Analysis of Algorithms I: Graph Representation and BFS

Xi Chen

Columbia University

An undirected graph $G = (V, E)$ is a pair in which V is a set of vertices (or nodes) and E is a set of edges (or arcs). Each edge is a set $\{u, v\}$ (or an unordered pair), where u and v are vertices in V. When $u = v$, we call it a self-loop. By convention, we usually use the notation (u, v) for an edge, instead of $\{u, v\}$, and we consider (u, v) and (v, u) to be the same edge. In particular, this means that if $(u, v) \in E$ then we also have $(v, u) \in E$ simply because $\{u, v\} = \{v, u\}$.

For example, the undirected graph in (b) on Page 1169 has:

$$E = \{(1,2),(1,5),(2,5),(3,6)\}$$

Again, the order in each pair does not mean anything here. Usually we use $n = |V|$ to denote the number of vertices and $m = |E|$ to denote the number of edges. Usually the vertices are numbered from $1, 2, \ldots$ to n so we have

$$V = \{1, 2, \ldots, n\}$$

For convenience, we will always assume that this is the case in all the discussions on graph algorithms.

We usually consider simple undirected graphs: no self-loops and no parallel (duplicate) edges. (Safe to assume this in all the graph algorithms we discuss in this course.) So we always have $m < n^2$. (Not a formal definition: When $m \ll n^2$ we call G a sparse graph and when $m = \Theta(n^2)$ we call G a dense graph.) If $(u, v) \in E$, we say v (or u) is adjacent to u (or v); we say v (or u) is a neighbor of u (or v); we say this edge (u, v) is incident on vertices u and v. The degree of a vertex u is then the number of edges that are incident on u. Review Appendix B.4 for other notation.

One way to represent an undirected graph is to use its adjacency matrix: Given $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, we refer to the following $n \times n$ $\{0, 1\}$-matrix $\mathbf{A}$ as the adjacency matrix of G:

$$A_{i,j} = \begin{cases} 1 & \text{if } (i, j) \in E\text{; and} \\ 0 & \text{otherwise} \end{cases}$$

Note that $\mathbf{A}$ is always symmetric: $A_{i,j} = A_{j,i}$ (why?).

A directed graph $G = (V, E)$ is a pair, where V is a set of vertices and E is a set of (directed) edges. Here each edge is an ordered pair (u, v) with u, v being vertices in V. The order does matter here: $(u, v) \in E$ does not imply $(v, u) \in E$. For example, check the example (a) on Page 1169.

The directed graphs $G = (V, E)$ we consider may have self-loops (e.g., the graph in Figure 22.2 on Page 590 has a self-loop $(6, 6)$ in E), but we always assume that there are no parallel edges in E and thus, $m \leq n^2$. Similarly, if $(u, v) \in E$, we say v is adjacent to u or there is an edge from u to v. We also say this edge (u, v) is incident from (or leaves) vertex u and is incident to (or enters) vertex v. The in-degree of a vertex u is the number of edges incident to it (or enter it) and the out-degree is the number of edges incident from it (or leave it). The degree of a vertex is then its in-degree plus its out-degree.

Similarly, a directed graph $G = (V, E)$, where $V = \{1, 2, \ldots, n\}$, can be represented by its adjacency matrix $\mathbf{A}$:

$$A_{i,j} = \begin{cases} 1 & \text{if } (i, j) \in E\text{; and} \\ 0 & \text{otherwise} \end{cases}$$

But unlike undirected graphs, $\mathbf{A}$ is not symmetric in general!

We will also work on weighted (undirected or directed) graphs $G = (V, E)$, in which each edge has an associated weight. We usually denote these edge weights using a weight function

$$w : E \to \mathbb{R}$$

So the weight of an edge $(u, v) \in E$ is $w(u, v)$, with

$$w(u, v) = w(v, u)$$

when the graph G is undirected. If G is directed, then $w(u, v)$ and $w(v, u)$ are different in general.

For each vertex $u \in V$, we let $N(u)$ denote the following set of vertices: If G is undirected, then $N(u)$ is the set of all neighbors of u, so $|N(u)| = \text{deg}(u)$; and if G is directed, then $N(u)$ is the set of all v such that $(u, v) \in E$, so $|N(u)| = \text{out-deg}(u)$.

The adjacency-list representation of a graph $G = (V, E)$ consists of $n = |V|$ lists (sometimes we may use doubly linked lists for easy deletion) one for each $u \in V$: the adjacency list $\mathrm{Adj}(u)$ of $u \in V$ contains all the vertices $v \in N(u)$. See examples on Page 590.

A quick question: If $G = (V, E)$ has $m = |E|$, then what is the total length of all the $n = |V|$ lists? The answer depends on whether G is undirected or directed:

1. If G is undirected, then every edge (u, v) leads to two entries: v in $\mathrm{Adj}(u)$ and u in $\mathrm{Adj}(v)$. So the total length is $2m$.
2. If G is directed, then every edge (u, v) leads to exactly one entry: v in $\mathrm{Adj}(u)$. So the total length is m.

Also we can adapt adjacency lists to represent a weighted graph (undirected or directed): For each edge $(u, v) \in E$, just store the weight $w(u, v)$ of $(u, v) \in E$ with vertex v in $\mathrm{Adj}(u)$.

Comparison between the two representations: Given a graph $G = (V, E)$ with $n = |V|$ and $m = |E|$, the space used is

1. Adjacency matrix: n^2
2. Adjacency list: m

So space-wise the adjacency list representation is more efficient (unless the graph is dense: $m = \Theta(n^2)$).

However, the other (sometimes more important) difference between these two representations is the following: Given a vertex $u \in V$, the enumeration of edges incident on (or from) u (or equivalently, the enumeration of all vertices v with $(u, v) \in E$) takes time

1. Matrix: $\Theta(n)$ (to go through a binary string of length n)
2. List: linear in the length of $\mathrm{Adj}(u)$ (or equivalently, $\deg(u)$ if G is undirected, and out-$\deg(u)$ if G is directed)

So the list representation always performs better. But the matrix representation of course has its own advantage: To decide if $(u, v) \in E$, it only uses $O(1)$ time, while the list representation has worst-case running time linear in the length of $\mathrm{Adj}(u)$ (why?).

Usually we do not consider the issue of which representation to use until the algorithm analysis stage. If the input graphs are sparse and we make a lot of enumerations of edges incident to a vertex in our algorithm, then the list representation usually fits better. If the algorithm uses a lot of random access of edges in G, then the matrix representation may perform better. We will see the use of both representations in different graph algorithms.

In the rest of this class, we describe Breadth-first search (BFS) and its applications. We start with the following problem: Given a (directed or undirected) graph $G = (V, E)$ and a "source" vertex $s \in V$, which vertices are reachable from s? Formally, if G is undirected, then we say $v \in V$ is reachable from s if there is a path connecting s and v. If G is directed, then we say v is reachable from s if there is a directed path "from" s "to" v:

$$s, u_1, \ldots, u_k, v$$

for some $k \geq 0$ such that $(s, u_1), (u_1, u_2), \ldots, (u_k, v) \in E$. (Note that when G is undirected, the set of vertices reachable from s is indeed the connected component that contains s. Prove it.)

Consider the following "generic" search algorithm:

```
1  set $R = \{s\}$
2  while there is an edge from $R$ to $V - R$ do
3      let $(u, v) \in E$ be such an edge with $u \in R$, $v \in V - R$
4      set $R = R \cup \{v\}$
```

Note that we did not specify which edge (u, v) to pick if at the beginning of a while-loop, there are multiple edges from R to $V - R$. As we will see, different strategies give us different algorithms that are useful in different contexts. Before that, we show that the algorithm is always correct: upon termination, R has all vertices reachable from s, no matter which strategy is used.

