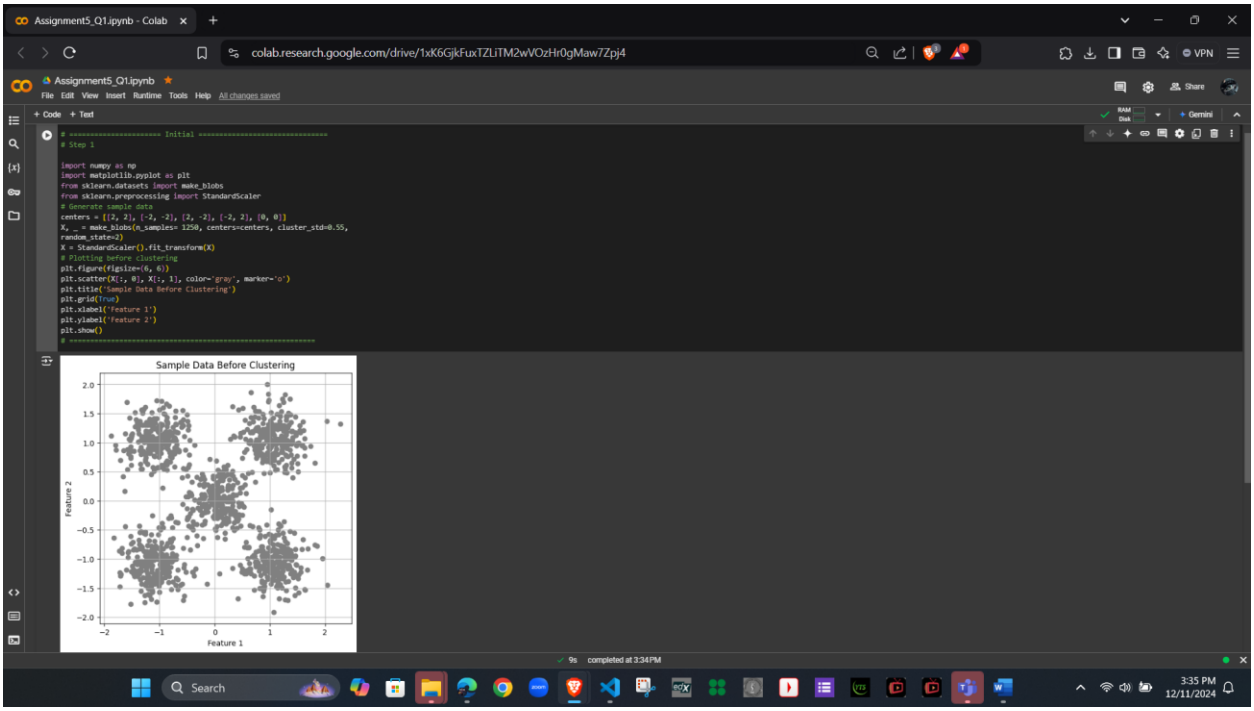
Assignment 5: Practical Application of DB-SCAN and A* Algorithm

Q1. DB-SCAN

Step 1: Visualize Data Before Clustering

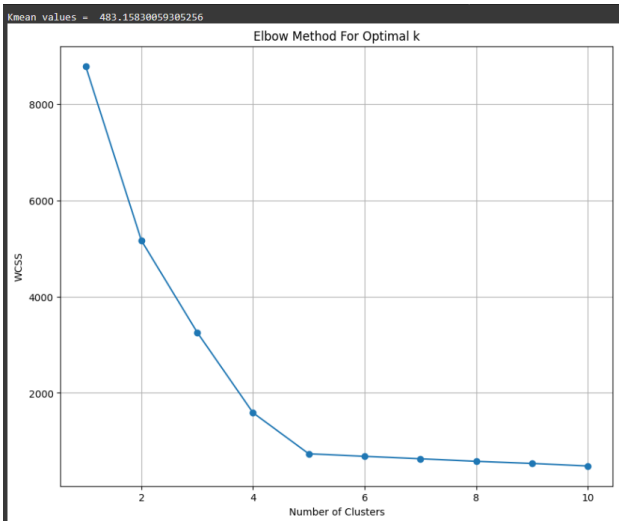
- Using the provided code and plotting the graph



Step 2: Implement K-means Clustering and Determine Optimal Clusters

- K-means and elbow method code
- Plotting the elbow method graph

```python
# Step 2
import numpy as np
import matplotlib.pyplot as plt
from sklearn.cluster import KMeans
from sklearn.datasets import make_blobs

# Generate synthetic data
X, _ = make_blobs(n_samples=1250, centers=centers, cluster_std=0.55, random_state=2)

# Calculate WCSS for different numbers of clusters
wcss = []
for i in range(1, 11):  # Testing 1 to 10 clusters
    kmeans = KMeans(n_clusters=i, init='k-means++', max_iter=300, n_init=10,
random_state=0)
    kmeans.fit(X)
    wcss.append(kmeans.inertia_)  # inertia_ is the WCSS for the model

print("Kmean values = ",kmeans.inertia_)
# Plotting the WCSS to observe the 'Elbow'
plt.figure(figsize=(10, 8))
plt.plot(range(1, 11), wcss, marker='o')
plt.title('Elbow Method For Optimal k')
plt.xlabel('Number of Clusters')
plt.ylabel('WCSS')
plt.grid(True)
plt.show()
```

