Module 4: Portfolio Milestone

Brady Chin

Colorado State University Global

CSC506-1: Design and Analysis of Algorithms

Dr. Dong Nguyen

December 8th, 2024

Portfolio Milestone

In this milestone, I implemented Dijkstra and Bellman-Ford's algorithms to calculate the shortest path between two points in a weighted diagram. I then created several test cases to ensure that the algorithms are corrects and giving expected outputs.

When executing these algorithms, I created it so that you will need to run the test scripts to see the outputs of each algorithm. These are named "[algorithm_name]_test_inputs.py".

Dijkstra Data

I created 4 weighted diagrams to test various inputs and different cases Dijkstra's algorithm. Table 1 shows the each test case. These cases were given various starting nodes and targets and were tested on different weighted diagrams.

Table 1: *Test cases for the Dijkstra algorithm.*

Case	Test	Start Vertex	Target	Weighted Diagram
1	Standard test. Start from point A and find shortest paths to all vertices.	A	All	'A': [('B', 5), ('C', 3), ('E', 11)], 'B': [('A', 5), ('C', 1), ('F', 2)], 'C': [('A', 3), ('B', 1), ('D', 1), ('E', 5)], 'D': [('C', 1), ('E', 9), ('F', 3)], 'E': [('A', 11), ('C', 5), ('D', 9)], 'F': [('B', 2), ('D', 3)]
2	Start from a vertex other than A. Hard code a specific target vertex.	B	E	A': [('B', 4), ('C', 2)], 'B': [('C', 3), ('D', 2), ('E', 3)], 'C': [('B', 1), ('D', 4), ('E', 5)], 'D': [], 'E': [('D', 1)]
3	Start from a vertex other than A. Find shortest path to all other nodes.	B	All	A': [('B', 2), ('C', 5)], 'B': [('C', 1), ('D', 4)], 'C': [('D', 2)], 'D': []
4	Request a shortest path that does not exist.	D	C	A': [('B', 3), ('C', 6)], 'B': [('C', 2), ('D', 1)], 'C': [('D', 4)], 'D': [('E', 5)], 'E': []

Table 2 shows the outputs for each test case. All cases resulted in the correct output and displayed the shortest path between the two points and the path itself.

Table 2: *Test case results for Dijkstra's algorithm.*

Case	Expected output	Actual output
1	A -> B: Distance: 4 Path: A -> C -> B A -> C: Distance: 3 Path: A -> C A -> D: Distance: 4 Path: A -> C -> D A -> E: Distance: 8 Path: A -> C -> E A -> F: Distance: 6 Path: A -> C -> B -> F	A -> B: Distance: 4 Path: A -> C -> B A -> C: Distance: 3 Path: A -> C A -> D: Distance: 4 Path: A -> C -> D A -> E: Distance: 8 Path: A -> C -> E A -> F: Distance: 6 Path: A -> C -> B -> F
2	B -> E: Distance: 3 Path: B -> E	B -> E: Distance: 3 Path: B -> E
3	B -> A: Distance: inf Path: B -> C: Distance: 1 Path: B -> C B -> D: Distance: 3 Path: B -> C -> D	B -> A: Distance: inf Path: B -> C: Distance: 1 Path: B -> C B -> D: Distance: 3 Path: B -> C -> D
4	D -> C: Distance: inf Path:	D -> C: Distance: inf Path:

Bellman-Ford Data

Using the same process to test the algorithm as Dijkstra, I created 4 test cases to test Bellman-Ford's algorithm. These tests were given various starting nodes, targets, and different weighted diagrams. In addition, since Bellman-Ford's algorithm is able to calculate weighted diagrams with negative weights, I added cases to ensure they were producing the correct outputs. Table 3 shows these test cases.

Table 3: *Test cases for the Bellman-Ford algorithm.*

Case	Test	Start Vertex	Target	Weighted Diagram
1	Standard test. Start from point A and find shortest paths to all vertices.	A	All	'A': [('B', 10), ('F', 8)], 'B': [('D',2)], 'C': [('B', 1)], 'D': [('C', -2)], 'E': [('B', -4), ('D', -1)], 'F': [('E', 1)]
2	Calculating shortest path would result in a negative weight cycle.	A	All	'A': [('B', 5)], 'B': [('C', -2)], 'C': [('D', 3)], 'D': [('B', -2)]
3	Start from a vertex other than A. Hard code a specific target vertex.	B	D	'A': [('B', 4), ('C', 2)], 'B': [('C', -1), ('D', 5)], 'C': [('D', 3), ('A', -2)], 'D': []
4	Start from a vertex other than A. Find shortest path to all other nodes.	B	All	'A': [('B', 1), ('C', 4)], 'B': [('C', -2), ('D', 2)], 'C': [('D', 3)], 'D': []

Table 4 shows the outputs for each test case. All cases resulted in the correct output and displayed the shortest path between the two points and the path itself.

