Module 11 : SHORTEST PATHS-DIJKSTRA'S ALGORITHM

Course Learning Outcomes:

1. Familiarize Dijkstra's Algorithm.
2. Differentiate Dijkstra's algorithm Version 1 and 2.
3. Know how to used short paths in an Algorithms.

Dijkstra's Algorithm

Is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It was conceived by computer scientist **Edsger W. Dijkstra** in 1956 and published three years later. The algorithm exists in many variants.

A common graph based problem is that we have some situation represented as a weighted digraph with edges labelled by non-negative numbers and need to answer the following question: For two particular vertices, what is the shortest route from one to the other?

Here, by "shortest route" we mean a path which, when we add up the weights along its edges, gives the smallest overall weight for the path. This number is called the length of the path. Thus, a shortest path is one with minimal length. Note that there need not be a unique shortest path, since several paths might have the same length. In a disconnected graph there will not be a path between vertices in different components, but we can take care of this by using ∞ once again to stand for "no path at all".

Note that the weights do not necessarily have to correspond to distances; they could, for example, be time (in which case we could speak of "quickest paths") or money (in which case we could speak of "cheapest paths"), among other possibilities. By considering "abstract" graphs in which the numerical weights are left uninterpreted, we can take care of all such situations and others. But notice that we do need to restrict the edge weights to be nonnegative numbers, because if there are negative numbers and cycles, we can have increasingly long paths with lower and lower costs, and no path with minimal cost.

Applications of shortest-path algorithms include internet packet routing (because, if you send an email message from your computer to someone else, it has to go through various email routers, until it reaches its final destination), train-ticket reservation systems (that need to figure out the best connecting stations), and driving route finders (that need to find an optimum route in some sense).

Course Module

Dijkstra's algorithm. In turns out that, if we want to compute the shortest path from a given start node s to a given end node z, it is actually most convenient to compute the shortest paths from s to all other nodes, not just the given node z that we are interested in. Given the start node, Dijkstra's algorithm computes shortest paths starting from s and ending at each possible node. It maintains all the information it needs in simple arrays, which are iteratively updated until the solution is reached. Because the algorithm, although elegant and short, is fairly complicated, we shall consider it one component at a time.

Overestimation of shortest paths. We keep an array D of distances indexed by the vertices. The idea is that D[z] will hold the distance of the shortest path from ***s*** to ***z*** when the algorithm finishes. However, before the algorithm finishes, D[z] is the best overestimate we currently have of the distance from s to z. We initially have D[s] = 0, and set D[z] = ∞ for all vertices z other than the start node s. Then the algorithm repeatedly decreases the overestimates until it is no longer possible to decrease them further. When this happens, the algorithm terminates, with each estimate fully constrained and said to be tight.

Improving estimates. The general idea is to look systematically for shortcuts. Suppose that, for two given vertices u and z, it happens that D[u] + weight[u][z] < D[z]. Then there is a way of going from s to u and then to z whose total length is smaller than the current overestimate D[z] of the distance from s to z, and hence we can replace D[z] by this better estimate. This corresponds to the code fragment.

```
if ( D[u] + weight[u][z] < D[z] )
   D[z] = D[u] + weight[u][z]
```

of the full algorithm given below. The problem is thus reduced to developing an algorithm that will systematically apply this improvement so that (1) we eventually get the tight estimates promised above, and (2) that is done as efficiently as possible.

Dijkstra's algorithm, Version 1. The first version of such an algorithm is not as efficient as it could be, but it is relatively simple and certainly correct. (It is always a good idea to start with an inefficient simple algorithm, so that the results from it can be used to check the operation of a more complex efficient algorithm.) The general idea is that, at each stage of the algorithm's operation, if an entry $D[u]$ of the array D has the minimal value among all the values recorded in D, then the overestimate $D[u]$ must actually be tight, because the improvement algorithm discussed above cannot possibly find a shortcut.

