'C', 'B') open_list = [(9, 1, 'F', 'A'), (9, 2, 'C', 'B')] Iteration 3: Pop F. Current is F. Add F to closed_list. Neighbors of F: A (closed), H (2,0) To H: g(H) = g(F) + 1 = 2. h(H) = 7. f(H) = 2 + 7 = 9. open_list gets (9, 2, 'H', 'F') open_list = [(9, 2, 'C', 'B'), (9, 2, 'H', 'F')] Iteration 4: Pop C. Current is C. Add C to closed_list. Neighbors of C: B (closed), D (0,3), G (1,2) To D: g(D)

= g(C) + 1 = 3. h(D) = 6. f(D) = 3 + 6 = 9. open_list gets (9, 3, 'D', 'C') To G: g(G) = g(C) + 1 = 3. h(G) = 6. f(G) = 3 + 6 = 9. open_list gets (9, 3, 'G', 'C') open_list = [(9, 2, 'H', 'F'), (9, 3, 'D', 'C'), (9, 3, 'G', 'C')] Iteration 5: Pop H. Current is H. Add H to closed_list. Neighbors of H: F (closed), M (3,0), I (2,1) To M: g(M) = g(H) + 1 = 3. h(M) = 6. f(M) = 3 + 6 = 9. open_list gets (9, 3, 'M', 'H') To I: g(I) = g(H) + 1 = 3. h(I) = 6. f(I) = 3 + 6 = 9. open_list gets (9, 3, 'I', 'H') open_list = [(9, 3, 'D', 'C'), (9, 3, 'G', 'C'), (9, 3, 'M', 'H'), (9, 3, 'I', 'H')] ... and so on. This process continues until Y is popped from the open_list. Let's simulate a few more steps, paying close attention to f-scores: It's tedious to do manually for every node. Let's trace a likely optimal path by intuitively moving towards the goal and minimizing f-scores. Path Exploration (Mental Walkthrough to find the path): A(f=9) A -> B(g=1, h=8, f=9) A -> F(g=1, h=8, f=9) From B: B -> C(g=2, h=7, f=9) From F: F -> H(g=2, h=7, f=9) From C: C -> D(g=3, h=6, f=9), C -> G(g=3, h=6, f=9) From H: H -> I(g=3, h=6, f=9), H -> M(g=3, h=6, f=9) From G: G -> J(g=4, h=5, f=9) From I: I -> J(g=4, h=5, f=9) (This path has lower g if coming from G or I to J) Let's assume we take the path that leads to J as it's centrally located and seems promising. A -> B -> C -> G -> J (g=4) A -> F -> H -> I -> J (g=4) Let's pick A -> F -> H -> I -> J J (2,2) with g=4, h=5, f=9. From J: Neighbors: I (closed), G (closed), K (2,4) To K: g(K) = g(J) + 1 = 5. h(K) = 3. f(K) = 5 + 3 = 8. This is a lower f-score! open_list will prioritize K. came_from['K'] = 'J' Current node: K (2,4) with g=5, h=3, f=8. From K: Neighbors: J (closed), L (2,5), P (3,4) To L: g(L) = g(K) + 1 = 6. h(L) = 4. f(L) = 6 + 4 = 10. came_from['L'] = 'K' To P: g(P) = g(K) + 1 = 6. h(P) = 2. f(P) = 6 + 2 = 8. This is good! came_from['P'] = 'K' open_list now has (8, 6, 'P', 'K') (and potentially L with f=10 and others from earlier branches). P will be chosen. Current node: P (3,4) with g=6, h=2, f=8. From P: Neighbors: K (closed), O (3,3), Q (3,5), U (4,3) To O: g(O) = g(P) + 1 = 7. h(O) = 3. f(O) = 7 + 3 = 10. came_from['O'] = 'P' To Q: g(Q) = g(P) + 1 = 7. h(Q) = 3. f(Q) = 7 + 3 = 10. came_from['Q'] = 'P' To U: g(U) = g(P) + 1 = 7. h(U) = 2. f(U) = 7 + 2 = 9. came_from['U'] = 'P' open_list will likely contain U as the next best option from this branch. Current node: U (4,3) with g=7, h=2, f=9. From U: Neighbors: P (closed), T (4,2), X (5,3) To T: g(T) = g(U) + 1 = 8. h(T) = 3. f(T) = 8 + 3 = 11. came_from['T'] = 'U' To X: g(X) = g(U) + 1 = 8. h(X) = 1. f(X) = 8 + 1 = 9. came_from['X'] = 'U' open_list will likely contain X as the next best option. Current node: X (5,3) with g=8, h=1, f=9. From X: Neighbors: U (closed), Y (5,4) To Y: g(Y) = g(X) + 1 = 9. h(Y) = 0. f(Y) = 9 + 0 = 9. came_from['Y'] = 'X' Goal Reached! Y is popped from open_list. Now we reconstruct the path.

4. Reconstructing the Path Start from Y and backtrack using came_from: Y <- X X <- U U <- P P <- K K <- J J <- I I <- H H <- F F <- A So the path is: A -> F -> H -> I -> J -> K -> P -> U -> X -> Y The total cost (number of steps) is g(Y) = 9.