Some notation: During the execution of the algorithm, if we add v into R in a while loop because of (u, v), we say u discovers v.

Given two vertices u and v in V, we use $\delta(u, v)$ to denote the shortest-path distance (or simply distance) from u to v: the minimum number of edges in any path from u to v; if there is no path from u to v (i.e., v is not reachable from u), then we set $\delta(u, v) = \infty$. The following two lemmas are easy to prove:

Lemma

Let s be a vertex in V. If $(u, v) \in E$ then $\delta(s, v) \leq \delta(s, u) + 1$.

Lemma

Assume $\delta(s, v) = d \geq 2$ and let

$$s, u_1, \ldots, u_{d-1}, v$$

denote a path from s to v of length d, then $\delta(s, u_{d-1}) = d - 1$.

Lemma

Upon termination, R is the set of vertices reachable from s.

First, using induction it is easy to show that at any time (and in particular, when the algorithm terminates), all vertices in R are reachable from s. (Prove this as an exercise.)

Second, assume there is a vertex in $V - R$ reachable from s upon termination. Let v denote a vertex in $V - R$ with the smallest distance from s. Because R is the set we get upon termination, it must be the case that there is no edge from R to $V - R$ (because otherwise the algorithm should continue). Let $\delta(s, v) = d \geq 1$ then by definition, there is a path from s to v of length d:

$$s, u_1, \ldots, u_{d-1}, v$$

Consider the following two cases: $u_{d-1} \in R$ or $\notin R$:

1. If $u_{d-1} \in R$, this contradicts with the assumption that R is the set we get upon termination (why?).
2. If $u_{d-1} \notin R$, contradict with the assumption that v has the minimum distance from s among all vertices in $V - R$ (why?).

This finishes the proof of the lemma.

Now we describe the breadth-first search in more details: First we maintain the set R by having an extra field called color with each vertex: u.color = white means that $u \in V - R$ at this moment and is not reached yet; u.color = black means that $u \in R$ at this moment and thus, is reachable from s. Note that the textbook uses three colors just for illustration purposes (see the footnote).

We use the following BFS strategy to pick an edge in each round: Choose an edge (u, v) where u is the earliest vertex added to R that has an edge $(u, v) \in E$ with $v \in V - R$ (white).

A straight-forward implementation for the BFS strategy is the following: Maintain a list L of all vertices in R (black) at the moment, ordered by the time they are added into R; Start with $L = (s)$; At the beginning of each while-loop, we go through the list from left to right (starting with s) and for each u in the list, check every $v \in N(u)$: If there is a white vertex $v \in N(u)$, we color v black (add v into R) and also append v to the end of L (because it is the newest vertex in R).

However, this implementation is not the most efficient. During the first $|N(s)|$ many while-loops, we keep adding vertices $v \in N(s)$ into R. Once we have discovered all vertices in $N(s)$ (meaning every $v \in N(s)$ is black), s can no longer be used to discover new reachable vertices any more. So we should remove s from L (though s remains black and remains in R) just because there is no need to go through $N(s)$ anymore, when trying to find an edge from R to $V - R$ at the beginning of each loop that follows.

This inspires us to maintain a first-in, first-out queue Q instead. All vertices in Q are ordered according the time they are added into R. Every time we need to find an edge between R and $V - R$, Dequeue a vertex u from Q and add all white vertices $v \in N(u)$ into R. (Note that we Dequeue u from Q because after adding all vertices in $N(u)$ into R, there is no need to explore u anymore.) Whenever we add a vertex v into R (by changing its color from white to black), we also Enqueue v into Q as well because we just discovered v and need to use $N(v)$ to expand R in the future. As Q is FIFO, v will be appended to the end of Q. This guarantees that vertices in Q are ordered by the time they are added into R.