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.
```

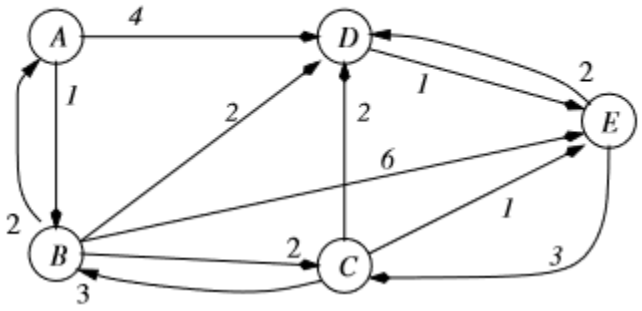
```
D[s] = 0   // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
   if ( z is not the start vertex s )
      D[z] = infinity    // This is certainly an overestimate.
}

// We use an auxiliary array 'tight' indexed by the vertices,
// that records for which nodes the shortest path estimates
// are ''known'' to be tight by the algorithm.

for ( each vertex z of the graph ) {
   tight[z] = false
}



// We now repeatedly update the arrays 'D' and 'tight' until
// all entries in the array 'tight' hold the value true.

repeat as many times as there are vertices in the graph {
   find a vertex u with tight[u] false and minimal estimate D[u]
   tight[u] = true
   for ( each vertex z adjacent to u )
      if ( D[u] + weight[u][z] < D[z] )
         D[z] = D[u] + weight[u][z]    // Lower overestimate exists.
}

// At this point, all entries of array 'D' hold tight estimates.
```

It is clear that when this algorithm finishes, the entries of D cannot hold under-estimates of the lengths of the shortest paths. What is perhaps not so clear is why the estimates it holds are actually tight, i.e. are the minimal path lengths. In order to understand why, first notice that an initial sub-path of a shortest path is itself a shortest path. To see this, suppose that you wish to navigate from a vertex s to a vertex z, and that the shortest path from s to z happens to go through a certain vertex u. Then your path from s to z can be split into two paths, one going from s to u (an initial sub-path) and the other going from u to z (a final sub-path). Given that the whole, unsplit path is a shortest path from s to z, the initial sub-path has to be a shortest path from s to u, for if not, then you could shorten your path from s to z by replacing the initial sub-path to u by a shorter path, which would not only give a shorter path from s to u but also from s to the final destination z. Now it follows that for any start vertex, there is a tree of shortest paths from that vertex to all other vertices. The reason is that shortest paths cannot have cycles. Implicitly, Dijkstra's algorithm constructs this tree starting from the root, that is, the start vertex.

If, as tends to be the case in practice, we also wish to compute the route of shortest path, rather than just its length, we also need to introduce a third array pred to keep track of the 'predecessor' or 'previous vertex' of each vertex, so that the path can be followed back from the end point to the start point. The algorithm can clearly also be adapted to work with non-weighted graphs by assigning a suitable weight matrix of 1s for connected vertices and 0s for non-connected vertices.

The time complexity of this algorithm is clearly O(n2) where n is the number of vertices, since there are operations of O(n) nested within the repeat of O(n).

A simple example. Suppose we want to compute the shortest path from A (node 0) to E (node 4) in the weighted graph we looked at before:



A direct implementation of the above algorithm, with some code added to print out the status of the three arrays at each intermediate stage, gives the following output, in which"oo" is used to represent the infinity symbol "∞":

```
Computing shortest paths from A

        |A      B       C       D       E
--------+-------------------------------------
D       |0      oo      oo      oo      oo
tight   |no     no      no      no      no
pred.   |none   none    none    none    none

Vertex A has minimal estimate, and so is tight.

Neighbour B has estimate decreased from oo to 1 taking a shortcut via A.
Neighbour D has estimate decreased from oo to 4 taking a shortcut via A.

        |A      B       C       D       E
--------+-------------------------------------
D       |0      1       oo      4       oo
tight   |yes    no      no      no      no
pred.   |none   A       none    A       none

Vertex B has minimal estimate, and so is tight.
```

```
Neighbour A is already tight.
Neighbour C has estimate decreased from oo to 3 taking a shortcut via B.
Neighbour D has estimate decreased from  4 to 3 taking a shortcut via B.
Neighbour E has estimate decreased from oo to 7 taking a shortcut via B.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       7
tight   |yes    yes     no      no      no
pred.   |none   A       B       B       B

Vertex C has minimal estimate, and so is tight.

Neighbour B is already tight.
Neighbour D has estimate unchanged.
Neighbour E has estimate decreased from 7 to 4 taking a shortcut via C.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     no      no
pred.   |none   A       B       B       C

Vertex D has minimal estimate, and so is tight.

