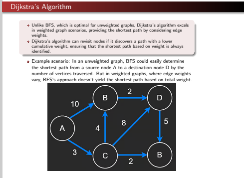
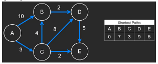


- Unlike BFS, which is optimal for unweighted graphs, Dijkstra's algorithm excels in weighted graph scenarios, providing the shortest path by considering edge weights.
- Example scenario: In an unweighted graph, BFS could easily determine the shortest path from a source node A to a destination node D by the number of vertices traversed. But in weighted graphs, where edge weights vary, BFS's approach doesn't yield the shortest path based on total weight.
- Dijkstra's algorithm systematically explores paths from the source node, prioritizing paths with the lowest cumulative weight, effectively finding the "lightest" path to each node.
- Key distinction: Unlike BFS, Dijkstra's algorithm can revisit nodes if it discovers a path with a lower cumulative weight, ensuring that the shortest path based on weight is always identified.
- This approach is essential in networks where paths have varying costs, distances, or any metric that adds weight to edges, making it a cornerstone algorithm for routing and navigation problems.

The Setup for Dijkstra's Algorithm

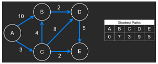
- Problem Statement: Starting from node A, determine the shortest path length to all other nodes in a weighted graph.
- Objective: Identify paths with the smallest total weight (cumulative edge weights), which we refer to as the "lightest" or shortest paths.
- Example Graph: Consider a graph with vertices A, B, C, D, E, and weighted edges such as [A,B,10], [A,C,3], [B,D,2], [C,B,4], [C,D,8], [C,E,2], [D,E,5].



- Problem Statement: Starting from node A, determine the shortest path length to all other nodes in a weighted graph.
- Objective: Identify paths with the smallest total weight (cumulative edge weights), which we refer to as the "lightest" or shortest paths.
- Example Graph: Consider a graph with vertices A, B, C, D, E, and weighted edges such as [A,B,10], [A,C,3], [B,D,2], [C,B,4], [C,D,8], [C,E,2], [D,E,5].
- Analysis:
  - From A to C: The shortest path is directly A -¿ C with a weight of 3, establishing the most efficient route despite other potential paths through intermediate nodes due to higher costs.
  - The graph demonstrates the principle of Dijkstra's algorithm in action, effectively navigating the weighted connections to find the minimum travel cost to each node from the source, A.
  - It's crucial that the graph does not contain negative weights to ensure the validity of the algorithm's assumptions and outcomes.

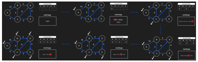
Walk-through of Dijkstra's Algorithm

- Initial Setup: Starting from vertex A, the distance from A to A is 0. Potential paths from A include A -¿ B with a weight of 10, and A -¿ C with a weight of 3.
- Path Exploration:
  - For vertex B, the shortest path is updated to 7 (A -¿ C -¿ B) since it is more efficient than directly from A to B.
  - For vertex C, the most direct and only path is A -¿ C, making the shortest path weight 3.
  - For vertex D, the shortest path is found through A -¿ C -¿ B -¿ D with a cumulative weight of 9.
  - For vertex E, the shortest path is A -¿ C -¿ E with a total weight of 5. It is noted that other paths exist but are not as efficient.



- Greedy Algorithm: Dijkstra's algorithm exemplifies a greedy approach by selecting the minimum weight edge at each step to ensure the optimal path is followed.
- Conclusion: This methodical exploration of paths and their weights, based on Dijkstra's algorithm, illustrates the process of finding the shortest, or "lightest", paths from a source vertex to all other vertices in a weighted graph.

Implementation of Dijkstra's Algorithm

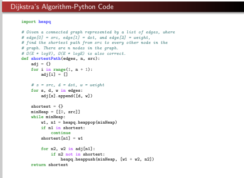


Min-Heap Utilization

The algorithm utilizes a min-heap to prioritize paths with the minimum weight. Each entry in the min-heap contains the node identifier and its associated cost, ensuring paths are sorted by their cost.

