Stan Antoniu 917

Graphs Algorithms

Dijkstra's Algorithm

Given a graph with non-negative costs, it computes the minimum-cost path between two vertices

Idea

Dijkstra's algorithm still relies on Bellman's optimality principle; however, it computes distances from the starting vertex in increasing order of the distances. This way, the distance from start to a given vertex does not have to be recomputed after a vertex is processed.

This way, Dijkstra's algorithm looks a bit like a breadth-first search traversal; however, the queue is replaced by a priority queue where the top vertex is the closest to the starting vertex.

If all costs are non-negative, the algorithm does not put a vertex in the priority queue once it was extracted and processed.

Algorithm (Python)

```
def dijkstraBackwards(self, start, end):
        '''
      Computes the minimum cost path from start to end, using Dijkstra's Algorithm,
      and the cost of this path.

        Preconditions:
        -start and end are integers
        -start and end are vertices from the graph
        -all costs are positive
        Postconditions:
        -returns RETURN_LIST, which is a list containing the cost of the minimum-
path, respectively another list representing the path itself.
        -returns [-1, []] if there is no path between start and end
        Errors:
        -Throws ValueError if start or end are not found in the graph.
        '''

        priority_queue = []
        dist = {}
        prev = {}
        found = False

        if end not in self.__vertices or start not in self.__vertices:
```

```
            raise ValueError("The START or/and END vertex is not found in the
graph!")

        heapq.heappush(priority_queue, (0, end))
        dist[end] = 0
        prev[end] = -1
        while (len(priority_queue) > 0):
            current_processed = heapq.heappop(priority_queue)[1] #priority_queue is
formed of 2-element tuples
            for neighbour in self.getInboundNeighbours(current_processed):
                if neighbour not in dist.keys():
                    prev[neighbour] = current_processed
                    dist[neighbour] = self.getEdgeCost(neighbour, current_processed)
+ dist[current_processed]
                    if neighbour == start:
                        found = True
                        break
                    heapq.heappush(priority_queue, (dist[neighbour], neighbour))

        if (found):
            path = []
            current_processed = start
            while current_processed != -1: #-1 is prev[end]
                path.append(current_processed)
                current_processed = prev[current_processed]
            return [dist[start], path]
        return [-1, []] #it means there is no path between start and end
```

As we can see, the algorithm uses the library heappq for implementing the priority queue.

Furthermore, the algorithm represents a “backwards Dijkstra”; that is a Dijkstra’s algorithm that searches backwards, from vertex END to vertex START.