DSA Dijkstra's Algorithm

‹ Previous Next ›

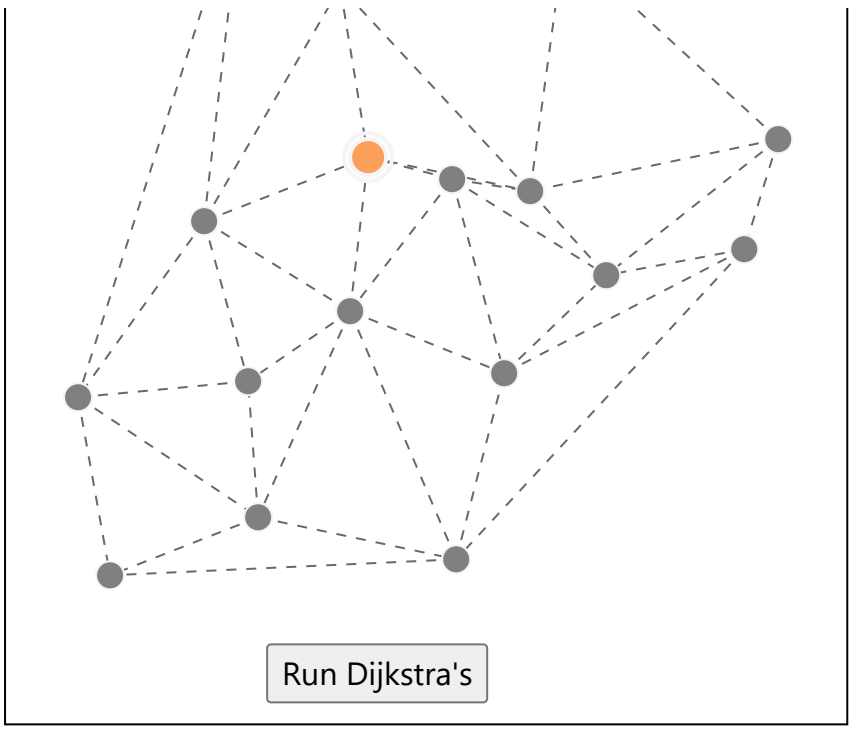
Dijkstra's shortest path algorithm was invented in 1956 by the Dutch computer scientist Edsger W. Dijkstra during a twenty minutes coffee break, while out shopping with his fiancée in Amsterdam.

The reason for inventing the algorithm was to test a new computer called ARMAC.

Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.

It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.



Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.

Dijkstra's algorithm is used for solving single-source shortest path problems for directed or undirected paths. Single-source means that one vertex is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.

Dijkstra's algorithm does not work for graphs with negative edges. For graphs with negative edges, the Bellman-Ford algorithm that is described on the next page, can be used instead.

To find the shortest path, Dijkstra's algorithm needs to know which vertex is the source, it needs a way to mark vertices as visited, and it needs an overview of the current shortest distance to each vertex as it works its way through the graph, updating these distances when a shorter distance is found.

How it works:

1. Set initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
2. Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
3. For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.

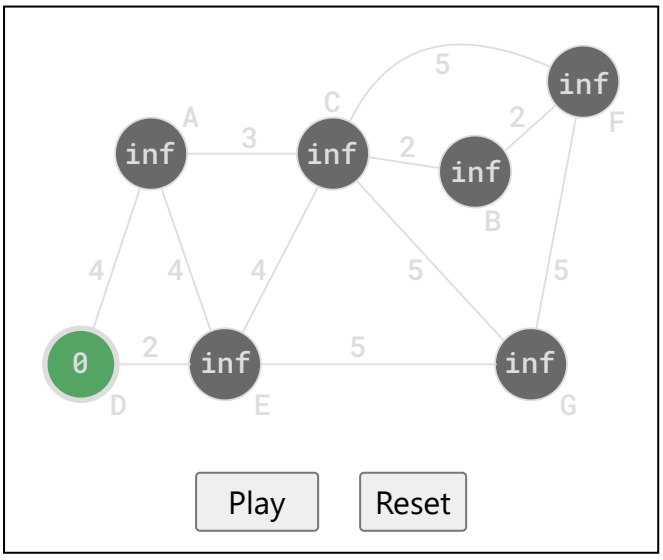
6. In the end we are left with the shortest path from the source vertex to every other vertex in the graph.

