Name: Kapil Wanaskar

Student ID: 016649880

Subject: CMPE 252: AI and Data Engineering

School: San Jose Satte University, California

Sem: Fall 2023

Homework 1

Instructions to Compile and Run Source Code

Prerequisites:

Ensure you have Python installed on your computer. If not, you can download and install Python from the official website. https://www.python.org/downloads/

Steps:

Step 1) Setup:

Place **"hw1.py"** python file in a directory of your choice.

Place the input file named **"input.txt"** in the same directory as **"hw1.py"** .

Step 2) Open Terminal or Command Prompt:

On Windows: Press **"Windows + R"** keys, type cmd, and press Enter.

On macOS: Press **"Cmd + Space"** , type terminal, and press Enter.

On Linux: Depending on the distro, you can use shortcuts or search for the terminal in the application menu.

Step 3) Navigate to the Directory:

Use the cd command followed by the directory path where you've saved hw1.py to navigate to the directory.

```
cd path_to_directory
```

Step 4) Run the Script:

Once in the correct directory, type the following command to execute the script:

```
python hw1.py
```

Step 5) Check the Output:

After running the script, an **"output.txt"** file will be generated in the same directory as the hw1.py script.

Open **"output.txt"** to view the results.

In [ ]:

CMPE252: Artificial Intelligence and Data Engineering

Homework 1

Due: Sep. 13, 9PM

Instructions. You must work on it individually. Please follow the submission instructions exactly. You need to submit your assignment online by the due date. It is okay to scan your answers and create the pdf submission.

Problem 1 — 100 points

Implement Dijkstra's algorithm to solve for minimum cost paths on a given graph. We will consider only 2D grid graphs in this assignment. The edge costs are the distances between the two vertices. Each vertex can be connected to at most eight of its neighbors (N, NW, W, SW, S, SE, E, NE).

Files. The link of files that you can download for this problem is `https://drive.google.com/drive/folders/12iGnOMY52mHQKYg-_nQ6jQG3lKweU6py?usp=sharing`

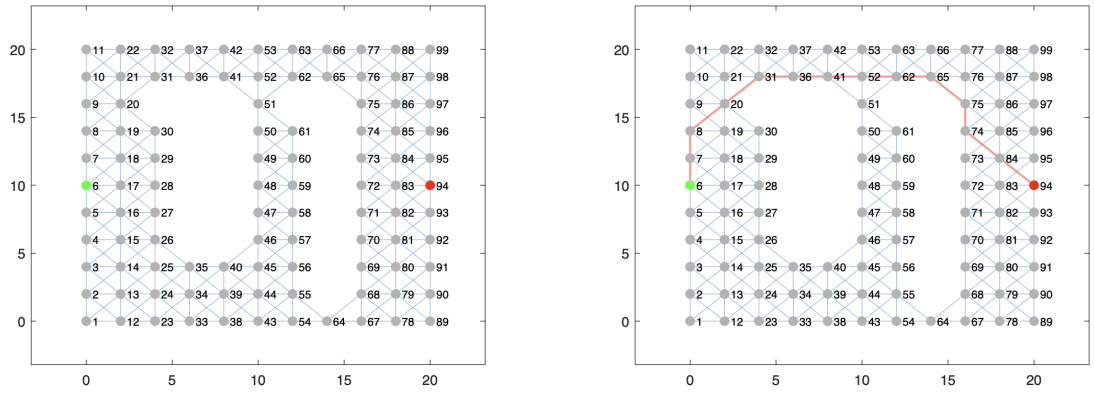
Input Description. You are given a grid environment. There are two input files corresponding to the given graph:

1. `input.txt`. This file has a specific format as given below:
   - The first line of the input file gives the total number of vertices, n, in the graph.
   - The second line gives the starting vertex index (indices go from 1 to n).
   - The third line gives the goal vertex index (indices go from 1 to n).
   - Starting from the fourth line, we have the edges in the graph specified in the form of an edge list. Each line specifies one edge: `i j wij` which indicates that there is a (directed) edge from `i` to `j` with a cost of `wij`. Note that this is a directed graph.
2. `coords.txt`. This file lists the x and y coordinates of the vertices starting with vertex numbered 1 on the first line.

