Mario Leston

Algoritmos em grafos

Contents

1 Shortest paths

1.1 Introduction

To begin, let us define our main character. A **simple graph** G is an ordered pair (V, E), where V is a finite set whose elements are called **vertices**, and E is a finite set of unordered pairs of distinct elements of V; the elements of E are called **edges**.[1] The vertex set of a simple graph G is often denoted by $V(G)$, and its edge set by $E(G)$. Moreover, we write uv instead of the longer $\{u, v\}$, usually when $\{u, v\}$ is an edge of G.

[1] The definition permits graphs with an empty vertex set—**empty** graphs. Although they can complicate statements by furnishing counterexamples, they are indispensable; we will keep this in mind and try not to forget that they may be a burden.

As the following definitions illustrate, many graph-theoretic concepts are highly intuitive and formalize everyday relationships among real-world entities. This helps explain their appeal and their wide range of practical applications.

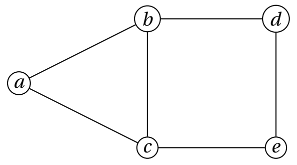


Figure 1.1: A simple graph with set of vertices $\{a, b, c, d, e\}$ and with set of edges $\{ab, ac, bc, bd, ce, de\}$.

Neighborhood of a vertex. For a simple graph G, vertices u and v of G are **neighbors** (or **adjacent**) if $uv \in E(G)$. We also say that v is a **neighbor** of u in G. The **(open) neighborhood** of u is the set

$$N_G(u) := \{v \in V(G) \mid uv \in E(G)\},$$

and we write $N(u)$ when G is clear from context. So N_G is a function that takes a vertex as input and returns a set of vertices as output, that is,[2] $N : V \to 2^V$.

[2] For a set A, 2^A is the set of all subsets of A.

Neighborhood of a set of vertices. Let G be a simple graph. We define $\widehat{N} : 2^V \to 2^V$ by setting

$$\widehat{N}(X) := \{y \in V \setminus X \mid \exists x \in X : y \in N(x)\}$$

for each $X \in 2^V$. Notice that $\widehat{N}(\{u\}) = N(u)$ for each vertex u of G. We will occasionally abuse notation and not distinguish between the functions N and $\widehat{N}$, whenever this causes no confusion.

1.2 *Cut and degree of a vertex*

Let u be a vertex of a simple graph G. The **cut** of u, denoted $\delta_G(u)$, is the set of edges of G that are incident with u. Formally,

$$\delta_G(u) := \{e \in E \mid u \in e\}.$$

The notation $\delta_G(u)$ is somewhat heavy; when no confusion can arise, we will simply write $\delta(u)$, or even just

$$\delta u.$$

The size (cardinality) of δu, denoted $\deg(u)$, is called the **degree** of u. Notice that (for simple graphs) $\deg(u) = |N(u)|$.

The next two facts are among the first emphasized in introductions to Graph Theory, and we will keep to that tradition. We also take the opportunity to introduce a piece of notation that will be useful.

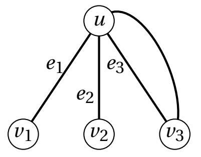


Figure 1.2: The figure shows the vertex u with $\delta u = \{e_1, e_2, e_3\}$. Thus $\deg(u) = 3$.

Let U be a *universe* set and let X be a subset of U, that is, $X \subseteq U$. The **characteristic function of** X **relative to** U is the function

$$\mathbb{1}_U^X : U \to \{0,1\}$$

such that

$$\mathbb{1}_U^X(u) = \begin{cases} 1, & \text{if } u \in X, \\ 0, & \text{if } u \notin X, \end{cases}$$

for each $u \in U$. Note that $|X| = \sum_{u \in U} \mathbb{1}_U^X(u)$. For brevity we will write $\mathbb{1}^X$ instead of its longer version, leaving the relevant universe implicit. In the lemma below we adopt this convention and write $\mathbb{1}^{\delta u}$ in place of $\mathbb{1}_E^{\delta u}$ for each vertex u.

Lemma 1.1. *If G is a simple graph, then* $\sum_{v \in V} \deg(v) = 2|E|$.

Proof. Suppose G is a simple graph. First observe that

$$\deg(v) = \sum_{e \in E} \mathbb{1}^{\delta v}(e)$$

for each $v \in V$. Moreover,

$$\sum_{v \in V} \mathbb{1}^{\delta v}(e) = 2$$

for each $e \in E$. Therefore,

$$\sum_{v \in V} \deg(v) = \sum_{v \in V} \sum_{e \in E} \mathbb{1}^{\delta v}(e) = \sum_{e \in E} \sum_{v \in V} \mathbb{1}^{\delta v}(e) = \sum_{e \in E} 2 = 2|E|.$$

□

Corollary 1.1. *Every simple graph has an even number of odd-degree vertices.*

Proof. Let G be a simple graph, let P be the set of even-degree vertices of G, and let I be the set of odd-degree vertices of G. Clearly

$$\sum_{v\in V}\deg(v)=\sum_{v\in P}\deg(v)+\sum_{v\in I}\deg(v).$$

By Lemma 1.1, $\sum_{v\in V}\deg(v)$ is even, and $\sum_{v\in P}\deg(v)$ is also even. It follows that $\sum_{v\in I}\deg(v)$ is even and therefore $|I|$ is even. □

Exercise 1.1. Let G be a simple graph and $X \subseteq V$. Prove that

$$\widehat{N}(X)=\Big(\bigcup_{x\in X}N(x)\Big)\setminus X.$$

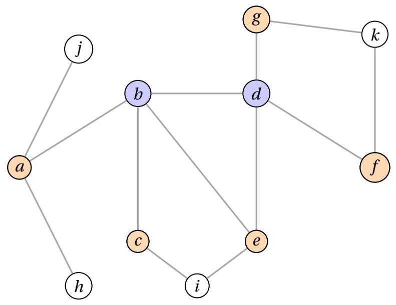


Figure 1.3: The set of vertices in orange is the neighborhood, $\widehat{N}(\{b,d\})$ of the set $\{b,d\}$ in blue.

