Weekly Challenge 14: K-edge Shortest Paths in Directed Graphs with Negative Weights

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

Objective

In doing this WC, you will:

- understand the basic algorithm/component/building block for various shortest path algorithms.
- demonstrate step-by-step working of the building block with examples and hand-drawn illustrations.
- implement the algorithm and verify correctness of your hand-crafted solutions.

The Problem

Ashraf Courier Service is responsible for delivering urgent messages between cities. The king has decreed that messages should reach their destination as quickly as possible, but couriers can take at most two roads to complete their journey.

Some cities are directly connected by a road, while others require passing through an intermediate city. The time taken to travel along each road varies, and couriers must always choose the fastest possible route.

Your task is to help the Ashraf Courier Service determine the shortest possible delivery time between a source city and all other cities while ensuring that no courier ever takes more than two roads in a single trip.

Task 1 (10 points)

For the graph shown in Figure 1, dry run Algorithm 1 shown in Figure 2 which finds at most 2-edge shortest distances from a source vertex a to all destination vertices k through intermediate vertices j. The source vertex would be S. You just need to submit **the final graph obtained after completing execution of the algorithm** as Figure 4. Please make sure the positioning of the vertices remains the same, i.e., do not morph the graph, just modify weights or add new edges.

Task 2 (10 points)

Implement Algorithm shown in Figure 2 and **visualize the resulting graph** using any graph visualization library. Include this image as Figure 5. You may use any programming language of your choice. Additionally, **mention in the caption** of Figure 5 what the output graph will illustrate if we run Algorithm 1 on this resulting graph one more time.

Task 3 (10 points)

Run Algorithm 1 for ***p-1*** times, where p is the total number of vertices in the graph (i.e., p = 10), and **visualize the resulting graph** using any graph visualization library. Include this image as Figure 6. Additionally, **mention in the caption** of Figure 6 what Algorithm 2 finds out.

Submission

1. Submit one PDF file including your solutions.
2. Submit one source file including your code.

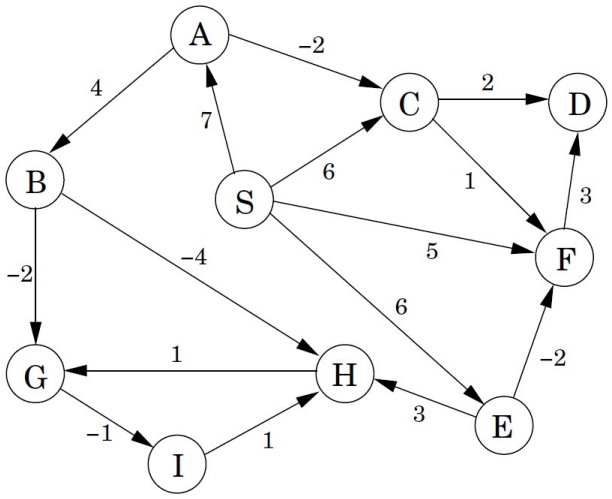

Figure 1: The sample input graph

```
For each vertex k of G do
    For each vertex j of G
        Find a two edge path from vertex a to vertex k
        passing through a vertex j, and
        assign min{w(a,k), w(a,j)+w(j,k)} to w(a,k)
```

Algorithm #1

Figure 2: Algorithm 1 finds at-most 2-edge distances from a source vertex $\boldsymbol{a}$ to all destinations $\boldsymbol{k}$ through all intermediate nodes $\boldsymbol{j}$. w(a,k) refers to the weight of the edge $a \rightarrow k$.

```
For x= 1 to p-1 do
    For each vertex k of G do
        For each vertex j of G
            Find a two edge path from vertex a to vertex k
            passing through a vertex j, and
            assign min{w(a,k), w(a,j)+w(j,k)} to w(a,k)
```

Algorithm #2

Figure 3: Algorithm 2 is designed by repeating Algorithm 1 (Figure 2) $\boldsymbol{p-1}$ times.