In the animation above, when a vertex is marked as visited, the vertex and its edges become faded to indicate that Dijkstra's algorithm is now done with that vertex, and will not visit it again.

Note: This basic version of Dijkstra's algorithm gives us the value of the shortest path cost to every vertex, but not what the actual path is. So for example, in the animation above, we get the shortest path cost value 10 to vertex F, but the algorithm does not give us which vertices (D->E->C->D->F) that make up this shortest path. We will add this functionality further down here on this page.

A Detailed Dijkstra Simulation

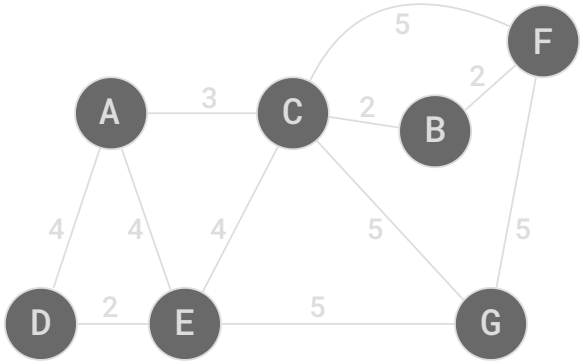
Run the simulation below to get a more detailed understanding of how Dijkstra's algorithm runs on a specific graph, finding the shortest distances from vertex D.



This simulation shows how distances are calculated from vertex D to all other vertices, by always choosing the next vertex to be the closest unvisited vertex from the starting point.

Manual Run Through

Consider the Graph below.

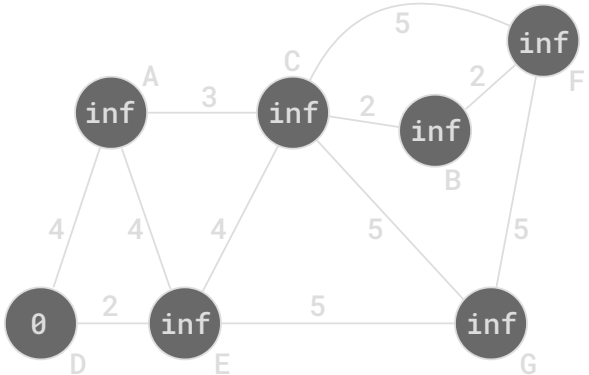


We want to find the shortest path from the source vertex D to all other vertices, so that for example the shortest path to C is D->E->C, with path weight 2+4=6.

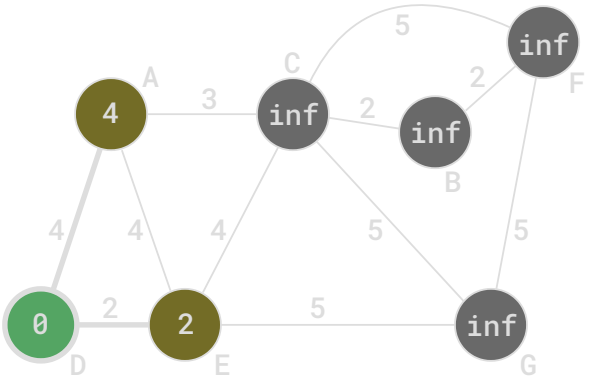
To find the shortest path, Dijkstra's algorithm uses an array with the distances to all other vertices, and initially sets these distances to infinite, or a very big number. And the distance to the vertex we start from (the source) is set to 0.

```
distances = [inf, inf, inf, 0, inf, inf, inf]
#vertices   [ A ,  B ,  C , D,  E ,  F ,  G ]
```

The image below shows the initial infinite distances to other vertices from the starting vertex D. The distance value for vertex D is 0 because that is the starting point.