Table 4: *Test case results for Bellman-Ford's algorithm.*

Case	Expected output	Actual output
1	A -> B: Distance: 5 Path: A -> F -> E -> B A -> C: Distance: 5 Path: A -> F -> E -> B -> D -> C A -> D: Distance: 7 Path: A -> F -> E -> B -> D A -> E: Distance: 9 Path: A -> F -> E A -> F: Distance: 8 Path: A -> F	A -> B: Distance: 5 Path: A -> F -> E -> B A -> C: Distance: 5 Path: A -> F -> E -> B -> D -> C A -> D: Distance: 7 Path: A -> F -> E -> B -> D A -> E: Distance: 9 Path: A -> F -> E A -> F: Distance: 8 Path: A -> F
2	Negative weight cycle detected.	Negative weight cycle detected.
3	B -> D: Distance: 2 Path: B -> C -> D	B -> D: Distance: 2 Path: B -> C -> D
4	B -> A: Distance: inf Path: B -> C: Distance: -2 Path: B -> C B -> D: Distance: 1 Path: B -> C -> D	B -> A: Distance: inf Path: B -> C: Distance: -2 Path: B -> C B -> D: Distance: 1 Path: B -> C -> D

Algorithm Development

Developing these algorithms were pretty challenging in my opinion. Lucky there were many resources that I had to help me with it. I used the courses zyBook's as assistance to develop the algorithm and make modifications that I felt were necessary. Dijkstra's algorithm was in chapter 7.13 (Pizzo, B., 2024). After making some modifications, I then used this as a starting point for Bellman-Ford's algorithm while referring to chapter 7.15 (Pizzo, B., 2024) for assistance.

Challenges

The first challenge I encountered was understanding the difference between Dijkstra and Bellman-Ford's algorithm. While they both compute the same thing, they use different techniques and have different limitations.

After completing each algorithm and testing the results, I noticed that Bellman-Ford's algorithm was giving incorrect results. This was because of the negative weight cycle. At first, I had difficulties understanding how this was resulting in an error and understanding what a negative weight cycle was (geeksforgeeks, 2023, September 4). After understanding this, I was able to create a check to exit the loop if a negative weight cycle was detected.

Future Developments

While the current algorithms I created give correct outputs, there will need to be some modifications in the for the future. I would like to improve the test scripts to make them more streamlined. I also think that these scripts are pretty introductory so I would like to develop a better method to test the algorithms.

The most important development I would like to make is using an Object-Oriented version. Using Object-oriented-programming is essential in python and allows for customization and scalability. The version I used is helpful for testing but as projects grow, this method would not be sustainable since it is less flexible.

Conclusion

I am confident that the algorithms have been properly developed and tested. There were challenges that were faced but I was able to find solutions to them. As I continue to analyze these algorithms in future milestone projects, I will be able to see where I can make changes and improvements.

Successful Execution

dijkstra.py and **dijkstra_test_inputs.py**

dijkstra.py

```python
# Run "dijkstra_test_input.py" to test this algorithm

def dijkstra(graph, start, target=''):  2 usages  Brady
    unvisited = list(graph)
    distances = {node: 0 if node == start else float('inf') for node in graph}
    paths = {node: [] for node in graph}
    paths[start].append(start)

    while unvisited:
        current = min(unvisited, key=distances.get)
        for node, distance in graph[current]:
            if distance + distances[current] < distances[node]:
                distances[node] = distance + distances[current]
                if paths[node] and paths[node][-1] == node:
                    paths[node] = paths[current][:]
                else:
                    paths[node].extend(paths[current])
                paths[node].append(node)
        unvisited.remove(current)

    targets_to_print = [target] if target else graph
    for node in targets_to_print:
        if node == start:
            continue
        print(f'{start} -> {node}:')
        print(f'  Distance: {distances[node]}')
        print(f'  Path: {" -> ".join(paths[node])}')

    return distances, paths
```

dijkstra_test_inputs.py

```python
from dijkstra import dijkstra

test_graphs = [
    [
        {
            'A': [('B', 5), ('C', 3), ('E', 11)],
            'B': [('A', 5), ('C', 1), ('F', 2)],
            'C': [('A', 3), ('B', 1), ('D', 1), ('E', 5)],
            'D': [('C', 1), ('E', 9), ('F', 3)],
            'E': [('A', 11), ('C', 5), ('D', 9)],
            'F': [('B', 2), ('D', 3)]
        },
        'A',
        ''
    ],
    [
        {
            'A': [('B', 4), ('C', 2)],
            'B': [('C', 3), ('D', 2), ('E', 3)],
            'C': [('B', 1), ('D', 4), ('E', 5)],
            'D': [],
            'E': [('D', 1)]
        },
        'B',
        'E'
    ],
    [
        {
            'A': [('B', 2), ('C', 5)],
            'B': [('C', 1), ('D', 4)],
            'C': [('D', 2)],
            'D': []
        },
        'B',
        ''
    ],
    [
        {
            'A': [('B', 3), ('C', 6)],
            'B': [('C', 2), ('D', 1)],
            'C': [('D', 4)],
            'D': [('E', 5)],
            'E': []
        },
        'D',
        'C'
    ]
]
```

