CS 341 - Algorithms, Spring 2017. University of Waterloo

Lecture 5: Breadth first search

We begin to study simple graph algorithms based on graph searching.

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

Today we study BFS and see some simple applications.

Graphs

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

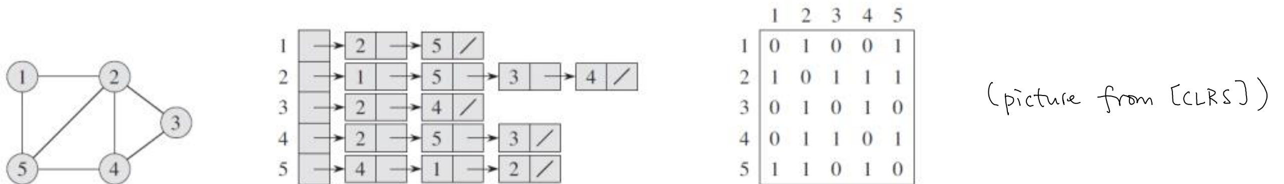
Graph representations

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 $A[i,j]=1$ if and only if $ij \in E$ (and $A[i,j]=0$ if and only if $ij \notin E$).

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 usually 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.

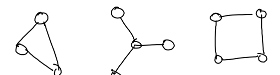
The adjacency list representation allows us to design algorithms with running time $O(m+n)$.

Graph connectivity

Given an undirected graph, we say that two vertices $u,v \in V$ are connected if there is a path from u to v.

A subset $S \subseteq V$ 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. three components

Some of the most basic questions about a graph are:

- to determine whether it is connected;
- to find all the connected components;
- to determine whether u,v are connected for given $u,v \in V$;
- to determine the shortest path connecting 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 you are searching for a person in a social network.

A natural strategy is to ask your friends, and then ask your friends to ask their friends, and so on.

A basic version of BFS can be described as follows.

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

Output: all vertices reachable from s

```
(Initialization)  visited[v] = false   for all v ∈ 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 a vertex u is dequeued, the for loop is executed $\deg(u)$ times.

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

Claim 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] is true then there is a path from S to V.

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

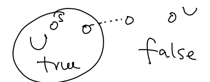
Now, suppose visited[v] is set to be true in the current step, inside the for loop of u. Then visited[u] = true at that time (because it was put on the queue).

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

Extending this path with the edge uv, there is a path from s to v.

$\Rightarrow$) Let U be the set of vertices with visited[v] = true at the end of the algorithm.

Then there are no edges with one endpoint in U and one endpoint in $V-S$, 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 but visited[v] = false.

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

With this claim, we see that BFS can be used to determine whether a graph is connected or not (by checking whether visited[v] = true for all $v \in V$), find the connected component containing v (the set of vertices with visited[v] = true), and whether there is a path from s to t (by checking whether visited[t] = true), all in linear time.

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

BFS tree

How to trace back a path from v to s?

This follows from the proof of the claim above.

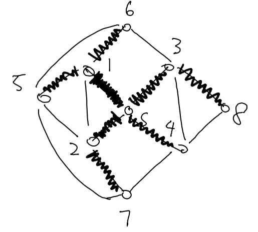
We can add an array parent[v].

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 s is reached.

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

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



Why it is a tree in the connected component in S ?

Say the connected component has n vertices.

Every vertex has one edge to its parent.

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

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

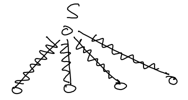
Therefore the edges (v, parent[v]) form a tree in the component containing s.

Shortest paths

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

To see this, let's think about how the BFS tree is created.

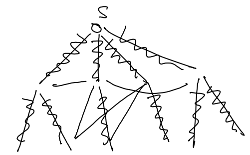
These edges record the first edges to visit a vertex.



Initially, s is the only vertex in the queue, and then every neighbor of s is visited with s as 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 other vertices (with distance at least two) are put in the queue.

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



Then, all vertices with distance one will be dequeued first, and all vertices with distance two will be enqueued before all other vertices of distance ≥ 3.

Repeat this argument inductively will show that the distances 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...)

To summarize, below is a BFS algorithm with the features 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 $v \in V$.

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.
```

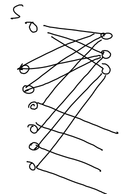
Exercise: Write the code for printing a shortest path from v to s.

Bipartite graphs

One application of BFS is to check whether a graph is bipartite.

There is not much creativity required 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, and so on.



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

Algorithm: let $L := \{v \in V \mid dist(s,v) \text{ is even}\}$ and $R := \{v \in V \mid dist(s,v) \text{ is odd}\}$.

If there are edges between two vertices in L or two vertices in R, return "non-bipartite".

Otherwise, return (L,R) as the bipartition.

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

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

Correctness: It is clear that if the algorithm returns (L,R) as the bipartition, then the graph is bipartite as there are no edges within L and R.

It remains to check whether the algorithm says "non-bipartite", the graph is really bipartite.

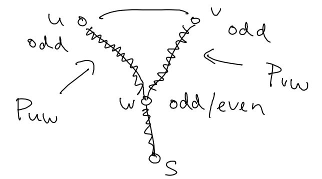
When can we say for sure a graph is non-bipartite?

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

One easy condition is when the graph has an odd cycle.

We will show that when the algorithm says "non-bipartite", the graph has an odd cycle.

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



We look at the BFS tree.

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

Since $dist(s,u)$ and $dist(s,v)$ are odd, regardless of whether $dist(w,s)$ is odd or even,

the sum of the path lengths of uw and vw is even.

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

So, when the algorithm says "non-bipartite", it is correct because the graph has an odd cycle. $\square$

Remark: - This also shows a linear time algorithm to find an odd cycle in the graph.

- We gave an algorithmic proof that a graph is bipartite if and only if it has no odd cycles.

- Having an odd cycle is a "short proof" that a graph is non-bipartite.

It is a much better explanation than say "try all possibilities but didn't work".

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