Output Description. You may implement your algorithm using one of the following programming languages: MATLAB, C/C++, Python, or Java. The output file `output.txt` must have two lines that contains the vertices along the shortest path as well as the traveling cost from the starting vertex to the above corresponding vertices. The output file must have the following format:

- The first line must list the indices of the vertices on the shortest path only - from the start to the goal vertex. You need to choose the one with a smaller index if there are multiple options. For example, if there are two paths with indices " 1 3 2 ..." and "1 3 4 ..." that have the same traveling cost, you need to choose the first one.
- The second line must list the traveling cost from the starting vertex to the above corresponding vertices.

A sample input and output file are given on canvas. For example, the `input.txt` file contains 99 vertices. The `coords.txt` file specifies the actual coordinates of the 99 vertices starting with vertex numbered 1 to vertex numbered 99.



(a) (b)

Figure 1: Example

(a) (b)

Figure 1: Example

Submission Description. A sample input and output file are given. The requirements of your submission are as follows

1. You need to generate a video showing the result in each step using the provided inputs. Your video should demonstrate the result of your algorithm in each step that contains the following: the constructed graph and the paths that have been explored/updated until the current step. In the last step, you should plot the path you select according to the above criterion. Each part should be with a different color to differentiate. For example, in MATLAB, you can use the examples shown on this page (`https://www.mathworks.com/help/matlab/ref/graph.plot.html#buzeikk`) to display the graph. The video should be produced using code/package/plugin, rather than recorded by a external hardware (e.g., camera). Your video output should be named `hw1.mp4`. A sample image is shown in Figure 1.

2. You need to submit your source code. Your implementation must produce a single executable or single script named `hw1` that reads input file `input.txt` (will be put in the same directory as the executable or script) and produce a output file named `output.txt` in the same directory. You don't need to put the video-generating function into your code. You don't need to upload any `input.txt` or `output.txt` file. We will test your code on instances other than the provided sample. Make sure you follow the input/output conventions exactly.

3. You need to submit a pdf report named `hw1.pdf` describing your implementation. You should also include instructions on how to compile and run your code.

4. Please note that your code should ideally produce results within a few seconds when using the provided `input.txt` test case and no animation-generating function is included, given that the problem size is very small. During our code testing, if your code (without video-generating function) cannot generate a result within a 3-minute time-frame using the same level of problem size, your code submission will be deemed unsuccessful.

In conclusion, you submission should contain the following: `hw1.pdf`, `hw1.mp4`, and `hw1`. *Submission needs to be uploaded as a zip file.*

"input.txt"

In [ ]:
```
with open("input.txt", "r") as file:
    lines = file.readlines()

    # Display the first 5 lines
    for line in lines[:5]:
        print(line, end='')  # end='' prevents double line breaks

    print("\n...")  # Ellipsis for clarity that some lines are skipped

    # Display the last 5 lines
    for line in lines[-5:]:
        print(line, end='')
```

```
99
6
94
1 2 2.000000
1 12 2.000000

...
98 97 2.000000
98 99 2.000000
99 87 2.828427
99 88 2.000000
99 98 2.000000
```

A summary of Dijkstra's algorithm:

1) Create a set to keep track of visited vertices.

2) Assign a tentative distance value to every vertex. Set the initial node's distance to zero and all other nodes to infinity.

3) For the current node, consider all its unvisited neighbors and calculate their tentative distances. If the newly calculated tentative distance is less than the current assigned value, update the distance.

4) After considering all neighbors of the current node, mark the node as visited.

5) Select the unvisited node with the smallest tentative distance, and set it as the new current node. Return to step 3.