Solution - Task 1 (Figure 4)

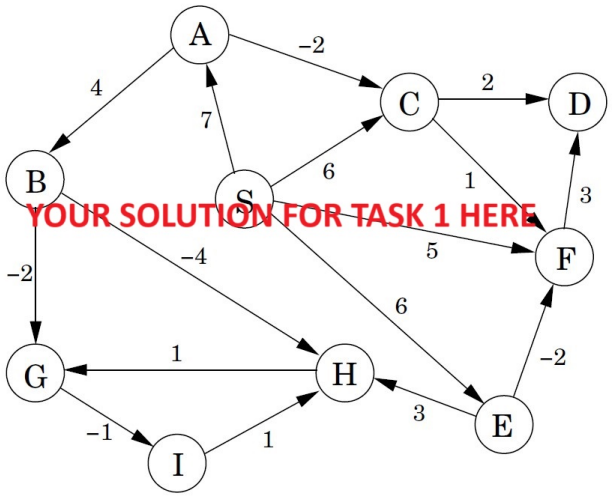

Figure 4: Modified graph showing the at-most 2-edge shortest distances from source vertex S to all other vertices for the graph in Figure 1.

Solution - Task 2 (Figure 5)

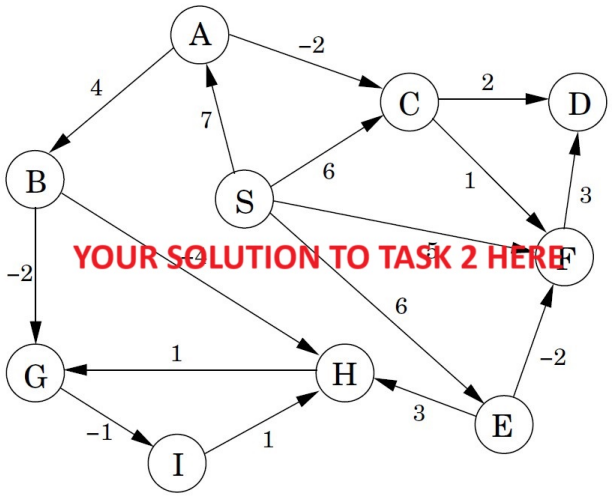

Figure 5: Code-generated graph showing the at-most 2-edge shortest distances from source vertex S to all other vertices for the graph in Figure 1.

Solution - Task 3 (Figure 6)

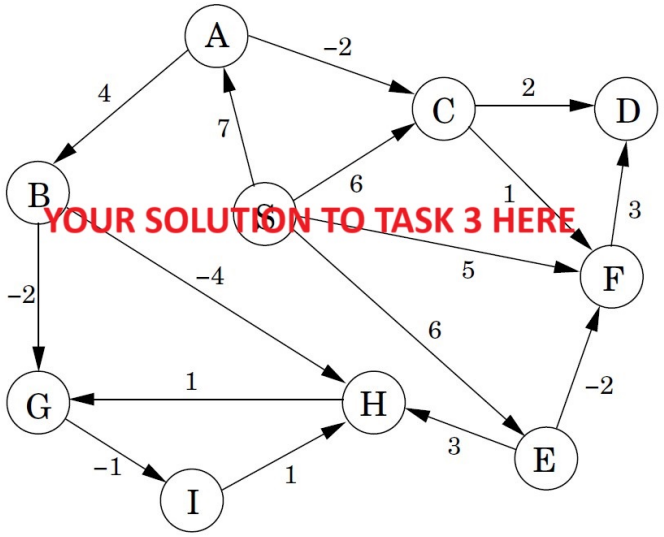

Figure 6: Code-generated graph showing shortest distances from source vertex S to all other vertices for the graph in Figure 1.

Weekly Challenge 14: K-edge Shortest Paths in Directed Graphs with Negative Weights

