CS 341 - Algorithms, Winter 2023. University of Waterloo

Lecture 5: Breadth First Search

We study simple graph algorithms based on graph searching.

There are two most common search methods: breadth first search (BFS) and depth first search (DFS).

Today we study BFS and see some applications. Next time we will study DFS.

Graphs

Many problems in computer science can be modeled as graph problems. See [KT 3.1] for discussions.

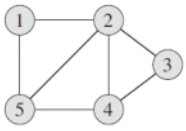
Graph Representations

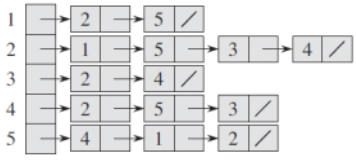
Let $G=(V,E)$ be an undirected graph. We use throughout that $n=|V|$ and $m=|E|$.

There are two standard representations of a graph.

One is the adjacency matrix: It is an $n \times n$ matrix A with $A[i,j] = \begin{cases} 1 & \text{if } ij \in E \\ 0 & \text{if } ij \notin E \end{cases}$.

Another is the adjacency list: Each vertex maintains a linked list of its neighbors.





	1	2	3	4	5
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

(picture from [CLRS])

We will mostly use the adjacency list representation, as its space usage depends on the number of edges, while we need to use $\Theta(n^2)$ space to store an adjacency matrix.

Only the adjacency list representation allows us to design algorithms with $O(m+n)$ word operations.

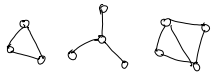
Graph Connectivity

Given a graph, we say two vertices are connected if there is a path from u to v.

A subset of vertices $S \subseteq V$ is connected if $u,v \in S$ are connected for all $u,v \in S$.

A graph is connected if $s,t \in V$ are connected for all $s,t \in V$.

A connected component is a maximally connected subset of vertices.



3 Components

Some of the most basic questions about a graph are:

1. to determine whether it is connected.
2. to find all the connected components.
3. to determine whether u,v are connected for given $u,v \in V$.

④ to output a shortest path between u and v for given $u, v \in V$.

Breadth first search (BFS) can be used to answer all these questions in $O(n+m)$ time.

Breadth First Search

To motivate breadth first search, imagine we are searching for a person in a social network. A natural strategy is to ask our friends, and then ask our friends to ask their friends, and so on. A basic version of BFS is described as follows.

Input: $G = (V, E)$, $s \in V$.

Output: all vertices reachable from s.

Initialization: visited[v] = false for all $v \in V$.

queue Q is empty. enqueue(Q, s). visited[s] = true.

```
While Q ≠ empty do
    u = dequeue(Q)
    for each neighbor v of u
        if visited[v] = false
            enqueue(Q, v).   visited[v] = true.
```

Time Complexity

Each vertex is enqueued at most once (when visited[v] = false).

When a vertex is dequeued, the for loop is executed for deg(v) iterations.

So, the total time complexity is $O(n + \sum_{v \in V} \deg(v)) = O(n+m)$.

Lemma There is a path from s to v if and only if visited[v] = true at the end.

Proof

$\Leftarrow$) We prove by induction on the number of steps of the algorithm that if visited[v] = true, then there is a path from s to v.

The base case is at the beginning when only visited[s] = true.

Now, supposed visited[v] is set to be true in the current step, inside the for loop of u.

Then, visited[u] = true at that time, because u was put in the queue.

By the induction hypothesis, there is a path from s to u.

Extending this path with the edge uv, we have found a path from s to v.

$\Rightarrow$) Let U be the set of vertices such that visited[v] is set to be true.

We would like to show that there is no path from s to any vertex in $V \setminus U$.

Note that there are no edges with one endpoint in U and another endpoint in $V \setminus U$, as otherwise we would have enqueued the other endpoint and set it to true.

Suppose for contradiction that there is a path from s to $v \in V \setminus U$.

Then there must exist an edge in the path that "crosses" the set U, a contradiction. □

The correctness of BFS is supported by the lemma.

With this claim, we see that this basic version of BFS can already be used to answer:

- whether the graph is connected or not, by checking whether visited[v] = true for all $v \in V$;
- the connected component containing s, by returning all the vertices with visited[v] = true;
- whether there is a path from s to v, by checking whether visited[v] = true.

Exercise: Find all connected components of the graph in $O(n+m)$ time.

BFS Tree

How to trace back a path from s to v (if such a path exists)?

This follows from the proof of the lemma above.

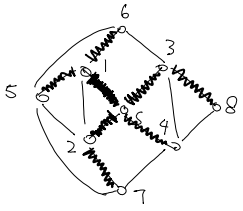
We can add an array parent[v].

When a vertex v is first visited, within the for loop of vertex u, then we set parent[v] = u.