Data Structures

- A min-heap to manage the exploration of paths based on ascending costs.
- A hashmap to track the shortest paths from the source to every vertex.



Dijkstra's Algorithm-Python Code

```python
import heapq

# Given a connected graph represented by a list of edges, where
# edge[0] = src, edge[1] = dst, and edge[2] = weight,
# find the shortest path from src to every other node in the
# graph. There are n nodes in the graph.
# O(E * logV), O(E * logE) is also correct.
def shortestPath(edges, n, src):
    adj = {}
    for i in range(1, n + 1):
        adj[i] = []

    # s = src, d = dst, w = weight
    for s, d, w in edges:
        adj[s].append([d, w])

    shortest = {}
    minHeap = [[0, src]]
    while minHeap:
        w1, n1 = heapq.heappop(minHeap)
        if n1 in shortest:
            continue
        shortest[n1] = w1

        for n2, w2 in adj[n1]:
            if n2 not in shortest:
                heapq.heappush(minHeap, [w1 + w2, n2])
    return shortest
```

- **Process Overview**: The algorithm starts with the source node inserted into the min-heap with a cost of 0. It then iterates, popping the node with the lowest cost from the heap and exploring its unvisited neighbors. Each neighbor, if not already visited, is added to the heap with its cumulative cost from the source.
- **Key Operations**:
  - Building an adjacency list (`adj`) from the given edges to facilitate graph traversal.
  - Maintaining a hashmap (`shortest`) to record the minimum distance from the source to each vertex.
  - Continuously updating the min-heap and hashmap until all possible paths are explored.

Time Complexity of Dijkstra's Algorithm

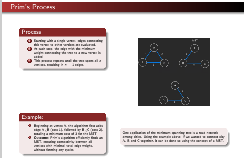
- **Overview**: Dijkstra's algorithm's time complexity is primarily influenced by the operations on the min-heap and the structure of the graph.
- **Complexity Analysis**:
  - The time complexity is denoted as $O(E \log E)$, where E represents the number of edges in the graph.
  - In a dense graph scenario, where each vertex is connected to every other vertex, the maximum number of edges approaches V^2, making the graph fully connected.
  - The algorithm's reliance on the min-heap for edge selection means that both insertion and removal operations are bound by $O(\log n)$ complexity, where n is the number of elements in the heap.
  - Given that, in the worst-case scenario, the heap may contain all edges of the graph, the overall time complexity sums up to $O(E \log E)$, accounting for the heap operations across all edges.
- **Implications**: This complexity indicates that while Dijkstra's algorithm is efficient for finding the shortest path in weighted graphs, its performance may vary significantly with the density of the graph and the distribution of edge weights.

Prim's Minimum Spanning Tree Algorithm

- Prim's algorithm is a greedy algorithm utilized for finding the minimum spanning tree (MST) of a weighted undirected graph.
- **Spanning Tree**: A subset of G's edges forming a tree that includes all the vertices with minimized total weight.
- **Key Characteristics**:
  - A tree cannot have cycles, implying an MST for a graph G with n nodes contains exactly $n-1$ edges.
  - The algorithm begins with an empty tree and iteratively adds the least weight edge from the graph that connects a vertex in the tree to a vertex outside.

Prim's algorithm is utilized for identifying a minimum spanning tree within a weighted undirected graph, functioning on the principle of a greedy algorithm, akin to Dijkstra's approach. The core objective is to find a spanning tree where the cumulative weight of its edges is minimized, effectively achieving cost efficiency similar to the goal in Dijkstra's algorithm.

A spanning tree of a given graph G is essentially a subset of G's edges that connects all vertices in G without creating any cycles and, importantly, minimizing the overall edge weight. This property underscores a crucial aspect of trees: they are inherently connected graphs devoid of cycles. Consequently, for a graph G comprising n nodes, the spanning tree derived from Prim's algorithm will contain precisely $n-1$ edges, ensuring full connectivity among all nodes while adhering to the tree structure's cyclic restriction.