Neighbour E has estimate unchanged.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     no
pred.   |none   A       B       B       C

Vertex E has minimal estimate, and so is tight.

Neighbour C is already tight.
Neighbour D is already tight.

        |A      B       C       D       E
--------+-----------------------------------
D       |0      1       3       3       4
tight   |yes    yes     yes     yes     yes
pred.   |none   A       B       B       C

End of Dijkstra's computation.

A shortest path from A to E is:  A B C E.
```

Once it is clear what is happening at each stage, it is usually more convenient to adopt a shorthand notation that allows the whole process to be represented in a single table. For example, using a "*" to represent tight, the distance, status and predecessor for each node at each stage of the above example can be listed more concisely as follows:

```
Stage  |   A        B         C         D         E
-------+-----------------------------------------------
1      |  0        oo        oo        oo        oo
2      |  0 *      1   A     oo        4   A     oo
3      |  0 *      1 * A     3   B     3   B     7   B
4      |  0 *      1 * A     3 * B     3   B     4   C
5      |  0 *      1 * A     3 * B     3 * B     4   C
6      |  0 *      1 * A     3 * B     3 * B     4 * C

A shortest path from A to E is:  A B C E.
```

Dijkstra's algorithm, Version 2. The time complexity of Dijkstra's algorithm can be improved by making use of a *priority queue* (e.g., some form of heap) to keep track of which node's distance estimate becomes tight next. Here it is convenient to use the convention that lower numbers have higher priority. The previous algorithm then becomes:

```
// Input:  A directed graph with weight matrix 'weight' and
//         a start vertex 's'.
// Output: An array 'D' of distances as explained above.

// We begin by buiding the distance overestimates.

D[s] = 0   // The shortest path from s to itself has length zero.

for ( each vertex z of the graph ) {
   if ( z is not the start vertex s )
      D[z] = infinity    // This is certainly an overestimate.
}

// Then we set up a priority queue based on the overestimates.

Create a priority queue containing all the vertices of the graph,
with the entries of D as the priorities

// Then we implicitly build the path tree discussed above.

while ( priority queue is not empty ) {
   // The next vertex of the path tree is called u.
   u = remove vertex with smallest priority from queue
   for ( each vertex z in the queue which is adjacent to u ) {
      if ( D[u] + weight[u][z] < D[z] ) {
         D[z] = D[u] + weight[u][z]    // Lower overestimate exists
         Change the priority of vertex z in queue to D[z]
      }
   }
}

// At this point, all entries of array 'D' hold tight estimates.
```

If the priority queue is implemented as a *Binary* or *Binomial heap*, initializing D and creating the priority queue both have complexity $O(n)$, where n is the number of vertices of the graph, and that is negligible compared to the rest of the algorithm. Then removing vertices and changing the priorities of elements in the priority queue require some rearrangement of the heap tree by "bubbling up", and that takes $O(\log_2 n)$ steps, because that is the maximum height of the tree. Removals happen $O(n)$ times, and priority changes happen $O(e)$ times, where e is the number of edges in the graph, so the cost of maintaining the queue and updating D is $O((e+n)\log_2 n)$. Thus, the total time complexity of this form of Dijkstra's algorithm is $O((e+n)\log_2 n)$. Using a *Fibonacci heap* for the priority queue allows priority updates of $O(1)$ complexity, improving the overall complexity to $O(e+n\log_2 n)$.

In a *fully connected* graph, the number of edges e will be $O(n^2)$, and hence the time complexity of this algorithm is $O(n^2\log_2 n)$ or $O(n^2)$ depending on which kind of priority queue is used. So, in that case, the time complexity is actually greater than or equal to the previous simpler $O(n^2)$ algorithm. However, in practice, many graphs tend to be much more *sparse* with $e = O(n)$. That is, there are usually not many more edges than vertices, and in this case the time complexity for both priority queue versions is $O(n\log_2 n)$, which is a clear improvement over the previous $O(n^2)$ algorithm.

References and Supplementary Materials

Books and Journals

1. John Bullinaria ; March 2019 ; Data Structures and Algorithms ; Birmingham, UK;
2. Michael T Goodrich ; Roberto Tamassia ; David Mount ; Second Edition ; Data Structures and Algorithm in C++
3. www.google.com
4. Merriam Webster Dictionary