In [ ]:
```
import heapq

def parse_input(filename):
    # parse input file
    with open(filename, 'r') as f:
        # read input file
        n = int(f.readline().strip())
        # number of vertices
        start = int(f.readline().strip())
        # start vertex
        goal = int(f.readline().strip())
        # goal vertex
```

```
        edges = {}
        # edges dictionary (key: vertex, value: list of tuples (neighbor, weigh
        for line in f:
            # read edges and weights and add to edges dictionary
            u, v, w = line.strip().split()
            # u: vertex, v: neighbor, w: weight
            u, v, w = int(u), int(v), float(w)
            # convert to int and float as needed
            if u not in edges:
            # add vertex to edges dictionary if not already present
                edges[u] = []
                # initialize list of neighbors and weights for vertex u in edge
            edges[u].append((v, w))
            # add neighbor and weight to list of neighbors and weights for vert
    return n, start, goal, edges
    # return number of vertices, start vertex, goal vertex, and edges dictionar

def dijkstra(n, start, goal, edges):
# Dijkstra's algorithm to solve for minimum cost paths on a given graph
    visited = set()
    # initialize visited set
    distances = {vertex: float('infinity') for vertex in range(1, n + 1)}
    # initialize distances dictionary (key: vertex, value: distance)
    previous_vertices = {vertex: None for vertex in range(1, n + 1)}
    # initialize previous vertices dictionary (key: vertex, value: previous ver
    distances[start] = 0
    # set distance of start vertex to 0
    vertices = [(0, start)]
    # initialize vertices list (tuple: (distance, vertex))
    while vertices:
    # while vertices list is not empty (i.e., there are still vertices to visit
        current_distance, current_vertex = heapq.heappop(vertices)
        # pop vertex with minimum distance from vertices list
        if current_vertex in visited:
        # if vertex has already been visited, continue
            continue
            # continue to next iteration of while loop
        visited.add(current_vertex)
        # add vertex to visited set (i.e., mark vertex as visited)
        for neighbor, weight in edges.get(current_vertex, []):
        # for each neighbor of current vertex
            distance = current_distance + weight
            # calculate distance to neighbor
            if distance < distances[neighbor]:
            # if distance to neighbor is less than current distance to neighbor
                distances[neighbor] = distance
                # update distance to neighbor
                previous_vertices[neighbor] = current_vertex
                # update previous vertex of neighbor
                heapq.heappush(vertices, (distance, neighbor))
                # push neighbor to vertices list
    path, current_vertex = [], goal
    # initialize path list and current vertex
    while previous_vertices[current_vertex] is not None:
    # while current vertex has a previous vertex
        path.append(current_vertex)
        # append current vertex to path list
        current_vertex = previous_vertices[current_vertex]
        # update current vertex to previous vertex
    if current_vertex != start:
```

```
    # if current vertex is not start vertex
        return [], []
        # return empty path and empty distances
    path.append(start)
    # append start vertex to path list
    return path[::-1], [distances[vertex] for vertex in path[::-1]]
    # return path list in reverse order and distances list in reverse order

def write_output(path, distances, filename="output.txt"):
# write output file
    with open(filename, 'w') as f:
    # write output file (path and distances)
        f.write(' '.join(map(str, path)) + '\n')
        # write path to output file (space-separated)
        f.write(' '.join(['{:.4f}'.format(dist) for dist in distances]) + '\n')
        # write distances to output file (space-separated, formatted to 4 decim

n, start, goal, edges = parse_input('input.txt')
# parse input file and store number of vertices, start vertex, goal vertex, and
path, distances = dijkstra(n, start, goal, edges)
# run Dijkstra's algorithm to solve for minimum cost paths on a given graph and
write_output(path, distances)
# write output file (path and distances) to output.txt file
```

"output.txt"

In [ ]:
```
with open("output.txt", "r") as file:
    content = file.read()
    print(content)
```