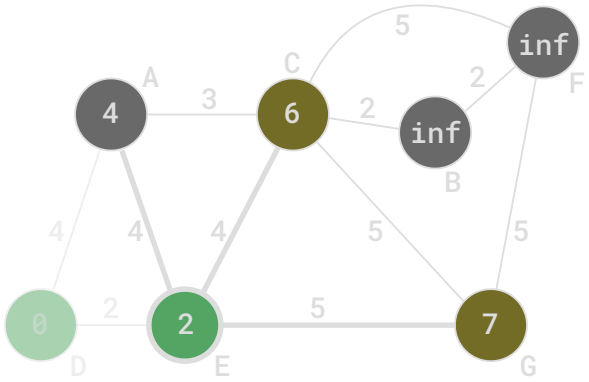


updating the distance values in this way is called 'relaxing'.



After relaxing vertices A and E, vertex D is considered visited, and will not be visited again.

The next vertex to be chosen as the current vertex must the vertex with the shortest distance to the source vertex (vertex D), among the previously unvisited vertices. Vertex E is therefore chosen as the current vertex after vertex D.

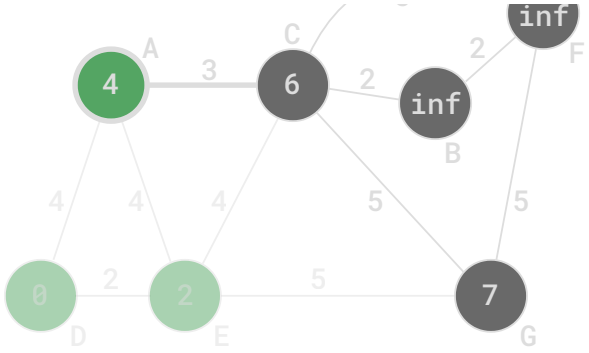


The distance to all adjacent and not previously visited vertices from vertex E must now be calculated, and updated if needed.

The calculated distance from D to vertex A, via E, is 2+4=6. But the current distance to vertex A is already 4, which is lower, so the distance to vertex A is not updated.

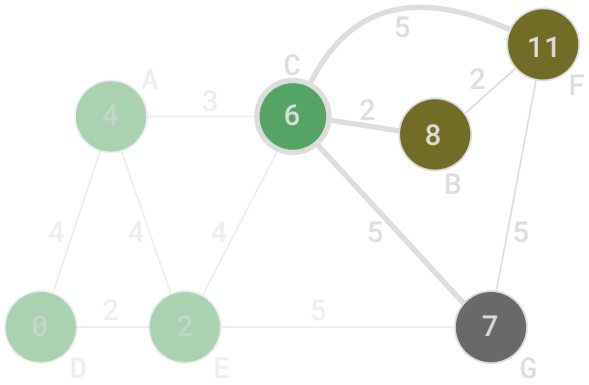
The distance to vertex C is calculated to be 2+4=6, which is less than infinity, so the distance to vertex C is updated.

Similarly, the distance to node G is calculated and updated to be 2+5=7.



The calculated distance to vertex C, via A, is 4+3=7, which is higher than the already set distance to vertex C, so the distance to vertex C is not updated.

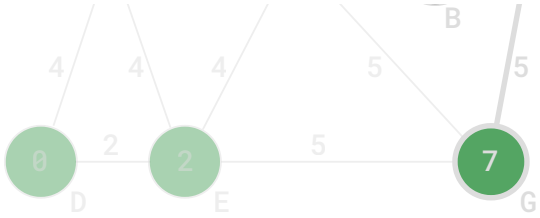
Vertex A is now marked as visited, and the next current vertex is vertex C because that has the lowest distance from vertex D between the remaining unvisited vertices.



Vertex F gets updated distance 6+5=11, and vertex B gets updated distance 6+2=8.