Now, to trace out a path from v to s, we just need to write a for loop that starts from v, and keep going to its parent until we reach vertex s.

For all vertices v reachable from s, the edges $(v, \text{parent}[v])$ form a tree, called the BFS tree.

Why is it a tree in the connected component containing s?



Say the connected component has n vertices.

Every vertex, except s, has one edge to its parent.

Theses edges can't form a cycle because the parent of a vertex is visited earlier.

So, these edges form an acyclic subgraph and there are $n-1$ edges (as s has no parent).

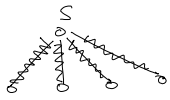
Therefore, the edges $(v, \text{parent}[v])$ must form a tree in the component containing s.

Shortest Paths

Not only can we trace back a path from v to s using a BFS tree,
this path is indeed a shortest path from s to v!

To see this, let's think about how a BFS tree was created.

These edges record the first edges to visit a vertex.



Initially, s is the only vertex in the queue, and then every neighbor of s
is visited with s being their parent, and these edges are put in the BFS tree.

At this time, all vertices with distance one from s are visited and are put in the queue,
before all other vertices with distance at least two from s are put in the queue.

A vertex v is said to have distance k from s if the shortest path length from s to v is k.

Then, all vertices with distance one will be dequeued, and then all vertices with distance two
will be enqueued before all other vertices with distance at least three.

Repeating this argument inductively will show that all the shortest path distances from s are
computed correctly, and a shortest path can be traced back from the BFS tree.

This is also very intuitive (friends before friends of friends before friends of friends of friends etc).

Being able to compute the shortest paths from s is the main feature of BFS.

We summarize below the BFS algorithm with shortest path distances included.

Input: $G = (V, E)$, $s \in V$.

Output: all vertices reachable from s and their shortest path distance from s.

```
Initialization:   visited[v] = false for all v ∈ V.
                  queue Q is empty.  enqueue(Q,s).  visited[s] = true.  distance[s] = 0.
While Q ≠ empty do
    u = dequeue(Q)
    for each neighbor v of u
        if visited[v] = false
            enqueue(Q,v).  visited[v] = true.  parent[v] = u.  distance[v] = distance[u] + 1.
```

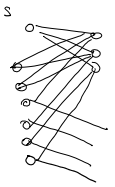
Exercise: Write the code for printing a shortest path from v to s.

Bipartite Graphs

One application of BFS is to check whether a graph is bipartite or not.

There is not much freedom allowed to design an algorithm for checking bipartiteness.

Given a vertex s, all its neighbors must be on the other side, and then neighbors of neighbors must be on the same side as s, and so on.



With this observation, we can run the BFS algorithm above and put all vertices with even distance from s on the same side as s and all other vertices on the other side.

Algorithm

- Let $L = \{ v \in V \mid dist(s,v) \text{ even} \}$ and $R = \{ v \in V \mid dist(s,v) \text{ odd} \}$.
- If there is an edge with both endpoints in L or both endpoints in R, then return "non-bipartite".
- Otherwise, return "bipartite" and (L,R) as the bipartition.

We assume the graph is connected, as otherwise we can solve the problem in each component.

The time complexity is $O(m+n)$ as we just do a BFS and then check every edge once.

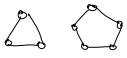
Correctness

It is clear that when the algorithm says "bipartite" it is correct, as (L,R) is indeed a bipartition.

The more interesting part is to show that when the algorithm say "non-bipartite", it is also correct.

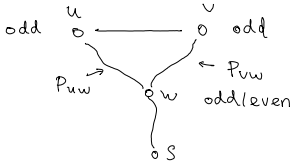
When can we say for sure that a graph is non-bipartite?

A necessary and sufficient condition is when the graph has an odd cycle (MATH 239)



Thus we would like to show that when the algorithm says "non-bipartite", the graph has an odd cycle.

Suppose WLOG that there is an edge uv between two vertices u,v in L.



We look at the BFS tree T.

Let w be the lowest common ancestor of u,v in T.

Since $dist(s,u)$ and $dist(s,v)$ are both odd (i.e. having the same parity), regardless of whether $dist(s,w)$ is even or odd, the sum of the path lengths of uw and vw on T is a even number.

This implies that $P_{vw} \cup P_{uw} \cup \{uv\}$ is an odd cycle.

So, when the algorithm says "non-bipartite", it is correct as the graph has an odd cycle. □

Some Remarks

① This provides an algorithmic proof that a graph is bipartite iff it has no odd cycles.

② This also provides a linear time algorithm to find an odd cycle of an undirected graph.

③ Having an odd cycle is a "short proof" of non-bipartiteness.

It is much better than saying "we tried all bipartitions but all failed".

References: [KT 3.1, 3.2], [DPV 4.1, 4.2], [CLRS 22.1, 22.2].