```
!pip install ortools
!pip install torch torch-geometric numpy matplotlib

Requirement already satisfied: ortools in
/usr/local/lib/python3.11/dist-packages (9.11.4210)
Requirement already satisfied: absl-py>=2.0.0 in
/usr/local/lib/python3.11/dist-packages (from ortools) (2.1.0)
Requirement already satisfied: numpy>=1.13.3 in
/usr/local/lib/python3.11/dist-packages (from ortools) (1.26.4)
Requirement already satisfied: pandas>=2.0.0 in
/usr/local/lib/python3.11/dist-packages (from ortools) (2.2.2)
Requirement already satisfied: protobuf<5.27,>=5.26.1 in
/usr/local/lib/python3.11/dist-packages (from ortools) (5.26.1)
Requirement already satisfied: immutabledict>=3.0.0 in
/usr/local/lib/python3.11/dist-packages (from ortools) (4.2.1)
Requirement already satisfied: python-dateutil>=2.8.2 in
/usr/local/lib/python3.11/dist-packages (from pandas>=2.0.0->ortools)
(2.8.2)
Requirement already satisfied: pytz>=2020.1 in
/usr/local/lib/python3.11/dist-packages (from pandas>=2.0.0->ortools)
(2024.2)
Requirement already satisfied: tzdata>=2022.7 in
/usr/local/lib/python3.11/dist-packages (from pandas>=2.0.0->ortools)
(2024.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.8.2-
>pandas>=2.0.0->ortools) (1.17.0)
Requirement already satisfied: torch in
/usr/local/lib/python3.11/dist-packages (2.5.1+cu121)
Requirement already satisfied: torch-geometric in
/usr/local/lib/python3.11/dist-packages (2.6.1)
Requirement already satisfied: numpy in
/usr/local/lib/python3.11/dist-packages (1.26.4)
Requirement already satisfied: matplotlib in
/usr/local/lib/python3.11/dist-packages (3.10.0)
Requirement already satisfied: filelock in
/usr/local/lib/python3.11/dist-packages (from torch) (3.16.1)
Requirement already satisfied: typing-extensions>=4.8.0 in
/usr/local/lib/python3.11/dist-packages (from torch) (4.12.2)
Requirement already satisfied: networkx in
/usr/local/lib/python3.11/dist-packages (from torch) (3.4.2)
Requirement already satisfied: jinja2 in
/usr/local/lib/python3.11/dist-packages (from torch) (3.1.5)
Requirement already satisfied: fsspec in
/usr/local/lib/python3.11/dist-packages (from torch) (2024.10.0)
Requirement already satisfied: nvidia-cuda-nvrtc-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-runtime-cu12==12.1.105
in /usr/local/lib/python3.11/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cuda-cupti-cu12==12.1.105 in
```

```
/usr/local/lib/python3.11/dist-packages (from torch) (12.1.105)
Requirement already satisfied: nvidia-cudnn-cu12==9.1.0.70 in
/usr/local/lib/python3.11/dist-packages (from torch) (9.1.0.70)
Requirement already satisfied: nvidia-cublas-cu12==12.1.3.1 in
/usr/local/lib/python3.11/dist-packages (from torch) (12.1.3.1)
Requirement already satisfied: nvidia-cufft-cu12==11.0.2.54 in
/usr/local/lib/python3.11/dist-packages (from torch) (11.0.2.54)
Requirement already satisfied: nvidia-curand-cu12==10.3.2.106 in
/usr/local/lib/python3.11/dist-packages (from torch) (10.3.2.106)
Requirement already satisfied: nvidia-cusolver-cu12==11.4.5.107 in
/usr/local/lib/python3.11/dist-packages (from torch) (11.4.5.107)
Requirement already satisfied: nvidia-cusparse-cu12==12.1.0.106 in
/usr/local/lib/python3.11/dist-packages (from torch) (12.1.0.106)
Requirement already satisfied: nvidia-nccl-cu12==2.21.5 in
/usr/local/lib/python3.11/dist-packages (from torch) (2.21.5)
Requirement already satisfied: nvidia-nvtx-cu12==12.1.105 in
/usr/local/lib/python3.11/dist-packages (from torch) (12.1.105)
Requirement already satisfied: triton==3.1.0 in
/usr/local/lib/python3.11/dist-packages (from torch) (3.1.0)
Requirement already satisfied: sympy==1.13.1 in
/usr/local/lib/python3.11/dist-packages (from torch) (1.13.1)
Requirement already satisfied: nvidia-nvjitlink-cu12 in
/usr/local/lib/python3.11/dist-packages (from nvidia-cusolver-
cu12==11.4.5.107->torch) (12.6.85)
Requirement already satisfied: mpmath<1.4,>=1.1.0 in
/usr/local/lib/python3.11/dist-packages (from sympy==1.13.1->torch)
(1.3.0)
Requirement already satisfied: aiohttp in
/usr/local/lib/python3.11/dist-packages (from torch-geometric)
(3.11.11)
Requirement already satisfied: psutil>=5.8.0 in
/usr/local/lib/python3.11/dist-packages (from torch-geometric) (5.9.5)
Requirement already satisfied: pyparsing in
/usr/local/lib/python3.11/dist-packages (from torch-geometric) (3.2.1)
Requirement already satisfied: requests in
/usr/local/lib/python3.11/dist-packages (from torch-geometric)
(2.32.3)
Requirement already satisfied: tqdm in /usr/local/lib/python3.11/dist-
packages (from torch-geometric) (4.67.1)
Requirement already satisfied: contourpy>=1.0.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (1.3.1)
Requirement already satisfied: cycler>=0.10 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (0.12.1)
Requirement already satisfied: fonttools>=4.22.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (4.55.3)
Requirement already satisfied: kiwisolver>=1.3.1 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (1.4.8)
Requirement already satisfied: packaging>=20.0 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (24.2)
```

```
Requirement already satisfied: pillow>=8 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (11.1.0)
Requirement already satisfied: python-dateutil>=2.7 in
/usr/local/lib/python3.11/dist-packages (from matplotlib) (2.8.2)
Requirement already satisfied: six>=1.5 in
/usr/local/lib/python3.11/dist-packages (from python-dateutil>=2.7-
>matplotlib) (1.17.0)
Requirement already satisfied: aiohappyeyeballs>=2.3.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (2.4.4)
Requirement already satisfied: aiosignal>=1.1.2 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (1.3.2)
Requirement already satisfied: attrs>=17.3.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (24.3.0)
Requirement already satisfied: frozenlist>=1.1.1 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (1.5.0)
Requirement already satisfied: multidict<7.0,>=4.5 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (6.1.0)
Requirement already satisfied: propcache>=0.2.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (0.2.1)
Requirement already satisfied: yarl<2.0,>=1.17.0 in
/usr/local/lib/python3.11/dist-packages (from aiohttp->torch-
geometric) (1.18.3)
Requirement already satisfied: MarkupSafe>=2.0 in
/usr/local/lib/python3.11/dist-packages (from jinja2->torch) (3.0.2)
Requirement already satisfied: charset-normalizer<4,>=2 in
/usr/local/lib/python3.11/dist-packages (from requests->torch-
geometric) (3.4.1)
Requirement already satisfied: idna<4,>=2.5 in
/usr/local/lib/python3.11/dist-packages (from requests->torch-
geometric) (3.10)
Requirement already satisfied: urllib3<3,>=1.21.1 in
/usr/local/lib/python3.11/dist-packages (from requests->torch-
geometric) (2.3.0)
Requirement already satisfied: certifi>=2017.4.17 in
/usr/local/lib/python3.11/dist-packages (from requests->torch-
geometric) (2024.12.14)
```