CS/CE 412/471 Algorithms: Design and Analysis

Spring 2025

Input Graph

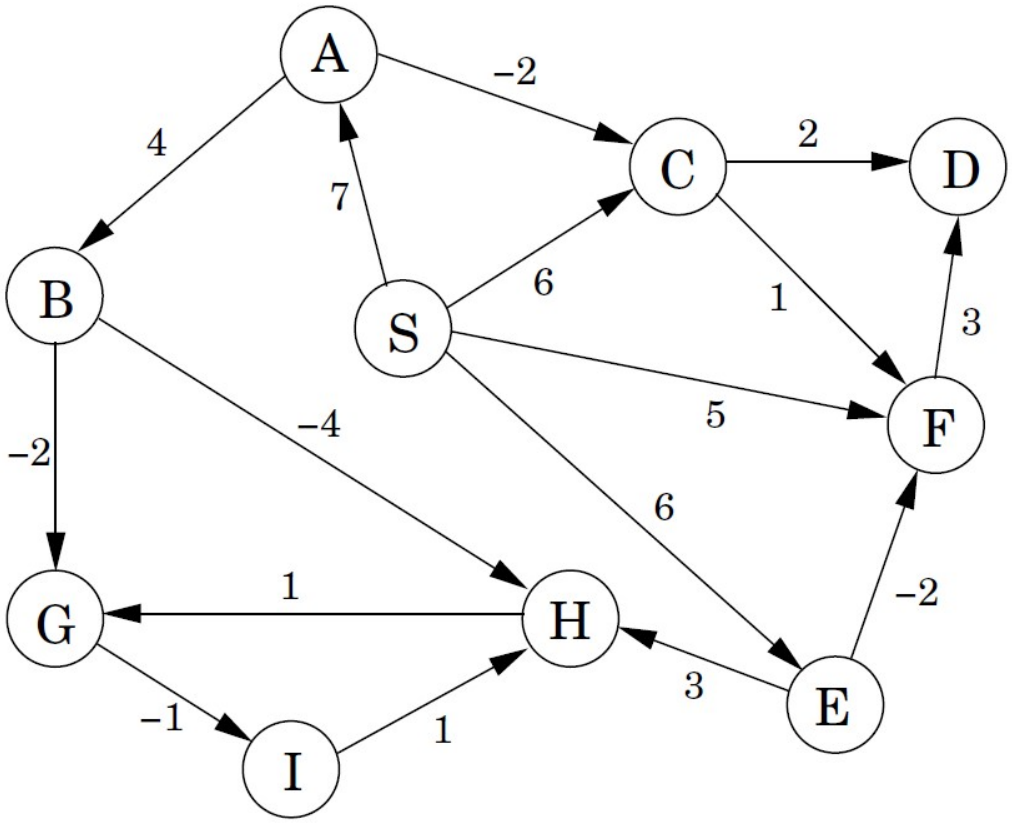

Figure 1: Input Graph

Solution - Task 1 (Figure 2)