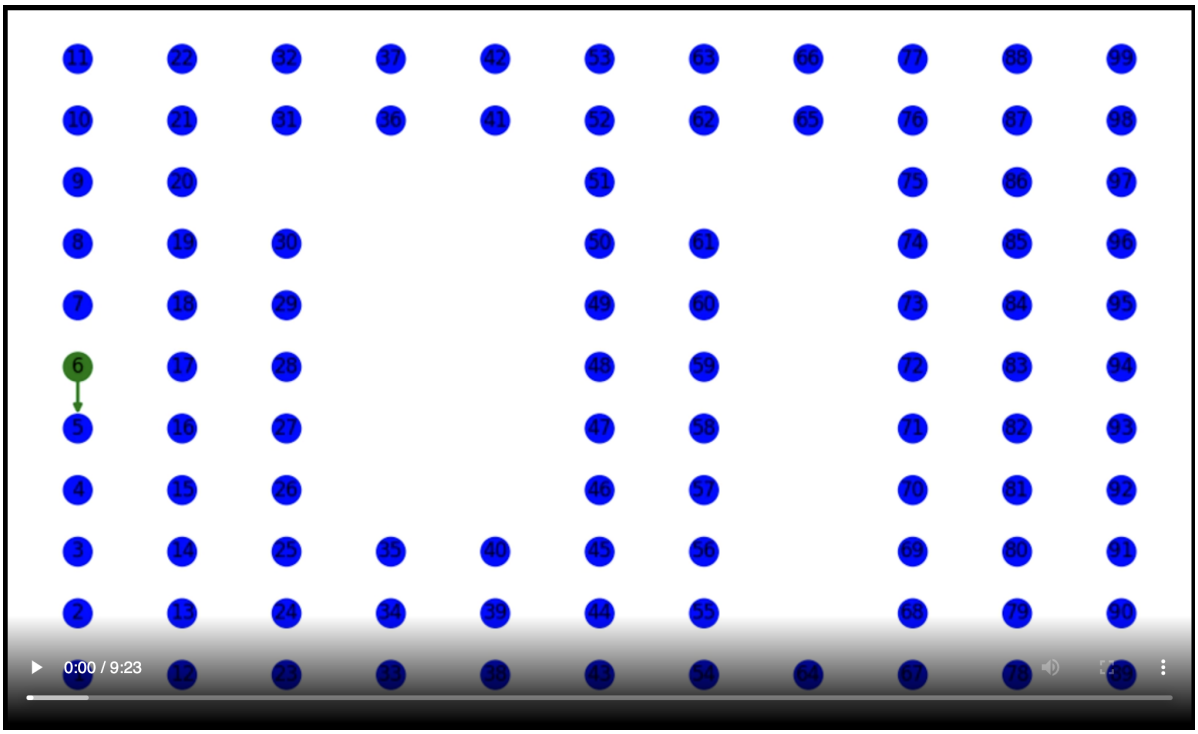
```
6 7 8 20 31 36 41 52 62 65 75 74 84 94
0.0000 2.0000 4.0000 6.8284 9.6569 11.6569 13.6569 15.6569 17.6569 19.6569 22.
4853 24.4853 27.3137 30.1421
```

"coords.txt"

In [ ]:
```
with open("coords.txt", "r") as file:
    lines = file.readlines()

    # Display the first 5 lines
    for line in lines[:5]:
        print(line, end='')  # end='' prevents double line breaks

    print("\n...")  # Ellipsis for clarity that some lines are skipped

    # Display the last 5 lines
    for line in lines[-5:]:
        print(line, end='')
```

```
0.0 0.0
0.0 2.0
0.0 4.0
0.0 6.0
0.0 8.0

...
20.0 12.0
20.0 14.0
20.0 16.0
20.0 18.0
20.0 20.0
```

Video Generation Code:



