Algorithms

Based on lectures by

Notes taken by yehelip

Winter 2025

These notes are not endorsed by the lecturers. I have revised them outside lectures to incorporate supplementary explanations, clarifications, and material for fun. While I have strived for accuracy, any errors or misinterpretations are most likely mine.

Contents

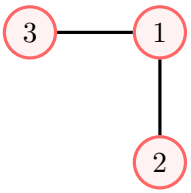
1 Preliminaries

This is a course about algorithms, and since many problems we are going to solve will require some knowledge about graphs we will write here some basic definitions so that we always have a reference if we forget something.

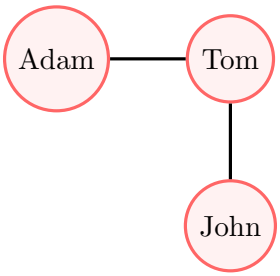
Definition 1.1 (Graph)**.** An ordered pair $G = (V, E)$ of V a set of vertices, and E a of $V \times V$ - a set of unordered pairs of vertices or "edges" is called a graph.

We can imagine graphs as nodes with names or values that represent them and the edges connect the nodes. For example we can consider the following graph

$$G = \left(\{1, 2, 3\}, \{(2, 3), (3, 1)\}\right) =$$



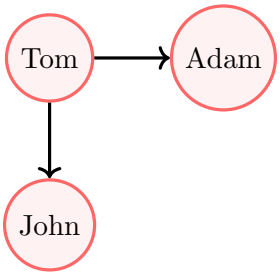
It looks very nice, we can think of it also as relations between people just by changing the names of the vertices:



This graph may represent that Adam knows Tom and John knows Adam but Adam and John don't know each other. We may also consider the relation of A loves B. In this case we need to define a new kind of graph, since even if Adam loves Tom, that doesn't mean that Tom loves Adam.

Definition 1.2 (Directed graph)**.** An ordered pair $G = (V, E)$ of V a set of vertices, and E a subset of $V \times V$ - a set of ordered pairs of vertices is called a dircted graph.

One example of a directed graph is:



Oh no! Tom loves two people, yet he isn't loved by anyone :(
Notice how changing the placement of the vertices doesn't change the graph itself.

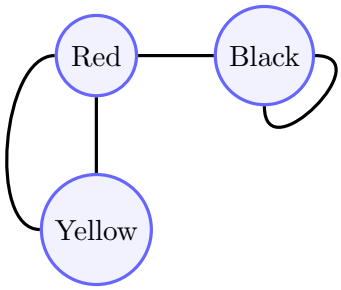
Definition 1.3 (Walk)**.** A walk is a finite or infinite sequence of edges which joins a sequence of vertices.

Definition 1.4 (Trail)**.** A trail is a walk in which all edges are distinct.

Definition 1.5 (Path)**.** A path is a trail in which all vertices (and therefore also all edges) are distinct.

Definition 1.6 (Simple graph)**.** A simple graph is a graph in which there are no edges from a vertex to itself, or multiple edges from on vertex to another.

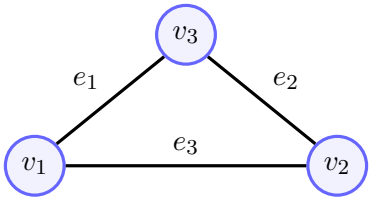
For example the following is not a simple graph:



Definition 1.7 (Circuit)**.** A circuit is a non-empty trail in which the first and last vertices are equal (closed trail).

Definition 1.8 (Cycle)**.** A cycle (or simple circuit) is a circuit in which only the first and last vertices are equal.

For example in the graph below:



We have the cycle $C = (v_1, e_1, v_3, e_2, v_2, e_3, v_1)$.

Definition 1.9 (Distance between vertices)**.** The distance between two vertices u, v is denoted as $\delta(u, v)$ and is the length of a shortest directed path from u to v consisting of arcs, provided at least one such path exists.

2 BFS

The BFS (Breadth First Search) algorithm is an algorithm that travereses all the vertices in the graph in a way breadth first approach, meaning that if we start at vertex s we will not traverse vertices in distance i from s before visiting all vertices of distance $< i$ from s. We can also use this algorithm to efficiently find the distance of any vertex from s.

Suppose we need to find the distance between the points s, v on a graph $G = (V, E)$, we could first look at u, then consider all of its neighbors and mark their distance from u as 1 and then consider all of the neighbors unvisited neighbors and mark their distance from u as 2 and so on. The reason this is called "Breadth First Search" is because we are going to mark all vertices with distance i from u before giving the vertices and distance $i + 1$ from u. What I just described is how to algorithm is supposed to work, but we need to be more exact and write psuedo-code for it, then prove its correctness, and calculate its complexity.