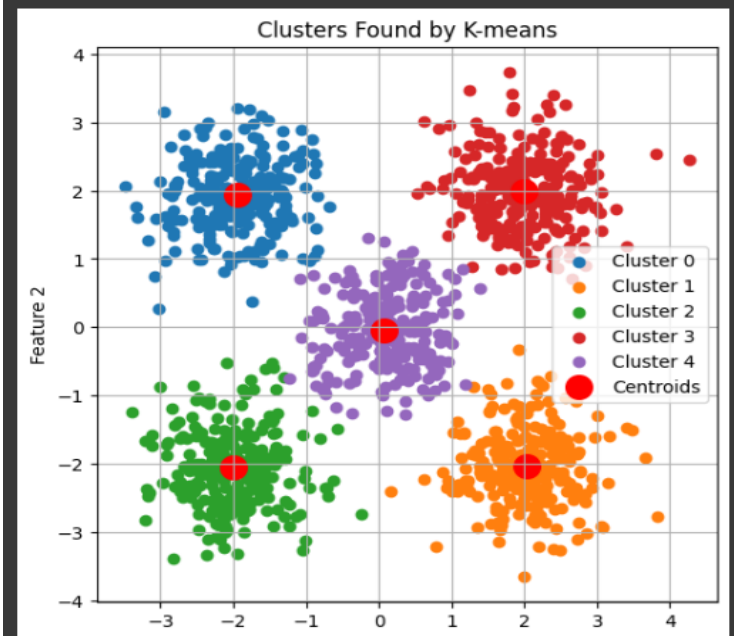


Kmean values = 483.15830059305256

Step 3: Visualize Data After Clustering

- Found by best K-means with K values from step 2
- Plotting the graph

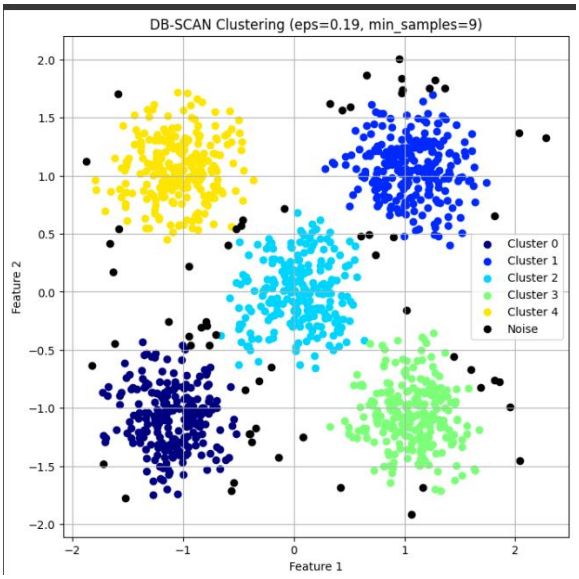
```
# step 3
optimal_k = 5  # Replace with the observed optimal k
kmeans = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Plotting clusters
plt.figure(figsize=(6, 6))
for i in range(optimal_k):
    plt.scatter(X[y_kmeans == i, 0], X[y_kmeans == i, 1], label=f'Cluster {i}')
plt.scatter(kmeans.cluster_centers_[:, 0], kmeans.cluster_centers_[:, 1], s=200, c='red', label='Centroids')
plt.title('Clusters Found by K-means')
plt.legend()
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



Step 4: Applied data from step1 with DB-SCAN

- Using the DB-SCAN algorithm to identify clusters
- Set epsilon parameters to 0.19
- Min_samples to 9
- Plotting the graph

```python
# step 4
from sklearn.cluster import DBSCAN

# Step 1: Generate Sample Data
centers = [[2, 2], [-2, -2], [2, -2], [-2, 2], [0, 0]]
X, _ = make_blobs(n_samples=1250, centers=centers, cluster_std=0.55, random_state=2)
X = StandardScaler().fit_transform(X)

# Step 2: Apply DB-SCAN Algorithm
eps = 0.19  # Epsilon parameter
min_samples = 9  # Minimum number of points to form a dense region

db = DBSCAN(eps=eps, min_samples=min_samples).fit(X)
labels = db.labels_

# Identify unique clusters
unique_labels = set(labels)

# Step 3: Visualize DB-SCAN Clustering Results
plt.figure(figsize=(8, 8))

# Plot each cluster
for label in unique_labels:
    if label == -1:  # Noise points
        color = 'black'
        label_name = 'Noise'
    else:
        # Use a colormap for clusters
        color = plt.cm.jet(float(label) / len(unique_labels))
        label_name = f'Cluster {label}'

    # Plot points for the current cluster
    plt.scatter(X[labels == label, 0], X[labels == label, 1],
                color=color, label=label_name, marker='o')