In [ ]:

```python
import heapq
# import heapq module (for priority queue)
import networkx as nx
# import networkx module (for graph visualization) and rename as nx
import matplotlib.pyplot as plt
# import matplotlib.pyplot module (for graph visualization) and rename as plt
from io import BytesIO
# import BytesIO module (for graph visualization)
import imageio.v2 as imageio
# import imageio module (for graph visualization)
import os
# import os module (for graph visualization)
os.environ['IMAGEIO_FFMPEG_EXE'] = '/opt/homebrew/bin/ffmpeg'
# set path to ffmpeg executable (for graph visualization)

def parse_input(filename):
    with open(filename, 'r') as f:
        n = int(f.readline().strip())
        start = int(f.readline().strip())
```

```
        goal = int(f.readline().strip())
        edges = {}
        for line in f:
            u, v, w = line.strip().split()
            u, v, w = int(u), int(v), float(w)
            if u not in edges:
                edges[u] = []
            edges[u].append((v, w))
    return n, start, goal, edges

def parse_coordinates(filename):
# parse coordinates file (for graph visualization)
    """
    Parses the 'coords.txt' file and returns a dictionary with vertex number as
    """
    coords = {}
    # initialize coordinates dictionary (key: vertex, value: (x, y) tuple)
    with open(filename, 'r') as f:
    # read coordinates file (for graph visualization)
        for index, line in enumerate(f, start=1):
        # for each line in coordinates file (for graph visualization) (enumerat
            x, y = map(float, line.strip().split())
            # read x and y coordinates and convert to float
            coords[index] = (x, y)
            # add vertex and (x, y) tuple to coordinates dictionary (for graph
    return coords
    # return coordinates dictionary (for graph visualization)


def dijkstra(n, start, goal, edges):
    visited = set()
    distances = {vertex: float('infinity') for vertex in range(1, n + 1)}
    previous_vertices = {vertex: None for vertex in range(1, n + 1)}
    distances[start] = 0
    vertices = [(0, start)]
    while vertices:
        current_distance, current_vertex = heapq.heappop(vertices)
        if current_vertex in visited:
            continue
        visited.add(current_vertex)
        for neighbor, weight in edges.get(current_vertex, []):
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                previous_vertices[neighbor] = current_vertex
                heapq.heappush(vertices, (distance, neighbor))
    path, current_vertex = [], goal
    while previous_vertices[current_vertex] is not None:
        path.append(current_vertex)
        current_vertex = previous_vertices[current_vertex]
    if current_vertex != start:
        return [], []
    path.append(start)
    return path[::-1], [distances[vertex] for vertex in path[::-1]]

def write_output(path, distances, filename="output.txt"):
    with open(filename, 'w') as f:
        f.write(' '.join(map(str, path)) + '\n')
        f.write(' '.join(['{:.4f}'.format(dist) for dist in distances]) + '\n')
```

```
def visualize_graph(edges, coords, active_vertex=None, partial_path=None, activ
    # visualize graph (for graph visualization)
    G = nx.DiGraph()
    # initialize graph (for graph visualization)

    # Add nodes and edges to the graph
    for u, v_w_list in edges.items():
    # for each vertex and list of neighbors and weights in edges dictionary (fo
        for v, w in v_w_list:
        # for each neighbor and weight in list of neighbors and weights
            G.add_edge(u, v, weight=w)
            # add edge to graph (for graph visualization)

    plt.figure(figsize=(10, 6))
    # set figure size (for graph visualization)

    node_colors = ['g' if node == active_vertex else 'b' for node in G.nodes()]
    # set node colors (for graph visualization)
    # green color for active vertex, blue color for all other vertices

    nx.draw_networkx_nodes(G, pos=coords, node_color=node_colors)
    # pos for coordinates and node_color for node colors (for graph visualizati
    nx.draw_networkx_labels(G, pos=coords)
    # draw node labels (for graph visualization)

    if partial_path:
    # if partial path is not empty
        path_edges = list(zip(partial_path[:-1], partial_path[1:]))
        # create list of edges in partial path
        nx.draw_networkx_edges(G, pos=coords, edgelist=path_edges, edge_color='
        # draw edges in partial path (for graph visualization)

    if active_edge:
    # if active edge is not empty
        nx.draw_networkx_edges(G, pos=coords, edgelist=[active_edge], edge_colo
        # draw active edge (for graph visualization)

    else:
    # if active edge is empty and partial path is empty (i.e., initial graph)
        nx.draw_networkx_edges(G, pos=coords)
        # draw all edges and set default edge color and position

    plt.axis('off')
    # turn off axis else it will show axis with vertex numbers
    plt.tight_layout()
    # tight layout as it will show axis with vertex numbers

    buf = BytesIO()