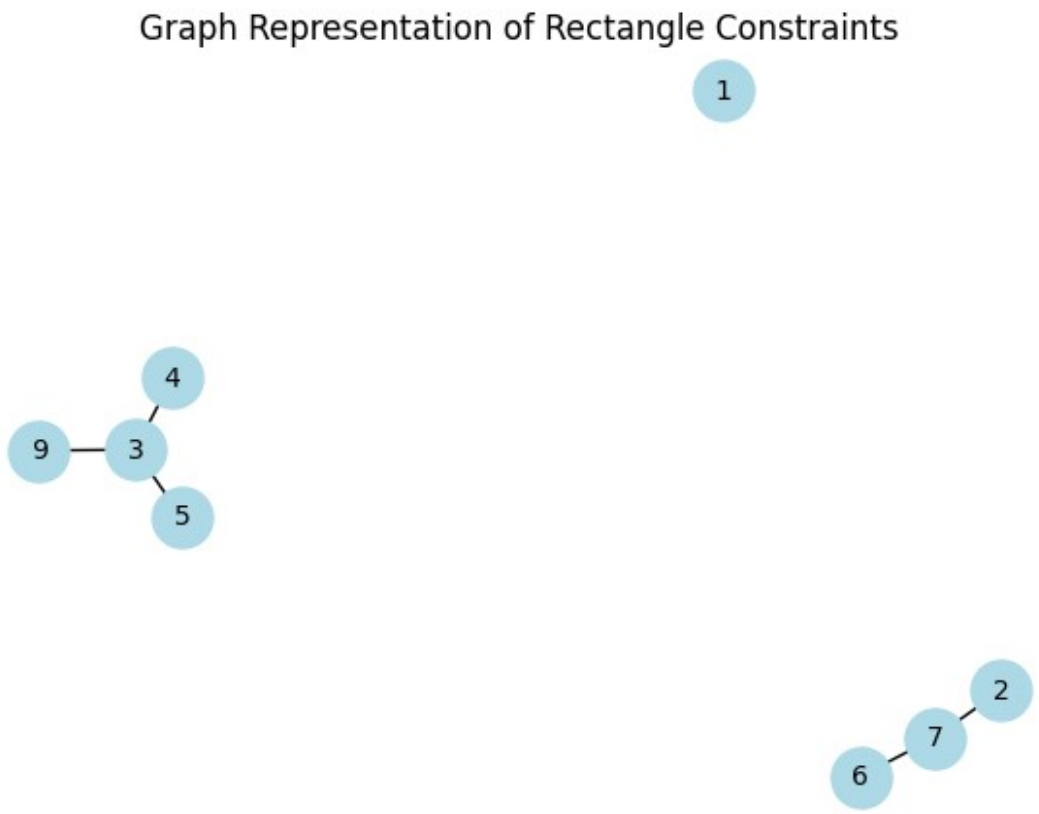
FIRST ITERATION

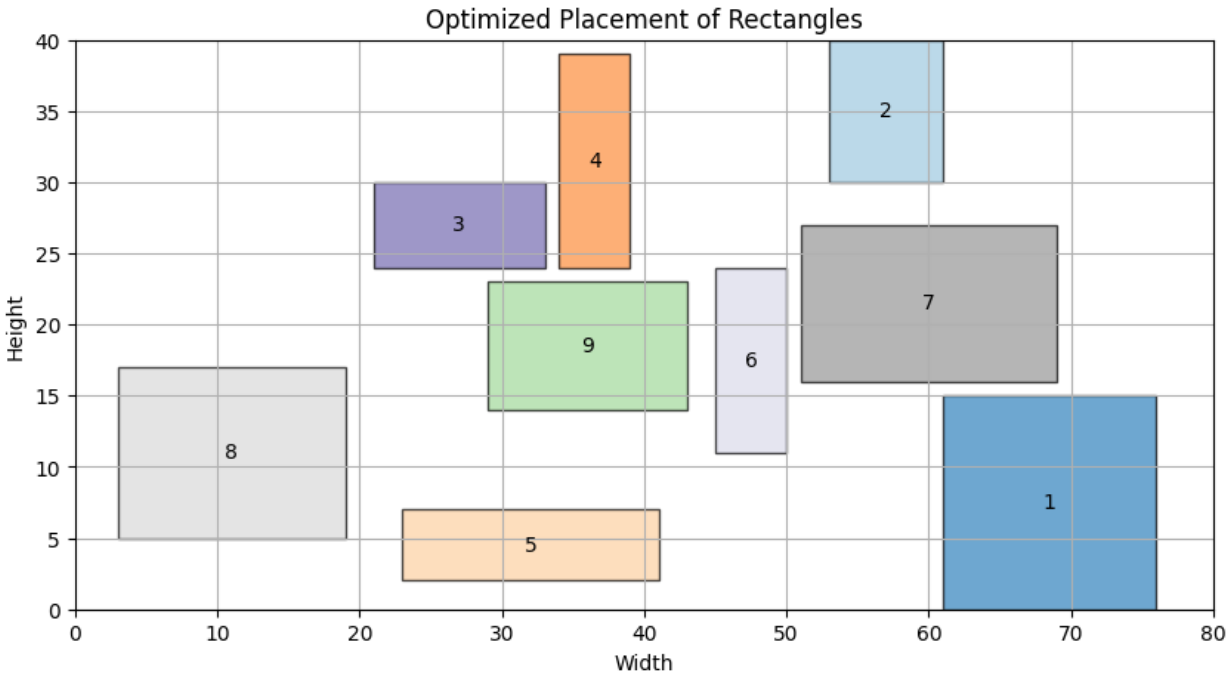
```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import random
```

```
BIN_WIDTH = 80
BIN_HEIGHT = 40

np.random.seed(42)

def generate_rectangle_sizes(num_rectangles, max_width, max_height):
    return [(random.randint(5, max_width), random.randint(5,
max_height)) for _ in range(num_rectangles)]

rectangles = generate_rectangle_sizes(9, 20, 15)

constraints = {
    1: {"position": "top"},
    2: {"position": "bottom"},
    3: {"close_to": [4, 5, 9]},
    7: {"close_to": [6, 2]},
}

G = nx.Graph()
for rect, props in constraints.items():
    G.add_node(rect)
    if "close_to" in props:
        for neighbor in props["close_to"]:
            G.add_edge(rect, neighbor)

plt.figure(figsize=(6, 4))
nx.draw(G, with_labels=True, node_color='lightblue', node_size=500,
font_size=10)
plt.title("Graph Representation of Rectangle Constraints")
plt.show()

bin_layout = np.zeros((BIN_HEIGHT, BIN_WIDTH), dtype=int)
placements = {}

def is_overlap(bin_layout, x, y, width, height):
    if x + width + 1 > BIN_WIDTH or y + height + 1 > BIN_HEIGHT or x <
0 or y < 0:
        return True
    return np.any(bin_layout[max(0, y - 1):min(BIN_HEIGHT, y + height
+ 1), max(0, x - 1):min(BIN_WIDTH, x + width + 1)] > 0)

def place_rectangle(rect_id, width, height, x=None, y=None):
    if x is None or y is None:
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        while is_overlap(bin_layout, x, y, width, height):
            x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
    bin_layout[y:y + height, x:x + width] = rect_id
```

```
    placements[rect_id] = (x, y, width, height)

place_rectangle(1, rectangles[0][0], rectangles[0][1],
x=random.randint(0, BIN_WIDTH - rectangles[0][0]), y=0)
place_rectangle(2, rectangles[1][0], rectangles[1][1],
x=random.randint(0, BIN_WIDTH - rectangles[1][0]), y=BIN_HEIGHT -
rectangles[1][1])

def place_close_to(rect_id, width, height, close_to_ids):
    placed_coords = [placements[close_id] for close_id in close_to_ids
if close_id in placements]
    if placed_coords:
        for px, py, pwidth, pheight in placed_coords:
            for dx in range(-width - 1, width + 2):
                for dy in range(-height - 1, height + 2):
                    x = max(0, min(BIN_WIDTH - width, px + dx))
                    y = max(0, min(BIN_HEIGHT - height, py + dy))
                    if not is_overlap(bin_layout, x, y, width,
height):
                        place_rectangle(rect_id, width, height, x, y)
                        return True
    return False

place_rectangle(4, rectangles[3][0], rectangles[3][1])
place_rectangle(5, rectangles[4][0], rectangles[4][1])
place_rectangle(9, rectangles[8][0], rectangles[8][1])
place_close_to(3, rectangles[2][0], rectangles[2][1], constraints[3]
["close_to"])
place_rectangle(6, rectangles[5][0], rectangles[5][1])
place_close_to(7, rectangles[6][0], rectangles[6][1], constraints[7]
["close_to"])

def place_remaining():
    for rect_id, (width, height) in enumerate(rectangles, start=1):
        if rect_id not in placements:
            while True:
                x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
                if not is_overlap(bin_layout, x, y, width, height):
                    place_rectangle(rect_id, width, height, x, y)
                    break

place_remaining()

colors = plt.cm.tab20c(np.linspace(0, 1, len(placements)))

plt.figure(figsize=(10, 5))
for idx, (rect_id, (x, y, width, height)) in
enumerate(placements.items()):
    color = colors[idx]
```

```
    plt.gca().add_patch(plt.Rectangle((x, y), width, height,
edgecolor='black', facecolor=color, alpha=0.7))
    plt.text(x + width / 2, y + height / 2, f"{rect_id}", ha='center',
va='center', color='black')

plt.xlim(0, BIN_WIDTH)
plt.ylim(0, BIN_HEIGHT)
plt.gca().set_aspect('equal', adjustable='box')
plt.title("Optimized Placement of Rectangles")
plt.xlabel("Width")
plt.ylabel("Height")
plt.grid(True)
plt.show()
```