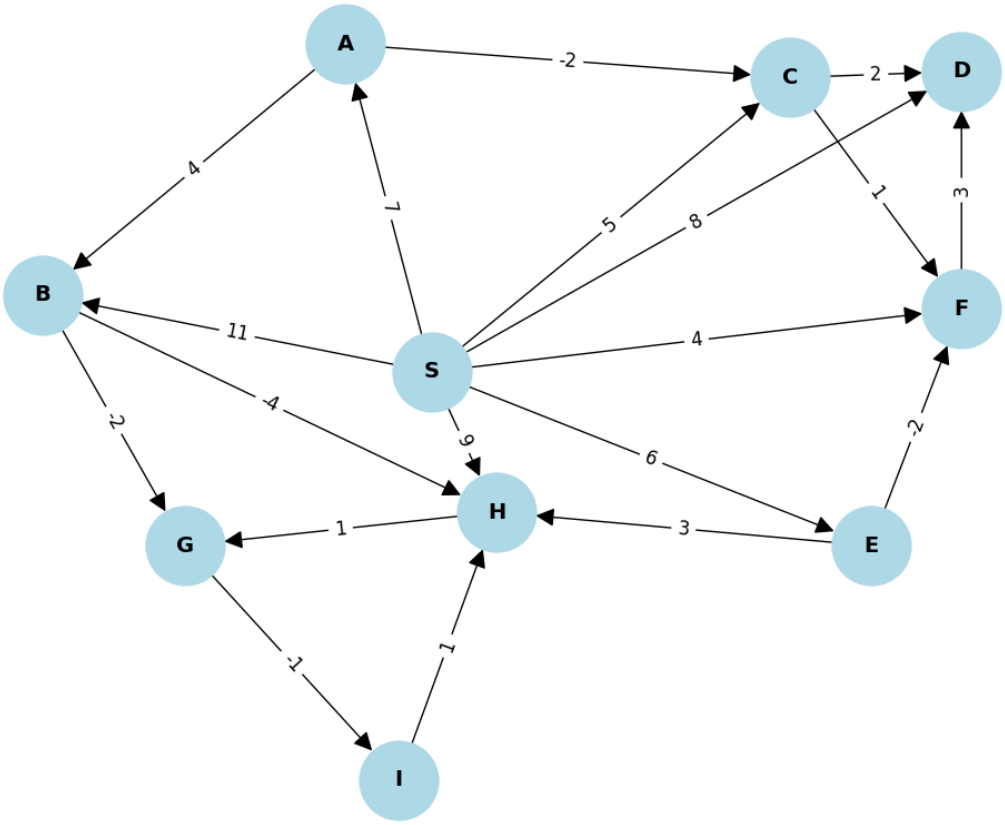

Figure 2: Modified graph showing the at-most 2-edge shortest distances from source vertex S to all other vertices for the graph in Figure 1.

Solution - Task 2 (Figure 3)

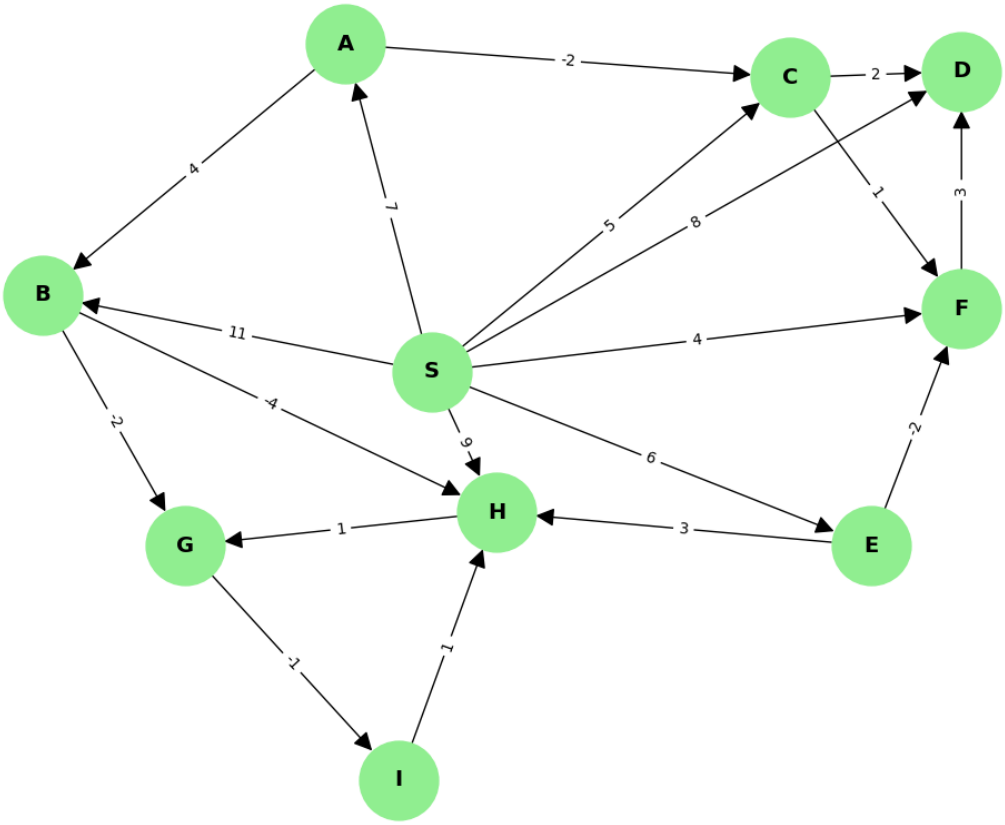

Figure 3: This graph shows the at-most 2-edge shortest distances from source vertex S to all other vertices after one complete application of Algorithm 1, where all possible 2-edge paths of the form S $\rightarrow$ j $\rightarrow$ k are evaluated. If we run Algorithm 1 on this resulting graph one more time, no further updates will occur, because all minimum distances via any 2-edge combination from S have already been found in this pass.

