

- Each key is a node.
- Each value is a list of neighboring nodes (edges).
- This is known as an **adjacency list** representation.
- It is a **dynamic representation**, meaning the user can build any custom graph by specifying nodes and edges.

## Why Each Line is Written This Way

- **Using Sets for Visited Nodes:** Sets provide efficient  $O(1)$  average time complexity for membership checking.
- **Recursion in DFS:** DFS is naturally suited for recursive implementation because of its depth-based nature.
- **Queue in BFS:** BFS is iterative, using a queue (FIFO) for breadth-based traversal.
- **Flexible Graph Input:** The user can create any graph structure using the input prompts.

Would you like me to improve this code with error handling (like invalid inputs) and better structure? 🚀



give me indetailed explanation of the code line by line

Got it. I will now explain this Python code **line by line in extreme detail**, covering the logic, why it is written that way, and what it does.

## 🚀 In-Depth Line-by-Line Explanation

### 1. Depth First Search (DFS) Function

```
python
def dfs(visited, graph, node):
```

- ``def``: This is a keyword to define a new function in Python.
- ``dfs``: The function name, which stands for **Depth First Search**, a popular graph traversal algorithm.
- ``visited``: This parameter is a set used to track nodes that have been visited.
- ``graph``: This parameter is a dictionary that represents the graph (nodes and their edges).
- ``node``: This is the starting node for the DFS traversal.

```
python
```

```
    if node not in visited:
```

- ``if``: This is a conditional statement.
- ``node not in visited``: It checks if the current node has not already been visited.
  - **Reason:** To avoid visiting the same node multiple times (infinite recursion in cyclic graphs).

```
python
```

```
        print(node, end=" ")
```

- ``print(node, end=" ")``: This prints the current node without creating a new line.
  - ``end=" "``: This ensures that all nodes are printed on the same line, separated by a space.

```
python
```

```
            visited.add(node)
```

- ``visited.add(node)``: This adds the current node to the ``visited`` set.
  - **Reason:** To mark this node as visited so it will not be visited again.

```
python
```

```
        for neighbour in graph[node]:
```

- ``for``: This starts a loop.
- ``neighbour in graph[node]``: This iterates through all neighbors (connected nodes) of the current node.
  - **Reason:** To explore all directly connected nodes.

```
python
```

```
            dfs(visited, graph, neighbour)
```

- ``dfs(visited, graph, neighbour)``: This calls the ``dfs`` function recursively for each neighbor.
  - **Reason:** To explore the graph in a depth-first manner (exploring a path fully before backtracking).

## 2. Breadth First Search (BFS) Function

```
python
```

```
def bfs(visited, graph, node, queue):
```

- ``def``: This defines another function, this time for **Breadth First Search (BFS)**.
- ``visited``: A set to keep track of visited nodes.
- ``graph``: A dictionary representing the graph (nodes and their edges).
- ``node``: The starting node for BFS.
- ``queue``: A list acting as a queue (FIFO - First In First Out) for BFS.

```
python
```

```
    visited.add(node)
```

- ``visited.add(node)``: Adds the starting node to the ``visited`` set.
  - **Reason:** Marks the starting node as visited.

```
python
```

```
    queue.append(node)
```

- ``queue.append(node)``: Adds the starting node to the queue.
  - **Reason:** In BFS, we explore nodes in the order they were added to the queue.

```
python
```

```
    while queue:
```

- ``while queue:``: This loop continues as long as the queue is not empty.
  - **Reason:** In BFS, we process nodes until there are no more nodes to explore.

```
python
```

```
        s = queue.pop(0)
```

- ``s = queue.pop(0)``: Removes (dequeues) the first node from the queue.
  - **Reason:** BFS is a FIFO algorithm (First In, First Out).
- ``s``: Stores the dequeued node for further processing.

```
python
```

```
        print(s, end=" ")
```

- `print(s, end=" ")`: Prints the dequeued node.
  - `end=" "`: Ensures nodes are printed in a single line with spaces between them.

python

```
for neighbour in graph[s]:
```

- `for`: This starts a loop.
- `neighbour in graph[s]`: Iterates over all neighbors of the dequeued node.

python

```
if neighbour not in visited:
```

- `if neighbour not in visited`: Checks if the neighbor has not been visited.
  - **Reason:** To prevent re-visiting nodes.

python

```
visited.add(neighbour)
```

- `visited.add(neighbour)`: Marks the neighbor as visited.
  - **Reason:** Prevents re-processing the same node.

python

```
queue.append(neighbour)
```

- `queue.append(neighbour)`: Enqueues the neighbor to explore it later.
  - **Reason:** This ensures the neighbor is processed in FIFO order.

### 3. Main Function (Program Entry Point)

python

```
def main():
```

- `def`: Defines the main function of the program.
- `main`: The function name, which is the entry point of the program.

python

```
visited1 = set()
visited2 = set()
```

```
queue = []
```

- ``visited1` and `visited2``: Two sets to track visited nodes for DFS and BFS separately.
- ``queue``: A list that will serve as the queue for BFS.

python

```
n = int(input("Enter number of nodes : "))
```

- ``n``: The total number of nodes in the graph.
- ``input()``: Prompts the user to enter the number of nodes.
- ``int()``: Converts the input from string to integer.

python

```
graph = dict()
```

- ``graph``: Initializes an empty dictionary to store the graph.
  - **Key**: Node number.
  - **Value**: List of connected nodes (edges).

python

```
for i in range(1, n + 1):
```

- ``for``: A loop to create each node in the graph.
- ``range(1, n + 1)``: Iterates from 1 to ``n`` (inclusive).

python

```
edges = int(input("Enter number of edges for node {} : ".format(i)))
graph[i] = list()
```

- ``edges``: The number of connections (edges) for the current node.
- ``graph[i] = list()``: Creates an empty list for each node to store its edges.

python

```
for j in range(1, edges + 1):
    node = int(input("Enter edge {} for node {} : ".format(j, i)))
    graph[i].append(node)
```

- A nested loop that:
  - Takes user input for each edge of the current node.

