Shortest Paths, and Dijkstra’s Algorithm: Overview

- Graphs with lengths/weights/costs on edges.
- Shortest paths in edge-weighted graphs
- Dijkstra’s classic algorithm for computing single-source shortest paths.
An **edge-weighted directed graph**, $G = (V, E, w)$, has a length/weight/cost function, $w : E \rightarrow \mathbb{N}$, which maps each edge $(u, v) \in E$ to a non-negative integer "length" (or "weight", or "cost"): $w(u, v) \in \mathbb{N}$.

We can **extend** the "length" function $w$ to a function $w : V \times V \rightarrow \mathbb{N} \cup \{\infty\}$, by letting $w(u, u) = 0$, for all $u \in V$, and letting $w(u, v) = \infty$ for all $(u, v) \notin E$.

Consider a directed path:

$$x_0 e_1 x_1 e_2 \ldots e_n x_n$$

from $u = x_0 \in V$ to $v = x_n \in V$, in graph $G = (V, E, w)$. The **length** of this path is defined to be: $\sum_{i=1}^{n} w(x_{i-1}, x_i)$.
Graphs with edge “length” (or “weight/cost”)

An **edge-weighted directed graph**, \( G = (V, E, w) \), has a **length/weight/cost** function, \( w : E \rightarrow \mathbb{N} \), which maps each edge \((u, v) \in E\) to a non-negative integer “length” (or “weight”, or “cost”): \( w(u, v) \in \mathbb{N} \).

We can **extend** the “length” function \( w \) to a function \( w : V \times V \rightarrow \mathbb{N} \cup \{\infty\} \), by letting \( w(u, u) = 0 \), for all \( u \in V \), and letting \( w(u, v) = \infty \) for all \((u, v) \notin E\).

Consider a directed path:
\[
x_0 e_1 x_1 e_2 \ldots e_n x_n
\]
from \( u = x_0 \in V \) to \( v = x_n \in V \), in graph \( G = (V, E, w) \). The **length** of this path is defined to be: \( \sum_{i=1}^{n} w(x_{i-1}, x_i) \).

**Question**: Given \( G \) and a pair of vertices \( u, v \in V \), how do we compute the length of the **shortest path** from \( u \) to \( v \)?
Dijkstra’s single-source shortest-path algorithm

Input: Edge-weighted graph, $G = (V, E, w)$, with (extended) weight function $w : V \times V \rightarrow \mathbb{N}$, and a source vertex $s \in V$.

Output: Function $L : V \rightarrow \mathbb{N} \cup \{\infty\}$, such that for all $v \in V$, $L(v)$ is the length of the shortest path from $s$ to $v$ in $G$.

Algorithm:

Initialize: $S := \{s\}; \quad L(s) := 0$;
Initialize: $L(v) := w(s, v)$, for all $v \in V - \{s\}$;

while $(S \neq V)$ do

$u := \text{arg min}_{z \in V - S} \{L(z)\}$

$S := S \cup \{u\}$

for all $v \in V - S$ such that $(u, v) \in E$ do

$L(v) := \min\{L(v), L(u) + w(u, v)\}$

end for

end while

Output function $L(\cdot)$.
Why does Dijkstra’s algorithm work?

**Claim:** The While loop of Dijkstra’s algorithm maintains the following **invariant** properties of the function $L$ and the set $S$:

1. $\forall v \in S, L(v)$ is the shortest path length from $s$ to $v$ in $G$.
2. $\forall v \in V - S, L(v)$ is the length of the shortest path from $s$ to $v$ which uses only vertices in $S \cup \{v\}$.
3. For all $u \in S$ and $v \in V - S$, $L(u) \leq L(v)$.

Note that the three invariants hold after initialization, just prior to the first iteration of the while loop.
The claim follows once we prove (on board) that if the invariants hold just prior to a while loop iteration then they hold just after.
Since each iteration adds one vertex to $S$, it follows that the algorithm halts, at which point $S = V$, and thus, by invariant (1.), the function $L : V \to \mathbb{N} \cup \{\infty\}$ is the correct answer.
Remarks on Dijkstra’s Algorithm

- If Dijkstra’s algorithm is implemented naively, it has running time $O(n^2)$, where $n = |V|$.

- With clever data structures (e.g., so called “Fibonacci Heaps”) Dijkstra’s algorithm can be implemented much more efficiently: essentially in time $O(m + n \log n)$ where, $n = |V|$ and $m = |E|$.

This increased efficiency can make a big difference on huge “sparse” graphs, where $m$ is much smaller than $n^2$ (e.g., when out-degree is a fixed constant, $m \in O(n)$).

- Dijkstra’s algorithm can be augmented to also output a description of a shortest path from the source vertex $s$ to every other vertex $v$.

We will not describe these extensions, and we will certainly not assume that you know them.
Graph Colouring
Graph Colouring

Suppose we have \( k \) distinct colours with which to colour the vertices of a graph. Let \([k] = \{1, \ldots, k\}\). For an undirected graph, \( G = (V, E) \), an admissible vertex \( k \)-colouring of \( G \) is a function \( c : V \rightarrow [k] \), such that for all \( u, v \in V \), if \( \{u, v\} \in E \) then \( c(u) \neq c(v) \).

For an integer \( k \geq 1 \), we say an undirected graph \( G = (V, E) \) is \( k \)-colourable if there exists a \( k \)-colouring of \( G \).

The **chromatic number** of \( G \), denoted \( \chi(G) \), is the *smallest positive integer* \( k \), such that \( G \) is \( k \)-colourable.
Some observations about Graph colouring

- Note that any graph $G$ with $n$ vertices in $n$-colourable.

- The $n$-Clique, $K_n$, i.e., the complete graph on $n$ vertices, has chromatic number $\chi(K_n) = n$. All its vertices must get assigned different colours in any admissible colouring.

- The clique number, $\omega(G)$, of a graph $G$ is the maximum positive integer $r \geq 1$, such that $K_r$ is a subgraph of $G$.

- Note that for all graphs $G$, $\omega(G) \leq \chi(G)$: if $G$ has an $r$-clique then it is not $(r - 1)$-colorable.

- However, in general, $\omega(G) \neq \chi(G)$. For instance, The 5-cycle, $C_5$, has $\omega(C_5) = 2 < \chi(C_5) = 3$. 
More observations about colouring

- As already mentioned, any bipartite graph is 2-colourable. Indeed, that is an equivalent definition of being bipartite.

- More generally, a graph $G$ is $k$-colourable precisely if it is $k$-partite, meaning its vertices can be partitioned into $k$ disjoint sets such that all edges of the graph are between nodes in different parts.
Algorithms/complexity of colouring graphs

To determine whether a $n$-vertex graph $G = (V, E)$ is $k$-colourable by “brute force”, we could try all possible colourings of $n$ nodes with $k$ colours.

**Difficulty:** There are $k^n$ such $k$-colouring functions $c : V \rightarrow [k]$.

**Question:** Is there an efficient (polynomial time) algorithm for determining whether a given graph $G$ is $k$-colourable?
To determine whether a $n$-vertex graph $G = (V, E)$ is $k$-colourable by “brute force”, we could try all possible colourings of $n$ nodes with $k$ colours.

**Difficulty:** There are $k^n$ such $k$-colouring functions $c : V \rightarrow [k]$.

**Question:** Is there an efficient (polynomial time) algorithm for determining whether a given graph $G$ is $k$-colourable?

**Answer:** No, no generally efficient (polynomial time) algorithm is known, and even the problem of determining whether a given graph is 3-colourable is **NP-complete**. (Even approximating the chromatic number of a given graph is NP-hard.)

In practice, there are heuristic algorithms that do obtain good colourings for many classes of graphs.
Applications of Graph Colouring (many)

Final Exam Scheduling

- There are \( n \) courses, \( \{1, \ldots, n\} \).

- Some courses have the same students registered for both, so their exams can’t be scheduled at the same time.

- Let \( G = (\{1, \ldots, n\}, E) \) be a graph such that \( \{i, j\} \in E \) if and only if \( i \neq j \) and courses \( i \) and \( j \) have a student in common.

- **Question:** What is the minimum number of exam time slots needed to schedule all \( n \) exams?

- **Answer:** This is precisely the chromatic number \( \chi(G) \) of \( G \).

  Furthermore, a \( k \)-colouring of \( G \) yields an *admissible schedule* of exams into \( k \) time slots, allowing all students to attend all their exams, as long as different “colors” are scheduled in disjoint time slots.