SECOND ITERATION

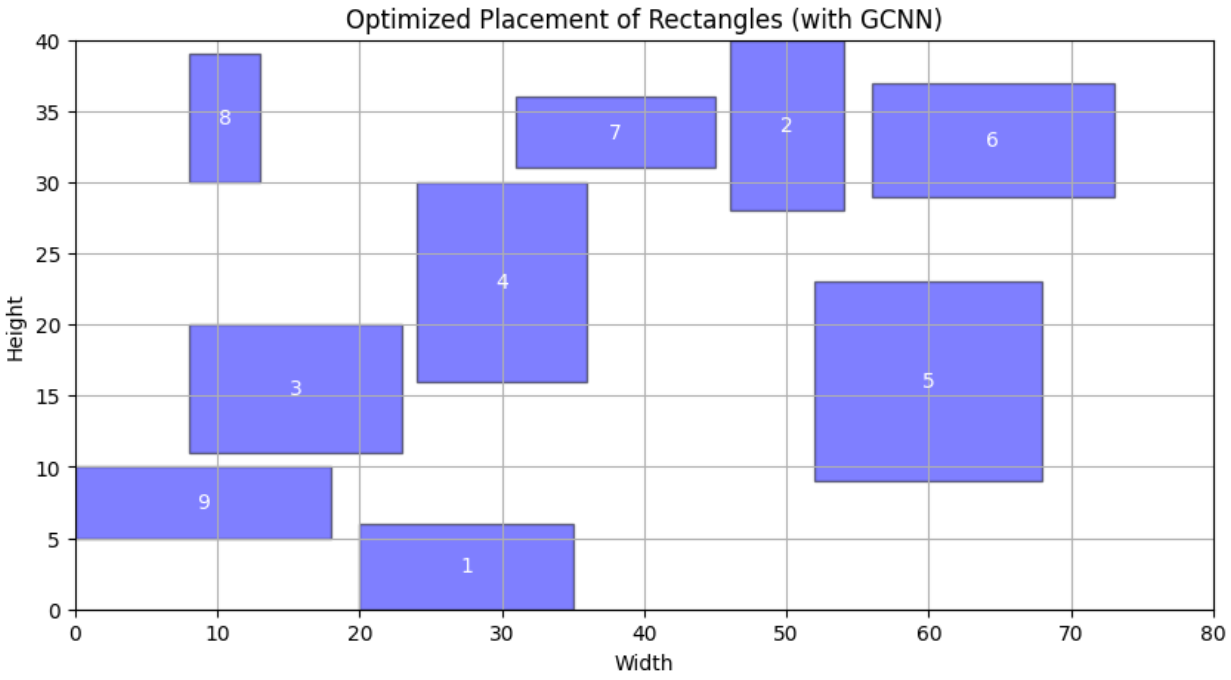
```
import numpy as np
import random
import torch
import torch.nn as nn
import torch.optim as optim
import matplotlib.pyplot as plt
import networkx as nx
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv

BIN_WIDTH = 80
BIN_HEIGHT = 40

np.random.seed(42)

def generate_rectangle_sizes(num_rectangles, max_width, max_height):
    return [(random.randint(5, max_width), random.randint(5,
max_height)) for _ in range(num_rectangles)]

rectangles = generate_rectangle_sizes(9, 20, 15)

constraints = {
    1: {"position": "top"},
    2: {"position": "bottom"},
    3: {"close_to": [4, 5, 9]},
    7: {"close_to": [6, 2]},
}
```

```
bin_layout = np.zeros((BIN_HEIGHT, BIN_WIDTH), dtype=int)
placements = {}

def is_overlap(bin_layout, x, y, width, height):
    if x + width + 1 > BIN_WIDTH or y + height + 1 > BIN_HEIGHT or x <
0 or y < 0:
        return True
    return np.any(bin_layout[max(0, y - 1):min(BIN_HEIGHT, y + height
+ 1), max(0, x - 1):min(BIN_WIDTH, x + width + 1)] > 0)

def place_rectangle(rect_id, width, height, x=None, y=None):
    if x is None or y is None:
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        while is_overlap(bin_layout, x, y, width, height):
            x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
    bin_layout[y:y + height, x:x + width] = rect_id
    placements[rect_id] = (x, y, width, height)

def place_close_to(rect_id, width, height, close_to_ids):
    placed_coords = [placements[close_id] for close_id in close_to_ids
if close_id in placements]
    if placed_coords:
        for px, py, pwidth, pheight in placed_coords:
            for dx in range(-width - 1, width + 2):
                for dy in range(-height - 1, height + 2):
                    x = max(0, min(BIN_WIDTH - width, px + dx))
                    y = max(0, min(BIN_HEIGHT - height, py + dy))
                    if not is_overlap(bin_layout, x, y, width,
height):
                        place_rectangle(rect_id, width, height, x, y)
                        return True
    return False

place_rectangle(1, rectangles[0][0], rectangles[0][1],
x=random.randint(0, BIN_WIDTH - rectangles[0][0]), y=0)
place_rectangle(2, rectangles[1][0], rectangles[1][1],
x=random.randint(0, BIN_WIDTH - rectangles[1][0]), y=BIN_HEIGHT -
rectangles[1][1])
place_rectangle(4, rectangles[3][0], rectangles[3][1])
place_rectangle(5, rectangles[4][0], rectangles[4][1])
place_rectangle(9, rectangles[8][0], rectangles[8][1])
place_close_to(3, rectangles[2][0], rectangles[2][1], constraints[3]
["close_to"])
place_rectangle(6, rectangles[5][0], rectangles[5][1])
place_close_to(7, rectangles[6][0], rectangles[6][1], constraints[7]
["close_to"])
```

```
def place_remaining():
    for rect_id, (width, height) in enumerate(rectangles, start=1):
        if rect_id not in placements:
            while True:
                x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
                if not is_overlap(bin_layout, x, y, width, height):
                    place_rectangle(rect_id, width, height, x, y)
                    break

place_remaining()

class GCNN(nn.Module):
    def __init__(self, num_features, hidden_dim, output_dim):
        super(GCNN, self).__init__()
        self.conv1 = GCNConv(num_features, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, output_dim)

    def forward(self, data):
        x, edge_index = data.x, data.edge_index
        x = self.conv1(x, edge_index)
        x = torch.relu(x)
        x = self.conv2(x, edge_index)
        return x

G = nx.Graph()
for rect, props in constraints.items():
    G.add_node(rect)
    if "close_to" in props:
        for neighbor in props["close_to"]:
            G.add_edge(rect, neighbor)

edges = [(edge[0] - 1, edge[1] - 1) for edge in list(G.edges())]
edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()

node_features = torch.tensor([list(rect) for rect in rectangles],
dtype=torch.float)

data = Data(x=node_features, edge_index=edge_index)

gcnn = GCNN(num_features=2, hidden_dim=16, output_dim=2)
optimizer = optim.Adam(gcnn.parameters(), lr=0.01)

def reward_function(placements, bin_layout, constraints):
    reward = 0
    area_used = np.sum(bin_layout > 0)
    reward += area_used

    for rect_id, props in constraints.items():
        if "close_to" in props:
```

```
            for neighbor_id in props["close_to"]:
                if neighbor_id in placements:
                    x1, y1, w1, h1 = placements[rect_id]
                    x2, y2, w2, h2 = placements[neighbor_id]
                    distance = np.linalg.norm(np.array([x1 + w1 / 2,
y1 + h1 / 2]) - np.array([x2 + w2 / 2, y2 + h2 / 2]))
                    if distance < 5:
                        reward += 10

    for rect_id, (x, y, w, h) in placements.items():
        if np.any(bin_layout[y:y + h, x:x + w] > 1):
            reward -= 50

    return torch.tensor(reward, dtype=torch.float, requires_grad=True)

def map_gcnn_to_placements(output):
    placements = {}
    for i, out in enumerate(output):
        rect_id = i + 1
        width, height = rectangles[i]
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        placements[rect_id] = (x, y, width, height)
    return placements

def train_rl(agent, optimizer, epochs=100):
    for epoch in range(epochs):
        optimizer.zero_grad()
        out = agent(data)
        placements = map_gcnn_to_placements(out)
        reward = reward_function(placements, bin_layout, constraints)

        reward.backward()
        optimizer.step()

def visualize_placements():
    plt.figure(figsize=(10, 5))
    for rect_id, (x, y, width, height) in placements.items():
        plt.gca().add_patch(plt.Rectangle((x, y), width, height,
edgecolor='black', facecolor='blue', alpha=0.5))
        plt.text(x + width / 2, y + height / 2, f"{rect_id}",
ha='center', va='center', color='white')

    plt.xlim(0, BIN_WIDTH)
    plt.ylim(0, BIN_HEIGHT)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title("Optimized Placement of Rectangles (with GCNN)")
    plt.xlabel("Width")
    plt.ylabel("Height")
    plt.grid(True)
```

```
    plt.show()

train_rl(gcnn, optimizer, epochs=100)

visualize_placements()
```



THIRD ITERATION

```
import numpy as np
import matplotlib.pyplot as plt
import networkx as nx
import random
import torch
import torch.nn as nn
import torch.optim as optim

BIN_WIDTH = 80
BIN_HEIGHT = 40

np.random.seed(42)

def generate_rectangle_sizes(num_rectangles, max_width, max_height):
    return [
        (random.randint(5, max_width), random.randint(5, max_height))
        for _ in range(num_rectangles)
    ]
```

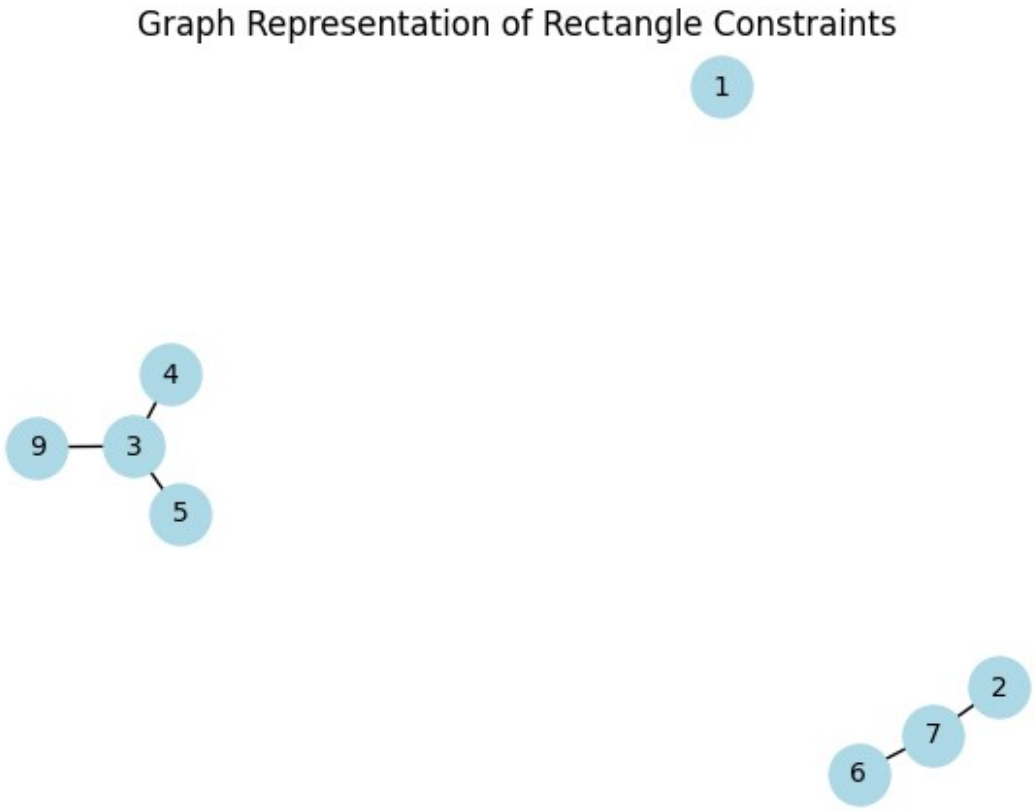
```
rectangles = generate_rectangle_sizes(9, 20, 15)

constraints = {
    1: {"position": "top"},
    2: {"position": "bottom"},
    3: {"close_to": [4, 5, 9]},
    7: {"close_to": [6, 2]},
}

G = nx.Graph()
for rect, props in constraints.items():
    G.add_node(rect)
    if "close_to" in props:
        for neighbor in props["close_to"]:
            G.add_edge(rect, neighbor)

plt.figure(figsize=(6, 4))
nx.draw(G, with_labels=True, node_color='lightblue', node_size=500,
font_size=10)
plt.title("Graph Representation of Rectangle Constraints")
plt.show()

bin_layout = np.zeros((BIN_HEIGHT, BIN_WIDTH), dtype=int)
placements = {}

def is_overlap(bin_layout, x, y, width, height):
    if x + width + 1 > BIN_WIDTH or y + height + 1 > BIN_HEIGHT or x <
0 or y < 0:
        return True
    return np.any(bin_layout[max(0, y - 1):min(BIN_HEIGHT, y + height
+ 1), max(0, x - 1):min(BIN_WIDTH, x + width + 1)] > 0)

def place_rectangle(rect_id, width, height, x=None, y=None):
    if x is None or y is None:
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        while is_overlap(bin_layout, x, y, width, height):
            x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
    bin_layout[y:y + height, x:x + width] = rect_id
    placements[rect_id] = (x, y, width, height)

place_rectangle(1, rectangles[0][0], rectangles[0][1],
x=random.randint(0, BIN_WIDTH - rectangles[0][0]), y=0)
place_rectangle(2, rectangles[1][0], rectangles[1][1],
x=random.randint(0, BIN_WIDTH - rectangles[1][0]), y=BIN_HEIGHT -
rectangles[1][1])

def place_close_to(rect_id, width, height, close_to_ids):
```