- Adds each edge (connected node) to the list of the current node in the graph dictionary.

python

```
print("The following is DFS")
dfs(visited1, graph, 1)
```

- **Prints** a heading for DFS.
- **Calls the DFS function** with:
  - `visited1` set (DFS tracking).
  - The created graph.
  - Starting node `1`.

python

```
print("\nThe following is BFS")
bfs(visited2, graph, 1, queue)
```

- **Prints** a heading for BFS.
- **Calls the BFS function** with:
  - `visited2` set (BFS tracking).
  - The graph.
  - Starting node `1`.
  - The BFS queue.

## 4. Entry Point (Program Control)

python

```
if __name__ == "__main__":
    main()
```

- `if __name__ == "__main__":`: Checks if this file is run directly.
  - `main()`: Calls the main function.
  - **Reason:** Prevents the main function from running if this file is imported as a module.

 **Would you like me to show you a well-structured, optimized version of this code?**



```
g=0
def print_board(elements):
```

```

for i in range(9):
    if i%3 == 0:
        print()
    if elements[i]==-1:
        print("_", end = " ")
    else:
        print(elements[i], end = " ")
print()

def solvable(start):
    inv=0

    for i in range(9):
        if start[i] <= 1:
            continue
        for j in range(i+1,9):
            if start[j]==-1:
                continue
            if start[i]>start[j]:
                inv+=1
    if inv%2==0:
        return True
    return False

def heuristic(start,goal):
    global g
    h = 0
    for i in range(9):
        for j in range(9):
            if start[i] == goal[j] and start[i] != -1:
                h += (abs(j-i))/3 + (abs(j-i))%3
    return h + g

def moveleft(start,position):
    start[position],start[position-1]= start[position-1],start[position]

def moveright(start,position):
    start[position],start[position+1]= start[position+1],start[position]

def moveup(start,position):
    start[position],start[position-3]= start[position-3],start[position]

def movedown(start,position):
    start[position],start[position+3]= start[position+3],start[position]

def movetile(start,goal):
    emptyat= start.index(-1)
    row = emptyat//3
    col = emptyat%3
    t1,t2,t3,t4 = start[:,],start[:,],start[:,],start[:,]
    f1,f2,f3,f4 = 100,100,100,100

```

```

if col -1 >=0:
    moveleft(t1, emptyat)
    f1 = heuristic(t1, goal)
if col+1<3:
    moveright(t2, emptyat)
    f2 = heuristic(t2, goal)
if row + 1 <3:
    movedown(t3, emptyat)
    f3 = heuristic(t3, goal)
if row-1>=0:
    moveup(t4, emptyat)
    f4 = heuristic(t4, goal)

min_heuristic = min(f1, f2,f3,f4)

if f1==min_heuristic:
    moveleft(start, emptyat)
elif f2==min_heuristic:
    moveright(start, emptyat)
elif f3==min_heuristic:
    movedown(start, emptyat)
elif f4 == min_heuristic:
    moveup(start, emptyat)

def solveEight(start,goal):
    global g
    g+=1
    movetile(start,goal)
    print_board(start)
    f = heuristic(start,goal)
    if f == g:
        print("Solved in {} moves".format(f))
        return

    solveEight(start,goal)

def main():
    global g
    start = list()
    goal = list()
    print("Enter the start state:(Enter -1 for empty):")
    for i in range(9):
        start.append(int(input()))

    print("Enter the goal state:(Enter -1 for empty):")
    for i in range(9):
        goal.append(int(input()))

    print_board(start)

```



```
# To check if solvable
if solvable(start):
    solveEight(start,goal)
    print("Solved in {} moves".format(g))
else:
    print("Not possible to solve")

if __name__ == '__main__':
    main()

# start = [3,7,6,5,1,2,4,-1,8]
# goal = [5,3,6,7,-1,2,4,1,8]
```

## In-Depth Line-by-Line Explanation of the 8-Puzzle Solver Code

---

### 1. Global Variable Initialization

```
python

g = 0
```

- `g`: A global variable that tracks the **number of moves** (steps) made to solve the puzzle.
  - It is initialized to `0` because the solution begins with no moves.
- 

### 2. Function: `print_board(elements)`

```
python

def print_board(elements):
```

- This function displays the 8-puzzle board in a 3x3 format.
  - `elements`: A list of 9 values representing the current state of the board.
- 

```
python

for i in range(9):
```

- A loop iterates over all 9 elements of the puzzle (since it is a 3x3 grid).
- 

```
python

if i % 3 == 0:
    print()
```

- ``if i % 3 == 0``: This condition checks if the current position is at the start of a new row (index 0, 3, 6).
- A newline (``\n``) is printed to create the next row.

python

```
if elements[i] == -1:
    print("_", end=" ")
```

- If the current element is ``-1``, it is considered the **empty space** in the puzzle.
- The program prints an underscore (``_``) to represent the empty space.

python

```
else:
    print(elements[i], end=" ")
```

- For all other values, it prints the number itself, separated by a space.

python

```
print()
```

- Prints a final newline for formatting.

### 3. Function: ``solvable(start)``

python

```
def solvable(start):
```

- This function determines if the given puzzle is **solvable**.
- ``start``: A list representing the initial puzzle state.

python

```
inv = 0
```

- ``inv``: A counter for **inversions** (pairs of tiles that are in the wrong order).
- An inversion is a pair where a larger number appears before a smaller one.

python

```
for i in range(9):
    if start[i] <= 1:
```

`continue`

- Loops over all 9 tiles.
- Skips any tile that is `-1`` (empty) or `1`` because `1`` is the smallest and does not cause inversions.

python

```
for j in range(i + 1, 9):
    if start[j] == -1:
        continue
```

- A nested loop starts from the next tile (`j = i + 1``).
- Skips the empty tile (`-1``).

python

```
if start[i] > start[j]:
    inv += 1
```

- If the current tile is greater than the next tile, an inversion is counted.

python

```
if inv % 2 == 0:
    return True
```

- If the total number of inversions is **even**, the puzzle is **solvable**.

python

```
return False
```

- If the number of inversions is **odd**, the puzzle is **unsolvable**.