Algorithm 1: BFS Algorithm

Input: Graph $G = (V, E)$, starting vertex s
Output: Visited vertices in BFS order

```
1 Initialize queue Q;
2 Mark all vertices as unvisited;
3 Mark the source vertex s as visited and enqueue s to Q;
4 while Q is not empty do
5     Dequeue a vertex v from Q;
6     for each neighbor u of v do
7         if u is not visited then
8             Mark u as visited;
9             Enqueue u to Q;
```

This is the basic BFS algorithm that just traverses the graph. If we want to also find the distances of all the vertices from s we can first initialize our approximaition distance function and correct it along the way.

Algorithm 2: BFS Algorithm for finding distances

Input: Graph $G = (V, E)$, starting vertex s
Output: λ such that $\lambda = \delta$

```
1 Initialize queue Q;
2 Initialize the λ(s, v) = ∞ for every v ∈ V;
3 Redefine λ(s, s) = 0 and enqueue s to Q;
4 while Q is not empty do
5     Dequeue the top vertex v from Q;
6     for each neighbor u of v do
7         if λ(s, u) = ∞ then
8             λ(s, u) = λ(s, v) + 1;
9             Enqueue u to Q;
```

We need to show find its complexity and show correctness. First we can find the complexity. We know that in the initizlization we go over all the vertices so the complexity there is $O(|V|)$, and then every iteration we do in $O(\deg(u))$ so the total complexity is:

$$O(|V|) + \sum_{u \in V} O(\deg(u)) = O(|V|) + O(|E|) = O(|V| + |E|)$$

To prove correctness we will show that $\delta(s,v) \leq \lambda(s,v)$ and $\lambda(s,v) \leq \delta(s,v)$. If we have $\delta(s,v) = \infty$ then the inequality is correct, otherwise we can show that by induction over the order the vertices enter the queue.
Base case
if v is the first vertex to enter the queue then we have $v = s$ and then:

$$\delta(s,s) = \lambda(s,s) = 0$$

Step
Now suppose we know that for all the previous vertices that entered the queue we have the inequlity, by the algorithm definition we get:

$$\lambda(s,v) = \lambda(s,u) + 1 \underbrace{\geq}_{\text{induction}} \delta(s,v) + 1 \geq \delta(s,v)$$

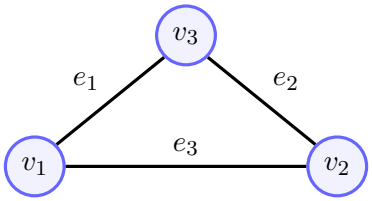
Which means that for all $v \in V$ we have $\delta(s,v) \leq \lambda(s,v)$ which completes one side of the proof. Try proving the other side yourself.

3 Topological Sorting

To talk about Topological sorting we first have to recall how we can define simple graphs. We can define graphs using an adjacency matrix such that:

1. $(A)_{ij}$ will be 1 if $(i, j) \in E$
2. $(A)_{ij}$ will be 0 if $(i, j) \notin E$

For example the adjacency matrix for the graph:



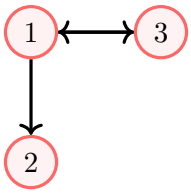
is:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix}$$

And with directed graphes we can define for $i \leq j$ as such:

1. $(A)_{ij}$ will be 1 if $(i, j) \in E$
2. $(A)_{ji}$ will be -1 if $(i, j) \in E$
3. $(A)_{ij}$ will be 0 if $(i, j) \notin E$

So for example for the graph:

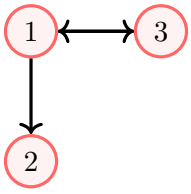


We get:

$$A = \begin{pmatrix} 0 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & 0 & 0 \end{pmatrix}$$

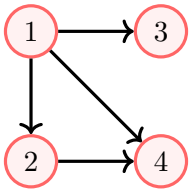
Definition 3.1 (Topological sorting)**.** A topological sorting of a directed graph is a linear ordering of its vertices such that for every directed edge (u, v) from vertex u to vertex v, u comes before v in the ordering.

For example a topological sorting of the graph above:

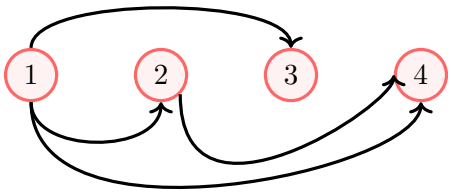


Is not possible. In fact such an ordering exists if and only if the graph we are trying to sort is a DAG.

For example here is a DAG:



A possible topological sorting is $(1, 2, 3, 4)$ because we get:



Algorithm 3: Kahn's Algorithm for Topological Sorting

Input: Directed Acyclic Graph (DAG) $G = (V, E)$
Output: Topologically sorted order of vertices

```
Compute in-degrees of all vertices;
Initialize queue Q with all vertices of in-degree 0; // Also known as sources
 Initialize empty list L;
while Q is not empty do
    Dequeue a vertex v from Q;
    Append v to L;
    for each neighbor u of v do
        Decrement in-degree of u;
        if in-degree of u becomes 0 then
            Enqueue u to Q;
if L contains all vertices then
    return L;
else
    Error: Graph has at least one cycle;
```