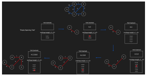
The concept underpinning Prim's algorithm acknowledges that a graph G can host multiple legitimate minimum spanning trees (MSTs), each potentially sharing an equivalent cost. The pivotal question then becomes: how do we isolate the MST from G? Prim's algorithm approaches this by initiating with a blank spanning tree. It then systematically traverses each vertex, selecting the neighbor vertex connected by the least weight edge. This process iterates until the tree comprises $n-1$ edges, aligning with the stipulation that a tree, by definition, should have one less edge than the number of nodes to remain acyclic and connected.

An illustrative scenario presents itself with starting at vertex A. Assuming the least costly connection from A leads to B with a weight of 1, followed by the selection of edge B to C weighing 2, the algorithm concludes upon achieving $n-1$ edges, culminating in a minimum total cost of 3. This forms the MST. In contrast, while other spanning trees within the graph might qualify as valid, they may not achieve the minimum possible cost, as depicted in the accompanying visual representation.

Prim's Algorithm: The Algorithmic Essence

Prim's algorithm, akin to Dijkstra's, adeptly finds the minimum spanning tree (MST) for a graph, utilizing a min-heap for efficient selection of the lowest-cost connections between vertices.

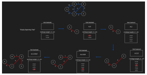
- Each node in the min-heap is characterized by ⟨weight, n1, n2⟩, indicating the cost to move from vertex *n1* to *n2*.
- A `visit` hashset tracks visited nodes to prevent cyclic paths, ensuring a growing tree structure.
- The MST is incrementally constructed by selecting the smallest available edge not forming a cycle.



Algorithm Termination Conditions

Prim's algorithm employs strategic checks to determine the completion of the MST construction:

- The algorithm proceeds until the min-heap is exhausted, indicating all possible edges have been considered.
- It halts once every vertex has been visited (`visit.size() == n`), signifying full coverage of the graph.
- A termination signal is also the attainment of $n - 1$ edges in the MST, aligning with the definition of a spanning tree.



Code Implementation for Prim's Algorithm

To transform an edge list into a usable structure for Prim's Algorithm, we construct an adjacency list that accounts for the undirected nature of the graph.

Building the Adjacency List

We initialize `adj`, a hashmap, to represent each vertex's connections:

- Key: Vertex
- Value: List of neighboring vertices and their connecting edge weights

This setup ensures that each edge's bidirectional nature is accurately represented, setting the stage for the MST construction.

```python
import heapq
# Given a list of edges of a connected undirected graph,
# with nodes numbered from 1 to n,
# return a list edges making up the minimum spanning tree.
def minimumSpanningTree(edges, n):
    adj = {}
    for i in range(1, n + 1):
        adj[i] = []
    for n1, n2, weight in edges:
        adj[n1].append([n2, weight])
        adj[n2].append([n1, weight])
```

Prim's Algorithm: Heap and MST Construction

In Prim's Algorithm, the selection of edges for the Minimum Spanning Tree (MST) relies on a priority queue, implemented as a min-heap, to ensure we're always processing the smallest weighted edge available.

Heap Initialization and MST Construction

- Initialize the minHeap with the source node's (node 1) neighbors.
- Use `mst` list to track edges of the MST.
- Use `visit` set to track visited nodes.

```
# Initialize the heap by choosing a single node
# (in this case 1) and pushing all its neighbors.
minHeap = []
for neighbor, weight in adj[1]:
    heapq.heappush(minHeap, [weight, 1, neighbor])

mst = []
visit = set()
visit.add(1)
while len(visit) < n:
    weight, n1, n2 = heapq.heappop(minHeap)
    if n2 in visit:
        continue

    mst.append([n1, n2])
    visit.add(n2)
    for neighbor, weight in adj[n2]:
        if neighbor not in visit:
            heapq.heappush(minHeap, [weight, n2, neighbor])
return mst
```

Kruskal's Algorithm Implementation

Implementation Strategy

- Utilize a minHeap to sort all edges by their weight to ensure we're always considering the edge with the minimum weight next.
- Directly work with edges, bypassing the need for an adjacency list, focusing on global minimum weights.