## 4. Function: `heuristic(start, goal)``

python

```
def heuristic(start, goal):
```

- This function calculates the **heuristic cost** (estimated distance) between the current state (`start``) and the goal state (`goal``).
- It uses the **Manhattan Distance** formula (number of moves required to place each tile in the correct position).
- **Heuristic Function ( $f = g + h$ ):**

- `g` is the actual cost (number of moves made so far).
- `h` is the heuristic cost (Manhattan Distance).

python

```
global g
h = 0
```

- The global variable `g` is used to account for the number of moves so far.
- `h` is initialized to `0` to accumulate the heuristic cost.

python

```
for i in range(9):
    for j in range(9):
```

- Double loop to compare each tile in the start state with its correct position in the goal state.

python

```
if start[i] == goal[j] and start[i] != -1:
```

- Checks if the current tile in the start state matches any tile in the goal state.
- Excludes the empty tile (`-1`).

python

```
h += (abs(j - i)) // 3 + (abs(j - i)) % 3
```

- This calculates the **Manhattan Distance**:
  - `(abs(j - i)) // 3` calculates vertical distance (row difference).
  - `(abs(j - i)) % 3` calculates horizontal distance (column difference).
- This value is added to the total heuristic cost `h`.

python

```
return h + g
```

- Returns the total heuristic cost (`h + g`).

## 5. Functions: Move Tiles

### 5.1 Moving Left

python

```
def moveleft(start, position):
    start[position], start[position - 1] = start[position - 1], start[position]
```

- Swaps the empty tile (`-1`) with the tile to its **left**.

## 5.2 Moving Right

python

```
def moveright(start, position):
    start[position], start[position + 1] = start[position + 1], start[position]
```

- Swaps the empty tile with the tile to its **right**.

## 5.3 Moving Up

python

```
def moveup(start, position):
    start[position], start[position - 3] = start[position - 3], start[position]
```

- Swaps the empty tile with the tile **above** it.

## 5.4 Moving Down

python

```
def movedown(start, position):
    start[position], start[position + 3] = start[position + 3], start[position]
```

- Swaps the empty tile with the tile **below** it.

# 6. Function: `movetile(start, goal)`

python

```
def movetile(start, goal):
```

- This function decides the **best move** based on the lowest heuristic value.

## 6.1 Calculating Heuristic for Each Move

- Four copies (`t1`, `t2`, `t3`, `t4`) represent four possible moves (left, right, up, down).
- Each move's heuristic value is calculated using the `heuristic` function.

## 6.2 Selecting the Best Move

- The minimum heuristic value is determined using `min()`.
  - The corresponding move is executed.
- 

## 7. Function: `solveEight(start, goal)`

```
python  
  
def solveEight(start, goal):
```

- This is a **recursive function** that solves the 8-puzzle using a **Greedy Best-First Search**.
- 

### 7.1 Move Tile

```
python  
  
    movetile(start, goal)
```

- Moves the empty tile in the best direction.
- 

### 7.2 Display the Board

```
python  
  
    print_board(start)
```

- Prints the current puzzle state.
- 

### 7.3 Check for Solution

```
python  
  
    if heuristic(start, goal) == g:
```

- If the heuristic value matches the move count `g`, the puzzle is solved.
- 

### 7.4 Recursive Call

```
python  
  
    solveEight(start, goal)
```

- Recursively calls itself until the puzzle is solved.
- 

## 8. Main Function

python

```
def main():
```

- Prompts user for initial and goal state of the puzzle.
- Checks if the puzzle is solvable.
- If solvable, solves it using `solveEight()`.
- If not, displays "Not possible to solve".

✅ **Would you like me to show you an optimized version of this code with better structure, clear comments, and improved performance?** 🚀



```
import heapq

def dijkstra(graph, start):
    # Distance to all nodes initially infinity
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    # Priority queue to get the node with the smallest distance
    pq = [(0, start)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        # Skip if we already found a better path
        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node]:
            distance = current_distance + weight

            # If new distance is smaller, update it
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return distances

# Define the graph as an adjacency list
graph = {
    'A': [('B', 4), ('C', 5)],
    'B': [('A', 4), ('C', 11), ('D', 9), ('E', 7)],
    'C': [('A', 5), ('B', 11), ('E', 3)],
    'D': [('B', 9), ('F', 2)],
    'E': [('B', 7), ('C', 3), ('F', 6)],
```

```

    'F': [('D', 2), ('E', 6)]
}

# Run Dijkstra from source node 'A'
distances = dijkstra(graph, 'A')

print("Shortest distances from A:")
for node in distances:
    print(f"{node}: {distances[node]}")

```

## In-Depth Line-by-Line Explanation of the Dijkstra Algorithm Code

---

### 1. Importing the `heapq` Module

```

python

import heapq

```

- `heapq`: A Python module providing an implementation of the **min-heap (priority queue)**.
  - It is used here because Dijkstra's algorithm requires efficient access to the smallest distance node.
- 

### 2. Function: `dijkstra(graph, start)`

```

python

def dijkstra(graph, start):

```

- This function implements **Dijkstra's algorithm**, which finds the **shortest paths from a starting node** to all other nodes in a weighted graph.
  - `graph`: A dictionary representing the graph as an adjacency list.
    - **Key**: The node.
    - **Value**: A list of tuples, where each tuple is `(neighbor, weight)`.
  - `start`: The starting node for finding shortest paths.
- 

### 3. Initializing Distances

```

python

distances = {node: float('inf') for node in graph}
distances[start] = 0

```

- `distances`: A dictionary that keeps track of the shortest known distance from the start node to each other node.
- `float('inf')`: Represents infinity, meaning all nodes initially have an unknown distance (infinity).



- `distances[start] = 0`: The starting node's distance is set to `0` because the distance from a node to itself is zero.

### Why Infinity?

- Setting the distance to infinity for all other nodes ensures that any discovered path will be shorter, making it possible to replace these values with actual shortest distances.

## 4. Initializing the Priority Queue

```
python

pq = [(0, start)]
```

- `pq`: A priority queue (min-heap) initialized with the starting node.
- The tuple `(0, start)` means:
  - `0`: The distance to the start node is zero.
  - `start`: The starting node itself.
- **Reason:** The priority queue will always pop the node with the smallest distance.