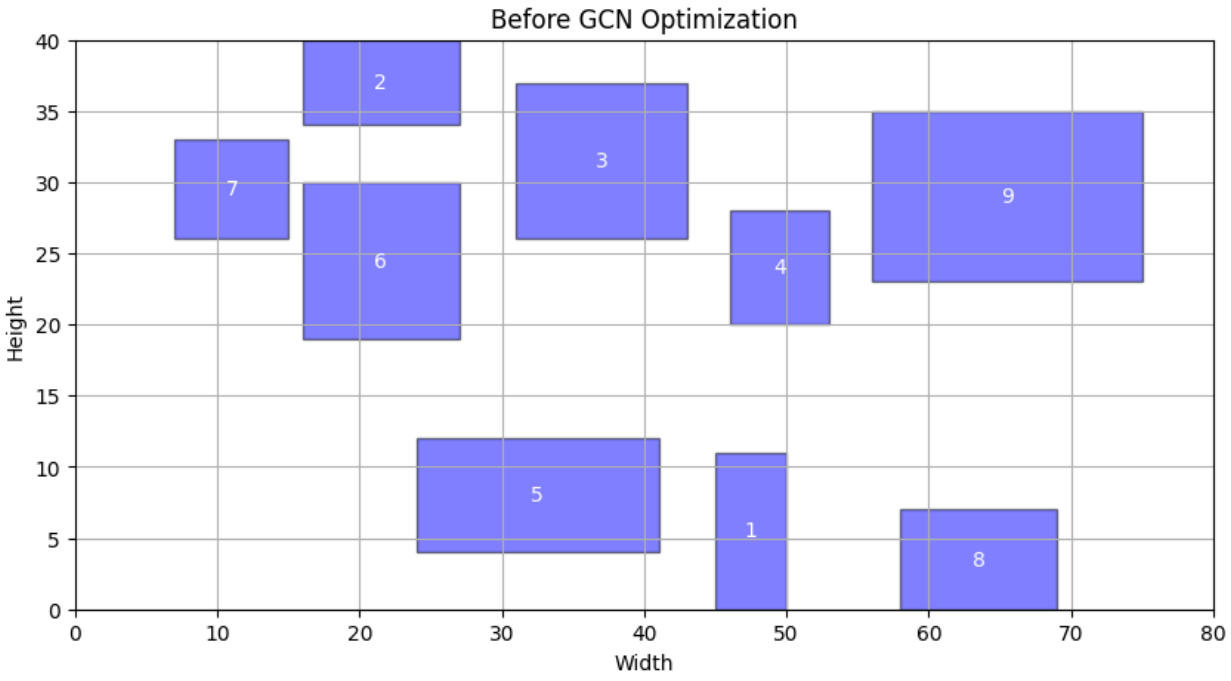
```
    placed_coords = [placements[close_id] for close_id in close_to_ids
if close_id in placements]
    if placed_coords:
        for px, py, pwidth, pheight in placed_coords:
            for dx in range(-width - 1, width + 2):
                for dy in range(-height - 1, height + 2):
                    x = max(0, min(BIN_WIDTH - width, px + dx))
                    y = max(0, min(BIN_HEIGHT - height, py + dy))
                    if not is_overlap(bin_layout, x, y, width,
height):
                        place_rectangle(rect_id, width, height, x, y)
                        return True
    return False

place_close_to(3, rectangles[2][0], rectangles[2][1], constraints[3]
["close_to"])
place_close_to(7, rectangles[6][0], rectangles[6][1], constraints[7]
["close_to"])

def place_remaining():
    for rect_id, (width, height) in enumerate(rectangles, start=1):
        if rect_id not in placements:
            while True:
                x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
                if not is_overlap(bin_layout, x, y, width, height):
                    place_rectangle(rect_id, width, height, x, y)
                    break

place_remaining()

class GCN(nn.Module):
    def __init__(self, in_features, out_features):
        super(GCN, self).__init__()
        self.conv1 = nn.Conv1d(in_channels=in_features,
out_channels=64, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=64,
out_channels=out_features, kernel_size=1)
        self.fc1 = nn.Linear(out_features, 128)
        self.fc2 = nn.Linear(128, 2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = x.permute(0, 2, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x.squeeze(0)

def prepare_graph_data(constraints, rectangles):
```

```
    num_rectangles = len(rectangles)
    adjacency_matrix = np.zeros((num_rectangles, num_rectangles))
    feature_matrix = np.zeros((num_rectangles, 4))

    for rect_id, props in constraints.items():
        if "close_to" in props:
            for neighbor in props["close_to"]:
                adjacency_matrix[rect_id - 1, neighbor - 1] = 1
                adjacency_matrix[neighbor - 1, rect_id - 1] = 1

    for rect_id, (width, height) in enumerate(rectangles, start=1):
        x, y, _, _ = placements.get(rect_id, (random.randint(0,
BIN_WIDTH - width), random.randint(0, BIN_HEIGHT - height), width,
height))
        feature_matrix[rect_id - 1] = [x, y, width, height]

    adjacency_matrix = torch.tensor(adjacency_matrix,
dtype=torch.float32)
    feature_matrix = torch.tensor(feature_matrix, dtype=torch.float32,
requires_grad=True)

    return adjacency_matrix, feature_matrix

def optimize_with_gcn():
    adjacency_matrix, feature_matrix = prepare_graph_data(constraints,
rectangles)
    model = GCN(in_features=4, out_features=2)
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    feature_matrix = feature_matrix.unsqueeze(0).transpose(1, 2)

    for epoch in range(5000):
        model.train()
        output = model(feature_matrix)

        if output.size(0) != len(rectangles) or output.size(1) != 2:
            raise ValueError("GCN output must have shape
[num_rectangles, 2]. Got: {}".format(output.shape))

        for i in range(output.size(0)):
            rect_id = i + 1
            if rect_id in [1, 2]:
                new_x = output[i][0].detach().numpy()
                _, y, width, height = placements[rect_id]

                new_x = max(0, min(BIN_WIDTH - width, int(new_x)))

                if not is_overlap(bin_layout, new_x, y, width,
height):
                    x, y, w, h = placements[rect_id]
```

```
                    bin_layout[y:y + h, x:x + w] = 0

                    placements[rect_id] = (new_x, y, width, height)
                    bin_layout[y:y + height, new_x:new_x + width] =
rect_id
                continue

            new_x, new_y = output[i].detach().numpy()
            _, _, width, height = placements[rect_id]

            new_x = max(0, min(BIN_WIDTH - width, int(new_x)))
            new_y = max(0, min(BIN_HEIGHT - height, int(new_y)))

            if not is_overlap(bin_layout, new_x, new_y, width,
height):
                x, y, w, h = placements[rect_id]
                bin_layout[y:y + h, x:x + w] = 0

                placements[rect_id] = (new_x, new_y, width, height)
                bin_layout[new_y:new_y + height, new_x:new_x + width]
= rect_id

        loss = torch.mean((output - feature_matrix[0, :2].transpose(0,
1)) ** 2)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 50 == 0:
            print(f"Epoch [{epoch}/5000], Loss: {loss.item():.4f}")

def plot_placement(placements, width, height, title=""):
    plt.figure(figsize=(10, 5))
    for rect_id, (x, y, w, h) in placements.items():
        plt.gca().add_patch(plt.Rectangle((x, y), w, h,
edgecolor='black', facecolor='blue', alpha=0.5))
        plt.text(x + w / 2, y + h / 2, f"{rect_id}", ha='center',
va='center', color='white')

    plt.xlim(0, width)
    plt.ylim(0, height)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(title)
    plt.xlabel("Width")
    plt.ylabel("Height")
    plt.grid(True)
    plt.show()

plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "Before GCN
Optimization")
```

```
optimize_with_gcn()
plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "After GCN
Optimization")
```

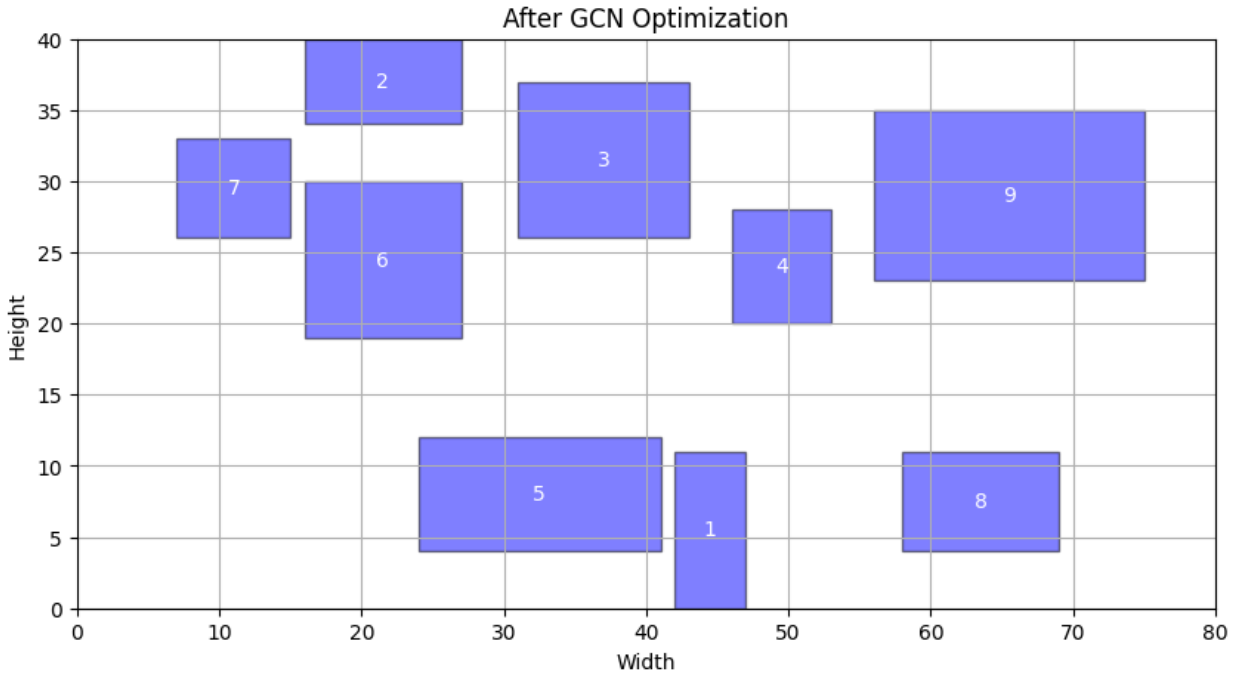





```
Epoch [0/5000], Loss: 904.3318
Epoch [50/5000], Loss: 115.5095
Epoch [100/5000], Loss: 10.6306
Epoch [150/5000], Loss: 1.9557
Epoch [200/5000], Loss: 0.6713
Epoch [250/5000], Loss: 0.2707
Epoch [300/5000], Loss: 0.1227
Epoch [350/5000], Loss: 0.0686
Epoch [400/5000], Loss: 0.0436
Epoch [450/5000], Loss: 0.0274
Epoch [500/5000], Loss: 0.0169
Epoch [550/5000], Loss: 0.0101
Epoch [600/5000], Loss: 0.0059
Epoch [650/5000], Loss: 0.0035
Epoch [700/5000], Loss: 0.0022
Epoch [750/5000], Loss: 0.0013
Epoch [800/5000], Loss: 0.0008
Epoch [850/5000], Loss: 0.0005
Epoch [900/5000], Loss: 0.0003
Epoch [950/5000], Loss: 0.0002
Epoch [1000/5000], Loss: 0.0001
Epoch [1050/5000], Loss: 0.0001
Epoch [1100/5000], Loss: 0.0001
Epoch [1150/5000], Loss: 0.0000
Epoch [1200/5000], Loss: 0.0000
Epoch [1250/5000], Loss: 0.0000
Epoch [1300/5000], Loss: 0.0000
Epoch [1350/5000], Loss: 0.0000
```

```
Epoch [1400/5000], Loss: 0.0000
Epoch [1450/5000], Loss: 0.0000
Epoch [1500/5000], Loss: 0.0000
Epoch [1550/5000], Loss: 0.0000
Epoch [1600/5000], Loss: 0.0000
Epoch [1650/5000], Loss: 0.0000
Epoch [1700/5000], Loss: 0.0000
Epoch [1750/5000], Loss: 0.0000
Epoch [1800/5000], Loss: 0.0000
Epoch [1850/5000], Loss: 0.0000
Epoch [1900/5000], Loss: 0.0000
Epoch [1950/5000], Loss: 0.0000
Epoch [2000/5000], Loss: 0.0000
Epoch [2050/5000], Loss: 0.0000
Epoch [2100/5000], Loss: 0.0000
Epoch [2150/5000], Loss: 0.0000
Epoch [2200/5000], Loss: 0.0000
Epoch [2250/5000], Loss: 0.0000
Epoch [2300/5000], Loss: 0.0000
Epoch [2350/5000], Loss: 0.0000
Epoch [2400/5000], Loss: 0.0000
Epoch [2450/5000], Loss: 0.0000
Epoch [2500/5000], Loss: 0.0000
Epoch [2550/5000], Loss: 0.0000
Epoch [2600/5000], Loss: 0.0000
Epoch [2650/5000], Loss: 0.0000
Epoch [2700/5000], Loss: 0.0000
Epoch [2750/5000], Loss: 0.0000
Epoch [2800/5000], Loss: 0.0000
Epoch [2850/5000], Loss: 0.0000
Epoch [2900/5000], Loss: 0.0000
Epoch [2950/5000], Loss: 0.0000
Epoch [3000/5000], Loss: 0.0000
Epoch [3050/5000], Loss: 0.0000
Epoch [3100/5000], Loss: 0.0000
Epoch [3150/5000], Loss: 0.0000
Epoch [3200/5000], Loss: 0.0000
Epoch [3250/5000], Loss: 0.0000
Epoch [3300/5000], Loss: 0.0000
Epoch [3350/5000], Loss: 0.0000
Epoch [3400/5000], Loss: 0.0000
Epoch [3450/5000], Loss: 0.0000
Epoch [3500/5000], Loss: 0.0000
Epoch [3550/5000], Loss: 0.0000
Epoch [3600/5000], Loss: 0.0000
Epoch [3650/5000], Loss: 0.0000
Epoch [3700/5000], Loss: 0.0000
Epoch [3750/5000], Loss: 0.0000
Epoch [3800/5000], Loss: 0.0000
```

```
Epoch [3850/5000], Loss: 0.0000
Epoch [3900/5000], Loss: 0.0000
Epoch [3950/5000], Loss: 0.0000
Epoch [4000/5000], Loss: 0.0000
Epoch [4050/5000], Loss: 0.0000
Epoch [4100/5000], Loss: 0.0000
Epoch [4150/5000], Loss: 0.0000
Epoch [4200/5000], Loss: 0.0000
Epoch [4250/5000], Loss: 0.0000
Epoch [4300/5000], Loss: 0.0000
Epoch [4350/5000], Loss: 0.0000
Epoch [4400/5000], Loss: 0.0000
Epoch [4450/5000], Loss: 0.0000
Epoch [4500/5000], Loss: 0.0000
Epoch [4550/5000], Loss: 0.0000
Epoch [4600/5000], Loss: 0.0000
Epoch [4650/5000], Loss: 0.0000
Epoch [4700/5000], Loss: 0.0000
Epoch [4750/5000], Loss: 0.0000
Epoch [4800/5000], Loss: 0.0000
Epoch [4850/5000], Loss: 0.0000
Epoch [4900/5000], Loss: 0.0000
Epoch [4950/5000], Loss: 0.0000
```



FOURTH ITERATION

```
import numpy as np
import matplotlib.pyplot as plt
```

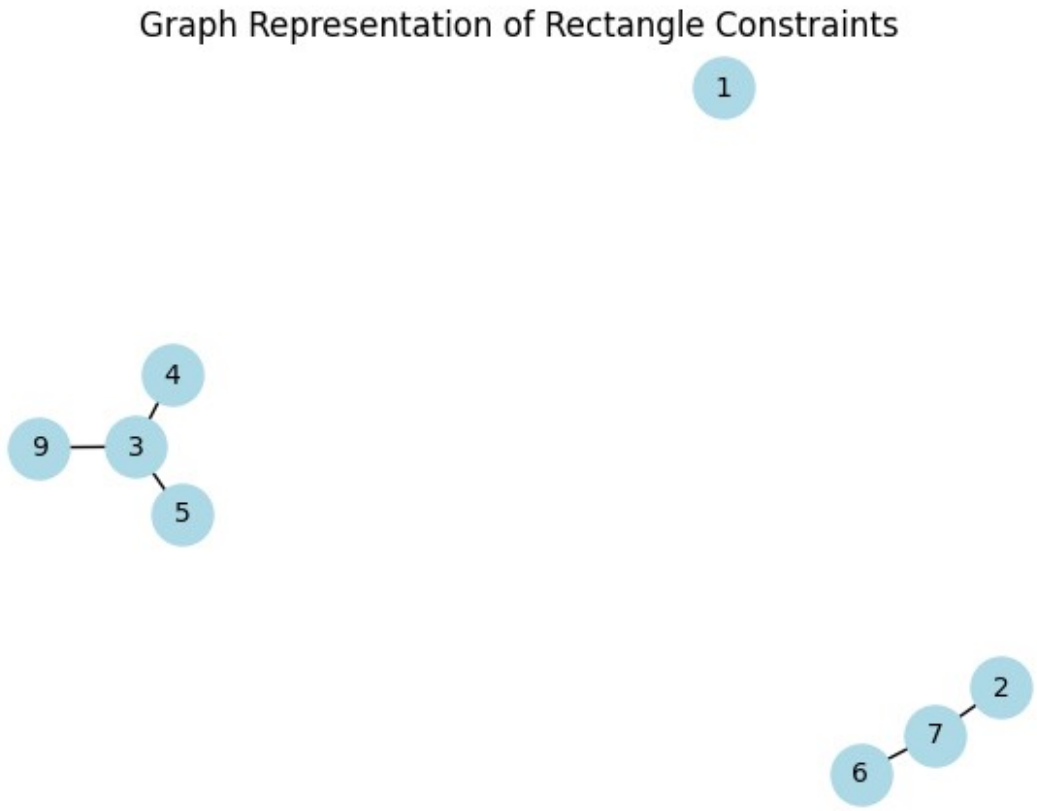
```
import networkx as nx
import random
import torch
import torch.nn as nn
import torch.optim as optim

BIN_WIDTH = 80
BIN_HEIGHT = 40

np.random.seed(42)

def generate_rectangle_sizes(num_rectangles, max_width, max_height):
    return [
        (random.randint(5, max_width), random.randint(5, max_height))
        for _ in range(num_rectangles)
    ]

rectangles = generate_rectangle_sizes(9, 20, 15)

constraints = {
    1: {"position": "top"},
    2: {"position": "bottom"},
    3: {"close_to": [4, 5, 9]},
    7: {"close_to": [6, 2]},
}

G = nx.Graph()
for rect, props in constraints.items():
    G.add_node(rect)
    if "close_to" in props:
        for neighbor in props["close_to"]:
            G.add_edge(rect, neighbor)

plt.figure(figsize=(6, 4))
nx.draw(G, with_labels=True, node_color='lightblue', node_size=500,
font_size=10)
plt.title("Graph Representation of Rectangle Constraints")
plt.show()

bin_layout = np.zeros((BIN_HEIGHT, BIN_WIDTH), dtype=int)
placements = {}

def is_overlap(bin_layout, x, y, width, height):
    if x + width + 1 > BIN_WIDTH or y + height + 1 > BIN_HEIGHT or x <
0 or y < 0:
        return True
    return np.any(bin_layout[max(0, y - 1):min(BIN_HEIGHT, y + height
+ 1), max(0, x - 1):min(BIN_WIDTH, x + width + 1)] > 0)

def place_rectangle(rect_id, width, height, x=None, y=None):
```

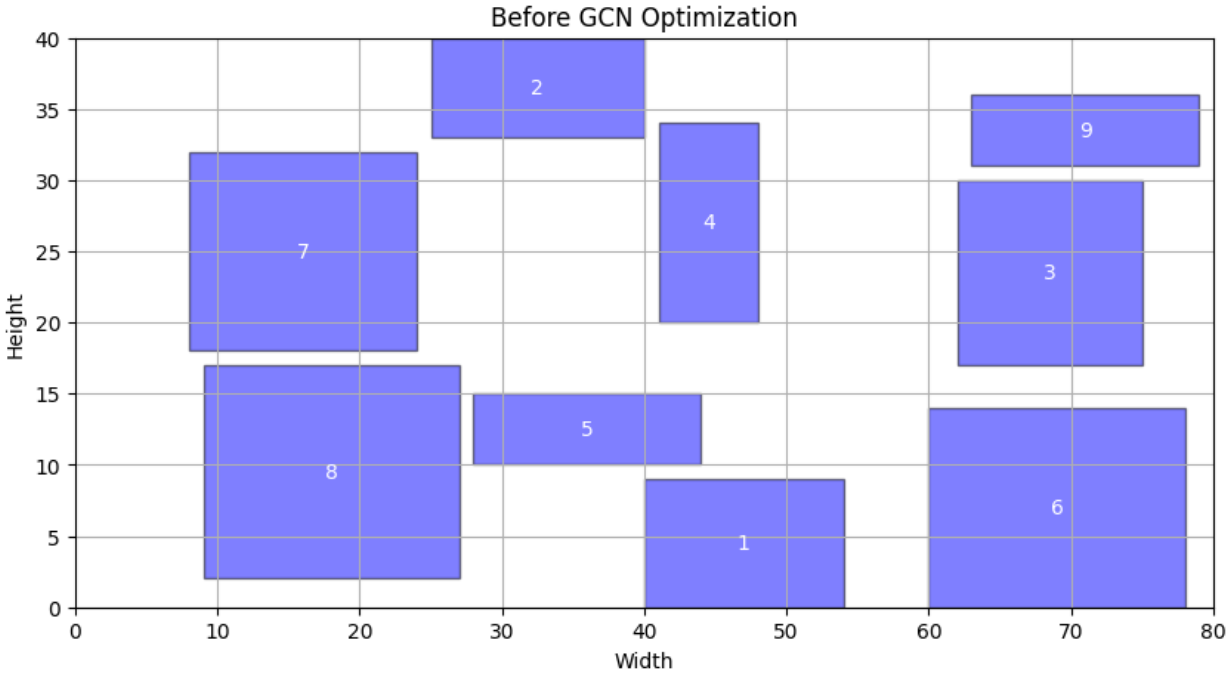
```
    if x is None or y is None:
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        while is_overlap(bin_layout, x, y, width, height):
            x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
    bin_layout[y:y + height, x:x + width] = rect_id
    placements[rect_id] = (x, y, width, height)

place_rectangle(1, rectangles[0][0], rectangles[0][1],
x=random.randint(0, BIN_WIDTH - rectangles[0][0]), y=0)
place_rectangle(2, rectangles[1][0], rectangles[1][1],
x=random.randint(0, BIN_WIDTH - rectangles[1][0]), y=BIN_HEIGHT -
rectangles[1][1])

def place_close_to(rect_id, width, height, close_to_ids):
    placed_coords = [placements[close_id] for close_id in close_to_ids
if close_id in placements]
    if placed_coords:
        for px, py, pwidth, pheight in placed_coords:
            for dx in range(-width - 1, width + 2):
                for dy in range(-height - 1, height + 2):
                    x = max(0, min(BIN_WIDTH - width, px + dx))
                    y = max(0, min(BIN_HEIGHT - height, py + dy))
                    if not is_overlap(bin_layout, x, y, width,
height):
                        place_rectangle(rect_id, width, height, x, y)
                        return True
    return False

place_close_to(3, rectangles[2][0], rectangles[2][1], constraints[3]
["close_to"])
place_close_to(7, rectangles[6][0], rectangles[6][1], constraints[7]
["close_to"])

def place_remaining():
    for rect_id, (width, height) in enumerate(rectangles, start=1):
        if rect_id not in placements:
            while True:
                x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
                if not is_overlap(bin_layout, x, y, width, height):
                    place_rectangle(rect_id, width, height, x, y)
                    break

place_remaining()

class GCN(nn.Module):
    def __init__(self, in_features, out_features):
        super(GCN, self).__init__()
```

```
        self.conv1 = nn.Conv1d(in_channels=in_features,
out_channels=64, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=64,
out_channels=out_features, kernel_size=1)
        self.fc1 = nn.Linear(out_features, 128)
        self.fc2 = nn.Linear(128, 2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = x.permute(0, 2, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x.squeeze(0)

def prepare_graph_data(constraints, rectangles):
    num_rectangles = len(rectangles)
    adjacency_matrix = np.zeros((num_rectangles, num_rectangles))
    feature_matrix = np.zeros((num_rectangles, 4))

    for rect_id, props in constraints.items():
        if "close_to" in props:
            for neighbor in props["close_to"]:
                adjacency_matrix[rect_id - 1, neighbor - 1] = 1
                adjacency_matrix[neighbor - 1, rect_id - 1] = 1

    for rect_id, (width, height) in enumerate(rectangles, start=1):
        x, y, _, _ = placements.get(rect_id, (random.randint(0,
BIN_WIDTH - width), random.randint(0, BIN_HEIGHT - height), width,
height))
        feature_matrix[rect_id - 1] = [x, y, width, height]

    adjacency_matrix = torch.tensor(adjacency_matrix,
dtype=torch.float32)
    feature_matrix = torch.tensor(feature_matrix, dtype=torch.float32,
requires_grad=True)

    return adjacency_matrix, feature_matrix

def optimize_with_gcn():
    adjacency_matrix, feature_matrix = prepare_graph_data(constraints,
rectangles)
    model = GCN(in_features=4, out_features=2)
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    feature_matrix = feature_matrix.unsqueeze(0).transpose(1, 2)

    for epoch in range(5000):
        model.train()
        output = model(feature_matrix)
```

```
        if output.size(0) != len(rectangles) or output.size(1) != 2:
            raise ValueError("GCN output must have shape
[num_rectangles, 2]. Got: {}".format(output.shape))

        for i in range(output.size(0)):
            rect_id = i + 1
            if rect_id in [1, 2]:
                continue

            new_x, new_y = output[i].detach().numpy()
            _, _, width, height = placements[rect_id]

            new_x = max(0, min(BIN_WIDTH - width, int(new_x)))
            new_y = max(0, min(BIN_HEIGHT - height, int(new_y)))

            if not is_overlap(bin_layout, new_x, new_y, width,
height):
                x, y, w, h = placements[rect_id]
                bin_layout[y:y + h, x:x + w] = 0

                placements[rect_id] = (new_x, new_y, width, height)
                bin_layout[new_y:new_y + height, new_x:new_x + width]
= rect_id

        loss = torch.mean((output - feature_matrix[0, :2].transpose(0,
1)) ** 2)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 50 == 0:
            print(f"Epoch [{epoch}/5000], Loss: {loss.item():.4f}")

def plot_placement(placements, width, height, title=""):
    plt.figure(figsize=(10, 5))
    for rect_id, (x, y, w, h) in placements.items():
        plt.gca().add_patch(plt.Rectangle((x, y), w, h,
edgecolor='black', facecolor='blue', alpha=0.5))
        plt.text(x + w / 2, y + h / 2, f"{rect_id}", ha='center',
va='center', color='white')

    plt.xlim(0, width)
    plt.ylim(0, height)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(title)
    plt.xlabel("Width")
    plt.ylabel("Height")
    plt.grid(True)
    plt.show()
```

```
plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "Before GCN
Optimization")
optimize_with_gcn()
plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "After GCN
Optimization")
```

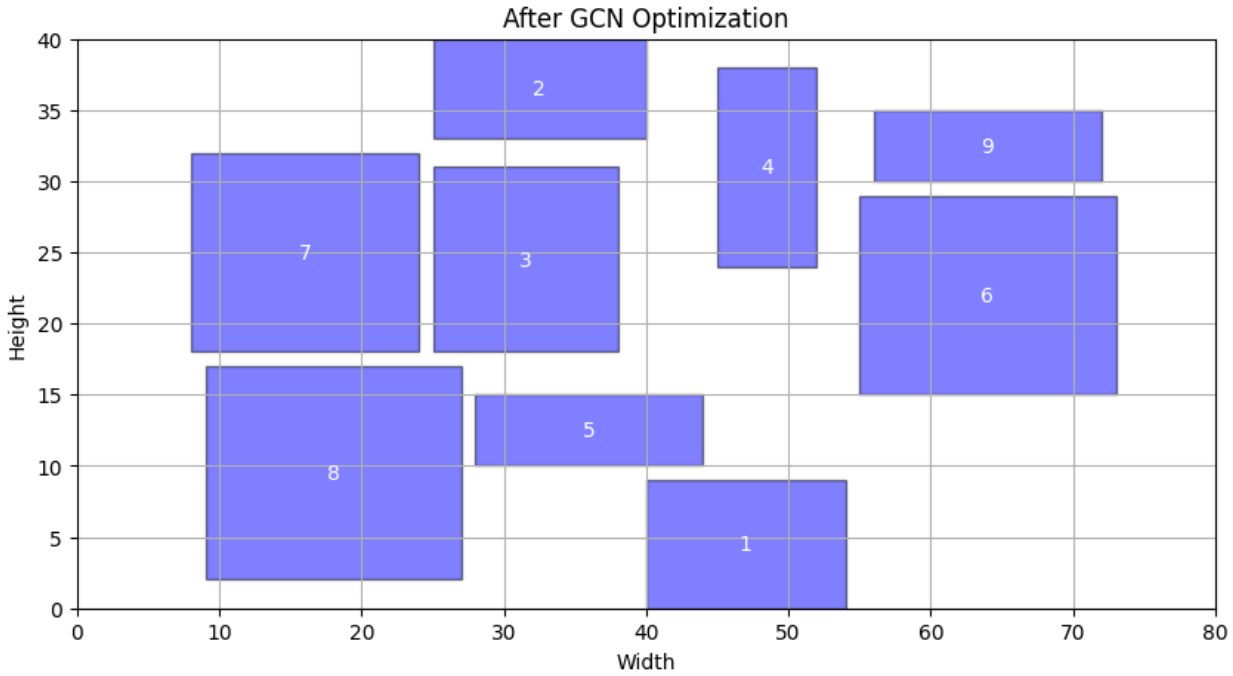





```
Epoch [0/5000], Loss: 1087.2544
Epoch [50/5000], Loss: 169.5217
Epoch [100/5000], Loss: 83.4068
Epoch [150/5000], Loss: 80.3617
Epoch [200/5000], Loss: 79.4989
Epoch [250/5000], Loss: 78.8301
Epoch [300/5000], Loss: 78.2761
Epoch [350/5000], Loss: 77.8084
Epoch [400/5000], Loss: 77.4010
Epoch [450/5000], Loss: 77.0308
Epoch [500/5000], Loss: 76.6753
Epoch [550/5000], Loss: 76.3186
Epoch [600/5000], Loss: 75.9474
Epoch [650/5000], Loss: 75.5524
Epoch [700/5000], Loss: 75.1295
Epoch [750/5000], Loss: 74.6745
Epoch [800/5000], Loss: 74.1876
Epoch [850/5000], Loss: 73.6724
Epoch [900/5000], Loss: 73.1358
Epoch [950/5000], Loss: 72.5877
Epoch [1000/5000], Loss: 72.0401
Epoch [1050/5000], Loss: 71.5055
Epoch [1100/5000], Loss: 70.9975
Epoch [1150/5000], Loss: 70.5289
Epoch [1200/5000], Loss: 70.1072
Epoch [1250/5000], Loss: 69.7410
Epoch [1300/5000], Loss: 69.4324
Epoch [1350/5000], Loss: 69.0799
```

```
Epoch [1400/5000], Loss: 68.7320
Epoch [1450/5000], Loss: 68.4992
Epoch [1500/5000], Loss: 68.3048
Epoch [1550/5000], Loss: 68.1317
Epoch [1600/5000], Loss: 67.9654
Epoch [1650/5000], Loss: 67.7973
Epoch [1700/5000], Loss: 67.6112
Epoch [1750/5000], Loss: 67.3690
Epoch [1800/5000], Loss: 67.0623
Epoch [1850/5000], Loss: 66.6597
Epoch [1900/5000], Loss: 66.1920
Epoch [1950/5000], Loss: 65.5112
Epoch [2000/5000], Loss: 64.3071
Epoch [2050/5000], Loss: 63.0739
Epoch [2100/5000], Loss: 61.9542
Epoch [2150/5000], Loss: 60.9648
Epoch [2200/5000], Loss: 60.1699
Epoch [2250/5000], Loss: 59.5990
Epoch [2300/5000], Loss: 59.2163
Epoch [2350/5000], Loss: 58.9700
Epoch [2400/5000], Loss: 58.8074
Epoch [2450/5000], Loss: 58.6929
Epoch [2500/5000], Loss: 58.6065
Epoch [2550/5000], Loss: 58.5381
Epoch [2600/5000], Loss: 58.4825
Epoch [2650/5000], Loss: 58.4367
Epoch [2700/5000], Loss: 58.3991
Epoch [2750/5000], Loss: 58.3684
Epoch [2800/5000], Loss: 58.3434
Epoch [2850/5000], Loss: 58.3233
Epoch [2900/5000], Loss: 58.3071
Epoch [2950/5000], Loss: 58.2941
Epoch [3000/5000], Loss: 58.2837
Epoch [3050/5000], Loss: 58.2752
Epoch [3100/5000], Loss: 58.2682
Epoch [3150/5000], Loss: 58.2625
Epoch [3200/5000], Loss: 58.2577
Epoch [3250/5000], Loss: 58.2536
Epoch [3300/5000], Loss: 58.2501
Epoch [3350/5000], Loss: 58.2470
Epoch [3400/5000], Loss: 58.2444
Epoch [3450/5000], Loss: 58.2420
Epoch [3500/5000], Loss: 58.2399
Epoch [3550/5000], Loss: 58.2380
Epoch [3600/5000], Loss: 58.2363
Epoch [3650/5000], Loss: 58.2347
Epoch [3700/5000], Loss: 58.2333
Epoch [3750/5000], Loss: 58.2320
Epoch [3800/5000], Loss: 58.2309
```

```
Epoch [3850/5000], Loss: 58.2298
Epoch [3900/5000], Loss: 58.2289
Epoch [3950/5000], Loss: 58.2280
Epoch [4000/5000], Loss: 58.2272
Epoch [4050/5000], Loss: 58.2266
Epoch [4100/5000], Loss: 58.2259
Epoch [4150/5000], Loss: 58.2254
Epoch [4200/5000], Loss: 58.2249
Epoch [4250/5000], Loss: 58.2245
Epoch [4300/5000], Loss: 58.2241
Epoch [4350/5000], Loss: 58.2237
Epoch [4400/5000], Loss: 58.2234
Epoch [4450/5000], Loss: 58.2233
Epoch [4500/5000], Loss: 58.2229
Epoch [4550/5000], Loss: 58.2227
Epoch [4600/5000], Loss: 58.2225
Epoch [4650/5000], Loss: 58.2223
Epoch [4700/5000], Loss: 58.2222
Epoch [4750/5000], Loss: 58.2220
Epoch [4800/5000], Loss: 58.2220
Epoch [4850/5000], Loss: 58.2218
Epoch [4900/5000], Loss: 58.2217
Epoch [4950/5000], Loss: 58.2217
```



FIFTH ITERATION

```
import numpy as np
import matplotlib.pyplot as plt
```

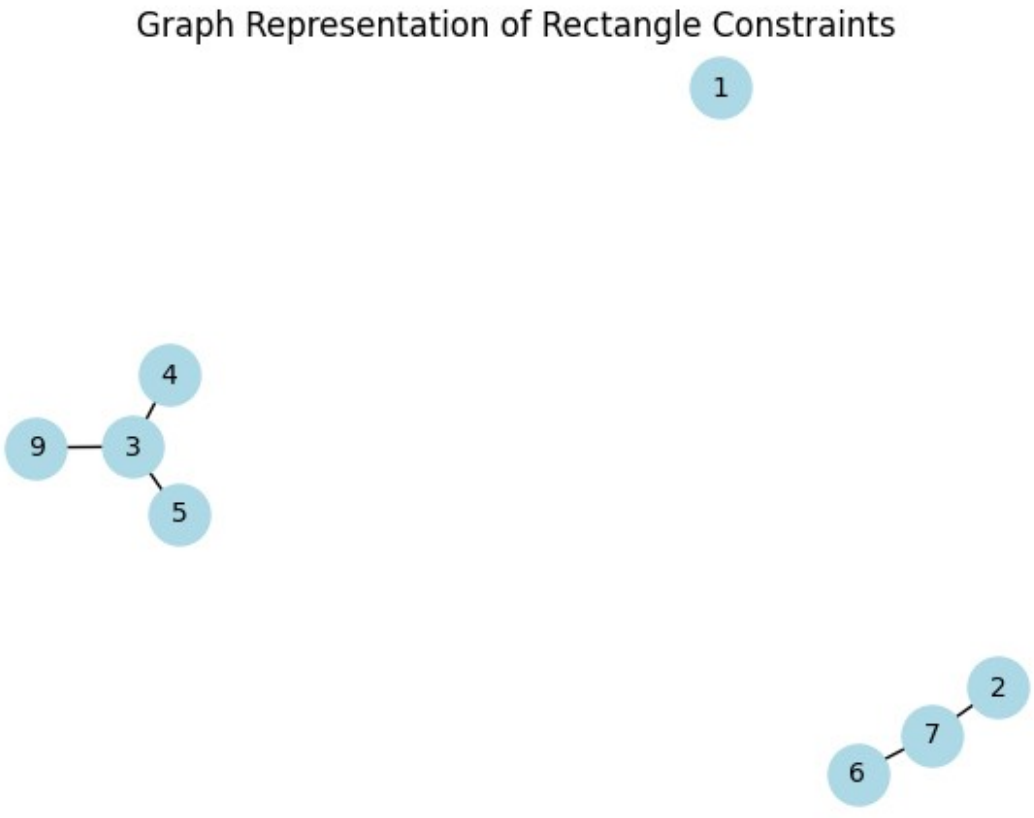
```
import networkx as nx
import random
import torch
import torch.nn as nn
import torch.optim as optim

BIN_WIDTH = 80
BIN_HEIGHT = 40

np.random.seed(42)

def generate_rectangle_sizes(num_rectangles, max_width, max_height):
    return [
        (random.randint(5, max_width), random.randint(5, max_height))
        for _ in range(num_rectangles)
    ]

rectangles = generate_rectangle_sizes(9, 20, 15)

constraints = {
    1: {"position": "top"},
    2: {"position": "bottom"},
    3: {"close_to": [4, 5, 9]},
    7: {"close_to": [6, 2]},
}

G = nx.Graph()
for rect, props in constraints.items():
    G.add_node(rect)
    if "close_to" in props:
        for neighbor in props["close_to"]:
            G.add_edge(rect, neighbor)

plt.figure(figsize=(6, 4))
nx.draw(G, with_labels=True, node_color='lightblue', node_size=500,
font_size=10)
plt.title("Graph Representation of Rectangle Constraints")
plt.show()

bin_layout = np.zeros((BIN_HEIGHT, BIN_WIDTH), dtype=int)
placements = {}

def is_overlap(bin_layout, x, y, width, height):
    if x + width + 1 > BIN_WIDTH or y + height + 1 > BIN_HEIGHT or x <
0 or y < 0:
        return True
    return np.any(bin_layout[max(0, y - 1):min(BIN_HEIGHT, y + height
+ 1), max(0, x - 1):min(BIN_WIDTH, x + width + 1)] > 0)

def place_rectangle(rect_id, width, height, x=None, y=None):
```

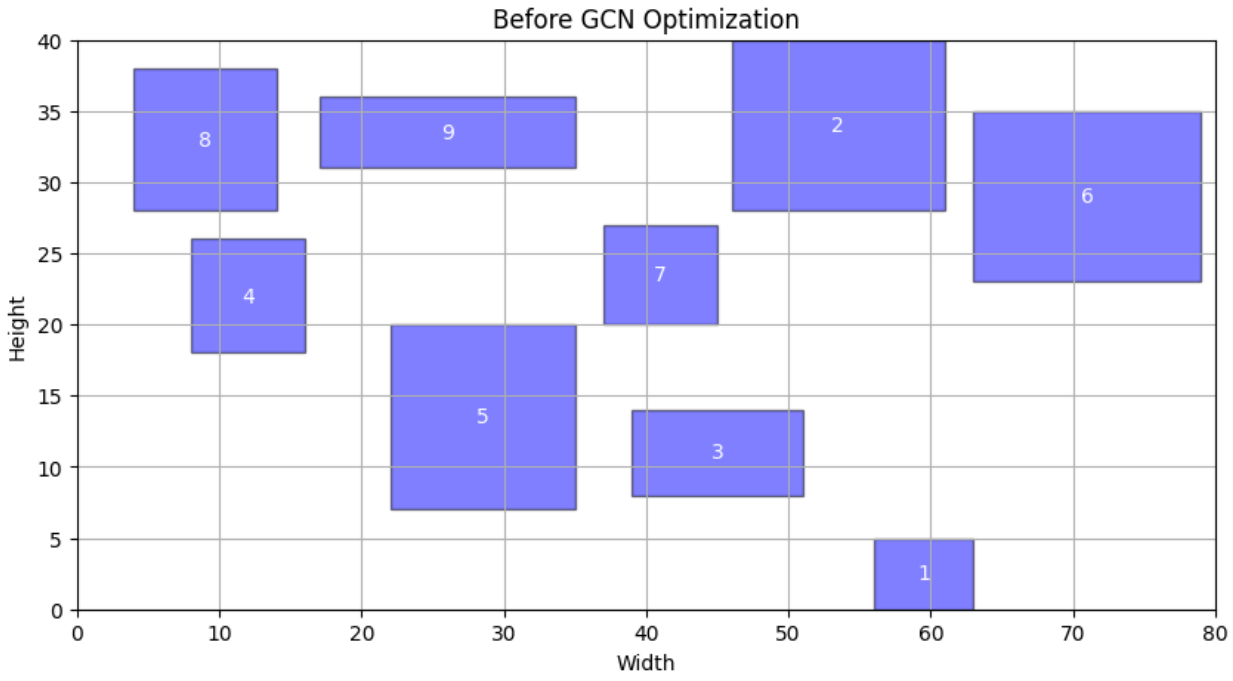
```
    if x is None or y is None:
        x, y = random.randint(0, BIN_WIDTH - width), random.randint(0,
BIN_HEIGHT - height)
        while is_overlap(bin_layout, x, y, width, height):
            x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
    bin_layout[y:y + height, x:x + width] = rect_id
    placements[rect_id] = (x, y, width, height)

place_rectangle(1, rectangles[0][0], rectangles[0][1],
x=random.randint(0, BIN_WIDTH - rectangles[0][0]), y=0)
place_rectangle(2, rectangles[1][0], rectangles[1][1],
x=random.randint(0, BIN_WIDTH - rectangles[1][0]), y=BIN_HEIGHT -
rectangles[1][1])

def place_close_to(rect_id, width, height, close_to_ids):
    placed_coords = [placements[close_id] for close_id in close_to_ids
if close_id in placements]
    if placed_coords:
        for px, py, pwidth, pheight in placed_coords:
            for dx in range(-width - 1, width + 2):
                for dy in range(-height - 1, height + 2):
                    x = max(0, min(BIN_WIDTH - width, px + dx))
                    y = max(0, min(BIN_HEIGHT - height, py + dy))
                    if not is_overlap(bin_layout, x, y, width,
height):
                        place_rectangle(rect_id, width, height, x, y)
                        return True
    return False

place_close_to(3, rectangles[2][0], rectangles[2][1], constraints[3]
["close_to"])
place_close_to(7, rectangles[6][0], rectangles[6][1], constraints[7]
["close_to"])

def place_remaining():
    for rect_id, (width, height) in enumerate(rectangles, start=1):
        if rect_id not in placements:
            while True:
                x, y = random.randint(0, BIN_WIDTH - width),
random.randint(0, BIN_HEIGHT - height)
                if not is_overlap(bin_layout, x, y, width, height):
                    place_rectangle(rect_id, width, height, x, y)
                    break

place_remaining()

class GCN(nn.Module):
    def __init__(self, in_features, out_features):
        super(GCN, self).__init__()
```

```
        self.conv1 = nn.Conv1d(in_channels=in_features,
out_channels=64, kernel_size=1)
        self.conv2 = nn.Conv1d(in_channels=64,
out_channels=out_features, kernel_size=1)
        self.fc1 = nn.Linear(out_features, 128)
        self.fc2 = nn.Linear(128, 2)

    def forward(self, x):
        x = torch.relu(self.conv1(x))
        x = torch.relu(self.conv2(x))
        x = x.permute(0, 2, 1)
        x = torch.relu(self.fc1(x))
        x = self.fc2(x)
        return x.squeeze(0)

def prepare_graph_data(constraints, rectangles):
    num_rectangles = len(rectangles)
    adjacency_matrix = np.zeros((num_rectangles, num_rectangles))
    feature_matrix = np.zeros((num_rectangles, 4))

    for rect_id, props in constraints.items():
        if "close_to" in props:
            for neighbor in props["close_to"]:
                adjacency_matrix[rect_id - 1, neighbor - 1] = 1
                adjacency_matrix[neighbor - 1, rect_id - 1] = 1

    for rect_id, (width, height) in enumerate(rectangles, start=1):
        x, y, _, _ = placements.get(rect_id, (random.randint(0,
BIN_WIDTH - width), random.randint(0, BIN_HEIGHT - height), width,
height))
        feature_matrix[rect_id - 1] = [x, y, width, height]

    adjacency_matrix = torch.tensor(adjacency_matrix,
dtype=torch.float32)
    feature_matrix = torch.tensor(feature_matrix, dtype=torch.float32,
requires_grad=True)

    return adjacency_matrix, feature_matrix

def optimize_with_gcn():
    adjacency_matrix, feature_matrix = prepare_graph_data(constraints,
rectangles)
    model = GCN(in_features=4, out_features=2)
    optimizer = optim.Adam(model.parameters(), lr=0.001)

    feature_matrix = feature_matrix.unsqueeze(0).transpose(1, 2)

    for epoch in range(5000):
        model.train()
        output = model(feature_matrix)
```

```
        if output.size(0) != len(rectangles) or output.size(1) != 2:
            raise ValueError("GCN output must have shape
[num_rectangles, 2]. Got: {}".format(output.shape))

        for i in range(output.size(0)):
            rect_id = i + 1
            if rect_id in [1, 2]:

                new_x = output[i][0].detach().numpy()
                _, y, width, height = placements[rect_id]

                new_x = max(0, min(BIN_WIDTH - width, int(new_x)))

                if not is_overlap(bin_layout, new_x, y, width,
height):
                    x, y, w, h = placements[rect_id]
                    bin_layout[y:y + h, x:x + w] = 0

                    placements[rect_id] = (new_x, y, width, height)
                    bin_layout[y:y + height, new_x:new_x + width] =
rect_id
                continue

            new_x, new_y = output[i].detach().numpy()
            _, _, width, height = placements[rect_id]

            new_x = max(0, min(BIN_WIDTH - width, int(new_x)))
            new_y = max(0, min(BIN_HEIGHT - height, int(new_y)))

            if not is_overlap(bin_layout, new_x, new_y, width,
height):
                x, y, w, h = placements[rect_id]
                bin_layout[y:y + h, x:x + w] = 0

                placements[rect_id] = (new_x, new_y, width, height)
                bin_layout[new_y:new_y + height, new_x:new_x + width]
= rect_id

        loss = torch.mean((output - feature_matrix[0, :2].transpose(0,
1)) ** 2)
        optimizer.zero_grad()
        loss.backward()
        optimizer.step()

        if epoch % 50 == 0:
            print(f"Epoch [{epoch}/5000], Loss: {loss.item():.4f}")

def plot_placement(placements, width, height, title=""):
    plt.figure(figsize=(10, 5))
```

```
    for rect_id, (x, y, w, h) in placements.items():
        plt.gca().add_patch(plt.Rectangle((x, y), w, h,
edgecolor='black', facecolor='blue', alpha=0.5))
        plt.text(x + w / 2, y + h / 2, f"{rect_id}", ha='center',
va='center', color='white')

    plt.xlim(0, width)
    plt.ylim(0, height)
    plt.gca().set_aspect('equal', adjustable='box')
    plt.title(title)
    plt.xlabel("Width")
    plt.ylabel("Height")
    plt.grid(True)
    plt.show()

plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "Before GCN
Optimization")
optimize_with_gcn()
plot_placement(placements, BIN_WIDTH, BIN_HEIGHT, "After GCN
Optimization")
```






```
Epoch [0/5000], Loss: 925.4675
Epoch [50/5000], Loss: 124.3803
Epoch [100/5000], Loss: 106.1619
Epoch [150/5000], Loss: 102.3072
Epoch [200/5000], Loss: 99.2896
Epoch [250/5000], Loss: 96.2614
Epoch [300/5000], Loss: 92.7668
Epoch [350/5000], Loss: 88.7624
Epoch [400/5000], Loss: 84.3428
Epoch [450/5000], Loss: 79.6214
Epoch [500/5000], Loss: 74.7360
Epoch [550/5000], Loss: 69.8698
Epoch [600/5000], Loss: 65.2318
Epoch [650/5000], Loss: 61.0037
Epoch [700/5000], Loss: 57.3265
Epoch [750/5000], Loss: 54.2747
Epoch [800/5000], Loss: 51.8789
Epoch [850/5000], Loss: 50.0851
Epoch [900/5000], Loss: 48.7997
Epoch [950/5000], Loss: 47.9139
Epoch [1000/5000], Loss: 47.3253
Epoch [1050/5000], Loss: 46.9278
Epoch [1100/5000], Loss: 46.6578
Epoch [1150/5000], Loss: 46.4848
Epoch [1200/5000], Loss: 46.3771
Epoch [1250/5000], Loss: 46.2870
Epoch [1300/5000], Loss: 46.2250
Epoch [1350/5000], Loss: 46.1733
```

```
Epoch [1400/5000], Loss: 46.1347
Epoch [1450/5000], Loss: 46.0952
Epoch [1500/5000], Loss: 46.0669
Epoch [1550/5000], Loss: 46.0396
Epoch [1600/5000], Loss: 46.0150
Epoch [1650/5000], Loss: 45.9913
Epoch [1700/5000], Loss: 45.9710
Epoch [1750/5000], Loss: 45.9586
Epoch [1800/5000], Loss: 45.9381
Epoch [1850/5000], Loss: 45.9249
Epoch [1900/5000], Loss: 45.9119
Epoch [1950/5000], Loss: 45.8962
Epoch [2000/5000], Loss: 45.8861
Epoch [2050/5000], Loss: 45.8686
Epoch [2100/5000], Loss: 45.8631
Epoch [2150/5000], Loss: 45.8610
Epoch [2200/5000], Loss: 45.8418
Epoch [2250/5000], Loss: 45.8358
Epoch [2300/5000], Loss: 45.8402
Epoch [2350/5000], Loss: 45.8235
Epoch [2400/5000], Loss: 45.8188
Epoch [2450/5000], Loss: 45.8140
Epoch [2500/5000], Loss: 45.8058
Epoch [2550/5000], Loss: 45.7992
Epoch [2600/5000], Loss: 45.7968
Epoch [2650/5000], Loss: 45.8020
Epoch [2700/5000], Loss: 45.7837
Epoch [2750/5000], Loss: 45.7761
Epoch [2800/5000], Loss: 45.7666
Epoch [2850/5000], Loss: 45.7628
Epoch [2900/5000], Loss: 45.7478
Epoch [2950/5000], Loss: 45.7226
Epoch [3000/5000], Loss: 45.6962
Epoch [3050/5000], Loss: 45.6911
Epoch [3100/5000], Loss: 45.6523
Epoch [3150/5000], Loss: 45.6153
Epoch [3200/5000], Loss: 45.5671
Epoch [3250/5000], Loss: 45.5186
Epoch [3300/5000], Loss: 45.4610
Epoch [3350/5000], Loss: 45.3967
Epoch [3400/5000], Loss: 45.3241
Epoch [3450/5000], Loss: 45.2443
Epoch [3500/5000], Loss: 45.1709
Epoch [3550/5000], Loss: 45.0911
Epoch [3600/5000], Loss: 45.0184
Epoch [3650/5000], Loss: 44.9239
Epoch [3700/5000], Loss: 44.8416
Epoch [3750/5000], Loss: 44.7624
Epoch [3800/5000], Loss: 44.7060
```

```
Epoch [3850/5000], Loss: 44.6376
Epoch [3900/5000], Loss: 44.5678
Epoch [3950/5000], Loss: 44.5540
Epoch [4000/5000], Loss: 44.4747
Epoch [4050/5000], Loss: 44.3996
Epoch [4100/5000], Loss: 44.3552
Epoch [4150/5000], Loss: 44.2789
Epoch [4200/5000], Loss: 44.2406
Epoch [4250/5000], Loss: 44.1663
Epoch [4300/5000], Loss: 44.1584
Epoch [4350/5000], Loss: 44.0960
Epoch [4400/5000], Loss: 44.0249
Epoch [4450/5000], Loss: 43.9939
Epoch [4500/5000], Loss: 43.9286
Epoch [4550/5000], Loss: 43.8949
Epoch [4600/5000], Loss: 43.8432
Epoch [4650/5000], Loss: 43.8106
Epoch [4700/5000], Loss: 43.7584
Epoch [4750/5000], Loss: 43.7033
Epoch [4800/5000], Loss: 43.6634
Epoch [4850/5000], Loss: 43.6228
Epoch [4900/5000], Loss: 43.5914
Epoch [4950/5000], Loss: 43.5197
```