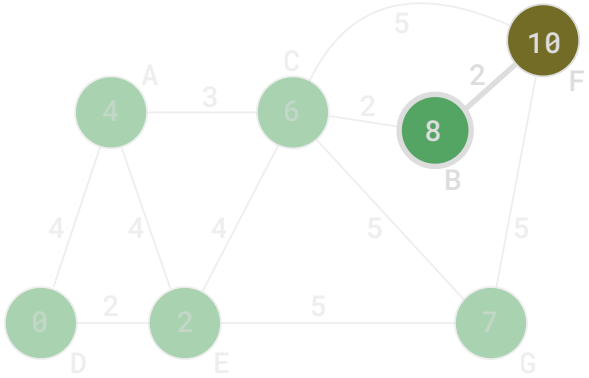
Calculated distance to vertex G via vertex C is 6+5=11 which is higher than the already set distance of 7, so distance to vertex G is not updated.

Vertex C is marked as visited, and the next vertex to be visited is G because is has the lowest distance between the remaining unvisited vertices.



Vertex F already has a distance of 11. This is lower than the calculated distance from G, which is 7+5=12, so the distance to vertex F is not updated.

Vertex G is marked as visited, and B becomes the current vertex because it has the lowest distance of the remaining unvisited vertices.



The new distance to F via B is 8+2=10, because it is lower than F's existing distance of 11.

Vertex B is marked as visited, and there is nothing to check for the last unvisited vertex F, so Dijkstra's algorithm is finished.

Every vertex has been visited only once, and the result is the lowest distance from the source vertex D to every other vertex in the graph.

Implementation of Dijkstra's Algorithm

To implement Dijkstra's algorithm, we create a `Graph` class. The `Graph` represents the graph with its vertices and edges:

```
1 class Graph:
2     def __init__(self, size):
```

```
11
```

set to `0`.

Line 4: `size` is the number of vertices in the graph.

Line 5: The `vertex_data` holds the names of all the vertices.

Line 7-10: The `add_edge` method is used to add an edge from vertex `u` to vertex `v`, with edge weight `weight`.

Line 12-14: The `add_vertex_data` method is used to add a vertex to the graph. The index where the vertex should belong is given with the `vertex` argument, and `data` is the name of the vertex.

The `Graph` class also contains the method that runs Dijkstra's algorithm:

```
16      def dijkstra(self, start_vertex_data):
17          start_vertex = self.vertex_data.index(start_vertex_data)
21
22          for _ in range(self.size):
```

```
                break

            visited[u] = True




                    distances[v] = alt

    return distances
```

Line 18-19: The initial distance is set to infinity for all vertices in the `distances` array, except for the start vertex, where the distance is 0.

Line 20: All vertices are initially set to `False` to mark them as not visited in the `visited` array.

Line 23-28: The next current vertex is found. Outgoing edges from this vertex will be checked to see if shorter distances can be found. It is the unvisited vertex with the lowest distance from the start.

means that all vertices that are reachable from the source have been visited.

Line 33: The current vertex is set as visited before relaxing adjacent vertices. This is more effective because we avoid checking the distance to the current vertex itself.

Line 35-39: Distances are calculated for not visited adjacent vertices, and updated if the new calculated distance is lower.

After defining the `Graph` class, the vertices and edges must be defined to initialize the specific graph, and the complete code for this Dijkstra's algorithm example looks like this:

Example

Python:

```
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight
            self.adj_matrix[v][u] = weight  # For undirected grap

    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

```

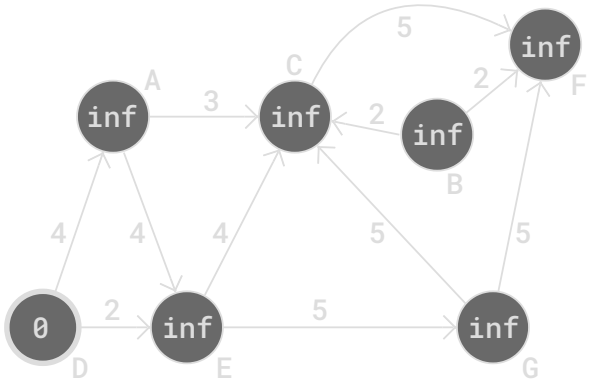
Run Example »

Dijkstra's Algorithm on Directed Graphs

To run Dijkstra's algorithm on directed graphs, very few changes are needed.