## 5. Main Loop: Exploring Nodes

```
python

while pq:
```

- `while pq:`: Runs until the priority queue is empty.
- **Reason:** This loop allows the algorithm to explore the graph until all reachable nodes are processed.

## 6. Popping the Nearest Node

```
python

current_distance, current_node = heapq.heappop(pq)
```

- `heapq.heappop(pq)`: Removes and returns the smallest distance node from the priority queue.
- `current_distance`: The smallest distance to the `current_node` found so far.

## 7. Skip if Already Processed

```
python
```

```
if current_distance > distances[current_node]:
    continue
```

- If the current distance is greater than the already known shortest distance to this node, we skip it.
- **Reason:** A better path has already been found.

## 8. Exploring Neighboring Nodes

```
python
```

```
for neighbor, weight in graph[current_node]:
```

- Loops through all neighboring nodes of the `current_node``.
- `neighbor``: The connected node.
- `weight``: The weight (distance) of the edge connecting `current_node`` to `neighbor``.

## 9. Calculating Distance

```
python
```

```
distance = current_distance + weight
```

- The **new distance** is calculated as:
  - `current_distance``: The distance to reach the `current_node``.
  - `weight``: The distance to the `neighbor`` from the `current_node``.

## 10. Updating Shortest Distance

```
python
```

```
if distance < distances[neighbor]:
    distances[neighbor] = distance
    heapq.heappush(pq, (distance, neighbor))
```

- If this new distance is **smaller than the previously known distance**:
  - The shortest distance to this `neighbor`` is updated in the `distances`` dictionary.
  - The `neighbor`` and its new distance are pushed to the priority queue.
- `heapq.heappush()``: Adds a new element to the priority queue.

## 11. Returning the Result

```
python
```

```
return distances
```

- The function returns the `distances` dictionary, which now contains the shortest distances from the starting node to all other nodes.

## 12. Defining the Graph (Adjacency List)

```
python
```

```
graph = {
    'A': [('B', 4), ('C', 5)],
    'B': [('A', 4), ('C', 11), ('D', 9), ('E', 7)],
    'C': [('A', 5), ('B', 11), ('E', 3)],
    'D': [('B', 9), ('F', 2)],
    'E': [('B', 7), ('C', 3), ('F', 6)],
    'F': [('D', 2), ('E', 6)]
}
```

- The graph is represented as an **adjacency list**, where:
  - **Keys:** Nodes ('A', 'B', 'C', 'D', 'E', 'F').
  - **Values:** Lists of tuples, each containing a connected neighbor and the weight (distance) of the edge.
- This graph is **undirected**, meaning each connection appears twice (e.g., ('A', 'B') and ('B', 'A')).

## 13. Running Dijkstra's Algorithm

```
python
```

```
distances = dijkstra(graph, 'A')
```

- Calls the `dijkstra` function with the graph and the starting node 'A'.
- The result (shortest distances) is stored in the `distances` dictionary.

## 14. Displaying the Shortest Distances

```
python
```

```
print("Shortest distances from A:")
for node in distances:
    print(f"{node}: {distances[node]}")
```

- This prints the shortest distance from the starting node 'A' to each other node.
- **Formatted Output:** Each node is displayed with its calculated shortest distance.



## 15. Sample Output

vbnet

Shortest distances from A:

A: 0  
B: 4  
C: 5  
D: 13  
E: 8  
F: 14

- `A`: The starting node, distance is `0`.
- `B`: The shortest distance is `4` ( $A \rightarrow B$ ).
- `C`: The shortest distance is `5` ( $A \rightarrow C$ ).
- `D`: The shortest distance is `13` ( $A \rightarrow B \rightarrow D$ ).
- `E`: The shortest distance is `8` ( $A \rightarrow C \rightarrow E$ ).
- `F`: The shortest distance is `14` ( $A \rightarrow B \rightarrow D \rightarrow F$ ).

 **Would you like me to show you an optimized version of this code with improved structure and clarity?** 



```
class DisjointSet:
    def __init__(self, n):
        self.parent = list(range(n))

    def find(self, x):
        if self.parent[x] != x:
            self.parent[x] = self.find(self.parent[x]) # Path compression
        return self.parent[x]

    def union(self, x, y):
        xroot = self.find(x)
        yroot = self.find(y)
        if xroot == yroot:
            return False
        self.parent[yroot] = xroot
        return True

def kruskal_mst(edges, n):
    # Sort edges based on weight
    edges.sort(key=lambda x: x[2])

    ds = DisjointSet(n)
    mst = []
    total_weight = 0
```

```

for u, v, weight in edges:
    if ds.union(u, v):
        mst.append((u, v, weight))
        total_weight += weight

return mst, total_weight

# Define edges (u, v, weight)
edges = [
    (0, 1, 4),
    (0, 7, 8),
    (1, 2, 8),
    (1, 7, 11),
    (2, 3, 7),
    (2, 8, 2),
    (2, 5, 4),
    (3, 4, 9),
    (3, 5, 14),
    (4, 5, 10),
    (5, 6, 2),
    (6, 7, 1),
    (6, 8, 6),
    (7, 8, 7),
]

n = 9 # Number of vertices (0 to 8)

mst, total_weight = kruskal_mst(edges, n)

print("Edges in MST:")
for u, v, weight in mst:
    print(f"{u} - {v}: {weight}")

print(f"Total weight of MST: {total_weight}")

```

## In-Depth Line-by-Line Explanation of the Kruskal's Algorithm Code using Disjoint Set (Union-Find)

### 1. Class: `DisjointSet` (Union-Find Data Structure)

```

python

class DisjointSet:

```

- This is a **Disjoint Set (Union-Find)** class, an efficient data structure used to keep track of connected components.
- Essential for cycle detection in Kruskal's algorithm.

## 2. Initializing the Disjoint Set

python

```
def __init__(self, n):
    self.parent = list(range(n))
```

- `__init__` (**Constructor**):
  - Takes the total number of elements (`n`) as an argument.
  - Initializes a list `self.parent`, where each element is its own parent.
  - This means each node is initially its own separate component.

