Hw2 - Cost-to-Go Algorithm

Student Name : Chen Kai Zhang

Student ID : 014806701

Overview

The algorithm implemented is based off the notation in the slide and Bellman-Ford algorithm (which I believe is the same).

$$V(x_i) = \min_{x_j in N(x_i)}{C(x_i,x_j) + V(x_j)}$$ $$V(x_G) = 0$$

- $C(x_i, x_j)$: cost of edge from x_i to x_j
  - if no edge then $C(x_i, x_j) = inf$
- $V(x_i)$: cost to go from x_i to x_G
  - **value function** : the minimum cost to reach the goal node from any node
- $N(x_i)$: vertices that are outgoing neighbors of x_i

The algorithm begins by calculating all V for all nodes by using

$$V(node) = \min_{neighbor}(cost(node \rightarrow neighbor) + V(neighbor))$$

... where if the node has no outgoing edges, then the $V(node)$ will be $\inf$.

By doing this, we have access to all the shortest paths from $node$ to the goal.

Afterwards, we can build the shortest path by going through the steps from start to goal by picking the next node with the least cost

$$C(current, next) + V(next)$$

Setup

1. Make and start a new python virtual environment
2. Run `pip install -r requirements.txt`
3. Run `python3 014806701.py -i <input-file> -s <start-node> -g <end-node>`

Code

The code is divided into two sections

1. Calculating V for each node
2. Computing the shortest path

1. Calculating V for each node

```python
Vs = {node: float("inf") for node in graph}
Vs[goal] = 0

nodes = list(graph.keys())

for _ in range(len(nodes) - 1):
    for node in nodes:
        if node == goal:
            continue
        if graph[node]:
            neighbors = [graph[node][neighbor] + Vs[neighbor] for neighbor
in graph[node]]
            Vs[node] = min(neighbors)
        else:
            Vs[node] = float("inf")
```

We begin by initializing `Vs` which will hold the V value for all our nodes. Since $V(G)$ will always be 0, we can define that immediately.

To populate `Vs`, we will

1. Go through each node in our graph
2. Check if the node has outgoing edges (`neighbors`)
3. Calculate the V for node to each neighbor
4. Set V of node to the smallest one

With `Vs` populated, we can move on to finding the shortest path.

2. Computing the shortest path

```python
path = []
path_cost = []
current = start

while current != goal:
    path.append(str(current))
    # The next best node will be the one with the smallest V()
    next = min(graph[current], key=lambda x: graph[current][x] + Vs[x])
    path_cost.append(str(graph[current][next] + Vs[next]))
    current = next

path.append(str(goal))
path_cost.append("0")
```

Similar to **Hw1 - Dijkstra's Algorithm**, we will hold the shortest path and the cost along the way in `path` and `path_cost` respectively. To find the shortest path, we start current and for each step we find the shortest path via $C(current,next)+V(next)$.

Extra : Identifying negative-weight cycles

```python
for node in nodes:
    if not graph[node]:
        continue
    for neighbor, weight in graph[node].items():
        if Vs[node] != float("inf") and Vs[node] + weight < Vs[neighbor]:
            raise ValueError("Graph contains negative-weight cycle")
```

Since Bellman Ford accepts negative weights, we could encounter negative weight cycles. To resolve this issue, we check for those and report `ValueError()` upon detection so the code wont fall into an infinite loop