plt.title(f'DB-SCAN Clustering (eps={eps}, min_samples={min_samples})')
plt.legend()
plt.grid(True)
plt.xlabel('Feature 1')
plt.ylabel('Feature 2')
plt.show()
```



Step 5: Interpreting Results

Interpretation of Results: K-means vs. DB-SCAN

1. K-means Clustering:
    - Output: K-means groups the data into a predefined number of clusters (k). It assigns every data point to the nearest cluster center.
    - Visualization: You can observe well-separated clusters if the data is spherical and evenly distributed.
    - Key Observations:
        - K-means successfully clustered the dataset into distinct groups when the clusters were clearly separated.
        - It assigned all points to clusters, meaning no noise was detected.
        - The elbow method was helpful in determining the optimal number of clusters.
2. DB-SCAN Clustering:
    - Output: DB-SCAN forms clusters based on density and marks points that don't belong to any cluster as noise (-1).
    - Visualization: DB-SCAN effectively separated clusters and identified outliers or noise points in the dataset.
    - Key Observations:
        - DB-SCAN handled non-spherical clusters well and separated clusters based on density rather than distance from the center.
        - It detected noise in areas where the density was too low to form a cluster.

Pros and Cons of Each Algorithm:

1. **K-means Clustering:**
    - Pros:
        - Fast and computationally efficient for large datasets.
        - Works well for spherical, well-separated clusters.
        - Easy to understand and implement.
    - Cons:
        - Requires predefining the number of clusters (k).
        - Sensitive to initialization of centroids and outliers.
        - Poor performance on datasets with non-spherical or overlapping clusters.
2. **DB-SCAN Clustering:**
    - Pros:
        - Can detect clusters of arbitrary shapes.
        - Identifies noise and handles outliers effectively.
        - Does not require the number of clusters to be predefined.
    - Cons:
        - Sensitive to the choice of eps and min_samples parameters.
        - Struggles with datasets that have varying densities.
        - Computationally expensive for large datasets.

Additional code:

```
# Number of clusters found by DB-SCAN (excluding noise)
num_clusters_dbscan = len(set(labels)) - (1 if -1 in labels else 0)
num_noise_points = list(labels).count(-1)

print(f"Number of clusters found by DB-SCAN: {num_clusters_dbscan}")
print(f"Number of noise points identified: {num_noise_points}")

# Compare K-means clusters and DB-SCAN clusters
optimal_k = 5  # Replace with the actual optimal k from the elbow method
kmeans = KMeans(n_clusters=optimal_k, init='k-means++', max_iter=300, n_init=10, random_state=0)
y_kmeans = kmeans.fit_predict(X)