### Example:

- For `n = 5`, the parent list would be:  
`[0, 1, 2, 3, 4]`
  - Node `0` is its own parent, node `1` is its own parent, and so on.

## 3. Method: `find(self, x)`

python

```
def find(self, x):
```

- This method finds the **root parent (representative)** of a given node `x`.
- It uses **Path Compression**, which is an optimization to flatten the structure.

python

```
if self.parent[x] != x:
    self.parent[x] = self.find(self.parent[x]) # Path compression
```

- If the node is **not its own parent**, it means it is part of a larger component.
- The method recursively finds the root parent.
- **Path Compression**: This makes future lookups faster by directly connecting nodes to their root.

### Example:

- Before Path Compression:

```
0 → 1 → 2 → 3
```

- After Path Compression:

```
0 → 3
1 → 3
2 → 3
```

python

```
return self.parent[x]
```

- Finally, it returns the root parent of the node.

#### 4. Method: `union(self, x, y)`

python

```
def union(self, x, y):
```

- This method **unites two components (x and y)** by connecting their root parents.

python

```
xroot = self.find(x)
yroot = self.find(y)
```

- Finds the root of each component using the `find` method.

python

```
if xroot == yroot:
    return False
```

- If both nodes share the same root, they are already in the same component (a cycle), so no union is performed.

python

```
self.parent[yroot] = xroot
return True
```

- If the nodes are in different components, one root is assigned as the parent of the other.
- This effectively merges the two components.

#### ✅ Disjoint Set Recap:

- The `find` method determines the root of a component.
- The `union` method connects two components, ensuring they become one.

## 5. Function: `kruskal\_mst(edges, n)`

python

```
def kruskal_mst(edges, n):
```

- This function implements **Kruskal's Algorithm** to find the **Minimum Spanning Tree (MST)** of a graph.
- `edges`: A list of edges, each defined as `(u, v, weight)`.
- `n`: The total number of vertices in the graph.

## 6. Sorting Edges by Weight

python

```
edges.sort(key=lambda x: x[2])
```

- The edges are sorted in **ascending order of their weight**.
- **Why?** Kruskal's algorithm always adds the smallest edge to the MST first.

### Example:

- If edges are:

css

```
[(0, 1, 4), (0, 7, 8), (1, 2, 8)]
```

- After sorting:

css

```
[(0, 1, 4), (0, 7, 8), (1, 2, 8)]
```

## 7. Initializing Data Structures

python

```
ds = DisjointSet(n)
mst = []
total_weight = 0
```

- `ds`: An instance of the `DisjointSet` class for cycle detection.
- `mst`: A list to store the edges of the Minimum Spanning Tree.
- `total_weight`: A counter to track the total weight of the MST.



## 8. Building the MST

python

```
for u, v, weight in edges:
```

- Loops through each edge (in ascending order).

python

```
if ds.union(u, v):
```

- Checks if the two vertices (`u` and `v`) can be connected without forming a cycle.
- If yes:
  - The edge is added to the MST.
  - The weight is added to the total weight.

python

```
mst.append((u, v, weight))
total_weight += weight
```

- The edge is added to the MST list.
- The total weight is incremented.

## 9. Returning the MST and Total Weight

python

```
return mst, total_weight
```

- The function returns:
  - The list of edges in the MST.
  - The total weight of the MST.

## 10. Defining the Graph Edges

python

```
edges = [
    (0, 1, 4),
    (0, 7, 8),
    (1, 2, 8),
    (1, 7, 11),
    (2, 3, 7),
    (2, 8, 2),
    (2, 5, 4),
```

```
(3, 4, 9),
(3, 5, 14),
(4, 5, 10),
(5, 6, 2),
(6, 7, 1),
(6, 8, 6),
(7, 8, 7),
]
```

- The graph is represented as a list of edges, each with:
  - Start node (`u`), End node (`v`), and Weight.

## 11. Number of Vertices

```
python

n = 9 # Number of vertices (0 to 8)
```

- The graph has `9` vertices (labeled from `0` to `8`).

## 12. Running Kruskal's Algorithm

```
python

mst, total_weight = kruskal_mst(edges, n)
```

- The `kruskal\_mst` function is called with the edges and the number of vertices.
- It returns:
  - `mst`: List of edges in the MST.
  - `total\_weight`: The total weight of the MST.

## 13. Displaying the MST

```
python

print("Edges in MST:")
for u, v, weight in mst:
    print(f"{u} - {v}: {weight}")

print(f"Total weight of MST: {total_weight}")
```

- Prints each edge in the MST.
- Prints the total weight of the MST.

## 14. Sample Output

yaml

```
Edges in MST:
6 - 7: 1
2 - 8: 2
5 - 6: 2
0 - 1: 4
2 - 5: 4
0 - 7: 8
2 - 3: 7
3 - 4: 9
Total weight of MST: 37
```

✓ **Would you like me to show you a fully optimized version of this code with better performance and clarity?** 🚀



```
import sys

class Graph():
    def __init__(self, vertices):
        self.V = vertices
        self.graph = [[0 for column in range(vertices)]
                       for row in range(vertices)]

    def printMST(self, parent):
        print("Edge \tWeight")
        total_weight = 0
        for i in range(1, self.V):
            weight = self.graph[parent[i]][i]
            print(parent[i] + 1, "-", i + 1, "\t", weight)
            total_weight += weight
        print("Total weight of MST:", total_weight)

    def minKey(self, key, mstSet):
        min = sys.maxsize
        min_index = -1

        for v in range(self.V):
            if key[v] < min and not mstSet[v]:
                min = key[v]
                min_index = v

        return min_index

    def primMST(self):
        key = [sys.maxsize] * self.V
        parent = [None] * self.V
        key[0] = 0
        mstSet = [False] * self.V
```

```

parent[0] = -1

for _ in range(self.V):
    u = self.minKey(key, mstSet)
    mstSet[u] = True

    for v in range(self.V):
        if self.graph[u][v] > 0 and not mstSet[v] and key[v] > self.graph[u][v]:
            key[v] = self.graph[u][v]
            parent[v] = u

self.printMST(parent)

if __name__ == '__main__':
    g = Graph(6)
    g.graph = [
        [0, 2, 0, 1, 4, 0], # Node 1
        [2, 0, 3, 0, 0, 7], # Node 2
        [0, 3, 0, 5, 0, 8], # Node 3
        [1, 0, 5, 0, 9, 0], # Node 4
        [4, 0, 0, 9, 0, 0], # Node 5
        [0, 7, 8, 0, 0, 0] # Node 6
    ]

    g.primMST()