Solution - Task 3 (Figure 4)

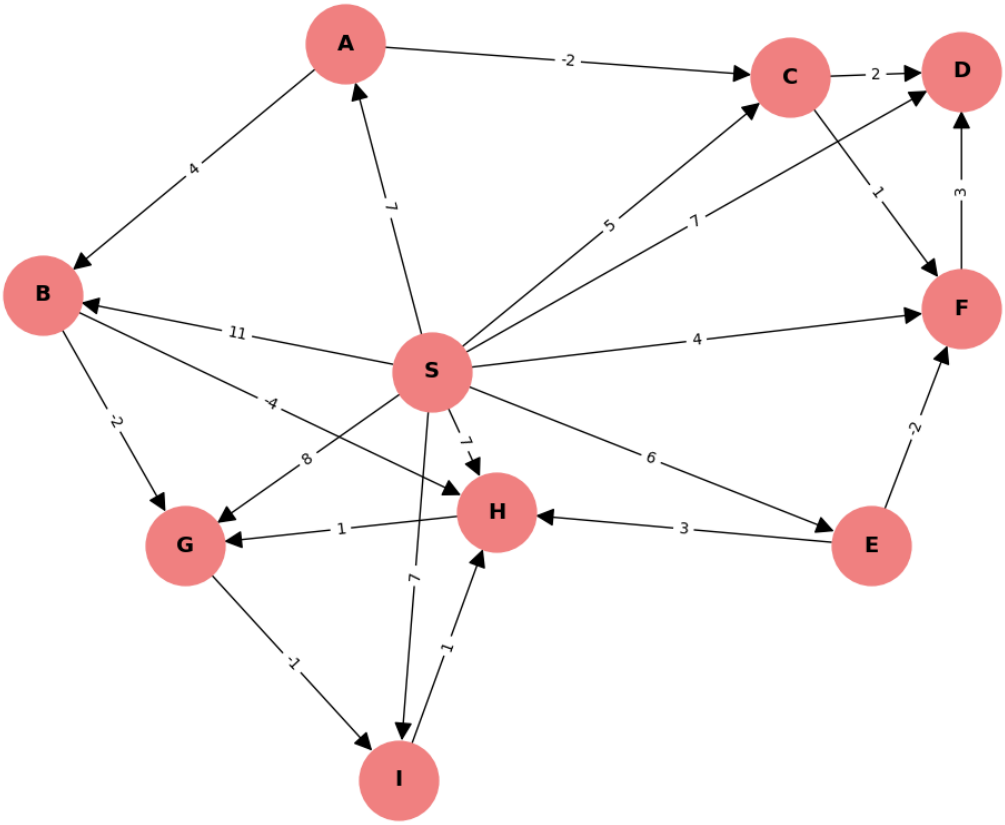

Figure 4: This graph displays the shortest distances between the source vertex S and every other vertex. All shortest pathways, including those requiring several 2-edge relaxations, are guaranteed to be fully captured by Algorithm 2. Like the Bellman-Ford method modified for 2-edge path logic, it ensures that even the most indirect optimal paths from S are found.

```
import networkx as nx
import matplotlib.pyplot as plt

# -------------------- Graph Definition --------------------
G = nx.DiGraph()
edges = [
    ('S', 'A', 7), ('S', 'C', 6), ('A', 'C', -2),
    ('A', 'B', 4), ('C', 'D', 2), ('C', 'F', 1), ('F', 'D', 3), ('S', 'F', 5),
    ('S', 'E', 6), ('E', 'F', -2), ('E', 'H', 3), ('H', 'G', 1),
    ('G', 'I', -1), ('I', 'H', 1), ('B', 'H', -4), ('B', 'G', -2)
]
G.add_weighted_edges_from(edges)

# -------------------- Task 1: One-time 2-edge pass --------------------
dist_task1 = {}
for node in G.nodes():
    if node == 'S':
        dist_task1[node] = 0
    elif G.has_edge('S', node):
        dist_task1[node] = G['S'][node]['weight']
    else:
        dist_task1[node] = float('inf')

for j in G.successors('S'):
    for k in G.successors(j):
        if j == k: continue
        w1 = G['S'][j]['weight']
        w2 = G[j][k]['weight']
        total_weight = w1 + w2
        if total_weight < dist_task1.get(k, float('inf')):
            dist_task1[k] = total_weight

final_graph_task1 = nx.DiGraph()
for u, v, w in edges:
    if u != 'S':
        final_graph_task1.add_edge(u, v, weight=w)
for node in dist_task1:
    if node != 'S' and dist_task1[node] != float('inf'):
        final_graph_task1.add_edge('S', node, weight=dist_task1[node])

# -------------------- Task 2: Full 1-time Algorithm 1 (all j, all k)
--------------------
dist_task2 = {}
for node in G.nodes():
    if node == 'S':
        dist_task2[node] = 0
    elif G.has_edge('S', node):
        dist_task2[node] = G['S'][node]['weight']
    else:
        dist_task2[node] = float('inf')

for k in G.nodes():
    if k == 'S': continue
    for j in G.nodes():
        if j == 'S': continue
        if G.has_edge('S', j) and G.has_edge(j, k):
            via_j = G['S'][j]['weight'] + G[j][k]['weight']
            if via_j < dist_task2[k]:
                dist_task2[k] = via_j
```

```
final_graph_task2 = nx.DiGraph()
for u, v, w in edges:
    if u != 'S':
        final_graph_task2.add_edge(u, v, weight=w)
for node in dist_task2:
    if node != 'S' and dist_task2[node] != float('inf'):
        final_graph_task2.add_edge('S', node, weight=dist_task2[node])

# -------------------- Task 3: Algorithm 2 (repeat Algorithm 1 p-1 times)
--------------------