Union-Find Data Structure

- Initialize the UnionFind structure with each vertex pointing to itself.
- For each edge popped from the minHeap:
  - Attempt to union the vertices of the edge.
  - If union is successful (no cycle created), include the edge in the MST.
  - If a cycle would be formed, discard the edge.
- Continue until the MST contains $n - 1$ edges.

```python
    # Given an list of edges of a connected undirected graph,
# with nodes numbered from 1 to n,
# return a list edges making up the minimum spanning tree.
def minimumSpanningTree(edges, n):
    minHeap = []
    for n1, n2, weight in edges:
        heapq.heappush(minHeap, [weight, n1, n2])

    unionFind = UnionFind(n)
    mst = []
    while len(mst) < n - 1:
        weight, n1, n2 = heapq.heappop(minHeap)
        if not unionFind.union(n1, n2):
            continue
        mst.append([n1, n2])
    return mst
```

Key Implementation Note

The UnionFind class, used here, facilitates efficient cycle detection and set management.

Topological Sort: The Idea

The Idea:

- Topological sort is a linear ordering of vertices in a directed acyclic graph (DAG), where for each directed edge uv, vertex u comes before v in the ordering.
- This sorting is particularly useful in scenarios where you need to understand the sequence of tasks or events, like prerequisites in university courses.
- In a DAG, if there's a path from vertex u to vertex v, u must appear before v in the order.
- It provides a way to execute tasks while respecting all precedence relationships.

Illustration: University Courses

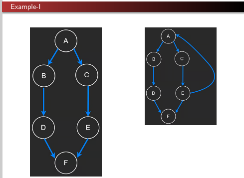
- Consider university courses as nodes in a DAG, where edges represent prerequisite requirements.
- Some courses (nodes) can be taken without prerequisites, while others depend on the completion of prior courses.
- A topological sort of this graph would provide an order in which to take the courses, ensuring all prerequisites are met.
- Example: If Course C depends on Courses A and B, then A and B will appear before C in the topological ordering.

Key Points in Topological Sorting

- It's applicable only to DAGs. A cyclic graph does not have a valid topological ordering.
- The order is not necessarily unique. There can be multiple valid topological sorts for a given graph.
- Topological sorts are used in scheduling tasks, ordering compiler tasks, resolving symbol dependencies in linkers, etc.

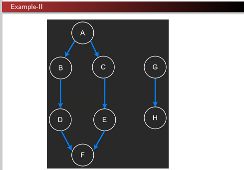
Topological Sort
The idea Topological sort is a way of sorting a directed acyclic graph (DAG) such that each node comes before its dependent nodes. A simple example of this is university courses. There are some courses that can be taken without any pre-requisites and then there are those that have pre-requisites, i.e. you cannot take them unless you have taken other courses first.
In other words, some courses can be taken independent of other courses and others have to be taken in a specific order. We can represent this scenario using a DAG, where the edges represent the dependencies between the courses.

So, if we have node C and it has node A and B as its dependents, A and B will appear before C in the topological ordering. What order they appear in is not important unless A and B also have a dependency on each other.



Suppose we are given the following directed acyclic graph (DAG). The topological ordering for this graph would be A,B,C,D,E,F. Notice that each node appears before its dependent node.

This is a rather simple example. We mentioned previously that topological sort works on acyclic graphs. What if we had a cycle in our graph? Let's take a slight modification of the graph above and apply the same concept to it. In this case, we have an edge coming out of E, going into A. The order would be: E,A,B,C,D,E,F. This actually contradicts the idea of topological sort since it is not possible to have E before A, and also after A. This would be like saying to take course A, you must take course E first, but to take course E, you must take course A first - it is a cycle.



Even if there are no cycles allowed, topological sort will still work on disconnected graphs. If we have two connected components in a graph, the ordering in which we place the vertices of the individual disconnected components does not matter as they are independent of each other. The graph has two connected components and one possible valid ordering could be A,B,C,D,E,F,G,H.