```

## In-Depth Line-by-Line Explanation of Prim's Algorithm Code using Adjacency Matrix

### 1. Class: `Graph`

python

```
class Graph():
```

- Defines a `Graph` class to represent an undirected graph using an **Adjacency Matrix**.
- This graph class is specifically for implementing **Prim's Minimum Spanning Tree (MST)** algorithm.

### 2. Initialization Method: `\_\_init\_\_`

python

```

def __init__(self, vertices):
    self.V = vertices
    self.graph = [[0 for column in range(vertices)] for row in range(vertices)]

```

- The `\_\_init\_\_` method initializes:

- `self.V`: The total number of vertices in the graph.
- `self.graph`: A `V x V` matrix (2D list) representing the graph.
  - Each value in the matrix is initially `0`, meaning no edges exist between nodes.
  - The graph is represented as an **adjacency matrix**, where `graph[i][j]` is the weight of the edge between node `i` and node `j`.

✓ **Example: If `vertices = 4`, the initial graph is:**

```
0 0 0 0
0 0 0 0
0 0 0 0
0 0 0 0
```

### 3. Method: `printMST(self, parent)`

python

```
def printMST(self, parent):
    print("Edge \tWeight")
    total_weight = 0
```

- This method is used to print the edges of the **Minimum Spanning Tree (MST)** and its total weight.
- `parent`: A list where each index represents a vertex, and the value at each index is the parent of that vertex in the MST.

### 4. Printing the MST Edges

python

```
for i in range(1, self.V):
    weight = self.graph[parent[i]][i]
    print(parent[i] + 1, "-", i + 1, "\t", weight)
    total_weight += weight
```

- The loop starts from `1` because the root of the MST has no parent (`parent[0] = -1`).
- For each vertex `i`:
  - It prints the edge between the vertex and its parent (`parent[i]`).
  - The weight of this edge is retrieved from the adjacency matrix.
  - The total weight of the MST is calculated by summing these edge weights.

✓ **Example Output:**

```
nginx
```

Edge	Weight
1 - 4	1
1 - 2	2
2 - 3	3
1 - 5	4
2 - 6	7

## 5. Printing Total Weight

```
python
```

```
print("Total weight of MST:", total_weight)
```

- This line prints the total weight of all edges in the MST.

## 6. Method: `minKey(self, key, mstSet)`

```
python
```

```
def minKey(self, key, mstSet):
```

- A helper method that returns the vertex with the **minimum key value** (smallest weight) that is **not yet included in the MST (mstSet)**.

## 7. Finding the Minimum Key

```
python
```

```
min = sys.maxsize
min_index = -1
```

- Initializes:
  - `min` to the maximum possible value (`sys.maxsize`).
  - `min\_index` to `-1`, which will be updated to the index of the minimum key.

```
python
```

```
for v in range(self.V):
    if key[v] < min and not mstSet[v]:
        min = key[v]
        min_index = v
```

- Loops through all vertices:
  - If the vertex has a smaller key value than the current `min` and is **not yet in the MST**, it updates `min` and `min\_index`.

- **Why?** This helps find the next vertex to be included in the MST.

### ✓ Example:

- If `key = [0, 2, 3, 1, 4, 7]` and `mstSet = [True, False, False, False, False, False]`, it will choose:
  - Vertex `3` (value `1`) because it is the smallest among those not in MST.

python

```
return min_index
```

- Returns the index of the vertex with the minimum key.

## 8. Method: `primMST(self)`

python

```
def primMST(self):
```

- This method implements **Prim's Algorithm** to build the MST.

## 9. Initialization

python

```
key = [sys.maxsize] * self.V
parent = [None] * self.V
key[0] = 0
mstSet = [False] * self.V
parent[0] = -1
```

- `key`: A list that stores the minimum weight edge for each vertex. Initially, all values are set to infinity (`sys.maxsize`).
- `parent`: A list that keeps track of the parent of each vertex in the MST.
- `mstSet`: A list of boolean values indicating whether a vertex is included in the MST (`True` means included).
- `key[0] = 0`: The starting vertex is set to `0` (arbitrary).
- `parent[0] = -1`: The first vertex has no parent.

## 10. Building the MST

python

```
for _ in range(self.V):
```

- Loops `V` times (once for each vertex).

python

```
u = self.minKey(key, mstSet)
mstSet[u] = True
```

- Finds the vertex with the smallest key value (`u`) that is not in the MST and includes it in the MST by marking `mstSet[u] = True`.

## 11. Updating Neighboring Vertices

python

```
for v in range(self.V):
    if self.graph[u][v] > 0 and not mstSet[v] and key[v] > self.graph[u][v]:
        key[v] = self.graph[u][v]
        parent[v] = u
```

- For each vertex `v`:
  - Checks if:
    - There is an edge between `u` and `v` (`graph[u][v] > 0`).
    - The vertex `v` is not yet in the MST.
    - The current key value of `v` is greater than the weight of the edge (`graph[u][v]`).
  - If true:
    - Updates `key[v]` to the weight of this edge.
    - Sets `parent[v]` to `u` (making `u` the parent of `v` in MST).

## ✓ 12. Printing the MST

python

```
self.printMST(parent)
```

- After constructing the MST, it calls the `printMST` method to display the MST.