    # initialize buffer, which is a BytesIO object (for graph visualization)
    plt.savefig(buf, format='png')
    # save figure to buffer as png
    buf.seek(0)
    # seek to the beginning of the buffer
    image_array = imageio.imread(buf)
    # read buffer as image array, helping to create animation (for graph visual

    plt.close()
    # close figure, guiding matplotlib to free memory

    return image_array
```

```
    # return image array (for graph visualization)


def dijkstra_visualized(n, start, goal, edges, coords):
# Dijkstra's algorithm to solve for minimum cost paths on a given graph (with v
    visited = set()
    # initialize visited set
    distances = {vertex: float('infinity') for vertex in range(1, n + 1)}
    # initialize distances dictionary (key: vertex, value: distance)
    previous_vertices = {vertex: None for vertex in range(1, n + 1)}
    # initialize previous vertices dictionary (key: vertex, value: previous ver
    distances[start] = 0
    # set distance of start vertex to 0
    vertices = [(0, start)]
    # initialize vertices list (tuple: (distance, vertex))

    images = []
    # initialize images list (for graph visualization)
    partial_paths = {}
    # initialize partial paths dictionary (key: vertex, value: partial path)

    while vertices:
        # while vertices list is not empty (i.e., there are still vertices to v
        current_distance, current_vertex = heapq.heappop(vertices)
        # pop vertex with minimum distance from vertices list
        if current_vertex in visited:
        # if vertex has already been visited,
            continue
            # continue to next iteration of while loop
        visited.add(current_vertex)
        # add vertex to visited set (i.e., mark vertex as visited)
        for neighbor, weight in edges.get(current_vertex, []):
        # for each neighbor of current vertex
            active_edge = (current_vertex, neighbor)
            # set active edge (for graph visualization)

            partial_path = [current_vertex]
            # initialize partial path (for graph visualization)
            prev_vertex = previous_vertices[current_vertex]
            # initialize previous vertex (for graph visualization)
            while prev_vertex is not None:
            # while previous vertex is not None (i.e., while current vertex has
                partial_path.append(prev_vertex)
                # append previous vertex to partial path (for graph visualizati
                prev_vertex = previous_vertices[prev_vertex]
                # update previous vertex to previous vertex of previous vertex
            partial_paths[current_vertex] = partial_path[::-1]
            # add partial path to partial paths dictionary (for graph visualiza

            image_array = visualize_graph(edges, coords, current_vertex, partia
            # visualize graph (for graph visualization)
            images.append(image_array)
            # append image array to images list (for graph visualization)

            distance = current_distance + weight
            # calculate distance to neighbor
            if distance < distances[neighbor]:
            # if distance to neighbor is less than current distance to neighbor
                distances[neighbor] = distance
                # update distance to neighbor
```

```
                previous_vertices[neighbor] = current_vertex
                # update previous vertex of neighbor
                heapq.heappush(vertices, (distance, neighbor))
                # push neighbor to vertices list

    path, current_vertex = [], goal
    # initialize path list and current vertex
    while previous_vertices[current_vertex] is not None:
    # while current vertex has a previous vertex
        path.append(current_vertex)
        # append current vertex to path list
        current_vertex = previous_vertices[current_vertex]
        # update current vertex to previous vertex
    if current_vertex != start:
    # if current vertex is not start vertex
        return [], []
        # return empty path and empty distances

    path.append(start)
    # append start vertex to path list (i.e., path list is now complete)

    image_array = visualize_graph(edges, coords, active_vertex=None, partial_pa
    # visualize graph (for graph visualization)
    images.append(image_array)
    # append image array to images list (for graph visualization)

    imageio.mimsave('hw1.mp4', images, fps=1)
    # save images list as mp4 file
    # mimsave() function from imageio module to save images list as mp4 file
    # fps=1 to set frame rate to 1 frame per second

    return path[::-1], [distances[vertex] for vertex in path[::-1]]
    # return path list in reverse order and distances list in reverse order

coords = parse_coordinates('coords.txt')
# parse coordinates file (for graph visualization)
n, start, goal, edges = parse_input('input.txt')
path, distances = dijkstra_visualized(n, start, goal, edges, coords)
write_output(path, distances)

print("Visualization saved as 'hw1.mp4'.")
# print message to indicate that visualization has been saved as 'hw1.mp4'
```