Breadth-first search(G, s): where $G = (V, E)$

```
1  for each vertex $u \in V - \{s\}$ do
2      set $u$.color $=$ white and $u.d = \infty$
3  set $s$.color $=$ black and $s.d = 0$
4  Enqueue$(Q, s)$
5  while $Q \neq \emptyset$ do
6      $u =$ Dequeue$(Q)$
7      for each $v \in N(u)$ do
8          if $v$.color $=$ white then
9              set $v$.color $=$ black and $v.d = u.d + 1$
10             Enqueue$(Q, v)$
11     $u$.color $=$ black
```

It is clear that we have $v \in V$ is reachable from s if and only if

$$v.\text{color} = \text{black}$$

upon termination, simply because any strategy works as we proved earlier. But one advantage of BFS is that if we add an extra field $v.d$ for each vertex $v \in V$ (as we did in the last slide), we can also obtain the distance $\delta(s, v)$ from s to every vertex v, stored in $v.d$.

Before proving that BFS computes the distances correctly, which representation should we adopt and what is the running time of BFS? First, if $u = \text{Dequeue}(Q)$ in a while-loop, then what is the total running time of this loop? It depends on the graph representation! We choose to use the list representation here because the for-loop is essentially an enumeration of vertices in $N(u)$ (Why does the list representation performs better here?). If we use the list representation, then this can be done by going through $\mathrm{Adj}(u)$ in time linear in $|N(u)|$. So the total running time of this while-loop is $O\big(|N(u)|\big)$.

Second, given a vertex $u \in V$, how many times can u be dequeued from Q during the execution of BFS? At most once! (why?) Therefore, the total running time is bounded by

$$O(n) + \sum_{u \in V} O\big(|N(u)|\big) = O(n) + O(m) = O(n+m)$$

where the first $O(n)$ term accounts for the initialization.

Finally, we work on the correctness of BFS. We start with some intuition of how BFS works. Check the example on Page 596. At the beginning, $Q = \langle s \rangle$, $s = 0$ and is the only black vertex. In the first while-loop, we dequeue s from Q; change its two neighbors w, r to black; and enqueue them into Q. Note that $Q = \langle w, r \rangle$ at this moment and we have

$$\delta(s, w) = \delta(s, r) = w.d = r.d = 1$$

Also note that r, w are the only vertices of distance 1 from s.

Then we dequeue w from Q; change its two neighbors t, x to black (note that s is already black so we ignore it); and enqueue them into Q. We then dequeue r from Q; change its neighbor v to black (again, s is black so we ignore it); and enqueue it into Q. After this while-loops, we have

$$Q = \langle t, x, v \rangle$$

and note that

$$\delta(s,t) = \delta(s,x) = \delta(s,v) = t.d = x.d = v.d = 2$$

Also check that these are the only vertices of distance 2 from s. The BFS algorithm then continues.

From this example we see that BFS adds vertices to Q wave by wave according their distances from s. It starts by discovering vertices with distance 1 from s first, i.e., vertices in $N(s)$. For each vertex $u \in N(s)$, it discovers those white vertices in $N(u)$; adds them into R; and enqueues them into Q. From the example, after all $u \in N(s)$ are dequeued and explored, Q consists of exactly those vertices of distance 2 from s. Then after all vertices of distance 2 are dequeued and explored, Q consists of exactly those vertices of distance 3 from s. And it continues. This is how BFS is named because it visits vertices of distance $1, 2, \ldots$ from s, level by level.

We show that upon termination:

$$\delta(s, v) = v.d, \quad \text{for all } v \in V$$

If v is not reachable from s, then as we showed earlier, upon termination v.color remains white ($\in V - R$). As a result, $v.d$ remains ∞ since we only change $v.d$ when changing its color from white to black (or equivalently, when adding v into R). So the equation above holds. To prove the equation for vertices reachable from s, we need the following lemma:

Lemma

Given two vertices u, v both reachable from s, if

$$\delta(s, u) < \delta(s, v)$$

then u is added into R earlier than v.

We prove it by induction on $\delta(s, u)$! (I used induction on $\delta(s, v)$ in class. It worked but is not the best proof ...) The basis is trivial: If $\delta(s, u) < \delta(s, v)$ and $\delta(s, u) = 0$, u must be s. The property then follows trivially because s is the first vertex in R.

Now assume that the claim holds for all u, v with

$$\delta(s,u) < \delta(s,v) \quad \text{and} \quad \delta(s,u) \leq k$$

for some $k \geq 0$. (This is our inductive hypothesis.) We need to show that it also holds for all vertices u, v with

$$\delta(s,u) < \delta(s,v) \quad \text{and} \quad \delta(s,u) = k+1$$

For this purpose, let $s, u_1, \ldots, u_k, u$ denote a path from s to u of length $k+1$. We must have (why?):

$$\delta(s,u_k) = k$$

Moreover, we know that by the end of the while loop in which u_k is dequeued from Q, u must be in R (why?).

On the other hand, let w denote the vertex that discovers v. This means that $(w, v) \in E$ and we add v into R during the while loop in which w is dequeued. By $(w, v) \in E$, we have:

$$\delta(s, w) \geq \delta(s, v) - 1 > \delta(s, u) - 1 = k = \delta(s, u_k)$$

Now we can apply our inductive hypothesis because

$$\delta(s, u_k) < \delta(s, w) \quad \text{and} \quad \delta(s, u_k) = k$$

(see the importance of picking the right induction variable!) It implies that u_k enters Q before w and thus, is also dequeued from Q earlier. It then follows that u enters Q before v (why?).

Finally we prove the correctness of BFS: Upon termination,

$$v.d = \delta(s, v), \quad \text{for any } v \in V \text{ reachable from } s.$$

We use induction on $\delta(s, v)$. Basis is trivial: If $\delta(s, v) = 0$ then we must have $v = s$ and thus, $s.d = 0 = \delta(s, s)$ upon termination. Induction step: Assume $v.d = \delta(s, v)$ for any v with $\delta(s, v) \leq k$, for some $k \geq 0$. We show that any v with $\delta(s, v) = k + 1$ has

$$v.d = \delta(s, v) = k + 1$$

To this end, we use u to denote the vertex that leads BFS to discover v. This means that u is black, just dequeued from Q, $(u, v) \in E$; and BFS sets

$$v.d = u.d + 1$$

when adding v into R and Q. On the other hand, let

$$s, u_1, u_2, \ldots, u_k, v$$

denote a path from s to v of length $k + 1$, then $\delta(s, u_k) = k$.

We claim that u must satisfy

$$\delta(s,u) = k$$

Otherwise (if $\delta(s,u) \neq k$), because

$$\delta(s,u) \geq \delta(s,v) - 1 = k$$

we have $\delta(s,u) > k$ and thus, by the last lemma u enters Q after u_{k+1}. This implies that u cannot be the vertex that discovers v (why?), contradicting with our assumption. So we must have $\delta(s,u) = k$. By the inductive hypothesis, we have

$$v.d = u.d + 1 = \delta(s,u) + 1 = k + 1$$

Also check the textbook for the definition of Breadth-first trees. By introducing another attribute $v.\pi$ for each vertex $v \in V$ (a pointer to another vertex, set to be nil at the beginning), one can simply add a line to set $v.\pi = u$ when u discovers v. Then upon the termination of BFS, these pointers give us a Breadth-first tree, with essentially the same running time.

Using BFS, we have the following algorithm that computes all the connected components of $G = (V, E)$:

1. Initialization: set all vertices to be white
2. Pick an arbitrary vertex $s \in V$, and call BFS(G, s)
3. Copy all the black vertices to an array (as the first connected component) and remove them from G
4. Pick another vertex $s' \in V$, if V is not empty yet
5. BFS(G, s') and repeat, until V is empty

Prove that the total running time is $O(n + m)$. This requires a more accurate upper bound for the running time of BFS(G, s): Show that it is linear in the number of vertices in the connected component that contains s.