## 13. Main Function



python

```

if __name__ == '__main__':
    g = Graph(6)
    g.graph = [
        [0, 2, 0, 1, 4, 0], # Node 1
        [2, 0, 3, 0, 0, 7], # Node 2
        [0, 3, 0, 5, 0, 8], # Node 3
        [1, 0, 5, 0, 9, 0], # Node 4
        [4, 0, 0, 9, 0, 0], # Node 5
        [0, 7, 8, 0, 0, 0]  # Node 6
    ]
    g.primMST()

```

- Creates a graph with 6 vertices.
- Defines an adjacency matrix with the edge weights.
- Calls the `primMST()` function to calculate and display the MST.

✅ **Would you like me to show you a fully optimized and modular version of Prim's Algorithm with more flexibility?** 🚀



```

def selectionSort(arr):
    n = len(arr)

    for i in range(n):
        min_idx = i
        for j in range(i+1, n):
            if arr[j] < arr[min_idx]:
                min_idx = j

        # Swap the found minimum element with the first element
        arr[i], arr[min_idx] = arr[min_idx], arr[i]

    return arr

# Main function
if __name__ == '__main__':
    arr = list(map(int, input("Enter the elements separated by space: ").split()))
    print("Original array:", arr)
    sorted_arr = selectionSort(arr)
    print("Sorted array:", sorted_arr)

```

🚀 **In-Depth Line-by-Line Explanation of Selection Sort Code**

## 1. Function Definition: `selectionSort`

```
python
```

```
def selectionSort(arr):
```

- Defines a function named `selectionSort` that takes an array (`arr`) as input.
- This function implements the **Selection Sort Algorithm**, which is an in-place, comparison-based sorting algorithm.

## 2. Calculating Length of the Array

```
python
```

```
n = len(arr)
```

- Calculates the length of the input array and stores it in variable `n`.
- This is useful to avoid recalculating the length during each loop iteration.

## 3. Outer Loop: Iterating Over the Array

```
python
```

```
for i in range(n):
```

- The outer loop starts from the first element (`i = 0`) and goes to the last element (`i = n-1`).
- At each step, the algorithm assumes the current element is the smallest.

### ✓ Example:

- If the array is `[64, 25, 12, 22, 11]` and `i = 0`, it assumes `64` is the smallest.

## 4. Initializing Minimum Index

```
python
```

```
min_idx = i
```

- Initializes the `min_idx` variable to the current index (`i`).
- This variable keeps track of the index of the smallest element found in the unsorted portion of the array.

## 5. Inner Loop: Finding the Minimum Element

python

```
for j in range(i+1, n):
```

- The inner loop starts from the next element (`i+1`) and goes to the end of the array.
- Its goal is to find the smallest element in the remaining unsorted portion of the array.

## 6. Comparing Elements

python

```
if arr[j] < arr[min_idx]:
    min_idx = j
```

- Checks if the current element (`arr[j]`) is smaller than the element at the current minimum index (`arr[min\_idx]`).
- If true, it updates `min\_idx` to `j`, making this new position the index of the smallest element.

### ✓ Example: (Step-by-step)

- Array: `[64, 25, 12, 22, 11]`
- For `i = 0`:
  - Compares `25 < 64`, so `min\_idx = 1`.
  - Compares `12 < 25`, so `min\_idx = 2`.
  - Compares `22 < 12` (no change).
  - Compares `11 < 12`, so `min\_idx = 4`.

## 7. Swapping the Smallest Element with the Current Position

python

```
arr[i], arr[min_idx] = arr[min_idx], arr[i]
```

- After finding the smallest element in the unsorted portion:
  - Swaps it with the first unsorted element (`arr[i]`).
- This ensures that the smallest element is placed in the correct position in the sorted portion.

### ✓ Example: Swap Step

- Array: `[64, 25, 12, 22, 11]`
- After the first pass (`i = 0`), the smallest element `11` is swapped:

```
csharp
```

```
[11, 25, 12, 22, 64]
```

- This process continues until the array is fully sorted.

## 8. Returning the Sorted Array

```
python
```

```
return arr
```

- Returns the sorted array after the sorting process is complete.

## 9. Main Function: User Input and Execution

```
python
```

```
if __name__ == '__main__':
```

- This block ensures that the following code only runs if this script is executed directly (not imported).

## 10. Taking User Input

```
python
```

```
arr = list(map(int, input("Enter the elements separated by space: ").split()))
```

- Takes user input in a single line (space-separated values).
- The `map(int, ...)` part converts each value to an integer.
- The `list()` function creates a list from these integer values.

### Example:

- Input:

```
csharp
```

```
Enter the elements separated by space: 64 25 12 22 11
```

- Output:

```
python
```

```
[64, 25, 12, 22, 11]
```

## 11. Displaying the Original Array

python

```
print("Original array:", arr)
```

- Prints the array before sorting.

## 12. Sorting the Array Using Selection Sort

python

```
sorted_arr = selectionSort(arr)
```

- Calls the `selectionSort` function to sort the array.
- The sorted array is stored in `sorted_arr`.

## 13. Displaying the Sorted Array

python

```
print("Sorted array:", sorted_arr)
```

- Prints the sorted array.



## Dry Run Example:

### Input:

csharp

Enter the elements separated by space: 64 25 12 22 11

### Step-by-Step Execution:

- **Initial Array:** `[64, 25, 12, 22, 11]`

#### Pass 1: (`i = 0`)

- Minimum Element: `11` (at index 4)
- Swap `11` with `64`:

csharp

```
[11, 25, 12, 22, 64]
```

### Pass 2: ( $i = 1$ )

- Minimum Element: `12` (at index 2)
- Swap `12` with `25`:

```
csharp
[11, 12, 25, 22, 64]
```

### Pass 3: ( $i = 2$ )

- Minimum Element: `22` (at index 3)
- Swap `22` with `25`:

```
csharp
[11, 12, 22, 25, 64]
```

### Pass 4: ( $i = 3$ )

- No swap needed, already in place:

```
csharp
[11, 12, 22, 25, 64]
```

### Pass 5: ( $i = 4$ )

- Only one element left, already sorted.