Path. For each $n \geqslant 1$, a **path** P of **length** $|P| := n-1$ in a simple graph G is a sequence

$$\langle u_0,u_1,\ldots,u_{n-1}\rangle$$

of vertices of G such that

$$u_{i-1}u_i \in E(G) \quad \text{for each } i \in [1:n).$$

The **set of edges** of P, denoted $E(P)$, is

$$\{u_{i-1}u_i \mid i \in [1:n)\}.$$

The **endpoints** of P are u_0 and u_{n-1}; we say that P is a path **joining** u_0 and u_{n-1}, or, more simply, an (u_0,u_{n-1})-path.[3] When all vertices $u_0,\ldots,u_{n-1}$ are distinct, we call P a **simple path**. For $i,k \in [n)$ with $i \leqslant k$, we write $P(u_i,u_k)$ for the path

$$\langle u_i,u_{i+1},\ldots,u_k\rangle.$$

[3] We will also use these alternative phrasings interchangeably in what follows (e.g., "a path from u_0 to u_{n-1}," "a path beginning at u_0 and ending at u_{n-1}," and "an (u_0,u_{n-1})-path").

[3] For each $m,n \in \mathbb{Z}$:

$$\begin{aligned}[m:n] &:= \{k\in\mathbb{Z} \mid m \leqslant k \leqslant n\},\\ [m:n) &:= \{k\in\mathbb{Z} \mid m \leqslant k < n\},\\ (m:n] &:= \{k\in\mathbb{Z} \mid m < k \leqslant n\},\\ (m:n) &:= \{k\in\mathbb{Z} \mid m < k < n\},\\ [n) &:= [0:n).\end{aligned}$$

Exercise 1.2. Let G be a simple graph and u,v vertices of G. Prove that if G has an (u,v)-path, then G also has a simple (u,v)-path.

Shortest path. We now turn to the main notion. Set[4] $\min\varnothing := \infty$. Let G be a simple graph. For each $s,t \in V(G)$, define

$$\text{dist}_G(s,t) := \min\{|P| \mid P \text{ is an } (s,t)\text{-path in } G\}. \tag{1.1}$$

[4] Here, ∞ is an element adjoined to $\mathbb{R}$ with the property that $r \leqslant \infty$ for all $r \in \mathbb{R}$.

If there is no (s,t)-path in G, then the set on the right-hand side of (1.1) is empty and the minimum is ∞; conversely, if $\text{dist}(s,t)=\infty$, then there is no (s,t)-path in G. When $\text{dist}(s,t)<\infty$, any (s,t)-path P with $|P|=\text{dist}(s,t)$ is called a **shortest** (s,t)-path.

Exercise 1.3. Let G be a simple graph, and let $u, v \in V(G)$. Prove that if P is a shortest (u, v)-path, then P is simple.

Exercise 1.4. Let G be a graph and $r \in V$. Show that

$$|\mathsf{dist}(r,u) - \mathsf{dist}(r,v)| \leqslant 1$$

for all $uv \in E$.

Shortest path problem. Our goal now is to develop an algorithm for the single–source shortest path problem: given a simple graph G and a fixed source vertex $r \in V(G)$, compute, for each $s \in V(G)$, a shortest (r, s)-path (if one exists).

1.3 Structure of shortest paths

Let G be a simple graph and fix a vertex $r \in V(G)$. For each $i \in \mathbb{N}$, the **slice of level** i of G with respect to r is the set

$$R^i := \{u \in V \mid \mathsf{dist}(r,u) = i\};$$

we also set

$$R^{\leqslant i} := \{u \in V \mid \mathsf{dist}(r,u) \leqslant i\}$$

We claim[5] that for each $i \in \mathbb{N}$,

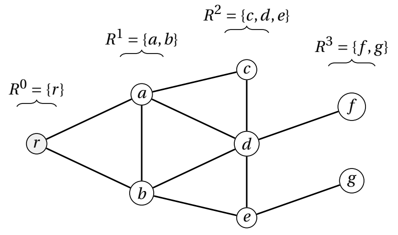
$$R^{i+1} = N(R^i) \setminus R^{\leqslant i}. \tag{1.2}$$

5 For sets X and Y, we denote by $X \setminus Y$ the set of all $x \in X$ such that $x \notin Y$.

Let us first prove that $R^{i+1} \subseteq N(R^i) \setminus R^{\leqslant i}$. Suppose $u \in R^{i+1}$, i.e., $\mathsf{dist}(r,u) = i+1$, and let P be a shortest (r,u)-path (so $|P| = i+1$). Let s be the predecessor of u on P. The (r,s)-prefix of P has length i, hence $\mathsf{dist}(r,s) \leqslant i$. If $\mathsf{dist}(r,s) < i$, then $P \cdot u$ is an (r,u)-path of length $< i+1$, a contradiction. Therefore $\mathsf{dist}(r,s) = i$, that is, $s \in R^i$, so $u \in N(R^i)$. Since $\mathsf{dist}(r,u) = i+1$, we also have $u \notin R^{\leqslant i}$. Hence $u \in N(R^i) \setminus R^{\leqslant i}$.

Let us now prove that $N(R^i) \setminus R^{\leqslant i} \subseteq R^{i+1}$. Suppose $u \in N(R^i) \setminus R^{\leqslant i}$. Since $u \notin R^{\leqslant i}$, we have $\mathsf{dist}(r,u) \geqslant i+1$. Let $s \in R^i$ be such that $u \in N(s)$. Now, $s \in R^i$ implies that there exists an (r,s)-path P with $|P| = i$. Hence[6] $P \cdot u$ is an (r,u)-path of length $i+1$, and thus $\mathsf{dist}(r,u) \leqslant i+1$, which, combined with $\mathsf{dist}(r,u) \geqslant i+1$, yields $\mathsf{dist}(r,u) = i+1$. Therefore, $u \in R^{i+1}$.