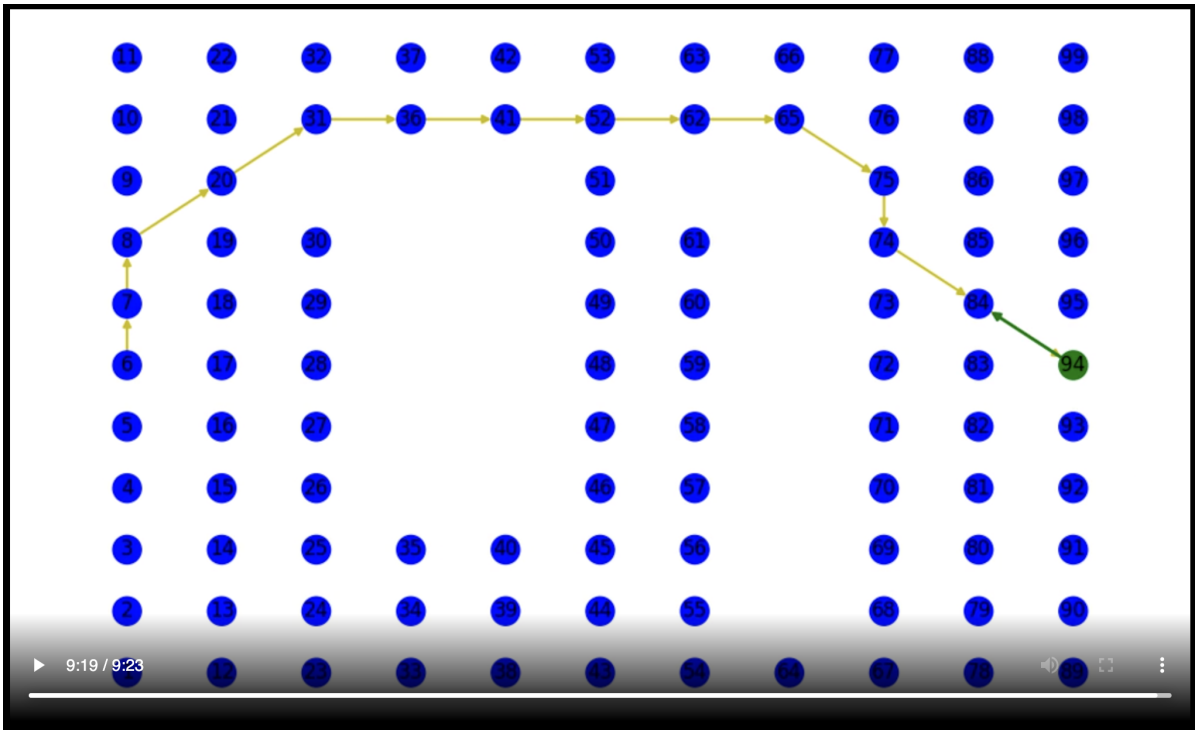
```
/var/folders/_n/wqs6gy2s2z95mhp37pm6p79w0000gn/T/ipykernel_14986/1864040001.p
y:86: DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will
be removed in NumPy 2.0. Please use `all` instead.
  nx.draw_networkx_edges(G, pos=coords, edgelist=[active_edge], edge_color
='g', width=2)
/var/folders/_n/wqs6gy2s2z95mhp37pm6p79w0000gn/T/ipykernel_14986/1864040001.p
y:83: DeprecationWarning: `alltrue` is deprecated as of NumPy 1.25.0, and will
be removed in NumPy 2.0. Please use `all` instead.
  nx.draw_networkx_edges(G, pos=coords, edgelist=path_edges, edge_color='y', w
idth=1.5)
IMAGEIO FFMPEG_WRITER WARNING: input image is not divisible by macro_block_siz
e=16, resizing from (1000, 600) to (1008, 608) to ensure video compatibility w
ith most codecs and players. To prevent resizing, make your input image divisi
ble by the macro_block_size or set the macro_block_size to 1 (risking incompat
ibility).
Visualization saved as 'hw1.mp4'.
```

This code give a much better visual representation of the steps of Dijkstra's algorithm.

The active vertex being explored are shown in green,

the edges being considered are shown in green, and

the current known shortest paths are shown in yellow.



In [ ]: