Exam 4 Notes

Dijkstra's Algorithm

Dijkstra's Algorithm is a classic algorithm in computer science, used to find the shortest paths from a source vertex to all other vertices in a weighted graph with non-negative edge weights. This algorithm is particularly useful in GPS networks to find the shortest path efficiently.

Algorithm Overview

The essence of Dijkstra's Algorithm lies in its use of a priority queue to repeatedly select the vertex with the smallest tentative distance from the source. The algorithm maintains several key structures during execution:

- A set, often referred to as `visited`, to keep track of vertices whose minimum distance from the source has already been determined.
- An array or dictionary `dist`, where `dist[v]` holds the shortest found distance from the source to vertex `v`.
- A priority queue that prioritizes vertices based on their tentative distance from the source, helping to explore the most promising vertex next.

Operation Per Iteration

During each iteration of Dijkstra's Algorithm, the following steps are executed:

1. Select the vertex `u` with the smallest tentative distance (`dist[u]`) from the priority queue that has not been visited yet.
2. Mark vertex `u` as visited. This means that the shortest path to `u` has been permanently determined.
3. For each neighboring vertex `v` of `u` that is not visited:
   (a) Calculate the alternative path distance to `v`, which is `dist[u]` plus the weight of the edge from `u` to `v`.
   (b) If this alternative distance is less than the currently known distance `dist[v]`, update `dist[v]` with the smaller distance and insert `v` into the priority queue with its updated distance.

Vertex Exploration Order

The order in which vertices are explored in Dijkstra's Algorithm is determined by the priority queue which prioritizes vertices based on their distance from the source. This structure ensures that at any step, the vertex with the lowest known distance is explored next, implementing a greedy strategy to always move towards the closest unvisited vertex. This property is crucial for guaranteeing that the found paths are the shortest.

Complexity

The time complexity of Dijkstra's Algorithm can vary depending on the implementation:

- Using a binary heap for the priority queue results in a complexity of $\mathcal{O}((V + E)\log(V))$, where V is the number of vertices and E is the number of edges.
- If a Fibonacci heap is used, the complexity can be improved to $\mathcal{O}(E + V\log(V))$.

This makes Dijkstra's algorithm very efficient for graphs with large numbers of vertices but relatively fewer edges.

Minimum Spanning Trees (MST)

A Minimum Spanning Tree (MST) is a subset of the edges of a connected, edge-weighted undirected graph that connects all the vertices together, without any cycles and with the minimal possible total edge weight. This concept is crucial in networks like telecommunication, where the goal is to lay out cables at the minimum possible cost.

Properties of MST

The properties of a Minimum Spanning Tree include:

- An MST for a graph with V vertices has exactly $V - 1$ edges.
- If all weights are distinct, the MST is unique; however, if weights are not distinct, multiple MSTs may exist.
- The total weight of all the edges in the MST is the minimum among all possible spanning trees of the graph.
- MSTs are acyclic and connect all vertices in the graph.

Kruskal's Algorithm

Kruskal's Algorithm is a popular method to find the MST of a graph. It follows a greedy approach and is typically easier to implement than Prim's algorithm. The steps are as follows:

1. Sort all the edges in non-decreasing order of their weight.
2. Initialize a forest F as a set of trees with each vertex in the graph as a separate tree.
3. For each sorted edge (u, v):
   (a) If u and v are in different trees (i.e., adding (u, v) does not form a cycle),
   (b) Add (u, v) to F, and merge the two trees in F into one (a union-find data structure is commonly used for this step).
4. The process continues until F contains exactly one tree, which is the MST.

Complexity of Kruskal's Algorithm

The time complexity of Kruskal's Algorithm mainly depends on the edge sorting step and the union-find operations:

- Sorting the edges takes $\mathcal{O}(E \log(E))$ time.
- Each union-find operation (both union and find) can be optimized to $\mathcal{O}(\log(V))$ using path compression and union by rank, making the overall complexity of managing the forest $\mathcal{O}(E \log(V))$.

Therefore, the total time complexity of Kruskal's Algorithm is $\mathcal{O}(E \log(E))$, which is effective for sparse graphs.