Run dijkstra_test_inputs

```
/usr/local/bin/python3.12 /Users/bradychin/Librar
--------------Test case 1---------------
- Start: A
- Target: All nodes

A -> B:
  Distance: 4
  Path: A -> C -> B
A -> C:
  Distance: 3
  Path: A -> C
A -> D:
  Distance: 4
  Path: A -> C -> D
A -> E:
  Distance: 8
  Path: A -> C -> E
A -> F:
  Distance: 6
  Path: A -> C -> B -> F


--------------Test case 2---------------
- Start: B
- Target: E

B -> E:
  Distance: 3
  Path: B -> E


--------------Test case 3---------------
- Start: B
- Target: All nodes

B -> A:
  Distance: inf
  Path:
B -> C:
  Distance: 1
  Path: B -> C
B -> D:
  Distance: 3
  Path: B -> C -> D


--------------Test case 4---------------
- Start: D
```

bellman_ford.py and **bellman_ford_test_inputs.py**

bellman_ford.py

```python
# Run "bellman_ford_test_input.py" to test this algorithm

def bellman_ford(graph, start, target=''):  2 usages  Brady
    distances = {node: 0 if node == start else float('inf') for node in graph}
    paths = {node: [] for node in graph}
    paths[start].append(start)

    for _ in range(len(graph)-1):
        for current in graph:
            for neighbour, distance in graph[current]:
                if distance + distances[current] < distances[neighbour]:
                    distances[neighbour] = distance + distances[current]
                    paths[neighbour] = paths[current][:] + [neighbour]

    # negative cycle check
    for current in graph:
        for neighbor, weight in graph[current]:
            if distances[current] + weight < distances[neighbor]:
                print("Negative weight cycle detected.")
                return None, None  # Negative cycle found

    targets_to_print = [target] if target else graph
    for node in targets_to_print:
        if node == start:
            continue
        print(f'{start} -> {node}:')
        print(f'  Distance: {distances[node]}')
        print(f'  Path: {" -> ".join(paths[node])}')

    return distances, paths
```

bellman_ford_test_inputs.py

```python
from bellman_ford import bellman_ford

test_graphs = [
    [
        {
            'A': [('B', 10), ('F', 8)],
            'B': [('D',2)],
            'C': [('B', 1)],
            'D': [('C', -2)],
            'E': [('B', -4), ('D', -1)],
            'F': [('E', 1)]
        },
        'A',
        ''
    ],
    [
        {
            'A': [('B', 5)],
            'B': [('C', -2)],
            'C': [('D', 3)],
            'D': [('B', -2)]
        },
        'A',
        ''
    ],
    [
        {
            'A': [('B', 4), ('C', 2)],
            'B': [('C', -1), ('D', 5)],
            'C': [('D', 3), ('A', -2)],
            'D': []
        },
        'B',
        'D'
    ],
    [
        {
            'A': [('B', 1), ('C', 4)],
            'B': [('C', -2), ('D', 2)],
            'C': [('D', 3)],
            'D': []
        },
        'B',
        ''
    ]
]

test_case = 1
for graph, start, target in test_graphs:
```

Run bellman_ford_test_inputs

```
/usr/local/bin/python3.12 /Users/bradychin/Librar
--------------Test case 1---------------
- Start: A
- Target: All nodes

A -> B:
  Distance: 5
  Path: A -> F -> E -> B
A -> C:
  Distance: 5
  Path: A -> F -> E -> B -> D -> C
A -> D:
  Distance: 7
  Path: A -> F -> E -> B -> D
A -> E:
  Distance: 9
  Path: A -> F -> E
A -> F:
  Distance: 8
  Path: A -> F


--------------Test case 2---------------
- Start: A
- Target: All nodes

Negative weight cycle detected.


--------------Test case 3---------------
- Start: B
- Target: D

B -> D:
  Distance: 2
  Path: B -> C -> D


--------------Test case 4---------------
- Start: B
- Target: All nodes

B -> A:
  Distance: inf
  Path:
B -> C:
  Distance: -2
  Path: B -> C
```

References

geeksforgeeks.org (2023, September 4) *Detect a Negative Weight Cycle in a Graph | (Bellman Ford).* https://www.geeksforgeeks.org/detect-negative-cycle-graph-bellman-ford/

Pizzo, B., (2024) *CSC506: Design and Analysis of Algorithms. 7.13 Python: Dijkstra's shortest path.* zyBooks. https://learn.zybooks.com/zybook/CSC506-1_5/chapter/7/section/13

Pizzo, B., (2024) *CSC506: Design and Analysis of Algorithms. 7.15 Python: Bellman-Ford's shortest path.* zyBooks. https://learn.zybooks.com/zybook/CSC506-1_5/chapter/7/section/15