COP 4530 - DATA STRUCTURES

DIJKSTRA'S ALGORITHM

BY KRISH VEERA AND JONATHAN KOCH

MECHANISM

01 DIJKSTRA VALUE INITIALIZED - TRACKS THE WEIGHTAGE THAT THE ALGORITHM ENCOUNTERED WHILE BEING EXECUTED.

02 DIJKSTRA VALUE FOR START NODE = 0 AND ALL THE OTHERS ARE MARKED ∞ SINCE NO NODES HAVE BEEN ENCOUNTERED YET.

03 AFTER PEEKING ALL THE EDGES CONNECTED TO THE NODE AND THEIR WEIGHT THE ALGORITHM THEN CHOOSES THE ONE WITH THE SMALLEST VALUE AND ADDS IT TO THE SHORTEST PATH.

04 THE NODE IS MARKED VISITED AND THE PROCESS IS REPEATED UNTIL THE DESTINATION HAS BEEN REACHED.

LIMITATION

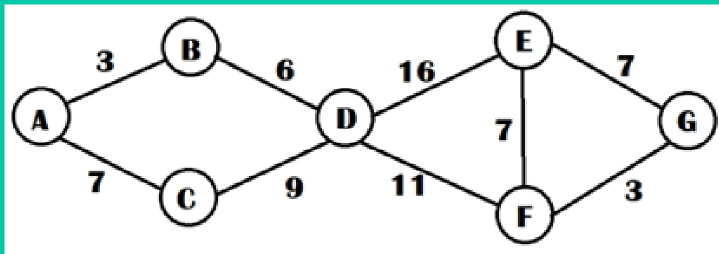
**THE DIJKSTRA'S ALGORITHM CAN ONLY WORK WITH POSITIVE WEIGHTS OF EDGES. THEREFORE, IT CAN ONLY BE IMPLEMENTED
ON A NON-NEGATIVE WEIGHTED GRAPH.**

TIME COMPLEXITY

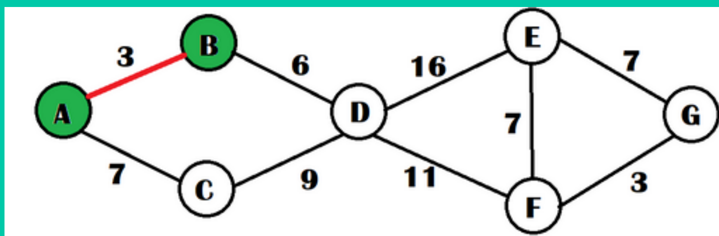
O(N+E*LOG(N))

N: Number of Nodes in the Weighted Graph
E: Number of Edges in the Weighted Graph

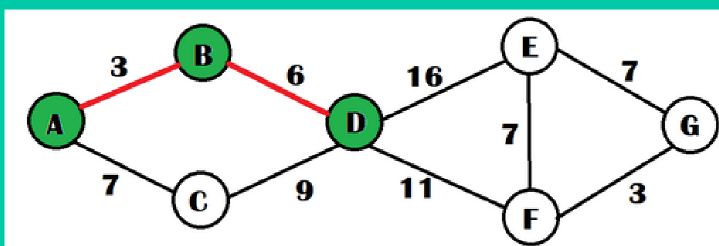
IMPLEMENTATION EXAMPLE



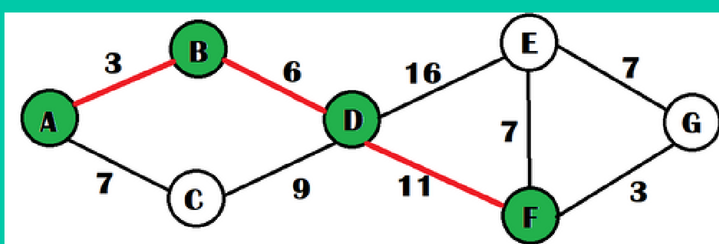
A	B	C	D	E	F	G
0	∞	∞	∞	∞	∞	∞



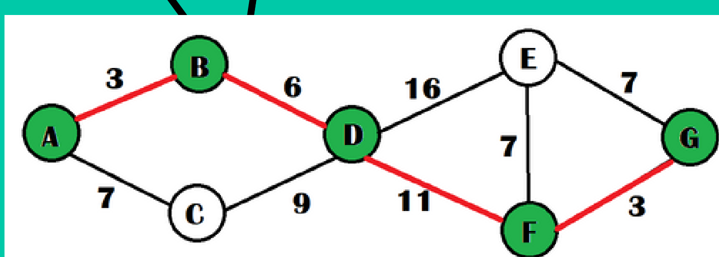
A	B	C	D	E	F	G
0	3	7	∞	∞	∞	∞



A	B	C	D	E	F	G
0	3	7	$3+6=9$	∞	∞	∞



A	B	C	D	E	F	G
0	3	7	9	$9+16=25$	$9+11=20$	∞



A	B	C	D	E	F	G
0	3	7	9	25	20	$20+3=23$

INITIALIZATION

```
unsigned long Graph::shortestPath(std::string startLabel, std::string endLabel,
    std::vector<std::string> &path) {
    GraphNode* startingNode = this->getVertex(startLabel);
    GraphNode* endingNode = this->getVertex(endLabel);
    /* node does not exist in list */
    if (startingNode == nullptr || endingNode == nullptr) return INFINITY;

    HeapQueue<Path*, Path::PathComparer> paths;
    std::map<std::string, unsigned long> dijkstraKey;
    for (auto it = this->nodes.begin(); it != this->nodes.end(); ++it) { /* O(N)*/
        if (it->first != startLabel) {
            dijkstraKey[it->first] = INFINITY;
        }
    }
    dijkstraKey[startLabel] = 0;
    paths.insert(new Path(startingNode));
```

ALGORITHM IMPLEMENTATION

```
while (paths.min()->current()->id != endLabel) {
    Path* shortestPath = paths.min();
    paths.removeMin(); /* */
    for (auto const& [currID, currLink]: shortestPath->pathsFrom()) { /* Each Edge in the Current Visiting Node O(Edges) */
        if (! shortestPath->nodeVisited(currID) ) {/* Dont Revisit old nodes*/
            Path* newLocation = new Path(*shortestPath); /* Copy the old Path */
            newLocation->visit(currLink->to); /* ... And visit the new Node for that Path  */
            unsigned long previousDistance = dijkstraKey[currID];
            /* Update the key along the way, and add the node if weve found a shorter path to another node. */
            if (newLocation->pathWeight() < previousDistance ) {
            dijkstraKey[currID] = newLocation->pathWeight();
            paths.insert(newLocation); /* O(log(E))*/
            }
        }
    }
}
Path* finalPath = paths.min();
paths.removeMin();
path = finalPath->toString();
return dijkstraKey[endLabel];
```