Graph Algorithms (Spring 2025)

2. Traversals, Connectivity, Shortest paths

Teacher: Arnaud Casteigts *Assistant: Matteo De Francesco*

In this class, we review basic graph algorithms that are commonly used as building blocks in more complex algorithms.

Keywords: graph search, depth-first search (DFS), breadth-first search (BFS), spanning trees, connectivity testing, decomposition into strongly connected components, weighted graphs, shortest paths, Disjktra's algorithm.

2.1 Graph search (DFS / BFS)

Depth-first search (DFS) and Breadth-first search (BFS) are two ways to *traverse* a graph. Many graph algorithms use them as building blocks. Many real-world programs also use it, e.g. searching for a file in a computer. The traversal starts at a given vertex, and proceeds by moving to other nodes repeatedly in the following way:

- Depth-first search: Loop over the local neighbors, visiting them recursively if they have not yet been visited.
- Breadth-first search: Loop over the local neighbors, adding them to a queue if they have not yet been visited. Then, unqueue the next node and repeat.

Several implementations are possible. The following has the advantage of using the same signature for both types of traversals:

```
def DFS(G, v, visited):
   visited[v] = True

   for ng in G.neighbors(node):
      if not visited[ng]:
          DFS(G, ng, visited)


G = ...
visited = [False] * G.number_of_nodes()
DFS(G, 0, visited) # or BFS
```

```
def BFS (G, v, visited):
    queue = [v]

    while len(queue) > 0:
        current = queue.pop(0)
        visited[current] = True
        for ng in G.neighbors(current):
            if not visited[ng] and ng not in queue:
                queue.append(ng)
```

The algorithms is the same for directed and undirected graphs, except that for directed graphs, we loop only over the *successors* of the current vertex. Intuitively, a DFS operates in a LIFO manner (Last In First Out, like a *stack*, implicitly used through recursion) and BFS in a FIFO manner (First In First Out, like a *queue*, explicitly used). The main advantage of the DFS is that it is very simple and "local" (no jumps in the graph), while the main advantage of the BFS is that it visits the nodes in increasing order of distance from the starting node: the nodes at distance 1 are visited before the nodes at distance 2, *etc.*

When implemented carefully, both algorithms have $O(m)$ time complexity.[1]

2.2 Connectivity test

Testing connectivity of a graph G can be done easily using traversals (DFS or BFS). If G is undirected, it is enough to run a single traversal starting at an arbitrary node, and check that all the nodes have been visited. If G is directed, testing strong connectivity is a bit more challenging. An obvious solution is to run a different traversal from *each* node, and check that each traversal visits all the nodes. However, this gives a complexity of $O(nm)$, which is not efficient. There exist a trick called Kosaraju's principle:

1. Run a traversal from an arbitrary node v and check that it visits all the nodes
2. Create another graph G^T by flipping the direction of all the arcs of G. This graph is called the *transpose* of G (because it's adjacency matrix is the transpose of that of G).
3. Run again a traversal from v in G^T and check that it visits all the nodes

The trick is that if v can reach all the other nodes in G^T, this means that v can *be reached by* all the other nodes in G (indeed, all paths are reversed in G^T). Thus, if steps 1 and 3 succeed, then v can both reach, and be reached by, all the other nodes, which implies that all the nodes can reach each other through v by composing these paths, thus G is strongly connected. Each of the three steps has time complexity $O(m)$, thus the overall algorithm has time complexity $O(m)$ as well.

2.3 Connected components

If the graph is not connected, we can use a similar strategy for computing the components.

2.3.1 Undirected graphs

The strategy is simply to start a traversal from the first unvisited vertex, repeatedly.

[1] Technically, it has $O(n + m)$ time complexity, where $+$ is to be understood as a max between the two. In the vast majority of interesting graphs, the number of edges is at least $O(n)$, so in these graphs $O(n+m)$ is the same as $O(m)$. Unless otherwise mentioned, we will always make this assumption.

```
def connected_components(G):
    visited = [False] * G.number_of_nodes()

    for v in range(G.nodes()):
        if (visited[v] == False):
            # new component to be explored
            DFS(G, v, visited)
```

When implemented correctly, the total cost is again $O(m)$, because every edge is considered by a single traversal.

2.3.2 Directed graphs

Computing the strongly connected components of a digraph is a bit more challenging. We will not review the algorithm, but you should know that it exists and that its time complexity is again $O(m)$. The algorithm actually produces a *decomposition into strongly connected components*, which is a new digraph where every node represents a strongly connected component, themselves connected by arcs in an acyclic way. Thus, the decomposition is a DAG, as illustrated in Figure 1.

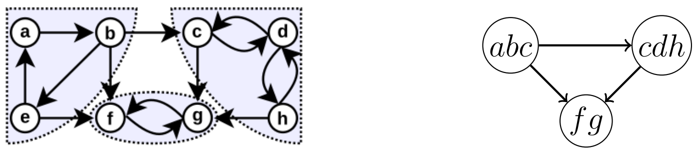

Figure 1: A digraph and its decomposition into strongly connected components.

2.4 Shortest paths

A path from u to v is *shortest* if no other path exists from u to v using fewer edges.

A shortest path from u to v can be computed using a simple BFS and using a list of parents (one for each node) as follows. Start a BFS from u. Every time a new node y is added to the queue while visiting node x, set `parent[y]=x`. We can stop after v is added to the queue. At this point, by following recursively `v, parent[v], parent[parent[v]], ...` until `u`, we have the shortest path from u to v (backward).

If we wait for the BFS to finish, then the `parent` list describes something stronger: a *tree* of shortest paths from u to all the other nodes that it can reach, as shown in Figure 2.

2.4.1 Weighted graphs

A **weighted graph** is a graph in which every edge is assigned a number (the weight). Formally, $G = (V, E, w)$, with $w : E \to \mathbb{R}$. Weighted graphs are used in many applications,

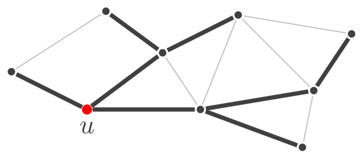

Figure 2: A tree of shortest paths from node u.

where the weights might represent costs, lengths or capacities. In a weighted graph, the *length* of a path refers to the *sum* of weights of its edges, and a path from u to v is *shortest* if this sum is minimum over all the possible paths from u to v.

The most common algorithm for computing shortest paths in weighted graphs is called Dijkstra's algorithm, this algorithm and its variants are used in countless applications, from transportation to network communications. The algorithm can be seen as a weighted version of the BFS, and similarly, it computes a *tree* of shortest paths from a given starting node. Note that the algorithm works only if the costs are non-negative.

Dijkstra's algorithm:

1. Initialize a cost of ∞ for each node, except the starting vertex (cost 0)
2. Find a node u of minimum cost that has not yet been visited
   $\rightarrow$ Mark u as visited
   $\rightarrow$ For each neighbor v of u, set $cost(v) = \min(cost(v), cost(u) + w(uv))$.
3. Repeat step 2 until all the nodes have been visited

Key property: After a node has been visited, its cost never changes because we picked the minimum cost node in each step. This makes Disjktra's algorithm a *greedy algorithm.* The best implementations of this algorithms run in time $O(m + n \log n)$. The outcome is illustrated in Figure 3.

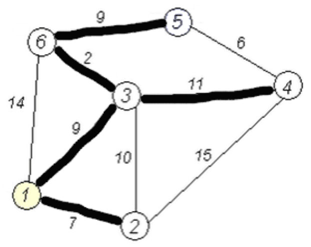

Figure 3: A tree of shortest paths from node 1, resulting from Dijkstra's algorithm.