# Evaluate clustering consistency
print(f"Number of clusters found by K-means: {optimal_k}")
```

```
Number of clusters found by DB-SCAN: 5
Number of noise points identified: 63
Number of clusters found by K-means: 5
```

Number of clusters found by DB-SCAN: 5

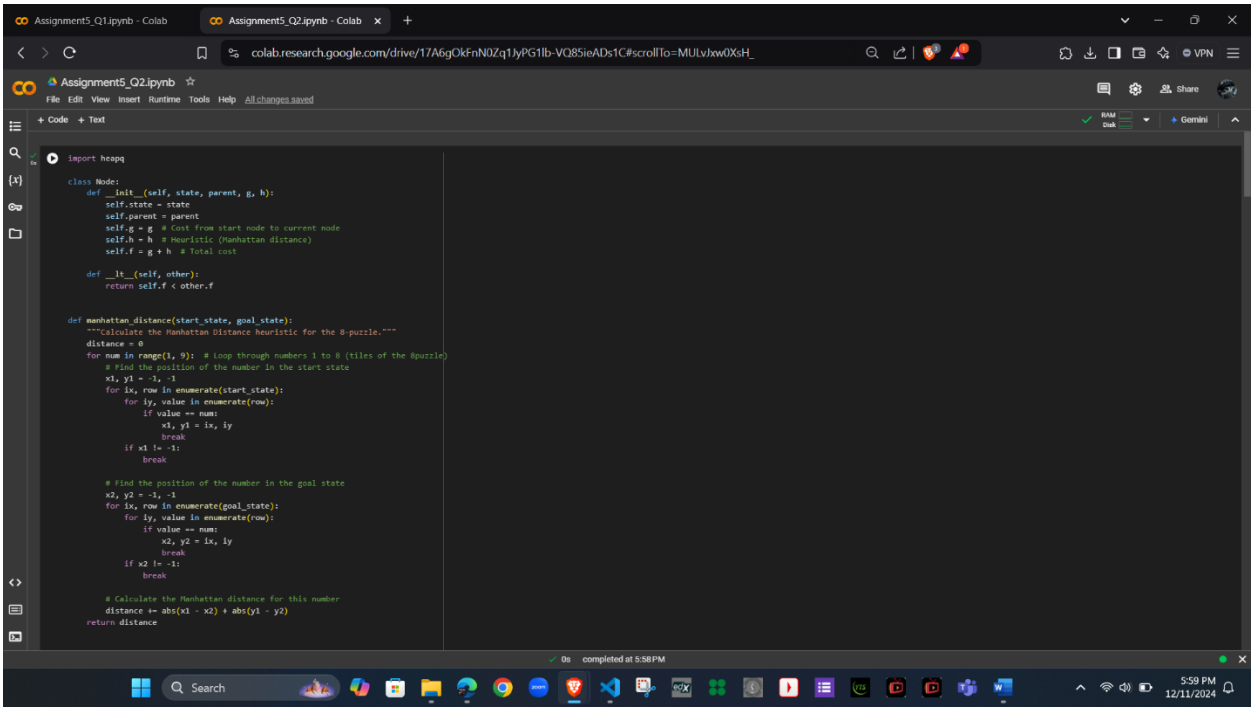
Number of noise points identified: 63

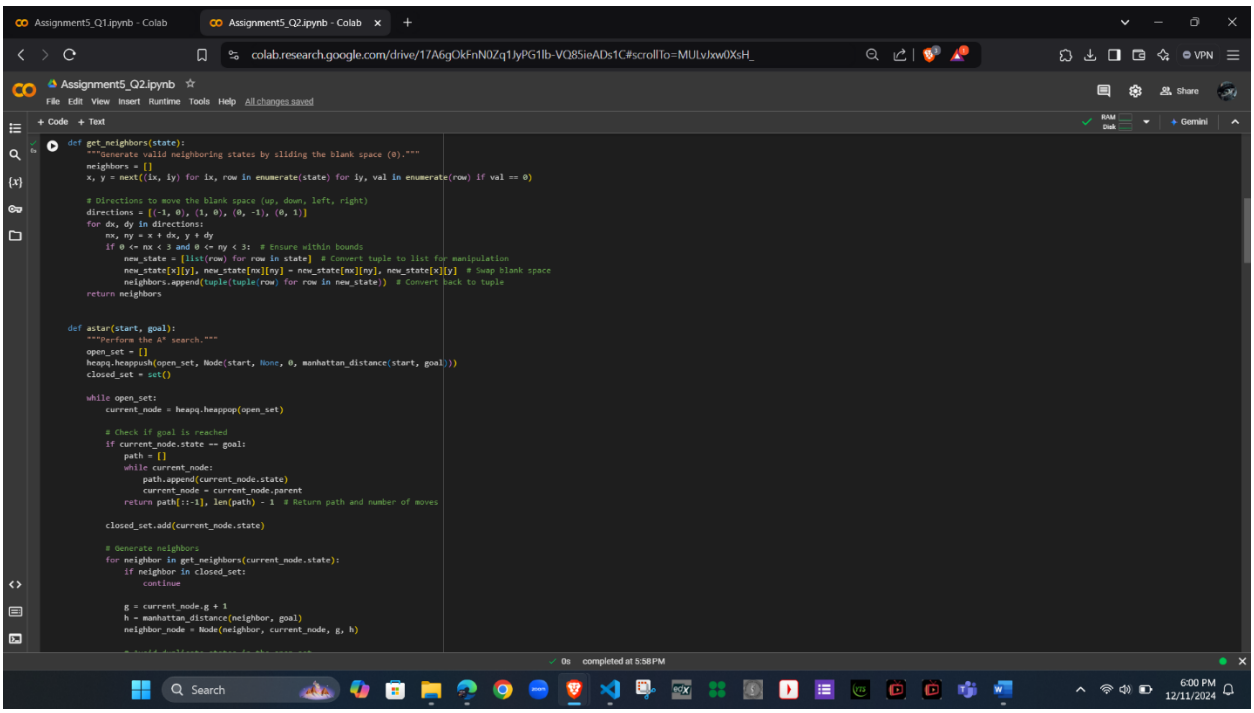
Number of clusters found by K-means: 5

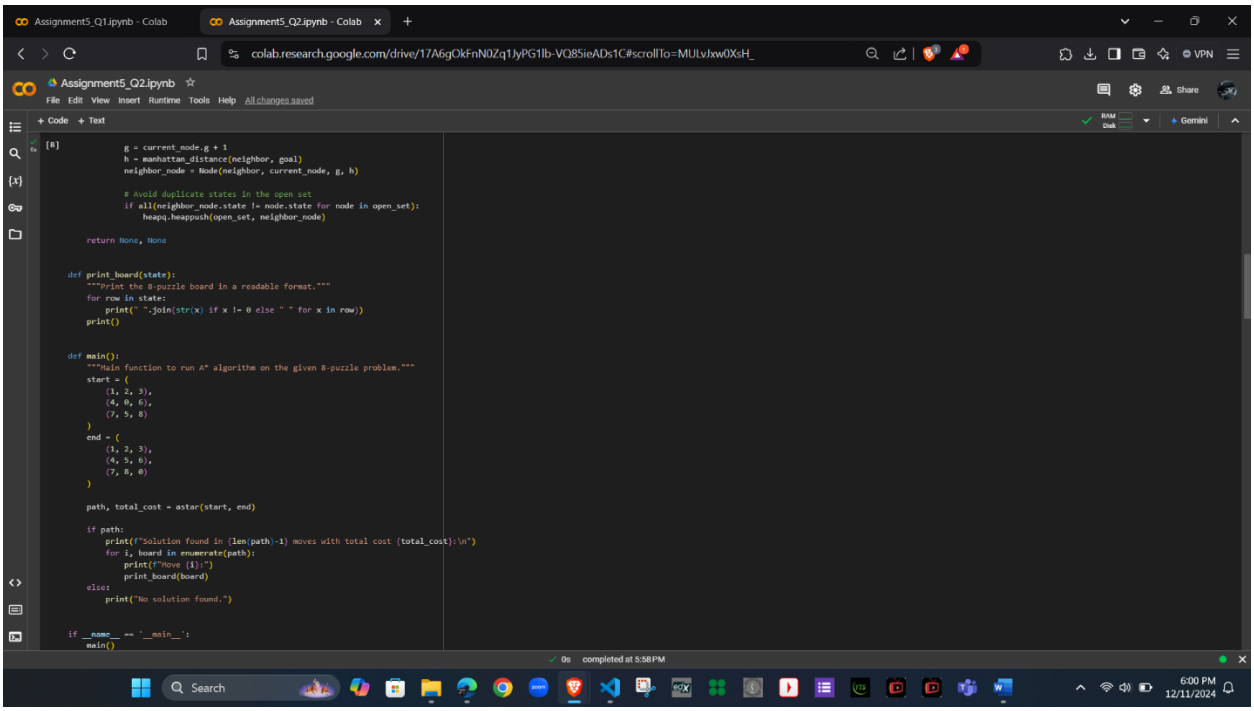