6 For sequences P and Q, we write $P \cdot Q$ for their concatenation; here, $P \cdot u$ abbreviates $P \cdot \langle u \rangle$.



$\underbrace{R^0 = \{r\}}$
$\underbrace{R^1 = \{a, b\}}$
$\underbrace{R^2 = \{c, d, e\}}$
$\underbrace{R^3 = \{f, g\}}$
r
a
b
c
d
e
f
g

Figure 1.4: Slices (nonempty) with respect to r.

This establishes (1.2).

It is clear that $R^0 = \{r\}$. Moreover, equation (1.2) allows us to compute a solution to the shortest–path problem. To make this precise, we first fix a compact representation of paths.

Let $\bot \notin V$ and consider the set $V \rightharpoonup V \cup \{\bot\}$ of partial functions.[7] The set of functions that **encode** r**-paths** (via predecessor pointers) is the subset $\mathscr{P}$ of $V \rightharpoonup V \cup \{\bot\}$ inductively defined by the following rules:

7 Let A and B be sets. A **partial function** from A to B is a subset f of $A \times B$ such that for $a \in A$ and $b, b' \in B$, if $(a,b), (a,b') \in f$, then $b = b'$. The set of all partial function from A to B is denoted by $A \rightharpoonup B$. As usual, we write $f : A \rightharpoonup B$ instead of $f \in A \rightharpoonup B$.

For a subset R of $A \times B$, we write $\mathtt{dom}(R)$ to denote the set of all $a \in A$ for which there exists $b \in B$ with $(a,b) \in R$.

- $\{(r,\bot)\} \in \mathscr{P}$;
- if $p \in \mathscr{P}$, $u \in V \setminus \texttt{dom}(p)$, and $t \in \texttt{dom}(p) \cap N(u)$, then $p \cup \{(u,t)\} \in \mathscr{P}$.

If $p \in \mathscr{P}$ and $u \in \texttt{dom}(p)$, then $p(u)$ is called the **predecessor** of u in p.

Each $p \in \mathscr{P}$ encodes, for every $u \in \texttt{dom}\, p$, an (r,u)-path obtained by following predecessors u, $p(u)$, $p^2(u), \ldots, r$; the chain terminates because $p(r) = \bot$. This can be formally defined as follows. Consider the function

$$p^* : \texttt{dom}(p) \to V^*$$

(where V^* is the set of finite sequences over V) given by:[8]

(i) $p^*(r) := \langle r \rangle$, and
(ii) $p^*(u) := p^*(p(u)) \cdot \langle u \rangle$ for each $u \in \texttt{dom}\, p \setminus \{r\}$.

[8] Why is this well-defined?

Exercise 1.5. Prove that if p encodes r-paths, then $p^*(u)$ is an (r,u)-path for each $u \in \texttt{dom}(p)$.

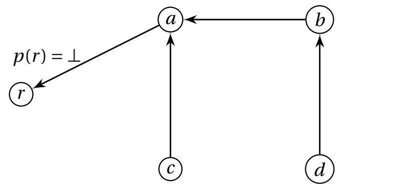
Exercise 1.6. Design an algorithm that, given a function p that encodes r-paths of some simple graph G for some vertex r of G, and a vertex $s \in \texttt{dom}(p)$, returns $p^*(s)$. Provide implementations in `Python` and `Clojure`.

Exercise 1.7. Design an algorithm that, given a function p that encodes r-paths of some simple graph G for some vertex r of G, returns a vertex $u \in \texttt{dom}(p)$ for which $|p^*(u)|$ is maximum. Provide implementations in `Python` and `Clojure`.

Exercise 1.8. Let p a function that encodes r-paths of some simple graph G for some vertex r of G. A **leaf** of p is a vertex $u \in \texttt{dom}(p)$ such that u is not the predecessor of any other vertex of $\texttt{dom}(p)$, that is, $p(v) \neq u$ for all $v \in \texttt{dom}(p)$. Design an algorithm that, given p, returns the set of all leaves of p. Provide implementations in `Python` and `Clojure`.

Exercise 1.9. Design an algorithm that, given a simple graph G, a vertex r of G, and a partial function $p : V \rightharpoonup V \cup \{\bot\}$, decides whether p encodes r-paths. Provide implementations in `Python` and `Clojure`.

We are now ready to present an algorithm for computing shortest paths. First, we fix how we will represent graphs computationally in *pseudocode* (and later in code). For our purposes it is convenient to represent a simple graph G by its neighborhood function $N : V \to 2^V$. Although we write G as a formal parameter of the algorithm, under these assumptions the only object we may query is the function N. We also assume access to $\texttt{dom}(N)$ so that, when needed, we can obtain the vertex set of G.



$p = \{(r,\bot),(a,r),(b,a),(c,a),(d,b)\}$
$p^*(r) = \langle r \rangle$, $p^*(a) = \langle r,a \rangle$ $p^*(b) = \langle r,a,b \rangle$
$p^*(d) = \langle r,a,b,d \rangle$, $p^*(c) = \langle r,a,c \rangle$

Figure 1.5: An arrow from vertex v to vertex u indicates that $p(v) = u$.

Single-source shortest paths (unweighted)

*The next algorithm receives a simple graph G, given by its neighborhood function N, and a vertex r of G. It returns an encoding of r-paths $p : V \rightharpoonup V \cup \{\perp\}$, which we call a **shortest r-path function**, such that* $\mathtt{dom}(p) = \{u \in V \mid \mathtt{dist}(r,u) < \infty\}$ *and $p^*(u)$ is a shortest path from r to u for all $u \in$* $\mathtt{dom}(p)$.

```
def mlp(G, r):
    $p, F, k \leftarrow \{(r,\perp)\}, \{r\}, 0$
    while $F \neq \varnothing$:
        $F' \leftarrow \varnothing$
        for $s \in F, u \in N(s)$:
            if $u \notin \mathtt{dom}(p)$:
                $F' \leftarrow F' \cup \{u\}$
                $p \leftarrow p \cup \{(u,s)\}$
        $F \leftarrow F'$
        $k \leftarrow k + 1$
    return $p$
```

Loop invariant. The following invariant holds at the beginning of each iteration:

(1.3) (i) p is a partial function that encodes r-paths;
(ii) $F = R^k$ and $\mathtt{dom}(p) = R^{\leq k}$; and
(iii) $p^*(u)$ is a shortest (r,u)-path for every $u \in \mathtt{dom}(p)$.

Exercise 1.10. Prove the invariant.

Exercise 1.11. Prove that the algorithm stops.

Exercise 1.12. Prove that the time complexity of `mlp`(G, r) is $\mathcal{O}(|V| + |E|)$.

Convention. When discussing implementations of algorithms, we adopt the following convention. Algorithms in pseudocode operate on mathematical entities, whereas implementations operate on representations of those entities. To streamline the exposition, we use a typographical convention: representations are set in `typewriter` font, and the corresponding mathematical entities are set in math font. Thus, if we say that `G` represents a graph, we denote that graph by G.

A simple `Python` *implementation.* We represent a simple graph G by a dictionary `G`, called an **adjacency-list** for G, whose keys are vertex representations and whose values are iterables of their neighbors. Thus, for the representation `u` of a vertex $u \in V$, the entry `G[u]` is a sequence whose elements are representations of the vertices in $N(u)$. The encoding of r-paths is likewise stored as a dictionary.

```python
def mlp(G, r):
    p, F = {r: None}, [r]
    while F:
        Fl = []
        for r in F:
            for u in G[r]:
                if u not in p:
                    Fl.append(u)
                    p[u] = r
        F = Fl
    return p
```

To avoid nested loops, we introduce a helper function `edges(G,F)`: given an adjacency-list representation `G`of a graph and an iterable `F` of representations of vertices, it returns a generator over all pairs `(s, u)` with `s in F` and `u in G[s]`.

```python
def edges(G, F):
    for s in F:
        for u in G[s]:
            yield s, u

def mlp(G, r):
    p, F = {r: None}, [r]
    while F:
        Fl = []
        for s, u in edges(G, F):
            if u not in p:
                Fl.append(u)
                p[u] = s
        F = Fl
    return p
```

A simple `Clojure` *implementation.* Next we present a simple `Clojure` implementation of algorithm `mlp`. A few comments are in order. We represent a simple graph G as a (`Clojure`) map `G` — corresponding to a dictionary in `Python` — whose keys are vertex representations and whose values are sequences of (representations of) their neighbors. We also call it an **adjacency-list** representaion. Thus, for the representation `u` of a vertex $u \in V$, the entry `(G u)` is a sequence whose elements are representations of the vertices in $N(u)$. The encoding of r-paths is likewise a map.

The function receives an adjacency-list representation `G` of a graph G and a representation `r` of a vertex of G. Assume `F` is a list whose elements are representations of the vertices of G at some level i (thus the elements in `F` represent R^i), and `p` is a representation of an encoding of r-paths. The expression

```
(for [s F, u (G s) :when (not (contains? p u))] [u s])
```

generates a sequence of pairs `[u s]` with $u \in R^{i+1}$ and $s \in R^i$. If two elements of this sequence, say `[u s]` and `[v t]`, share the same first component (so `u` and `v` are equal), then $p^*(s)$ and $p^*(t)$ have the same length, and therefore $p^*(s) \cdot \langle u \rangle$ and $p^*(t) \cdot \langle u \rangle$ also have the same length. By construction, in this case the map `q` retains the pair that occurs last.[9] Finally, observe that `F'` is a list whose elements are representations of the vertices in R^{i+1}.

[9] In Clojure, when building a map by successively associating the same key, the last association takes precedence.

```
(defn mlp [G r]
  (loop [F (list r), p {r nil}]
    (if (empty? F)
      p
      (let [q (into {}
                    (for [s F, u (G s)
                          :when (not (contains? p u))] [u s]))
            F' (map first q)
            p' (into p q)]
        (recur F' p')))))
```

Exercise 1.13. Provide an algorithm (and its `Python` and `Clojure` implementation) which receives a simple graph G and one of its vertices r and returns both a shortest r-path function p and a function $\ell : \mathtt{dom}(p) \to \mathbb{N}$ for which $\ell(u) = |p^*(u)|$ for all $u \in \mathtt{dom}(p)$.

Exercise 1.14. Design an algorithm that, given a nonempty simple graph G and a vertex r of G, returns the level partition $\{R^0, R^1, \ldots, R^h\}$ of G with respct to r where $h = \max\{\mathsf{dist}(r,u) \mid u \in V(G), \mathsf{dist}(r,u) < \infty\}$. Analyze its running time. Provide implementations in `Python` and `Clojure`.

1.4 Breadth-first search

Breadth-first search (BFS) is based on the following idea: informally, it can be viewed as a "lazy" version of `mlp`. Recall that `mlp` explores a simple graph G from a starting vertex r level by level. At the beginning of each iteration it maintains a set containing all vertices at some level i and replaces it with the set of vertices at level $i+1$. In BFS, instead, we maintain a sequence (a queue) that contains all vertices at level i and possibly some vertices at level $i+1$. Each iteration removes a vertex of level i from this sequence (it could be the front element of the sequence) and inspects its neighbors; any neighbor not seen before is appended to the end. This guarantees that all vertices at level i are processed before any vertex at level $i+1$.

For the BFS algorithm, we introduce some notation. Let X be a set and let $Q \in X^*$. Thus $Q : [n] \to X$ for some $n \in \mathbb{N}$, called the **length** of Q. When $n = 0$, Q is the **empty sequence**, denoted by ϵ. If $n \geqslant 1$, the tail of Q, denoted Q_{-0}, is the sequence obtained from Q by deleting its first element. Formally, $Q_{-0} : [n-1] \to X$ is defined by $(Q_{-0})_{(i)} := Q_{(i+1)}$ for each $i \in [n-1]$. Finally, for sequences $Q, Q' \in X^*$, let $Q \cdot Q'$ denote their concatenation, i.e., the sequence in X^* obtained by appending Q' to Q. We abuse notation and write $Q \cdot T$ when $T \subseteq X$. Since we work only with finite sets, we assume a fixed default enumeration of T, thereby producing a sequence in X^*. By $Q \cdot T$ we mean the concatenation of Q with that sequence.

Exercise 1.15. Provide a formal definition of concatenation.

Breadth-first search

The next algorithm receives a simple graph G, given by its neighborhood function N, and a vertex r of G. It returns an encoding of r-paths $p : V \rightharpoonup V \cup \{\bot\}$ such that $\mathtt{dom}(p) = \{u \in V \mid \mathsf{dist}(r,u) < \infty\}$ and $p^*(u)$ is a shortest path from r to u for all $u \in \mathtt{dom}(p)$.

```
def bfs(G, r):
    $Q, p \leftarrow \langle r \rangle, \{(r, \bot)\}$
    while $Q \neq \epsilon$:
        $s \leftarrow Q_{(0)}$
        $T \leftarrow N(s) \setminus \mathtt{dom}(p)$
        $p \leftarrow p \cup \{(t,s) \mid t \in T\}$
        $Q \leftarrow Q_{-0} \cdot T$
    return $p$
```

An attentive reader will notice that, in algorithm `bfs`, the sequence Q behaves like a FIFO queue: (1) only the first element is examined, (2) removals occur at the front, and (3) insertions occur at the back.

The following is a somewhat “functional” version of the algorithm, more amenable to a correctness proof than its “imperative” counterpart.

```
def bfs(G, r):
    def loop(Q, p):
        if $Q = \epsilon$: return $p$
        $T := N(s) \setminus \mathtt{dom}(p)$
        $p' := p \cup \{(t,s) \mid t \in T\}$
        $Q' := Q_{-0} \cdot T$
        return loop($Q'$, $p'$)

    return loop($\langle r \rangle$, $\{(r,\perp)\}$)
```

The algorithm `loop` is tail-recursive, so it is essentially a disguised form of an iterative algorithm.

Let G be a nonempty simple graph and r one of its vertices. For brevity, let us write $\mathsf{dist}(u)$ instead of $\mathsf{dist}(r,u)$ for each $u \in V$. The following invariant holds at the start of each iteration:[10]

(1.4) (i) $\{r\} \cup Q \subseteq \mathtt{dom}(p)$ and $N(u) \subseteq \mathtt{dom}(p)$ for each $u \in \mathtt{dom}(p) \setminus Q$;
(ii) p encodes r-paths and $p^*(u)$ is a shortest (r,u)-path for each $u \in \mathtt{dom}(p)$; and
(iii) $\mathsf{dist}(Q_{(i-1)}) \leqslant \mathsf{dist}(Q_{(i)}) \leqslant \mathsf{dist}(Q_{(0)}) + 1$ and for each $i \in [1 : |Q|)$.

[10] In set-theoretic contexts we (by abuse of notation) identify a sequence with the set of its entries, ignoring order and multiplicities. Thus, for instance, let X be a set, let $X' \subseteq X$, and let $s \in X^*$ be a sequence of length n. We write $s \subseteq X'$ to mean that $\{s_{(i)} \mid i \in [n)\} \subseteq X'$.

Assume that invariant (1.4) holds for the pair Q, p at the start of an arbitrary iteration and $Q \neq \epsilon$. Recall that

$$\begin{aligned} s &:= Q_{(0)}, \\ T &:= N(s) \setminus \mathtt{dom}(p), \\ p' &:= p \cup \{(t,s) \mid t \in T\}, \\ Q' &:= Q_{-0} \cup T. \end{aligned}$$

We will show that p' and Q' satisfy (i), (ii), and (iii).

It is easy to see that (i) holds.

For the proof of (ii), observe that it is immediate from the definition that p' encodes r-paths. By construction, $\mathtt{dom}(p') = \mathtt{dom}(p) \dot\cup T$.[11] Suppose that $t \in \mathtt{dom}(p')$. If $t \in \mathtt{dom}(p)$, we are done. We claim that

[11] For sets A and B, we write $A \dot\cup B$ for their disjoint union; that is, $A \dot\cup B$ denotes $A \cup B$ under the stipulation that $A \cap B = \varnothing$.

(1.5) if $t \in T$, then $(p')^*(t)$ is a shortest (r,t)-path ($\therefore \mathsf{dist}(t) = \mathsf{dist}(s) + 1$).

Suppose that $t \in T$. Let $P := \langle u_0, u_1, \ldots, u_{k-1} \rangle$ be an (r,t)-path (so $u_0 = r$ and $u_{k-1} = t$). Because $t \neq r$, we have $k \geqslant 2$. Since $u_0 \in \mathtt{dom}(p)$ (by (i)) and $u_{k-1} \notin \mathtt{dom}(p)$, there exists a largest $i \in [0 : k-1)$ such that $u_i \in \mathtt{dom}(p)$ and $u_{i+1} \notin \mathtt{dom}(p)$. Now, $u_i \in Q$. Indeed, if $u_i \in \mathtt{dom}(p) \setminus Q$, then (i) implies that $u_{i+1} \in \mathtt{dom}(p)$, which is a contradiction. Now, s and u_i are both in $Q \subseteq \mathtt{dom}(p)$. Thus $p^*(s)$ is a shortest (r,s)-path and $p^*(u_i)$ is a shortest (r,u_i)-path. Since $s = Q_{(0)}$, we have $|p^*(s)| = \mathsf{dist}(s) \leqslant \mathsf{dist}(u_i) = |p^*(u_i)|$. Moreover, $|P(r,u_i)| \geqslant |p^*(u_i)|$. But then

$$|P| \geqslant |P(r,u_i)| + 1 \geqslant |p^*(u_i)| + 1 \geqslant |p^*(s)| + 1 = |(p')^*(t)|.$$

Thus $(p')^*(t)$ is a shortest (r,t)-path. This completes the proof of (1.5) and of (ii).

The proof of (iii) follows directly from (1.5), since Q_{-0} is sorted by dist and $\mathsf{dist}(t) = \mathsf{dist}(s) + 1 \leqslant \mathsf{dist}((Q_{-0})_{[0]}) + 1$ for each $t \in T$. □

Termination. To prove termination, we introduce a *bounding function*: a function of the data structures manipulated by the loop that returns an integer upper bound on the number of iterations still to be performed. Such a function b must satisfy the following. For all Q and p, if Invariant (1.4) holds for the pair Q, p and $Q \neq \epsilon$, then

(1.6) (i) $b(Q', p') < b(Q, p)$; and
(ii) $b(Q, p) > 0$.

Here is a function b that takes a sequence Q and a partial function p and outputs[12]

$$b(Q,p) \;:=\; \big|V \setminus \mathtt{dom}(p)\big| \;+\; |Q|.$$

[12] Notice that $|Q|$ is the length of the sequence Q.

We will show that b is a bound function. We claim that, if Invariant (1.4) holds for Q, p and $Q \neq \epsilon$, then

$$b(Q', p') \;=\; b(Q,p) - 1,$$

In particular $b(Q,p) > 0$ and $b(Q', p') < b(Q,p)$.

Indeed, let $U := V \setminus \mathtt{dom}(p)$. Since $\mathtt{dom}(p') = \mathtt{dom}(p) \dot\cup T$ and $T \subseteq V \setminus \mathtt{dom}(p)$, we get

$$|V \setminus \mathtt{dom}(p')| \;=\; |U| - |T|.$$

Moreover, $Q' = Q_{-0} \cdot T$ is obtained from Q by removing its first element and appending all $t \in X$, hence

$$|Q'| \;=\; |Q| - 1 + |T|.$$

Therefore,

$$b(Q', p') \;=\; \big(|U| - |T|\big) + \big(|Q| - 1 + |T|\big) \;=\; \big(|U| + |Q|\big) - 1 \;=\; b(Q,p) - 1.$$

If $Q \neq \epsilon$, then $|Q| \geqslant 1$, so $b(Q,p) \geqslant 1$ and the measure is strictly decreasing. Therefore, b is a bound function.

Now it is routine to prove that the algorithm halts. The proof proceeds by induction on $b(Q,p)$. Let Q and p be such that Invariant (1.4) holds. If $Q = \epsilon$, then the algorithm halts, and there is nothing more to prove. Assume $Q \neq \epsilon$. Since $Q \neq \epsilon$ and Invariant (1.4) holds, by (1.6)(ii) we have $b(Q,p) > 0$, and by (1.6)(i) we have $b(Q', p') < b(Q,p)$. The next iteration begins with Q' and p'; therefore, by the induction hypothesis applied to Q' and p', the algorithm halts. □

Running time. Note that each iteration of the loop performs at most $c(1 + \deg(s))$ primitive operations for a suitable constant c. We claim that the following is also a loop invariant:

(1.7) The number of primitive operations performed by the loop is at most

$$c\Big(|\mathtt{dom}(p) \setminus Q| + \sum_{u \in \mathtt{dom}(p)\setminus Q} \deg(u)\Big).$$

This is clear at the beginning of the first iteration. Assume it holds at the beginning of an arbitrary iteration and that $Q \neq \epsilon$. Let p' and Q' be as above. Observe that

$$\mathtt{dom}(p') \setminus Q' = (\mathtt{dom}(p) \setminus Q) \cup \{s\},$$

and thus the number of primitive operations performed by the loop at the beginning of the next iteration is at most

$$c\Big(|\mathtt{dom}(p) \setminus Q| + \sum_{u \in U} \deg(u)\Big) + c(1 + \deg(s)) = c\Big(|\mathtt{dom}(p') \setminus Q'| + \sum_{u \in U'} \deg(u)\Big),$$

where $U := \mathtt{dom}(p) \setminus Q$ and $U' := \mathtt{dom}(p') \setminus Q'$, which completes the induction and proves (1.7).

For the time complexity, when the algorithm halts, the number of primitive operations performed by the loop is at most

$$c\Big(|\mathtt{dom}(p)| + \sum_{u \in \mathtt{dom}(p)} \deg(u)\Big) \leqslant c\big(|V| + 2|E|\big).$$

Since initialization takes constant time, the running time of the algorithm is $\mathcal{O}(|V| + |E|)$. □

We now turn to the proof of correctness of the `bfs` algorithm. We begin with a preparatory lemma.

Lemma 1.2. *Let G be a nonempty simple graph, let r be a vextex of G, and let $X \subseteq V$. If $r \in X$ and $\bigcup_{x \in X} N(x) \subseteq X$, then*

$$\big\{u \in V \mid \textit{there exists an } (r,u)\textit{-path in } G\big\} \subseteq X.$$

Proof. Let $u \in V$ and suppose that there exists an (r,u)-path $P := \langle u_0, \ldots, u_k\rangle$ in G (so $u_0 = r$ and $u_k = u$). We prove by induction on i that $u_i \in X$ for all $i \in [0:k]$. Let $i \in [0:k]$. For $i = 0$, we have that $u_0 = r \in X$ holds by hypothesis. Suppose $i > 0$ and assume that $u_{i-1} \in X$. Since P is a path, then $u_i \in N(u_{i-1})$. Beacuse $\bigcup_{x \in X} N(x) \subseteq X$ and $u_{i-1} \in X$, we have $N(u_{i-1}) \subseteq X$, whence $u_i \in X$. Thus $u \in X$, as required. □

Theorem 1.1. *Let G be a nonempty simple graph and let $r \in V$. Then,* `bfs(`G`, `r`)` *returns an encoding of r-paths $p : V \rightharpoonup V \cup \{\perp\}$ such that*

- $\mathtt{dom}(p) = \{u \in V \mid \mathtt{dist}(r,u) < \infty\}$ *and*
- *$p^*(u)$ is a shortest path from r to u for all $u \in \mathtt{dom}(p)$.*

Proof. By termination and Invariant 1.4, the call `bfs(`G`, `r`)` returns an encoding of r-paths such that

(i) $r \in \mathsf{dom}(p)$ and $N(u) \subseteq \mathsf{dom}(p)$ for each $u \in \mathsf{dom}(p)$; and

(ii) $p^*(u)$ is a shortest (r,u)-path for each $u \in \mathsf{dom}(p)$.

It remains to show that $\mathsf{dom}(p) = \{u \in V \mid \mathsf{dist}(u) < \infty\}$.

It is clear that $\mathsf{dom}(p) \subseteq \{u \in V \mid \mathsf{dist}(u) < \infty\}$, since $p^*(u)$ is an (r,u)-path for each $u \in \mathsf{dom}(p)$. To see that $\{u \in V \mid \mathsf{dist}(u) < \infty\} \subseteq \mathsf{dom}(p)$, assume that $u \in V$ and $\mathsf{dist}(u) < \infty$. Hence there exists an (r,u)-path in G. By (i) $r \in \mathsf{dom}(p)$ and $N(u) \subseteq \mathsf{dom}(p)$ for each $u \in \mathsf{dom}(p)$, whence, by Lemma 1.2, we have that $u \in \mathsf{dom}(p)$, as required. □

A simple Python *implementation.* Now we exhibit a simple Python implementation. For this, we will use deque from the collections library.

```python
from collections import deque

def bfs(G, r):
    Q, p = deque([r]), {r: None}
    while Q:
        s = Q.popleft()
        T = (t for t in G[s] if t not in p)
        for t in T:
            p[t] = s
            Q.append(t)
    return p
```

A simple Clojure *implementation.* The Clojure implementation uses the standard-library queue clojure.lang.PersistentQueue and resembles the "functional" version of the bfs algorithm.

```clojure
(defn bfs [G r]
  (loop [Q (conj PersistentQueue/EMPTY r)
         p {r nil}]
    (if (empty? Q)
      p
      (let [s (peek Q),
            T (filter #(not (contains? p %1)) (G s))
            Q' (into (pop Q) T)
            p' (into p (for [t T] [t s]))]
        (recur Q' p')))))
```

The use of persistent data structures allows a compact representation of paths. The following code illustrates this: rather than returning an encoding of paths, it returns a map from vertices to their paths.

```
(defn bfs-paths [G r]
  (loop [Q (conj PersistentQueue/EMPTY r), p {r [r]}]
    (if (empty? Q)
      p
      (let [s (peek Q)
            T (for [t (G s) :when (not (contains? p t))] t)
            Q' (into (pop Q) T)
            p' (into p (for [t T] [t (conj (p s) t)]))]
        (recur Q' p')))))
```

Exercise 1.16. Provide Clojure code that, given an adjacency-list representation G of a simple graph and a representation r of one of its vertices, returns a map that, for each vertex representation s, maps s to a sequence of paths–each path a sequence of vertex representations–listing all shortest (r,s)-paths in G.

Exercise 1.17. Consider the following program — an implementation of BFS. Explain how it works.

```
(defn it-bfs [G r]
  (loop [G G, s r, Q (conj PersistentQueue/EMPTY s), p {r nil}]
    (cond
      (nil? s) p
      (empty? (G s)) (recur G (peek Q) (pop Q) p)
      :else (let [[t & T] (G s),
                  G' (assoc G s T),
                  p' (assoc p t s)]
              (if (contains? p t)
                (recur G' s Q p)
                (recur G' s (conj Q t) p'))))))
```

Exercise 1.18. Design an algorithm that, given a simple graph G and vertices $a,b,c \in V(G)$, computes

$$\gamma(a,b,c) := \min\big\{|E(P_1) \cup E(P_2)| \;\big|\; P_1 \text{ is an } (a,c)\text{-path, } P_2 \text{ is a } (b,c)\text{-path}\big\}.$$

Hint. Show that

$$\gamma(a,b,c) = \min\big\{\mathrm{dist}(a,x) + \mathrm{dist}(b,x) + \mathrm{dist}(c,x) \mid x \in V\big\}.$$

Provide implementations in Python and Clojure.

Exercise 1.19. Let G be a simple graph, and let $w : E \to \{0,1\}$ be a weight function. The **weight** of a path $P := \langle u_0, u_1, \ldots, u_{k-1}\rangle$, denoted $w(P)$, is

$$w(P) = \sum_{i\in[1:k)} w(u_{i-1}u_i).$$

For vertices r and s of G, we say that an (r,s)-path P is w-**minimum** if $w(P) \leqslant w(Q)$ for every (r,s)-path Q. Design an algorithm that, given a simple graph G, a weight function w, and a vertex r of G, finds, for each $s \in V$, an (r,s)-path that is w-minimum (when one exists).

1.5 *Subgraphs*

Let G and H be simple graphs. We say that H is a **subgraph** of G, written $H \subseteq G$, if

$$V(H) \subseteq V(G) \quad \text{and} \quad E(H) \subseteq E(G).$$

In words, H is a subgraph of G whenever the following conditions hold:

- every vertex of H is a vertex of G, and
- if uv is an edge of H, then uv is also an edge of G.

A subgraph H of G is said to be **spanning** if $V(H) = V(G)$.

1.5.1 *Induced subgraph*

Let G be a simple graph, and let $X \subseteq V$. An edge whose endpoints both lie in X is said to be **induced** by X. The **set of edges induced** by X is denoted by $E_G[X]$ (or simply $E[X]$); that is,

$$E[X] = \{e \in E \mid e \subseteq X\}.$$

The subgraph of G **induced** by X, denoted $G[X]$, is the simple graph

$$(X, E[X]).$$

1.6 *Connected components*

A simple graph G is **connected** if for every $u, v \in V$ there exists a path joining u and v. A **connected component** (or simply a **component**) of a nonempty simple graph G is a maximal subset[13] $X \subseteq V$ such that the simple graph $G[X]$ is connected; that is, a subset $X \subseteq V$ such that $G[X]$ is connected and, for all $Y \subseteq V$, if $Y \supseteq X$ and $G[Y]$ is connected, then $Y = X$.[14] Observe that if X is a component of a nonempty simple graph G, then $X \neq \emptyset$, since $G[\{v\}]$ is obviously connected for every vertex v of G. It is often convenient to identify a connected component X of a nonempty simple graph G with the simple graph $G[X]$. As usual, we will adopt this convention whenever no confusion can arise.

[13] Recall that given a nonempty collection $\mathscr{X}$ of subsets of some set, we say that an element $X \in \mathscr{X}$ is **maximal** if

$$Y \supseteq X \Rightarrow Y = X$$

for every $Y \in \mathscr{X}$.

[14] Equivalently, $G[Y]$ is not connected whenever $X \subset Y \subseteq V$.

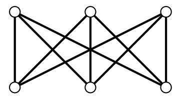


Figure 1.6: The figure illustrates a connected graph.

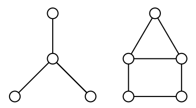


Figure 1.7: A graph and its two connected components.

Lemma 1.3. *Let G be a simple graph and let $X_1, X_2 \subseteq V$. If $G[X_1]$ and $G[X_2]$ are connected and $X_1 \cap X_2 \neq \emptyset$, then $G[X_1 \cup X_2]$ is also connected.*

Proof. It suffices to show that there exists a path from x_1 to x_2 for each $x_1 \in X_1$ and each $x_2 \in X_2$. So let $x_1 \in X_1$ and $x_2 \in X_2$, and let $x \in X_1 \cap X_2$. Since $G[X_1]$ is connected and $x, x_1 \in X_1$, there exists a path P_1 from x_1 to x. Similarly, there exists a path P_2 from x to x_2. It follows that $P_1 \cdot P_2$ is a path from x_1 to x_2. We conclude that $G[X_1 \cup X_2]$ is connected. □

Corollary 1.2. *If X_1 and X_2 are connected components of a nonempty simple graph G, then $X_1 = X_2$ or $X_1 \cap X_2 = \emptyset$.*

Proof. Suppose that G is a nonempty simple graph and X_1, X_2 are connected components of G with $X_1 \cap X_2 \neq \emptyset$. By Lemma 1.3, $G[X_1 \cup X_2]$ is connected. The maximality of X_1 implies that $X_1 = X_1 \cup X_2$. The maximality of X_2 implies that $X_2 = X_1 \cup X_2$. Hence $X_1 = X_2$. □

Corollary 1.3. *The set of connected components of a nonempty simple graph is a partition of its vertex set.* □

A **partition** of a set U is a set of nonempty, pairwise disjoint subsets of U whose union is U. For example, $\{\{1,4\},\{2,5\},\{3,6,7\}\}$ is a partition of the set $[7]$.

The number of connected components of a nonempty simple graph G is denoted by $\kappa(G)$. Every vertex of a graph belongs to exactly one connected component of the graph; the connected component that contains a given vertex x is called the **territory** of x.

Exercise 1.20. Let G be a nonempty simple graph, and let $r \in V$. Show that G is connected if and only if there exists a path joining r and s for all $s \in V$.

Exercise 1.21. Prove that a simple graph G is connected if and only if

$$\widehat{N}(X) \neq \varnothing$$

for all $\varnothing \subset X \subset V$.

Exercise 1.22. Suppose that a simple graph G has exactly two vertices, say u and v, of odd degree. Show that there exists a path in G joining the vertices u and v.

Exercise 1.23. Prove that, if G is a simple graph in which every vertex $v \in V$ satisfies $\deg(v) \geqslant |G|/2$, then G is connected.

1.6.1 *Breadth-first search and components*

How can breadth-first search be used to determine the connected components of a nonempty simple graph? We first show how breadth-first search recovers the territory of a given vertex in a nonempty simple graph, and we leave to the exercises a straightforward modification that computes all connected components.

Let G be a nonempty simple graph and let $r \in V$. Notice that `bfs(`G`, `r`)` returns an encoding p of r-paths such

$$\mathsf{dom}(p) = \{s \in V \mid \mathsf{dist}(r,s) < \infty\}.$$

Let C be the territory of r in G. We will show that $\mathsf{dom}(p) = C$. Now,

$$\begin{aligned} s \in C \quad &\equiv \quad \text{there exists an } (r,s)\text{-path in } G \\ &\equiv \quad \mathsf{dist}(r,s) < \infty \\ &\equiv \quad s \in \mathsf{dom}(p), \end{aligned}$$

whence $\mathsf{dom}(p) = C$.

Territory of a vertex

The next algorithm receives a simple graph G, given by its neighborhood function N, and a vertex r of G. It returns the territory of r in G.

```
def territory(G, r):
    return dom(bfs(G, r))
```

Exercise 1.24. Design an algorithm `components(`G`)`, based on breadth-first search, that computes all connected components of a nonempty simple graph G. It is up to you to decide what the algorithm returns. Provide a working `Python` and `Clojure` implementation.