CS120: Intro. to Algorithms and their Limitations	Prof. Salil Vadhan
Lecture 12: Graph Search (cont.)	
Harvard SEAS - Fall 2021	*Oct. 12, 2021*

1 Announcements

Recommended Reading:

- Roughgarden II Sec 7.3–7.4, 8.3
- CLRS 22.0–22.2
- Salil OH Wed 10-11 on zoom
- Midterm in class Thursday!

2 Representations of Graphs

By convention, we usually use n to denote the number of vertices in a graph, m the number of edges.

Adjacency Matrix Representation:

A matrix A with $|V|$ rows and $|V|$ columns indexed by elements of V, where

$$A_{u,v} = \begin{cases} 1 & (u,v) \in E \\ 0 & \text{else.} \end{cases}$$

Adjacency List Representation:

Here we have two arrays, an outdegree array A_{deg} where

$$A_{deg}[v] = \deg(v)$$

and a neighbor array

$$\text{Nbr}[v] = \{u : (v,u) \in E\}$$

That is, each element of $\text{Nbr}[v]$ is an array holding the neighbors of v.

Q: What are the sizes of these representations?

Adjacency Matrix Representation requires an $n \times n$ matrix of Boolean values, and so takes $\Theta(n^2)$ space. Adjacency List Representation requires $\Theta(n+m)$ space, since we require two arrays of length n and m total space for storing neighbors.

Thus, the adjacency list representation is more space efficient, as long as the number of edges is below n^2, and is much more compact for *sparse* graphs (where $m = O(n)$). But note that our adjacency matrix requires n^2 *bits*, whereas the adjacency list requires m *words*, so for *dense* graphs ($m = \Omega(n^2)$)we can make the adjacency matrix a bit more compact by packing $w \geq \log_2 n$ bits into a w-bit word.

Q: How much time to convert between them?
Time $\Theta(n^2)$ to convert between them. (The lower bound comes from the fact that the adjacency matrix takes n^2 time to read or write, so we can't go faster.)
Q: Which do we prefer for algorithms?
Except when otherwise stated, we will use the *adjacency list* representation of graphs. This was important for achieving $O(n+m)$ runtime in BFS, as we needed to be able to enumerate the vertices leaving a vertex u in time $O(1 + d_{out}(u))$, rather than time $O(n)$.

3 More Graph Search

Q: How to actually find a shortest *path*, not just the distance?

Maintain an auxiliary array A_{pred} of size $|V|$, where $A_{pred}[v]$ holds the vertex u that we "discovered" u from. That is, if we add v to the frontier when exploring the neighbors of u, set $A_{pred}[v] = u$. After the completion of BFS, we can reconstruct the path from s to t using this predecessor array.
Observation: BFS actually solves the following computational problem:

Input	: A digraph $G = (V, E)$ and a vertex $s \in V$
Output	: For every vertex v, $\text{dist}_G(s, v)$ and, if $\text{dist}_G(s, v) < \infty$, a path p_v from s to v of length $\text{dist}_G(s, v)$

Computational Problem SingleSourceShortestPaths

We have proven:

Theorem 3.1. *There is an algorithm that solves SingleSourceShorestPaths in time $O(n + m)$ on digraphs with n vertices and m edges in adjacency list representation.*

The algorithm we have seen (BFS) only works on unweighted graphs; algorithms for weighted graphs are covered in CS124.

4 Other Forms of Graph Search

Another very useful form of graph search that you may have seen is *depth-first search* (DFS). We won't cover it in CS120, but DFS and some of its applications are covered in CS124.

We will, however, briefly discuss a randomized form of graph search, namely *random walks*, and use it to solve the *decision* problem of STConnectivity on undirected graphs.

Input	: A graph $G = (V, E)$ and vertices $s, t \in V$
Output	: YES if there is a path from s to t in G, and NO otherwise

Computational Problem UndirectedSTconnectivity

`RandomWalk`(G, s, ℓ)
Input : A digraph $G = (V, E)$, a vertices $s, t \in V$, and a walk-length ℓ
Output : YES or NO
$v = s$;
foreach $i = 1, \ldots, \ell$ **do**
    if $v = t$ **then return** *YES*;
    $j =$ `random`$(d_{out}(v))$;
    $v = j$'th out-neighbor of v;
return ∞

Q: What is the advantage of this algorithm over BFS?
While BFS needs $\Omega(n)$ words of memory in addition to the space required to store the input, this algorithm uses a *constant* number of words of memory while running.

It can be shown that if G is an *undirected* graph with n vertices and m edges, then for an appropriate choice of $\ell = O(mn)$, with high probability `RandomWalk`(G, s, ℓ) will visit all vertices reachable from s. Thus, we obtain a *Monte Carlo* algorithm for UndirectedSTConnectivity.

Theorem 4.1. *UndirectedSTConnectivity can be solved by a Monte Carlo randomized algorithm with arbitrarily small error probability in time $O(mn)$ using only $O(1)$ words of memory in addition to the input.*