Q2. A* algorithms

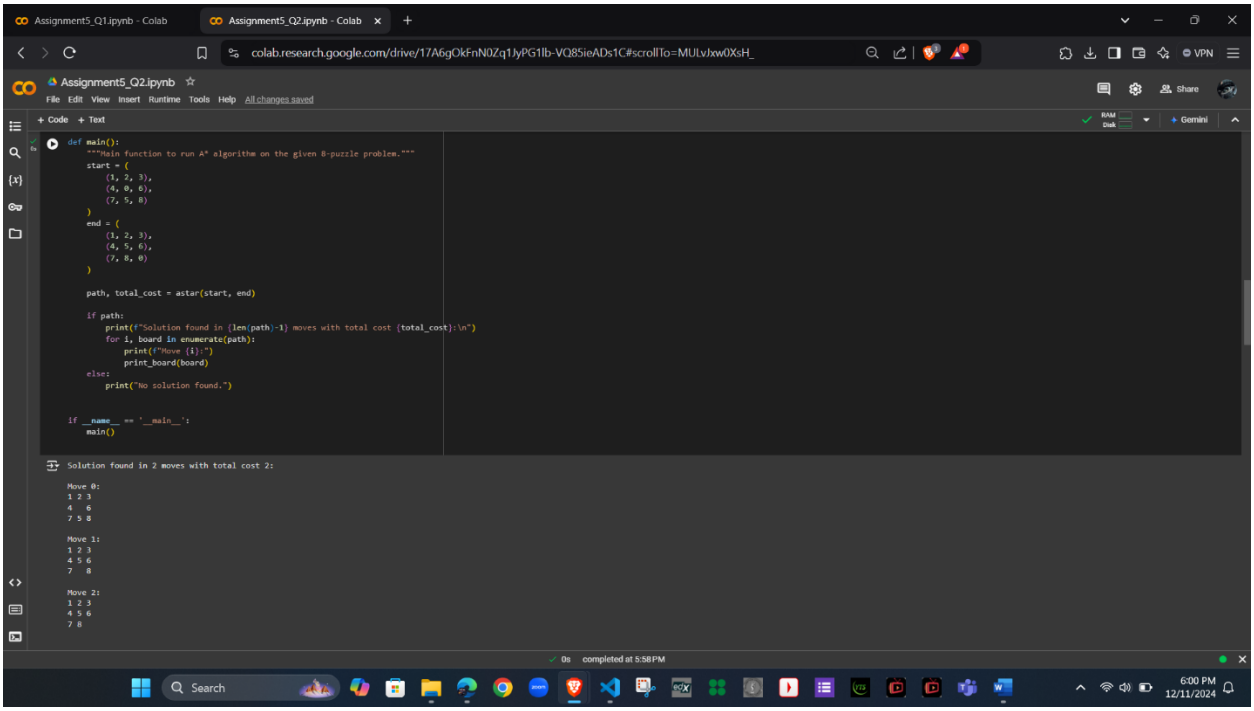
Step 1: Task Setup

- Find the shortest path to the goal state.





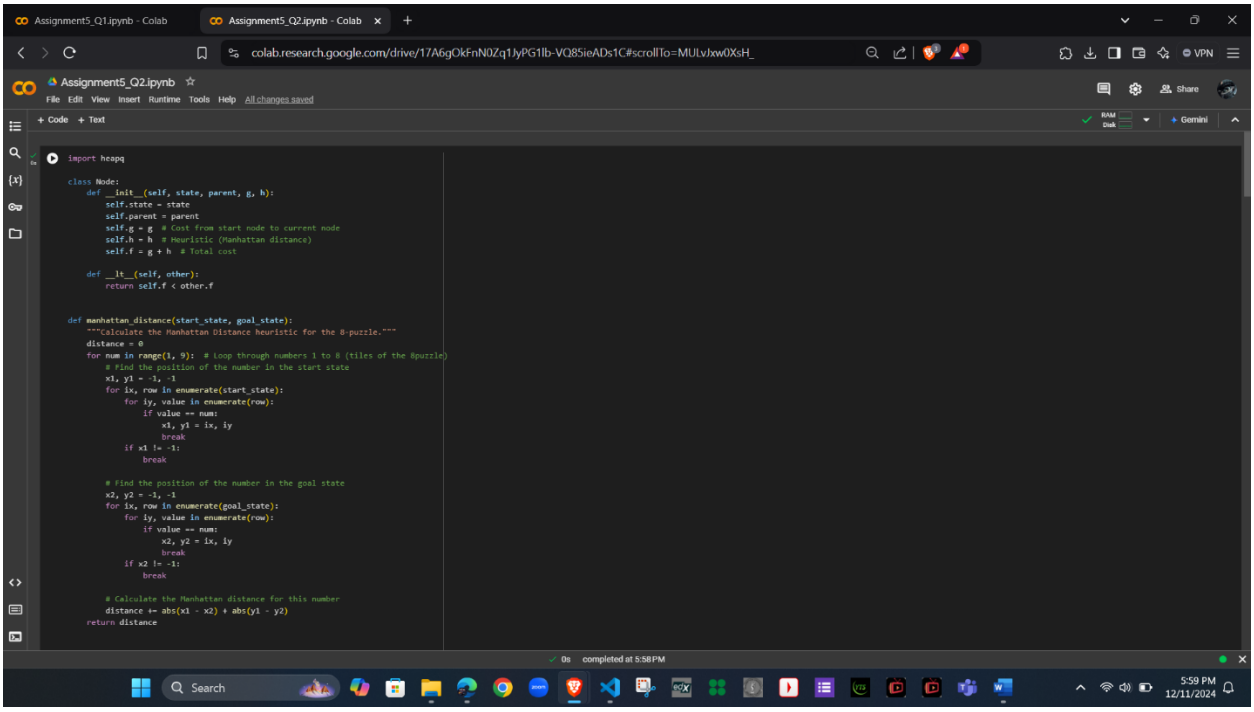




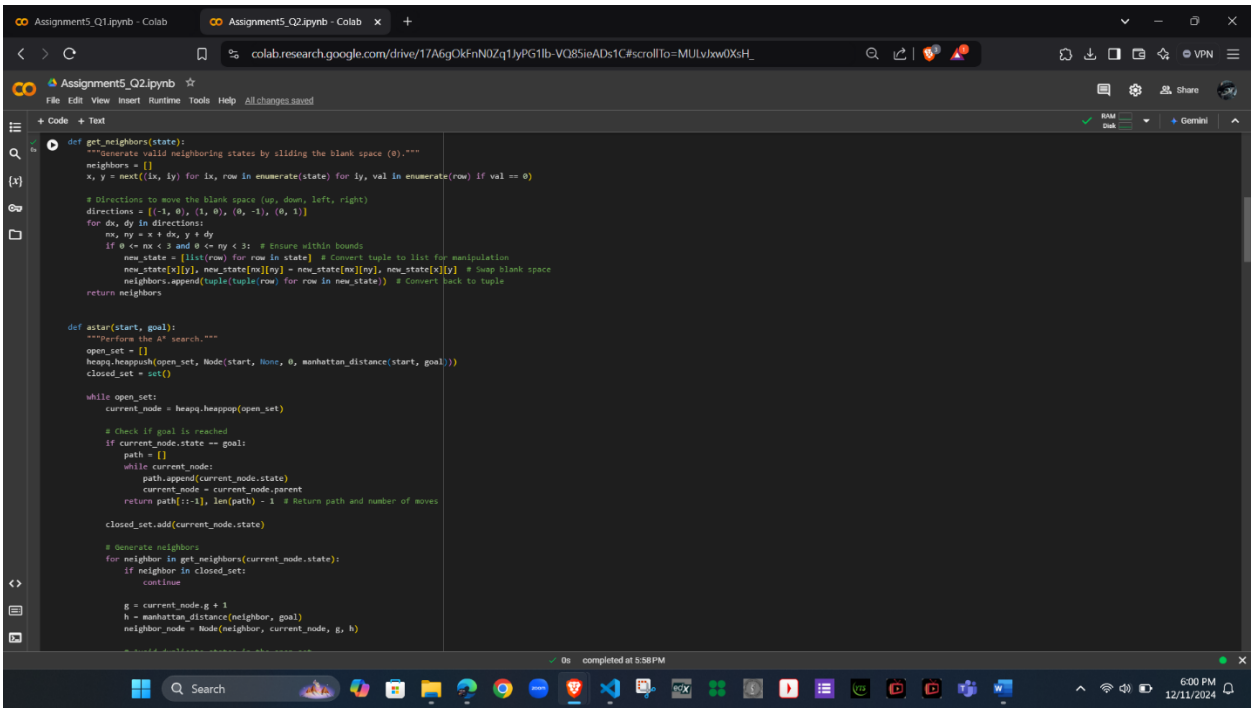
Solution found in 2 moves with total cost = 2