Depth First Search (DFS) and Breadth First Search (BFS)

Depth First Search (DFS) and Breadth First Search (BFS) are two fundamental algorithms used for exploring vertices and edges of a graph. These methods are pivotal for numerous applications in computer science, from pathfinding to analyzing networks.

Depth First Search (DFS)

DFS explores as far as possible along each branch before backing up. Here's how it works:

1. Start at the chosen vertex, marking it as visited.
2. Recursively visit any adjacent vertex that has not been visited yet, marking it as visited when you visit it.
3. This recursive visiting continues until all vertices connected by a path to the starting vertex are visited.

DFS can be implemented using a stack, either through the function call stack via recursion or an explicit stack data structure.

Breadth First Search (BFS)

BFS explores the neighbor nodes at the present depth prior to moving on to nodes at the next depth level. Here's the procedure:

1. Start at the chosen vertex, marking it as visited and enqueue it.
2. While the queue is not empty:
   (a) Dequeue the next vertex from the front of the queue.
   (b) Visit all its adjacent vertices, mark them as visited, and enqueue any vertex that has not been visited yet.

BFS uses a queue data structure to ensure that vertices are explored in the order they are reached.

Strongly Connected Components (SCC)

A Strongly Connected Component (SCC) of a directed graph is a maximal subset of vertices such that every vertex in the subset is reachable from every other vertex in the subset.

Properties of SCC

- In a directed graph, an SCC forms a cycle which means there is a path from each vertex to every other vertex in the same component.
- The union of all SCCs of a graph forms the entire graph.
- SCCs are the "meta-nodes" in the Condensed Graph of the original graph where each SCC is a node, and edges between SCCs represent paths in the original graph.

Finding SCCs

The common algorithms for finding all SCCs in a graph are:

1. Tarjan's Algorithm: It uses DFS to locate SCCs and can find all SCCs in one go, marking each strongly connected component as it goes.
2. Kosaraju's Algorithm: This method involves two passes of DFS. The first pass orders the vertices in decreasing finish time. In the second pass, the graph is reversed and DFS is applied in the order determined in the first pass.

Both algorithms run in linear time, $\mathcal{O}(V + E)$, where V is the number of vertices and E is the number of edges.

Reconstructing Graphs from Strongly Connected Components (SCC)

Given the SCCs of a directed graph, one might wonder about the structure of the original graph. Particularly, questions often arise regarding the minimum and maximum number of edges that the graph could potentially have based on its SCCs.

Understanding SCCs in Graph Reconstruction

Example: Consider a directed graph with 6 nodes divided into three maximal SCCs: {1,2}, {3,4}, {5,6}.

Graph Reconstruction from SCCs

Given these SCCs, we know each SCC forms a subgraph where every node is reachable from every other node within the same SCC. The challenge lies in determining the possible connections between these SCCs and the edges within each SCC.

Minimum Number of Edges

The minimum number of edges in the graph is achieved by:

- Including the minimum number of edges required to keep each SCC strongly connected.
- Not adding any additional edges between these SCCs.

Calculation:

- Each SCC with 2 nodes requires at least 2 edges to be strongly connected (forming a cycle).
- Therefore, for SCCs {1,2}, {3,4}, and {5,6}, we need at least 2+2+2 = 6 edges.

Thus, the minimum number of edges is 6.

Maximum Number of Edges

The maximum number of edges is computed by:

- Including all possible edges within each SCC to maintain strong connectivity.
- Adding all possible directed edges between different SCCs.

Calculation:

- Within each SCC of 2 nodes, we can have 2 direct edges (one in each direction), making it 4 possible edges (full connectivity).
- Between each pair of SCCs (say, between {1,2} and {3,4}), there can be 4 directed edges (1 to 3, 1 to 4, 2 to 3, 2 to 4).

- There are 3 pairs of such SCCs ({1,2} & {3,4}, {1,2} & {5,6}, and {3,4} & {5,6}), contributing 4*3 = 12 inter-SCC edges.
- Total edges within SCCs: 4+4+4 = 12.

Therefore, the maximum number of edges is 12 + 12 = 24.