Similarly to the change we needed for cycle detection for directed graphs, we just need to remove one line of code so that the adjacency matrix is not symmetric anymore.

Let's implement this directed graph and run Dijkstra's algorithm from vertex D.



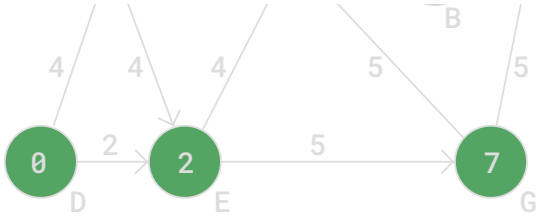
Here is the implementation of Dijkstra's algorithm on the directed graph, with D as the source vertex:

```python
class Graph:
    def __init__(self, size):
        self.adj_matrix = [[0] * size for _ in range(size)]
        self.size = size
        self.vertex_data = [''] * size

    def add_edge(self, u, v, weight):
        if 0 <= u < self.size and 0 <= v < self.size:
            self.adj_matrix[u][v] = weight


    def add_vertex_data(self, vertex, data):
        if 0 <= vertex < self.size:
            self.vertex_data[vertex] = data

    def dijkstra(self, start_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        distances = [float('inf')] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i

            if u is None:
                break

            visited[u] = True

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
```

```
g = Graph(7)

g.add_vertex_data(0, 'A')
g.add_vertex_data(1, 'B')
g.add_vertex_data(2, 'C')
g.add_vertex_data(3, 'D')
g.add_vertex_data(4, 'E')
g.add_vertex_data(5, 'F')
g.add_vertex_data(6, 'G')

g.add_edge(3, 0, 4)  # D -> A, weight 5
g.add_edge(3, 4, 2)  # D -> E, weight 2
g.add_edge(0, 2, 3)  # A -> C, weight 3
g.add_edge(0, 4, 4)  # A -> E, weight 4
g.add_edge(4, 2, 4)  # E -> C, weight 4
g.add_edge(4, 6, 5)  # E -> G, weight 5
g.add_edge(2, 5, 5)  # C -> F, weight 5
g.add_edge(1, 2, 2)  # B -> C, weight 2
g.add_edge(1, 5, 2)  # B -> F, weight 2
g.add_edge(6, 5, 5)  # G -> F, weight 5

# Dijkstra's algorithm from D to all vertices
print("Dijkstra's Algorithm starting from vertex D:\n")
distances = g.dijkstra('D')
for i, d in enumerate(distances):
    print(f"Shortest distance from D to {g.vertex_data[i]}: {d}")
```

Run Example »

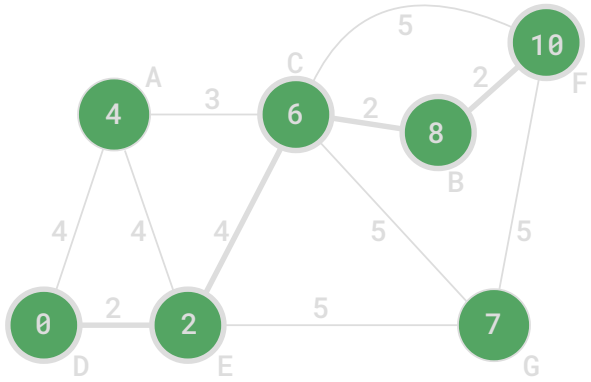
The image below shows us the shortest distances from vertex D as calculated by Dijkstra's algorithm.



This result is similar to the previous example using Dijkstra's algorithm on the undirected graph. However, there's a key difference: in this case, vertex B cannot be visited from D, and this means that the shortest distance from D to F is now 11, not 10, because the path can no longer go through vertex B.

Returning The Paths from Dijkstra's Algorithm

With a few adjustments, the actual shortest paths can also be returned by Dijkstra's algorithm, in addition to the shortest path values. So for example, instead of just returning that the shortest path value is 10 from vertex D to F, the algorithm can also return that the shortest path is "D->E->C->B->F".



To return the path, we create a `predecessors` array to keep the previous vertex in the shortest path for each vertex. The `predecessors` array can be used to backtrack to find the shortest path for every vertex.