Step 2: Testing

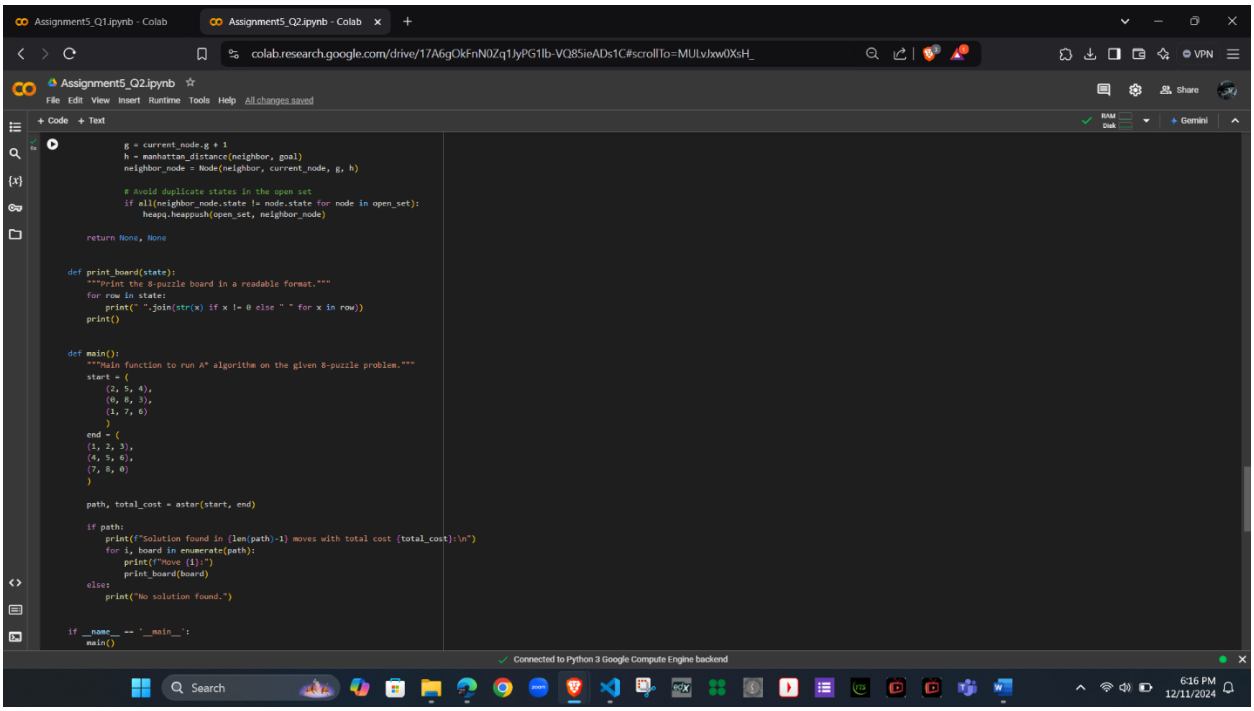
- Testing the A* algorithm with alternative start points, such as: ((2, 5, 4), (0, 8, 3), (1, 7, 6)).



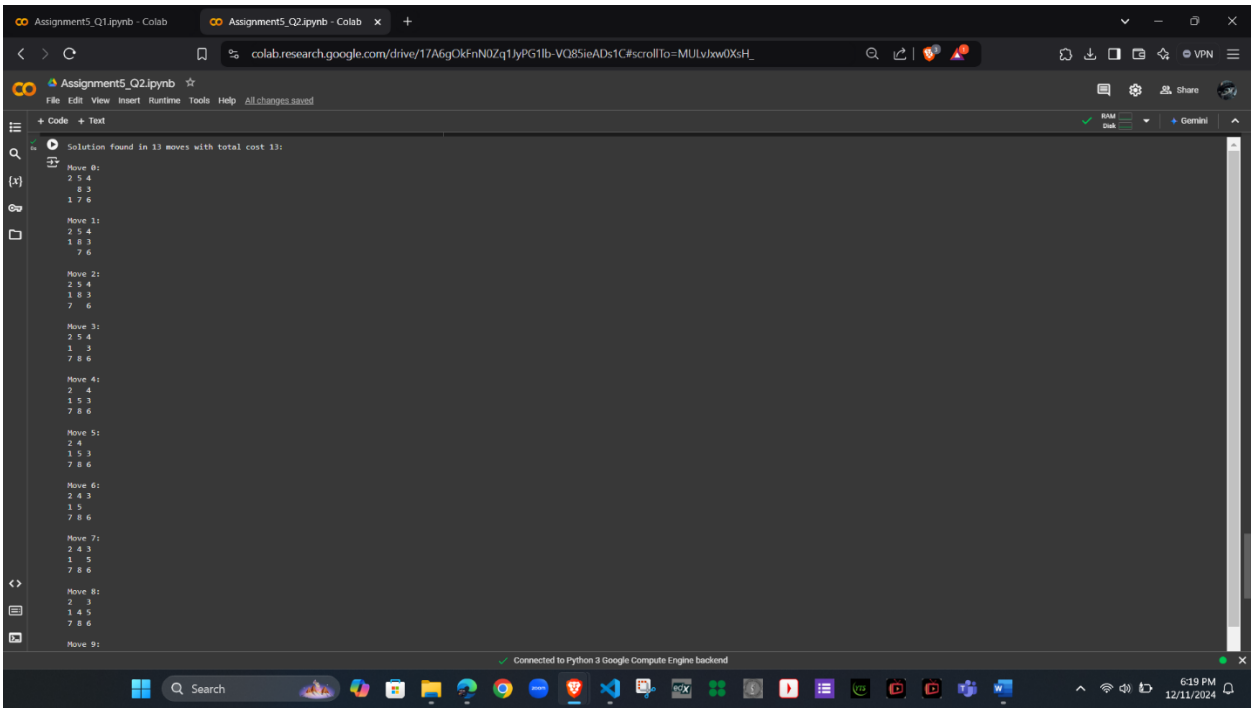
```
import heapq

