# Lecture 12

# Distance in Graphs

---

This lecture introduces the notion of a *weighted graph* and explains how some choices of weights permit us to define a notion of distance in a graph.

**Reading:**
The material in this lecture comes from Chapter 3 of

> Dieter Jungnickel (2013), *Graphs, Networks and Algorithms*, 4th edition, which is (available online via SpringerLink.

---

## 12.1 Adding weights to edges

The ideas and applications that will occupy us for the next few lectures involve both directed and undirected graphs and will include one of the most important applications in the course, which involves the scheduling of large complex projects. To begin with, we introduce the notion of *edge weights*.

**Definition 12.1.** *Given a graph $G(V, E)$, which may be either directed or undirected, we can associate* ***edge weights*** *with $G$ by specifying a function $w : E \to \mathbb{R}$. We will write $G(V, E, w)$ to denote the graph $G(V, E)$ with edge weights given by $w$ and we will call such a graph a* ***weighted graph****.*

We will write $w(a, b)$ to indicate the weight of the edge $e = (a, b)$ and if $G(V, E, w)$ is an undirected weighted graph we will require $w(a, b) = w(b, a)$ for all $(a, b) \in E$.

Note that Definition 12.1 allows the weights to be negative or zero. That's because, as we'll see soon, the weights can represent many things. If the vertices represent places, then we could define a weight function $w$ so that, for an edge $e = (a, b) \in E$, the weight $w(e)$ is:

- the distance from $a$ to $b$;
- the time it takes to travel from $a$ to $b$, in which case it may happen that $w(a, b) \neq w(b, a)$;

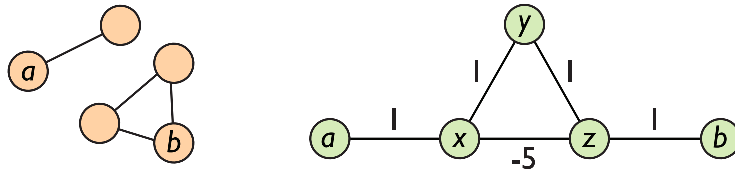


Figure 12.1: *In the graph at left there are no walks from $a$ to $b$ and so, by convention, we define $d(a,b) = \infty$. The graph at right, which has edge weights as indicated, illustrates a more serious problem. The cycle specified by the vertex sequence $(x, y, z, x)$ has negative weight and so there is no minimal-weight path from $a$ to $b$ and hence no well-defined distance $d(a,b)$.*

- the profit made when we send a shipping container from $a$ to $b$. This could easily be negative if we had to bring an empty container back from someplace we'd sent a shipment.

In any case, once we've defined weights for edges, it's natural to define the weight of a walk as follows.

**Definition 12.2.** *Given a weighted graph $G(V,E,w)$ and a walk from $a$ to $b$ defined by the vertex sequence*

$$a = v_0, \ldots, v_\ell = b,$$

*so that the its edges are $e_j = (v_{j-1}, v_j)$, then the* ***weight of the walk*** *is*

$$\sum_{j=1}^{\ell} w(e_j).$$

## 12.2 A notion of distance

Given two vertices $a$ and $b$ in a weighted graph $G(V,E,w)$, we might try to define a distance $d(a,b)$ from $a$ to $b$ as

$$d(a,b) = \min \{ w(\omega) \,|\, \omega \text{ is a walk from } a \text{ to } b \},$$

but two issues, both of which are illustrated in Figure 12.1 present themselves immediately:

(1) What if there aren't any walks from $a$ to $b$? In this case, by convention, we define $d(a,b) = \infty$.

(2) What if some cycle in $G$ has negative weight? As we will see below, this leads to insurmountable problems and so we'll just have to exclude this possibility.

The problem with cycles of negative weight is illustrated at the right in Figure 12.1. The graph has $V = \{a, x, y, z, b\}$ and edge weights

$$w(a,x) = w(x,y) = w(y,z) = w(z,b) = 1 \qquad \text{and} \qquad w(x,z) = -5.$$

The cycle specified by the vertex sequence $(z, x, y, z)$ thus has weight

$$w(z, x) + w(x, y) + w(y, z) = -5 + 1 + 1 = -3$$

and one can see that this presents a problem for our definition of $d(a, b)$ by considering the sequence of walks:

| *Walk* | *Weight* |
|---|---|
| $(a, x, y, z, b)$ | $1 + 1 + 1 + 1 = 4$ |
| $(a, x, y, z, x, y, z, b)$ | $4 + (-5 + 1 + 1) = 1$ |
| $(a, x, y, z, x, y, z, x, y, z, b)$ | $4 - 2 \times 3 = -2$ |
| $\vdots$ | |
| $(a, x, y, z, \underbrace{x, y, z, \ldots, x, y, z}_{k \text{ times around the cycle}}, b)$ | $4 - k \times 3 = 4 - 3k.$ |

There *is* no walk of minimal weight from $a$ to $b$: one can always find a walk of lower weight by tracing over the negative-weight cycle a few more times. We could escape this problem by defining $d(a, b)$ as the weight of a minimal-weight *path*[1], but instead we will exclude the problematic cases explicitly:

**Definition 12.3.** *Suppose $G(V, E, w)$ is a weighted graph that does not contain any cycles of negative weight. For vertices $a$ and $b$ we define the* ***distance function*** *$d : V \times V \to \mathbb{R}$ as follows:*

- *$d(a, a) = 0$ for all $a \in V$;*
- *$d(a, b) = \infty$ if there is no walk from $a$ to $b$;*
- *$d(a, b)$ is the weight of a minimal-weight walk from $a$ to $b$ when such walks exist.*

### A warning

The word "distance" in Definition 12.3 is potentially misleading in that it is perfectly possible to find weighted graphs in which $d(a, b) < 0$ for some (or even all) $a$ and $b$. Further, it's possible that in a directed graph there may be vertices $a$ and $b$ such that $d(a, b) \neq d(b, a)$. If we want our distance function to have the all the properties that the word "distance" normally suggests, it's helpful to recall (or learn for the first time) the definition of a *metric on a set $X$*. It's a function $d : X \times X \to \mathbb{R}$ with the following properties:

**Non-negativity** $d(x, y) \geq 0 \ \forall x, y \in X$ and $d(x, y) = 0 \iff x = y$;

**symmetry** $d(x, y) = d(y, x) \ \forall x, y \in X$;

**triangle inequality** $d(x, y) + d(y, z) \geq d(x, z) \ \forall x, y, z \in X$.

If $d$ is a metric on $X$ we say that the pair $(X, d)$ constitute a *metric space*. It's not hard to prove (see the Problem Sets) that if $G(V, E, w)$ is a weighted, undirected graph in which $w(e) > 0 \ \forall e \in E$, then the function $d : V \times V \to \mathbb{R}$ from Definition 12.3 is a metric on the vertex set $V$.

[1] A path cannot revisit a vertex and hence cannot trace over a cycle.

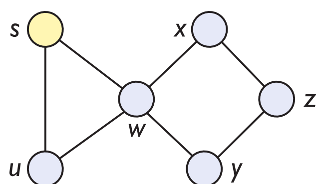


Figure 12.2: *A SSSP problem in which all the edge weights are 1 and the source vertex s is shown in yellow.*

## 12.3 Shortest path problems

Once one has a definition for distance in a weighted graph $G(V, E, w)$, two natural problems present themselves:

**Single-Source Shortest Path (SSSP):** Given a vertex $s$—the so-called *source vertex*—compute $d(s, v)\ \forall v \in V$.

**All-Pairs Shortest Path:** Compute $d(u, v)\ \forall u, v \in V$.

We will develop algorithms to solve the first of these, but not the second. Of course, if one has an algorithm for SSSP, one can also solve the second by applying the SSSP algorithm with each vertex as the source, though there are more efficient approaches as well.

### 12.3.1 Uniform weights & Breadth First Search

The simplest SSSP problems are those in undirected weighted graphs where all edges have the same weight, say, $w(e) = 1\ \forall e \in E$. In this case one can use an algorithm called *Breadth First Search (BFS)*, which is one of the fundamental tools of algorithmic graph theory. I'll present the algorithm twice, once informally, by way of an example, and then again in a sufficiently detailled way that one could, for example, implement it in MATLAB. As we're working on a single-source problem, it's convenient to define

$$d(v) \equiv d(s, v),$$

where $s$ is the source vertex. Our goal is then to compute $d(v)$ for all vertices in the graph.

To illustrate the main ideas, we'll use BFS to compute $d(v)$ for all the vertices in the graph pictured in Figure 12.2:

- Set $d(s) = 0$.
- Set $d(v) = 1$ for all $s$'s neighbours. That is, set $d(v) = 1$ for all vertices $v \in A_s = \{u, w\}$.
- Set $d(v) = 2$ for those vertices that (a) are adjacent to vertices $t$ with $d(t) = 1$ and (b) have not yet had a values of $d(v)$ assigned.

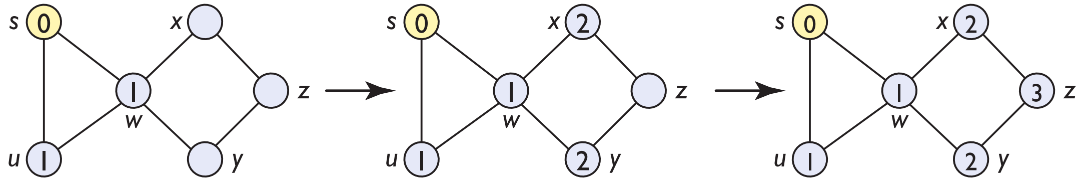


Figure 12.3: *The leftmost graph shows the result of the first two stages of the informal BFS algorithm: we set $d(s) = 0$ and and $d(v) = 1$ for all $v \in A_s$. In the second stage we set $d(v) = 2$ for neighbours of vertices $t$ with $d(t) = 1 \cdots$ and so on.*

- Set $d(v) = 3$ for those vertices that (a) are adjacent to vertices $t$ with $d(t) = 2$ and (b) have not yet had a values of $d(v)$ assigned.

This process is illustrated in Figure 12.3 and, for the current example, these four steps assign a value of $d(v)$ to every vertex. In Section 12.4 we will return to this algorithm and rewrite it in a more general way, but I'd like to conclude this section by discussing why this approach works.

### 12.3.2 Bellman's equations

BFS, and indeed all the shortest-path algorithms we'll study, work because of a characterisation of minimum-weight walks due to Richard Bellman. Suppose $G(V, E, w)$ is a weighted graph (either directed or undirected) on $|V| = n$ vertices. Specify a single-source shortest path problem by numbering the vertices so that the source vertex $s$ comes first, $v_1 = s$, and assemble the edge weights into an $n \times n$ matrix $w$ whose entries are given by

$$w_{k,j} = \begin{cases} w(v_k, v_j) & \text{if } (v_k, v_j) \in E \\ \infty & \text{otherwise} \end{cases}$$

Then we have the following theorem, which captures the idea that a minimal-weight path from $v_1$ to $v_j$ consists of a minimal-weight path from $v_1$ to one of $v_j$'s neighbours—say $v_k$ such that $(v_k, v_j) \in E$—followed by a final step from $v_k$ to $v_j$.

**Theorem 12.4** (Bellman's equations)**.** *The quantities $u_j = d(v_1, v_j)$ satisfy the equations*

$$u_1 = 0 \qquad \text{and} \qquad u_j = \min_{k \neq j}(u_k + w_{k,j}) \qquad \text{for } 2 \leq j \leq n. \tag{12.1}$$

*Further, if all cycles in $G(V, E, w)$ have positive weight, then the equations (12.1) have a unique solution.*

## 12.4 Appendix: BFS revisited

The last two steps of our informal introduction to BFS had the general form

- Set $d(v) = j + 1$ for those vertices that (a) are adjacent to vertices $t$ with $d(t) = j$ and (b) have not yet had a values of $d(v)$ assigned.

The main technical problem in formalising the algorithm is to find a systematic way of working our way outward from the source vertex. A data structure from computer science called a *queue* provides an elegant solution. It's an ordered list that we'll write from left-to-right, so that a queue containing vertices might look like

$$Q = \{x, z, u, w, \ldots, a\},$$

where $x$ is the first entry in the queue and $a$ is the last.

Just as in a well-behaved bus or bakery queue, we "serve" the vertices in order (left to right) and require that any new items added to the queue get added at the end (the right). There are two operations that one can perform on a queue:

**push:** add a new entry onto the end of the queue (i.e. at the right);

**pop:** remove the first (i.e. leftmost) entry and, typically, do something with it.

Thus if our queue is $Q = \{b, c, a\}$ and we push a vertex $d$ onto it the result is

$$\{b, c, a\} \xrightarrow{\text{push } x \text{ onto } Q} \{b, c, a, x\},$$

while if we pop the queue we get

$$\{b, c, a\} \xrightarrow{\text{pop } Q} \{c, a\}.$$

We can use this idea to organise the order in which we visit the vertices in BFS. Our goal will be to compute $d(v) = d(s, v)$ for all vertices in the graph and we'll start by setting $d(s) = 0$ and

$$d(v) = \infty \ \forall v \neq s.$$

This has two advantages: first, $d(v) = \infty$ is the correct value for any vertex that is not reachable from $s$ and second, it serves as a way to indicate that, as far as our algorithm is concerned, we have yet to visit vertex $v$ and so $d(v)$ is yet-to-be-determined.

We'll then work our way through the vertices that lie in the same connected component as $s$ by

- pushing a vertex $v$ onto the end of the queue whenever we set $d(v)$, beginning with the source vertex $s$ and
- popping vertices off the queue in turn, working through the adjacency list of the popped vertex $u$ and examining its neighbours $w \in A_u$ in turn, setting $d(w) = d(u) + 1$ whenever $d(w)$ is currently marked as yet-to-be-determined.

**Algorithm 12.5** (BFS for SSSP). *Given an undirected graph $G(V, E)$ and a distinguished source vertex $s \in V$, assume uniform edge weights $w(e) = 1\ \forall e \in E$ and find the distances $d(v) = d(s, v)$ for all $v \in V$.*

(1) *Set things up:*

| | |
|---|---|
| $d(v) \leftarrow \infty \quad \forall v \neq s \in V$ | *Set $d(v) = \infty$ to indicate that it is yet-to-be-determined* |
| $d(s) \leftarrow 0$ | *Note that we know $d(s, s) = 0$* |
| $Q \leftarrow \{s\}$ | *Get ready to process s's neighbours* |

(2) *Main loop: continues until the queue is empty*

```
While ( Q ≠ ∅ ) {
        Pop a vertex u off the left end of Q.

        Examine each of u's neighbours
        For each  w ∈ A_u {
            If(  d(w) = ∞  ) then {
                Set d(w) and get ready to process w's neighbours
                d(w) ← d(u) + 1
                Push w onto the right end of Q.
            }
        }
}
```

Figure 12.4 illustrates the early stages of applying the algorithm to a small graph, while Table 12.1 provides a complete account of the computation: a set of PowerPoint-like slides illustrating this computation is available from the course web page.

**Remarks**

- If $d(u) = \infty$ when the algorithm finishes, then $u$ and $s$ lie in separate connected components.
- Because the computation works through adjacency lists, each edge gets considered at most twice and so the algorithm requires $O(|E|)$ steps, where a step consists of checking whether $d(u) = \infty$ and, if so, updating its value.
- It is possible to prove by induction on the lengths of the shortest paths, that BFS really does compute the distance $d(s, v)$: interested readers should see Jungnickel's Theorem 3.3.2.

| $u$ | $w \in A_u$ | Action | Resulting Queue |
|---|---|---|---|
| – | – | *Start* | $\{S\}$ |
| S | A | set $d(A) = 1$ and push $A$ | $\{A\}$ |
| S | C | set $d(C) = 1$ and push $C$ | $\{A, C\}$ |
| S | G | set $d(G) = 1$ and push $G$ | $\{A, C, G\}$ |
| A | B | set $d(B) = 2$ and push $B$ | $\{C, G, B\}$ |
| A | S | none, as $d(S) = 0$ | $\{C, G, B\}$ |
| C | D | set $d(D) = 2$ and push $D$ | $\{G, B, D\}$ |
| C | E | set $d(E) = 2$ and push $E$ | $\{G, B, D, E\}$ |
| C | F | set $d(F) = 2$ and push $F$ | $\{G, B, D, E, F\}$ |
| C | S | none | $\{G, B, D, E, F\}$ |
| G | F | none | $\{B, D, E, F\}$ |
| G | H | set $d(H) = 2$ and push $H$ | $\{B, D, E, F, H\}$ |
| G | S | none | $\{B, D, E, F, H\}$ |
| B | A | none | $\{D, E, F, H\}$ |
| D | C | none | $\{E, F, H\}$ |
| E | C | none | $\{F, H\}$ |
| E | H | none | $\{F, H\}$ |
| F | C | none | $\{H\}$ |
| F | G | none | $\{H\}$ |
| H | E | none | $\{\}$ |
| H | G | none | $\{\}$ |

Table 12.1: *A complete record of the execution of the BFS algorithm for the graph in Figure 12.4. Each row corresponds to one pass through the innermost loop of Algorithm 12.5, those steps that check whether* $d(w) = \infty$ *and act accordingly. The table is divided into sections—separated by horizontal lines—within which the algorithm works through the adjacency list of the most recently-popped vertex* $u$*.*

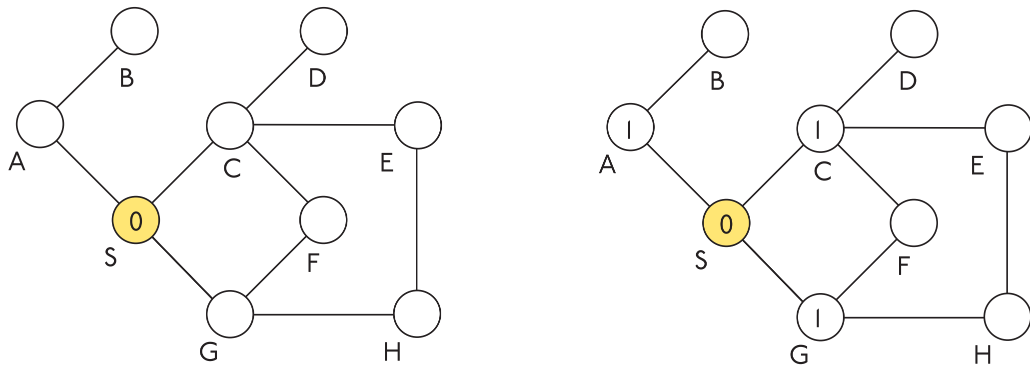


Figure 12.4: *In the graphs above the source vertex $s$ is shown in yellow and a vertex $v$ is shown with a number on it if, at that stage in the algorithm, $d(v)$ has been determined (that is, if $d(v) \neq \infty$). The graph at left illustrates the state of the computation just after the initialisation: $d(s)$ has been set to $d(s) = 0$, all other vertices have $d(v) = \infty$ and the queue is $Q = \{s\}$. The graph at right shows the state of the computation after we have popped $s$ and processed its neighbours: $d(v)$ has been determined for $A$, $C$ and $G$ and they have been pushed onto the queue, which is now $Q = \{A, C, G\}$.*