p = len(G.nodes())
dist_task3 = {}
for node in G.nodes():
    if node == 'S':
        dist_task3[node] = 0
    elif G.has_edge('S', node):
        dist_task3[node] = G['S'][node]['weight']
    else:
        dist_task3[node] = float('inf')

for _ in range(p - 1):  # Repeat Algorithm 1 p-1 times
    temp_dist = dist_task3.copy()
    for k in G.nodes():
        if k == 'S': continue
        for j in G.nodes():
            if j == 'S': continue
            if dist_task3[j] != float('inf') and G.has_edge(j, k):
                via_j = dist_task3[j] + G[j][k]['weight']
                if via_j < temp_dist[k]:
                    temp_dist[k] = via_j
    dist_task3 = temp_dist

final_graph_task3 = nx.DiGraph()
for u, v, w in edges:
    if u != 'S':
        final_graph_task3.add_edge(u, v, weight=w)
for node in dist_task3:
    if node != 'S' and dist_task3[node] != float('inf'):
        final_graph_task3.add_edge('S', node, weight=dist_task3[node])

# -------------------- Estimated Node Positions According to Original Graph
--------------------
pos = {
    'S': (-1.016, 0.040), 'A': (-1.5, 2), 'B': (-3.2, 0.5),
    'C': (1, 1.8), 'D': (1.96, 1.838), 'F': (1.96, 0.417),
    'E': (1.45, -1), 'H': (-0.651, -0.799), 'I': (-1.2, -2.4),
    'G': (-2.4, -1)
}

# -------------------- Plot & Save Task 1 --------------------
plt.figure(figsize=(10, 8))
nx.draw(final_graph_task1, pos, with_labels=True, node_color='lightblue',
node_size=2600,
        font_size=14, font_weight='bold', arrowsize=25)
nx.draw_networkx_edge_labels(final_graph_task1, pos,
edge_labels=nx.get_edge_attributes(final_graph_task1, 'weight'))
plt.title("Task 1: One-time 2-Edge Relaxation")
plt.tight_layout()
```

```
plt.savefig("task1.png")
plt.show()

# -------------------- Plot & Save Task 2 --------------------
plt.figure(figsize=(10, 8))
nx.draw(final_graph_task2, pos, with_labels=True, node_color='lightgreen',
node_size=2600,
        font_size=14, font_weight='bold', arrowsize=25)
nx.draw_networkx_edge_labels(final_graph_task2, pos,
edge_labels=nx.get_edge_attributes(final_graph_task2, 'weight'))
plt.title("Figure 5: Full Algorithm 1 (1 pass) Output")
plt.tight_layout()
plt.savefig("task2.png")
plt.show()

# -------------------- Plot & Save Task 3 --------------------
plt.figure(figsize=(10, 8))
nx.draw(final_graph_task3, pos, with_labels=True, node_color='lightcoral',
node_size=2600,
        font_size=14, font_weight='bold', arrowsize=25)
nx.draw_networkx_edge_labels(final_graph_task3, pos,
edge_labels=nx.get_edge_attributes(final_graph_task3, 'weight'))
plt.title("Figure 6: Algorithm 2 (p−1 passes)")
plt.tight_layout()
plt.savefig("task3.png")
plt.show()
```