### Final Output:

```
less
Original array: [64, 25, 12, 22, 11]
Sorted array: [11, 12, 22, 25, 64]
```

## ✓ Why is Selection Sort Not Efficient for Large Arrays?

- Time Complexity:  $O(n^2)$ 
  - Outer loop runs  $n$  times.
  - Inner loop (finding minimum) runs  $n - i$  times for each outer loop.
- Space Complexity:  $O(1)$  (in-place, no extra memory).
- Suitable for small arrays, but slow for large ones.

✓ Would you like me to show you a highly optimized and modular version of Selection Sort with both ascending and descending order options? 🚀



```
def issafe(arr,x,y,n):
    for row in range(x):
        if arr[row][y] ==1:
            # Checking column attack
            return False
    row = x
    col = y
    #Checking Diagonal Attack
    while row>=0 and col>=0:
        if arr[row][col]==1:
            return False
        row-=1
        col-=1

    row = x
    col = y
    #Checking Anti Diagonal Attack
    while row>=0 and col<n:
        if arr[row][col]==1:
            return False
        row-=1
        col+=1

    return True

def nQueen(arr,x,n):
    if x>=n:
        return True

    for col in range(n):
        if issafe(arr,x,col,n):
            arr[x][col]=1
            if nQueen(arr,x+1,n):
                return True
            arr[x][col] = 0

    return False

def main():
    n = int(input("Enter number of Queens : "))
    arr = [[0]*n for i in range(n)]

    if nQueen(arr,0,n):
        for i in range(n):
            for j in range(n):
```

```

        print(arr[i][j],end=" ")
    print()

if __name__ == '__main__':
    main()

```

## In-Depth Line-by-Line Explanation of N-Queens Solution Code

### 1. `is\_safe` Function:

```

python

def is_safe(arr, x, y, n):

```

- This function checks if it's safe to place a queen at position `(x, y)` on the chessboard.
- The board is represented by `arr`, where `arr[i][j]` is `1` if there is a queen and `0` otherwise.
- `x` and `y` represent the current row and column of the queen being placed.
- `n` is the size of the board (`n x n`).

### 2. Checking Column Attack:

```

python

for row in range(x):
    if arr[row][y] == 1:
        return False

```

- Loops through all rows above the current row `x`.
- If a queen is found in the same column `y`, it returns `False`, meaning it's not safe to place the queen at `(x, y)` due to a column attack.

### 3. Checking Diagonal Attack:

```

python

row = x
col = y
while row >= 0 and col >= 0:
    if arr[row][col] == 1:
        return False
    row -= 1
    col -= 1

```

- This loop checks the **left-upper diagonal** (top-left direction) of the board.
- It moves diagonally by decrementing both the `row` and `col`.
- If a queen is found along this diagonal, it returns `False`.



## 4. Checking Anti-Diagonal Attack:

```
python

row = x
col = y
while row >= 0 and col < n:
    if arr[row][col] == 1:
        return False
    row -= 1
    col += 1
```

- This loop checks the **right-upper diagonal** (top-right direction) of the board.
- It moves diagonally by decrementing the `row` and incrementing the `col`.
- If a queen is found along this diagonal, it returns `False`.

## 5. Return `True` if No Attacks Found:

```
python

return True
```

- If no column or diagonal attacks are detected, the function returns `True`, indicating that it's safe to place a queen at `(x, y)`.

## 6. `nQueen` Function:

```
python

def nQueen(arr, x, n):
```

- This function is used to solve the N-Queens problem.
- It attempts to place queens row by row starting from row `x` on the board.

## 7. Base Case:

```
python

if x >= n:
    return True
```

- The base case checks if `x` (the current row) has reached or exceeded `n`.
- If so, all queens have been successfully placed, and the function returns `True` to indicate the solution has been found.

## 8. Try Placing Queen in All Columns of Row `x`:

python

```
for col in range(n):
    if issafe(arr, x, col, n):
```

- The `for` loop tries placing a queen in each column `col` of the current row `x`.
- It calls `issafe(arr, x, col, n)` to check if placing the queen at `(x, col)` is safe.
- If it's safe, the queen is placed by setting `arr[x][col] = 1`.

## 9. Recursive Step:

python

```
if nQueen(arr, x + 1, n):
    return True
```

- After placing the queen, the function recursively tries to solve the problem for the next row (`x + 1`).
- If it finds a solution, it returns `True`.

## 10. Backtracking:

python

```
arr[x][col] = 0
```

- If placing a queen in column `col` of row `x` doesn't lead to a solution (i.e., the recursive call returns `False`), it removes the queen (backtracking) by setting `arr[x][col] = 0`.
- This allows the algorithm to try the next column in row `x`.

## 11. Return `False` if No Solution:

python

```
return False
```

- If no valid placement is found for row `x`, the function returns `False`, signaling that no solution exists for this configuration.

## 12. `main` Function:

python

```
def main():
    n = int(input("Enter number of Queens : "))
    arr = [[0] * n for i in range(n)]
```

- The `main` function starts by taking user input for the number of queens `n`.
- It initializes an `n x n` board (`arr`) with all zeros, representing an empty board.

### 13. Calling `nQueen` to Solve the Problem:

python

```
if nQueen(arr, 0, n):
```

- The function `nQueen` is called to start placing queens from row `0`.
- If it returns `True`, a solution is found.

### 14. Printing the Solution:

python

```
for i in range(n):
    for j in range(n):
        print(arr[i][j], end=" ")
    print()
```

- If a solution is found, the `arr` matrix (representing the board) is printed row by row.
- `1` represents a queen, and `0` represents an empty space.

### 15. Run Check for Execution:

python

```
if __name__ == '__main__':
    main()
```

- This condition ensures that the `main()` function is only executed when the script is run directly (not when imported as a module).

### ✓ Example Walkthrough:

For `n = 4` (4-Queens Problem):

- **Initial Board:**

csharp

```
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
[0, 0, 0, 0]
```

- **Solution (One of the Possible Configurations):**

```
0 0 0 1
1 0 0 0
0 0 1 0
0 1 0 0
```

### ✓ Time Complexity:

- Worst case:  $O(n!)$  due to the backtracking nature of the problem (since we try to place queens in all positions).
- Best case:  $O(n^2)$  when the solution is found early.

### ✓ Would you like me to explain how this algorithm can be optimized further? 🚀