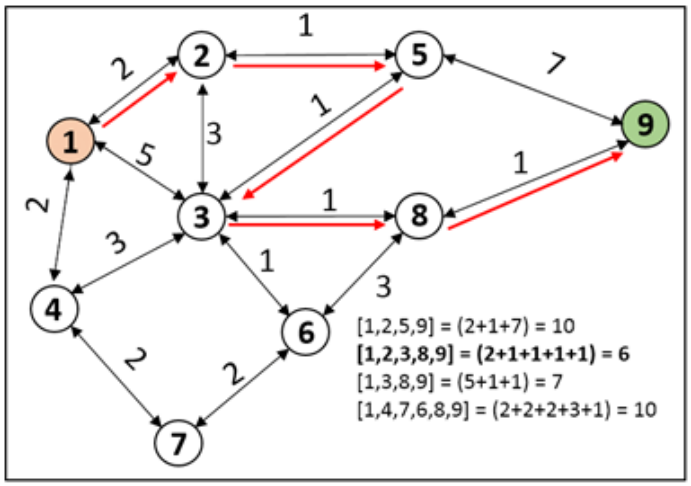
[A11] DijkstraPQShortestPath

Grafische Beschreibung



Textuelle Beschreibung

Der Dijkstra Algorithmus sucht innerhalb eines gewichteten Graphen den kostengünstigsten Weg zwischen zwei Knoten. Von einem Startknoten ausgehend werden die, im nächsten Schritt erreichbaren, Knoten mit dem geringsten Gewicht, ausgehend von den bereits erreichten Knoten, abgearbeitet. In jedem Durchlauf wird aus der Menge der erreichbaren Knoten, der Knoten mit den geringsten Kosten aus der Menge der bereits erreichten Knoten hinzugefügt. Dadurch entsteht ein neuer Graph mit den minimalsten Entfernungen zwischen allen Knoten.

In unserem Fall wurde der Algorithmus mittels Heap implementiert. Somit wird in mehreren Durchläufen jeder einzelne Knoten bearbeitet. Wichtig ist, dass bei Bedarf die Kosten des Weges zu besagtem Knoten abgepasst werden müssen. Die Entfernungen vom Startknoten zum jeweiligen Knoten werden im distance[] Array und die Vorgänger des jeweiligen Knotens werden im predecessor[]Array gespeichert.

Laufzeit

O((V+E)log(V))

Begründung: Der Algorithmus wurde mit einem Heap implementiert.

Wichtige Elemente

```
protected boolean calculatePath(int from, int to) {
   //PriorityQueue<Integer> heap = new PriorityQueue<>();
   VertexHeap heap = new VertexHeap();
   for (int i = 0; i < graph.numVertices(); i++) { // loop to add all vertices to
heap
```

```
   heap.insert(new Vertex(i, distance[i]));   // add new Vertex / Node to heap and
 set initial distance to 9999 (from distance array)
   predecessor[i] = -1;    // set predecessor to -1
   }
   distance[from] = 0;     // set distance from starting node to 0
   heap.setCost(from, 0);  // set cost/priority for starting node to 0 in heap
//heap.printHeap();
   while(!heap.isEmpty()){
        Vertex v = heap.remove();  // get top element (smallest priority/cost) and
 remove it from heap
        for (WeightedEdge edge : graph.getEdges(v.getVertex())) {      // iterate
through list of edges from current element (removed from top of heap)
            if((edge.weight + v.getCost()) < distance[edge.to_vertex]) {  // check
 if way is better (cost) than existing one
                predecessor[edge.to_vertex] = v.getVertex();   // set predecessor
 node
                distance[edge.to_vertex] = edge.weight + v.getCost(); // distance
 to target node calculated from edge weight and distance to current node
                heap.setCost(edge.to_vertex, distance[edge.to_vertex]);   // set
new priority / cost in heap
               }
           }
        }
     return true;
    }
```