Example

Python:

```
    def dijkstra(self, start_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        distances = [float('inf')] * self.size

        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i

            if u is None:
                break

            visited[u] = True

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
                        distances[v] = alt


        return distances, predecessors

```

```
# Dijkstra's algorithm from D to all vertices
print("Dijkstra's Algorithm starting from vertex D:\n")
distances, predecessors = g.dijkstra('D')
for i, d in enumerate(distances):
    path = g.get_path(predecessors, 'D', g.vertex_data[i])
    print(f"{path}, Distance: {d}")
```

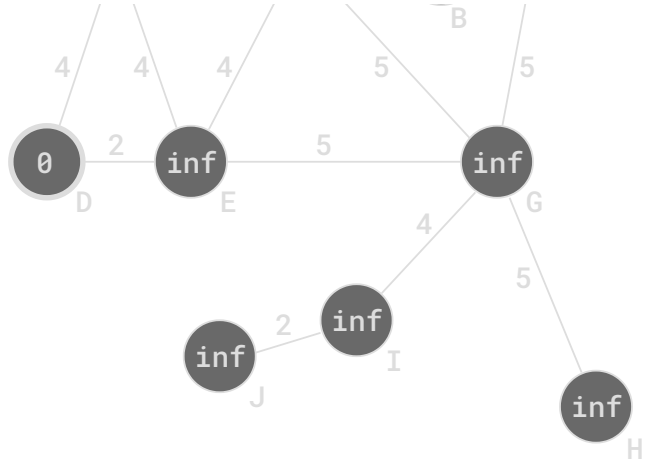
Run Example »

Line 7 and 29: The `predecessors` array is first initialized with `None` values, then it is updated with the correct predecessor for each vertex as the shortest path values are updated.

Line 33-42: The `get_path` method uses the `predecessors` array and returns a string with the shortest path from start to end vertex.

Dijkstra's Algorithm with a Single Destination Vertex

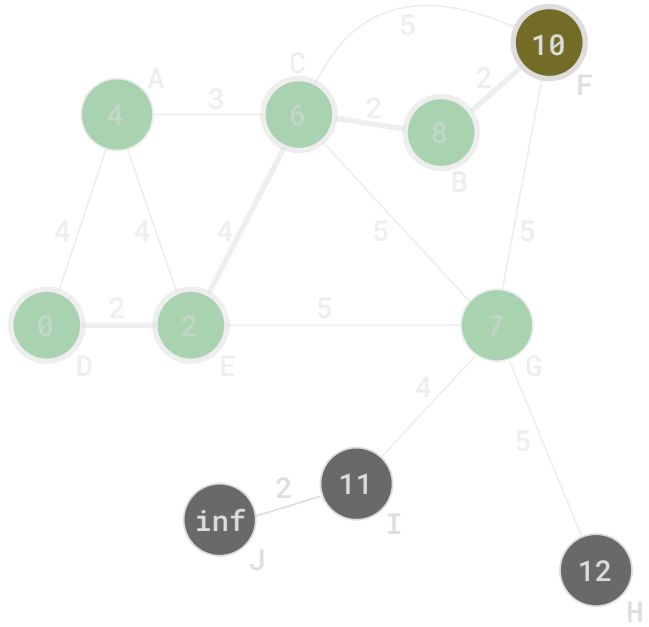
Let's say we are only interested in finding the shortest path between two vertices, like finding the shortest distance between vertex D and vertex F in the graph below.



Dijkstra's algorithm is normally used for finding the shortest path from one source vertex to all other vertices in the graph, but it can also be modified to only find the shortest path from the source to a single destination vertex, by just stopping the algorithm when the destination is reached (visited).

This means that for the specific graph in the image above, Dijkstra's algorithm will stop after visiting F (the destination vertex), before visiting vertices H, I and J because they are farther away from D than F is.

Below we can see the status of the calculated distances when Dijkstra's algorithm has found the shortest distance from D to F, and stops running.