class Node:
    def __init__(self, state, parent, g, h):
        self.state = state
        self.parent = parent
        self.g = g  # Cost from start node to current node
        self.h = h  # Heuristic (Manhattan distance)
        self.f = g + h  # Total cost

    def __lt__(self, other):
        return self.f < other.f


def manhattan_distance(start_state, goal_state):
    """Calculate the Manhattan Distance heuristic for the 8-puzzle."""
    distance = 0
    for num in range(1, 9):  # Loop through numbers 1 to 8 (tiles of the 8puzzle)
        # Find the position of the number in the start state
        x1, y1 = -1, -1
        for ix, row in enumerate(start_state):
            for iy, value in enumerate(row):
                if value == num:
                    x1, y1 = ix, iy
                    break
            if x1 != -1:
                break

        # Find the position of the number in the goal state
        x2, y2 = -1, -1
        for ix, row in enumerate(goal_state):
            for iy, value in enumerate(row):
                if value == num:
                    x2, y2 = ix, iy
                    break
            if x2 != -1:
                break

        # Calculate the Manhattan distance for this number
        distance += abs(x1 - x2) + abs(y1 - y2)
    return distance
```



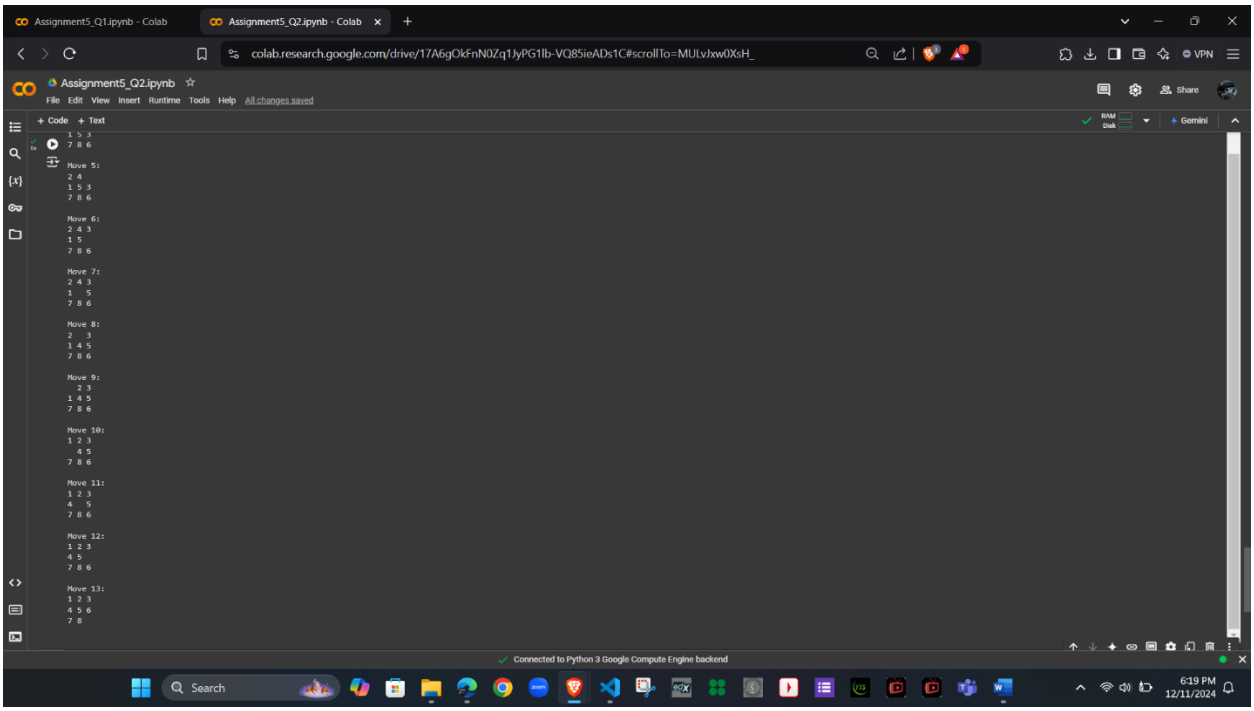
```
def get_neighbors(state):
    """Generate valid neighboring states by sliding the blank space (0)."""
    neighbors = []
    x, y = next((ix, iy) for ix, row in enumerate(state) for iy, val in enumerate(row) if val == 0)

    # Directions to move the blank space (up, down, left, right)
    directions = [(-1, 0), (1, 0), (0, -1), (0, 1)]
    for dx, dy in directions:
        nx, ny = x + dx, y + dy
        if 0 <= nx < 3 and 0 <= ny < 3:  # Ensure within bounds
            new_state = [list(row) for row in state]  # Convert tuple to list for manipulation
            new_state[x][y], new_state[nx][ny] = new_state[nx][ny], new_state[x][y]  # Swap blank space
            neighbors.append(tuple(tuple(row) for row in new_state))  # Convert back to tuple
    return neighbors


def astar(start, goal):
    """Perform the A* search."""
    open_set = []
    heapq.heappush(open_set, Node(start, None, 0, manhattan_distance(start, goal)))
    closed_set = set()

    while open_set:
        current_node = heapq.heappop(open_set)

        # Check if goal is reached
        if current_node.state == goal:
            path = []
            while current_node:
                path.append(current_node.state)
                current_node = current_node.parent
            return path[::-1], len(path) - 1  # Return path and number of moves

        closed_set.add(current_node.state)

        # Generate neighbors
        for neighbor in get_neighbors(current_node.state):
            if neighbor in closed_set:
                continue

            g = current_node.g + 1
            h = manhattan_distance(neighbor, goal)
            neighbor_node = Node(neighbor, current_node, g, h)
```



```
                g = current_node.g + 1
                h = manhattan_distance(neighbor, goal)
                neighbor_node = Node(neighbor, current_node, g, h)

                # Avoid duplicate states in the open set
                if all(neighbor_node.state != node.state for node in open_set):
                    heapq.heappush(open_set, neighbor_node)

        return None, None


    def print_board(state):
        """Print the 8-puzzle board in a readable format."""
        for row in state:
            print(" ".join(str(x) if x != 0 else " " for x in row))
        print()


    def main():
        """Main function to run A* algorithm on the given 8-puzzle problem."""
        start = (
            (2, 5, 4),
            (0, 8, 3),
            (1, 7, 6)
            )
        end = (
        (1, 2, 3),
        (4, 5, 6),
        (7, 8, 0)
        )

        path, total_cost = astar(start, end)

        if path:
            print(f"Solution found in {len(path)-1} moves with total cost {total_cost}:\n")
            for i, board in enumerate(path):
                print(f"Move {i}:")
                print_board(board)
        else:
            print("No solution found.")


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



```
Solution found in 13 moves with total cost 13:

Move 0:
2 5 4
  8 3
1 7 6

Move 1:
2 5 4
1 8 3
  7 6

Move 2:
2 5 4
1 8 3
7   6

Move 3:
2 5 4
1   3
7 8 6

Move 4:
2   4
1 5 3
7 8 6

Move 5:
2 4
1 5 3
7 8 6

Move 6:
2 4 3
1 5
7 8 6

Move 7:
2 4 3
1   5
7 8 6

Move 8:
2   3
1 4 5
7 8 6

Move 9:
```



Solution found in 13 moves with total cost 13