In the image above, vertex F has just got updated with distance 10 from vertex B. Since F is the unvisited vertex with the lowest distance from D, it would normally be the next current

destination vertex:

Example

Python:

```
class Graph:
    # ... (existing methods)

    def dijkstra(self, start_vertex_data, end_vertex_data):
        start_vertex = self.vertex_data.index(start_vertex_data)
        end_vertex = self.vertex_data.index(end_vertex_data)
        distances = [float('inf')] * self.size
        predecessors = [None] * self.size
        distances[start_vertex] = 0
        visited = [False] * self.size

        for _ in range(self.size):
            min_distance = float('inf')
            u = None
            for i in range(self.size):
                if not visited[i] and distances[i] < min_distance:
                    min_distance = distances[i]
                    u = i




            visited[u] = True
            print(f"Visited vertex: {self.vertex_data[u]}")

            for v in range(self.size):
                if self.adj_matrix[u][v] != 0 and not visited[v]:
                    alt = distances[u] + self.adj_matrix[u][v]
                    if alt < distances[v]:
                        distances[v] = alt
```

```
# Example usage
g = Graph(7)
# ... (rest of the graph setup)
distance, path = g.dijkstra('D', 'F')
print(f"Path: {path}, Distance: {distance}")
```

Run Example »

Line 20-23: If we are about to choose the destination vertex as the current vertex and mark it as visited, it means we have already calculated the shortest distance to the destination vertex, and Dijkstra's algorithm can be stopped in this single destination case.

Time Complexity for Dijkstra's Algorithm

With V as the number of vertices in our graph, the time complexity for Dijkstra's algorithm is

$$O(V^2)$$

The reason why we get this time complexity is that the vertex with the lowest distance must to be search for to choose the next current vertex, and that takes $O(V)$ time. And since this must to be done for every vertex connected to the source, we need to factor that in, and so we get time complexity $O(V^2)$ for Dijkstra's algorithm.

By using a Min-heap or Fibonacci-heap data structure for the distances instead (not yet explained in this tutorial), the time needed to search for the minimum distance vertex is reduced from $O(V)$ to $O(\log V)$, which results in an improved time complexity for Dijkstra's algorithm

$$O(V \cdot \log V + E)$$

Where V is the number of vertices in the graph, and E is the number of edges.

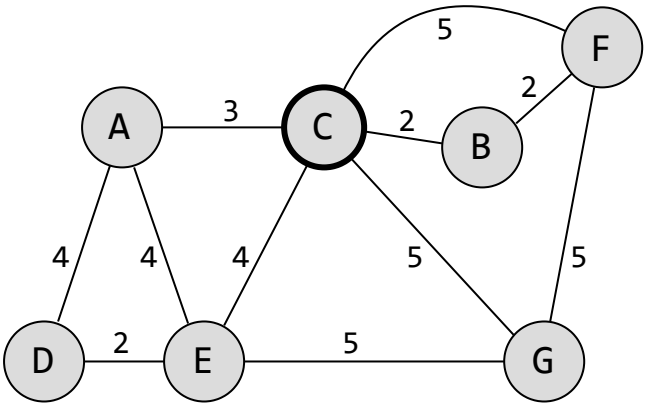
The improvement we get from using a Min-heap data structure for Dijkstra's algorithm is especially good if we have a large and sparse graph, which means a graph with a large number of vertices, but not as many edges.

DSA Exercises

Test Yourself With Exercises

Exercise:

Using Dijkstra's algorithm to find the shortest paths from vertex C in this graph:



What is the next vertex to be visited after C is visited?

```
Using Dijkstra's algorithm,
the next vertex to be visited
after vertex C is vertex   .
```

Submit Answer »

Start the Exercise

Top Tutorials

HTML Tutorial
CSS Tutorial
JavaScript Tutorial
How To Tutorial
SQL Tutorial
Python Tutorial
W3.CSS Tutorial
Bootstrap Tutorial
PHP Tutorial
Java Tutorial
C++ Tutorial
jQuery Tutorial

Top References

HTML Reference
CSS Reference
JavaScript Reference
SQL Reference
Python Reference
W3.CSS Reference
Bootstrap Reference
PHP Reference
HTML Colors